A Dynamic Approach to Characterizing Termination of General Logic Programs

YI-DONG SHEN
Chinese Academy of Sciences
JIA-HUAI YOU, LI-YAN YUAN and SAMUEL S. P. SHEN
University of Alberta
and
QIANG YANG
Simon Fraser University

We present a new characterization of termination of general logic programs. Most existing termination analysis approaches rely on some static information about the structure of the source code of a logic program, such as modes/types, norms/level mappings, models/interargument relations, and the like. We propose a dynamic approach that employs some key dynamic features of an infinite (generalized) SLDNF-derivation, such as repetition of selected subgoals and recursive increase in term size. We also introduce a new formulation of SLDNF-trees, called generalized SLDNF-trees. Generalized SLDNF-trees deal with negative subgoals in the same way as Prolog and exist for any general logic programs.

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; D.1.2 [**Programming Techniques**]: Automatic Programming; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Logic and constraint programming*

General Terms: Languages, Theory

Additional Key Words and Phrases: Termination analysis, dynamic characterization, Prolog

1. INTRODUCTION

For a program in any computer language, in addition to having to be logically correct, it should be terminating. Due to frequent use of recursion in logic

The work of Yi-Dong Shen is supported in part by the Chinese National Natural Science Foundation and Trans-Century Training Program Foundation for the Talents by the Chinese Ministry of Education. Qiang Yang is supported by NSERC and IRIS III grants.
Authors' addresses: Yi-Dong Shen, Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100080, China; email: ydshen@ios.ac.cn; Jia-Huai You and Li-Yan Yuan, Department of Computing Science, University of Alberta, Edmonton, Canada T6G 2H1; email: {you,yuan}@cs.ualberta.ca; Samuel S. P. Shen, Department of Mathematics, University of Alberta, Edmonton, Canada T6G 2H1; email: shen@ualberta.ca; Qiang Yang, School of Computing Science, Simon Fraser University, Burnaby, BC, Canada V5A 1S6; email: qyang@cs.sfu.ca.

programming, however, a logic program may more likely be non-terminating than a procedural program. Termination of logic programs then becomes an important topic in logic programming research. Because the problem is extremely hard (undecidable in general), it has been considered as a *never-ending story*; see Schreye and Decorte [1993] for a comprehensive survey.

The goal of termination analysis is to establish a characterization of termination of a logic program and design algorithms for automatic verification. A lot of methods for termination analysis have been proposed in the last decade. A majority of these existing methods are the *norm-* or *level mapping-based* approaches, which consist of inferring mode/type information, inferring norms/level mappings, inferring models/interargument relations, and verifying some well-founded conditions (constraints). For example, Ullman and Van Gelder [1988] and Plümer [1990b, 1990a] focused on establishing a decrease in term size of some recursive calls based on interargument relations; Apt, Bezem and Pedreschi [Apt and Pedreschi 1993; Bezem 1992], and Bossi, Cocco and Fabris [Bossi et al. 1994] provided characterizations of Prolog left-termination based on level mappings/norms and models; Verschaetse [1992], Decorte, De Schreye and Fabris [Decorte et al. 1993], and Martin, King and Soper [Martin et al. 1997] exploited inferring norms/level mappings from mode and type information; De Schreye and Verschaetse [Schreye and Verschaetse 1995], Brodsky and Sagiv [1991], and Lindenstrauss and Sagiv [1997] discussed automatic inference of interargument/size relations; De Schreye, Verschaetse and Bruynooghe [Schreye et al. 1992] addressed automatic verification of the well-founded constraints. Very recently, Decorte, De Schreye and Vandecasteele [Decorte et al. 1999] presented an elegant unified termination analysis that integrates all the above components to produce a set of constraints that, when solvable, yields a termination proof.

It is easy to see that the above methods are *compile-time* (or *static*) approaches in the sense that they make termination analysis only relying on some *static* information about the structure (of the source code) of a logic program, such as modes/types, norms (i.e. term sizes of atoms of clauses)/level mappings, models/interargument relations, and the like. Our observation shows that some *dynamic* information about the structure of a concrete infinite SLDNF-derivation, such as *repetition* of selected subgoals and *recursive increase* in term size, plays a crucial role in characterizing the termination. Such dynamic features are hard to capture by applying a compile-time approach. This suggests that methods of extracting and utilizing dynamic features for termination analysis are worth exploiting.

In this note, we present a *dynamic* approach by employing dynamic features of an infinite (generalized) SLDNF-derivation to characterize termination of general logic programs. In Section 2, we introduce a notion of a generalized SLDNF-tree, which is the basis of our method. Roughly speaking, a generalized SLDNF-tree is a set of standard SLDNF-trees augmented with an ancestor-descendant relation on their subgoals. In Section 3, we define a key concept, *loop goals*, which captures both repetition of selected subgoals and recursive increase in term size of these subgoals. We then prove a necessary and sufficient condition for an infinite generalized SLDNF-derivation in terms of loop goals.

This condition allows us to establish a dynamic characterization of termination of general logic programs (Section 4). In Section 5, we mention the related work, and in Section 6 we conclude the article with our future work.

1.1 Preliminaries

We present our notation and review some standard terminology of logic programs as described in Lloyd [1987].

Variables begin with a capital letter, and predicate, function and constant symbols with a lower case letter. A term is a constant, a variable, or a function of the form $f(T_1, \dots, T_m)$ where f is a function symbol and the T_is are terms. An atom is of the form $p(T_1, \dots, T_m)$ where p is a predicate symbol and the T_is are terms. A literal is of the form A or $\neg A$ where A is an atom. Let A be an atom/term. The size of A, denoted $|A|$, is the number of occurrences of function symbols, variables and constants in A. By $\{A_i\}_{i=1}^{n}$ we denote a sequence A_1, $A_2, \dots, A_n$. Two atoms, A and B, are said to be *variants* if they are the same up to variable renaming.

Lists are commonly used terms. A list is of the form [] or $[T|L]$ where T is a term and L is a list. For our purpose, the symbols [,] and | in a list are treated as function symbols.

Definition 1.1. A (general) *logic program* is a finite set of clauses of the form

$$A \leftarrow L_1, \dots, L_n$$

where A is an atom and L_is are literals. When $n = 0$, the "$\leftarrow$" symbol is omitted. A is called the *head* and $L_1, \dots, L_n$ is called the *body* of the clause. If a general logic program has no clause with negative literals like $\neg A$ in its body, it is called a *positive* logic program.

Definition 1.2. A *goal* is a headless clause $\leftarrow L_1, \dots, L_n$ where each literal L_i is called a *subgoal*. $L_1, \dots, L_n$ is called a (concrete) *query*.

The initial goal, $G_0 = \leftarrow L_1, \dots, L_n$, is called a *top* goal. Without loss of generality, we shall assume throughout the paper that a top goal consists only of one atom, that is, $n = 1$ and L_1 is a positive literal.

Definition 1.3. A *control strategy* consists of two rules, one for selecting a goal from among a set of goals and the other for selecting a subgoal from the selected goal.

The second rule in a control strategy is usually called a *selection* or *computation* rule in the literature. To facilitate our presentation, throughout the article we choose to use the best-known *depth-first, left-most* control strategy (used in Prolog) to describe our approach (It can be adapted to any other fixed control strategies). So the *selected* subgoal in each goal is the left-most subgoal. Moreover, the clauses in a logic program are used in their textual order.

Trees are commonly used to represent the search space of a top-down proof procedure. For convenience, a node in such a tree is represented by $N_i : G_i$ where N_i is the name of the node and G_i is a goal labeling the node. Assume no two nodes have the same name. Therefore, we can refer to nodes by their names.

2. GENERALIZED SLDNF-TREES

In order to characterize infinite derivations more precisely, in this section we extend the standard SLDNF-trees [Lloyd 1987] to include some new features. We first define the ancestor-descendant relation on selected subgoals. Informally, A is an ancestor subgoal of B if the proof of A needs (or in other words goes via) the proof of B. For example, let $M :\leftarrow A, A_1, \ldots, A_m$ be a node in an SLDNF-tree, and $N :\leftarrow B_1, \ldots, B_n, A_1, \ldots, A_m$ be a child node of M that is generated by resolving M on the subgoal A with a clause $A \leftarrow B_1, \ldots, B_n$. Then A at M is an ancestor subgoal of all B_is at N. However, such relationship does not exist between A at M and any A_j at N. It is easily seen that all B_is at N inherit the ancestor subgoals of A at M.

The ancestor-descendant relation can be explicitly expressed using an *ancestor list*. The ancestor list of a subgoal A at a node M, denoted $AL_{A@M}$, is of the form $\{(N_1, D_1), \ldots, (N_l, D_l)\}$ ($l \geq 0$), where for each $(N_i, D_i) \in AL_{A@M}$, N_i is a node name and D_i a subgoal such that D_i at N_i is an ancestor subgoal of A at M. For instance, in the above example, the ancestor list of each B_i at node N is $AL_{B_i@N} = \{(M, A)\} \cup AL_{A@M}$ and the ancestor list of each A_i at node N is $AL_{A_i@N} = AL_{A_i@M}$.

Let $N_i : G_i$ and $N_k : G_k$ be two nodes and A and B be the selected subgoals in G_i and G_k, respectively. We use $A \prec_{anc} B$ to denote that A is an ancestor subgoal of B. When A is an ancestor subgoal of B, we refer to B as a *descendant subgoal* of A, N_i as an *ancestor node* of N_k, and N_k as a *descendant node* of N_i.

Augmenting SLDNF-trees with ancestor lists leads to the following definition of SLDNF*-trees.

Definition 2.1 *(SLDNF*-trees).* Let P be a general logic program, $G_r = \leftarrow A_r$ a goal with A_r an atom, and R the depth-first, left-most control strategy. The *SLDNF*-tree* $T_{N_r:G_r}$ for $P \cup \{G_r\}$ via R is defined inductively as follows.

(1) $N_r : G_r$ is its root node, and the tree is completed once a node marked as **LAST** is generated or when all its leaf nodes have been marked as $\Box_t$, $\Box_f$ or $\Box_{fl}$.

(2) For each node $N_i :\leftarrow L_1, \ldots, L_m$ in the tree that is selected by R, if $m = 0$ then (1) N_i is a *success* leaf marked as $\Box_t$ and (2) if $AL_{A_r@N_r} \neq \emptyset$ then N_i is also a node marked as **LAST**. Otherwise (i.e. $m > 0$), we distinguish between the following three cases.

   (a) L_1 is a positive literal. For each clause $B \leftarrow B_1, \ldots, B_n$ in P such that L_1 and B are unifiable, N_i has a child node
   $$N_s :\leftarrow (B_1, \ldots, B_n, L_2, \ldots, L_m)\theta$$
   where θ is an mgu (i.e. most general unifier) of L_1 and B, the ancestor list for each $B_k\theta$, $k \in \{1, \ldots, n\}$, is $AL_{B_k\theta@N_s} = \{(N_i, L_1)\} \cup AL_{L_1@N_i}$, and the ancestor list for each $L_k\theta$, $k \in \{2, \ldots, m\}$, is $AL_{L_k\theta@N_s} = AL_{L_k@N_i}$. If there exists no clause in P whose head can unify with L_1 then N_i has a single child node—a *failure* leaf marked as $\Box_f$.

(b) $L_1 = \neg A$ is a ground negative literal. Let $T_{N_{i+1}:\leftarrow A}$ be an (subsidiary) SLDNF*-tree for $P \cup \{\leftarrow A\}$ via R with $AL_{A@N_{i+1}} = AL_{L_1@N_i}$. We have the following four cases:
   i. $T_{N_{i+1}:\leftarrow A}$ has a success leaf. Then N_i has a single child node—a failure leaf marked as $\Box_f$.
   ii. $T_{N_{i+1}:\leftarrow A}$ has no success leaf but has a flounder leaf. Then N_i has a single child node—a flounder leaf marked as $\Box_{fl}$.
   iii. All branches of $T_{N_{i+1}:\leftarrow A}$ end with a failure leaf. Then N_i has a single child node

$$N_s :\leftarrow L_2, \ldots, L_m$$

   with $AL_{L_k@N_s} = AL_{L_k@N_i}$ for each $k \in \{2, \ldots, m\}$.
   iv. Otherwise, N_i has no child node. It is the last node of $T_{N_r:G_r}$ so that it is marked as **LAST**.

(c) $L_1 = \neg A$ is a non-ground negative literal. Then N_i has a single child node—a *flounder* leaf marked as $\Box_{fl}$.

Starting from the root node $N_r : G_r$, we expand the nodes of the SLDNF*-tree $T_{N_r:G_r}$ following the depth-first order. The expansion for $T_{N_r:G_r}$ stops when either a node marked as **LAST** is generated or all of its leaf nodes have been marked as $\Box_t$, $\Box_f$ or $\Box_{fl}$.

In this article we do not consider floundering—a situation where a non-ground negative subgoal is selected by R (see the case 2c). See Chan [1988] for a discussion of that topic.

We first prove the following.

THEOREM 2.2. *Let $T_{N_{i+1}:\leftarrow A}$ be a subsidiary SLDNF*-tree built for proving a negative subgoal $L_1 = \neg A$ at a node N_i (see the case 2b). Then $AL_{A@N_{i+1}} \neq \emptyset$.*

PROOF. Note that $AL_{A@N_{i+1}} = AL_{L_1@N_i}$. Since the subgoal L_1 at N_i is negative, N_i cannot be the root node of the SLDNF*-tree that contains N_i. Therefore, L_1 at N_i has at least one ancestor subgoal (i.e. the subgoal at the root node of the tree), which means $AL_{A@N_{i+1}} \neq \emptyset$. $\Box$

In order to solve a top goal $G_0 = \leftarrow A_0$, we build an SLDNF*-tree $T_{N_0:\leftarrow A_0}$ for $P \cup \{G_0\}$ via R with $AL_{A_0@N_0} = \emptyset$. It is easy to see that $T_{N_0:\leftarrow A_0}$ is an enhancement of the standard SLDNF-tree for $P \cup \{G_0\}$ via R with the following three new features.

(1) Each node N_i is associated with an ancestor list $AL_{L_j@N_i}$ for each L_j of its subgoals. In particular, subgoals of a subsidiary SLDNF*-tree $T_{N_{i+1}:\leftarrow A}$ built for solving a negative subgoal $L_1 = \neg A$ at N_i inherit the ancestor list $AL_{L_1@N_i}$ (see the case 2b). This bridges the ancestor-descendant relationships across SLDNF*-trees and is especially useful in identifying infinite derivations across SLDNF*-trees (see Example 2.1). Note that a negative subgoal will never be an ancestor subgoal.
(2) In a standard SLDNF-tree, to handle a ground negative subgoal $L_1 = \neg A$ at N_i a full subsidiary SLDNF-tree FT for $P \cup \{\leftarrow A\}$ via R must be generated. In an SLDNF*-tree, however, the subsidiary SLDNF*-tree $T_{N_{i+1}:\leftarrow A}$ may not

include all branches of FT because it will terminate at the first success leaf (see the case 2 where by Theorem 2.2 $AL_{A@N_{i+1}} \neq \emptyset$). The intuition behind this is that it is absolutely unnecessary to exhaust the remaining branches of FT because they would never generate any new answers for A (and $\neg A$). Such a pruning mechanism embedded in SLDNF*-trees is very useful in not only improving the efficiency of query evaluation but also avoiding some possible infinite derivations (see Example 2.2). In fact, Prolog performs the same pruning by using a *cut* operator to skip the remaining branches of FT once the first success leaf is generated (e.g. see SICStus Prolog [ISLAB 1998]).

(3) A well-known problem with the standard SLDNF-tree approach (formally called *SLDNF-resolution* [Clark 1978; Lloyd 1987]) is that for some programs, such as $P = \{A \leftarrow \neg A\}$ and $G_0 = \leftarrow A$, no SLDNF-trees exist [Apt and Doets 1994; Kunen 1989; Martelli and Tricomi 1992]. The main reason for this abnormality lies in the fact that to solve a negative subgoal $\neg A$ it generates a subsidiary SLDNF-tree FT for $P \cup \{\leftarrow A\}$ via R *which is supposed either to contain a success leaf or to consist of failure leaves*. When FT neither contains a success leaf nor finitely fails by going into an infinite derivation, the negative subgoal cannot be handled.

In contrast, SLDNF*-trees exist for any general logic programs. A ground negative subgoal $\neg A$ at a node N_i succeeds if all branches of the subsidiary SLDNF*-tree $T_{N_{i+1}:\leftarrow A}$ end with a failure leaf (see the Case 2(b)iii), and fails if $T_{N_{i+1}:\leftarrow A}$ has a success leaf (see the case 2(b)i). Otherwise, the value of the subgoal $\neg A$ is undetermined and thus N_i is marked as **LAST**, showing that it is the last node of the underlying SLDNF*-tree that can be finitely generated (see the Case 2(b)iv).[1] The tree is then completed here.

For convenience, we use dotted edges "$\cdots\triangleright$" to connect parent and child (subsidiary) SLDNF*-trees, so that infinite derivations across SLDNF*-trees can be clearly identified. Formally, we have

Definition 2.3. Let P be a general logic program, G_0 a top goal and R the depth-first, left-most control strategy. Let $T_{N_0:G_0}$ be the SLDNF*-tree for $P \cup \{G_0\}$ via R with $AL_{A_0@N_0} = \emptyset$. A *generalized SLDNF-tree* for $P \cup \{G_0\}$ via R, denoted GT_{G_0}, is rooted at $N_0 : G_0$ and consists of $T_{N_0:G_0}$ along with all its descendant SLDNF*-trees, where parent and child SLDNF*-trees are connected via "$\cdots\triangleright$".

Therefore, a path of a generalized SLDNF-tree may come across several SLDNF*-trees through dotted edges. Any such a path starting at the root node $N_0 : G_0$ is called a *generalized SLDNF-derivation*.

[1]This case occurs when either $T_{N_{i+1}:\leftarrow A}$ or some of its descendant SLDNF*-trees is infinite, or $T_{N_{i+1}:\leftarrow A}$ has an infinite number of descendant SLDNF*-trees. Note that **LAST** is used here only for the purpose of formulating an SLDNF*-tree – showing that N_i is the last node of the SLDNF*-tree. In practical implementation of SLDNF*-trees, in such a case N_i will never be marked by **LAST** since it requires an infinitely long time to build $T_{N_{i+1}:\leftarrow A}$ together with all of its descendant SLDNF*-trees. However, the SLDNF*-tree is always completed at N_i, whether N_i is marked by **LAST** or not, because (1) such a case occurs at most one time in an SLDNF*-tree and (2) it always occurs at the last generated node N_i.

$$\begin{tikzcd}
N_0\colon \leftarrow p(a) \arrow[d, "C_{p_1}"] & & \\
N_1\colon \leftarrow \neg p(f(a)) \arrow[r, dotted] & N_2\colon \leftarrow p(f(a)) \arrow[d, "C_{p_1}"] & \\
\textbf{LAST} & N_3\colon \leftarrow \neg p(f(f(a))) \arrow[r, dotted] & N_4\colon \leftarrow p(f(f(a))) \arrow[d, "C_{p_1}"] \\
 & \textbf{LAST} & \infty
\end{tikzcd}$$

Fig. 1. The generalized SLDNF-tree $GT_{\leftarrow p(a)}$.

Thus, there may occur two types of edges in a generalized SLDNF-derivation, "$\xrightarrow{C}$" and "$\cdots \triangleright$". For convenience, we use "$\Rightarrow$" to refer to either of them. Moreover, for any node $N_i : G_i$ we use L_i^1 to refer to the selected (i.e. left-most) subgoal in G_i.

Example 2.1. Let P_1 be a general logic program and G_0 a top goal, given by

$$\begin{array}{lll} P_1: & p(X) \leftarrow \neg p(f(X)). & C_{p_1} \\ G_0: & \leftarrow p(a). & \end{array}$$

The generalized SLDNF-tree $GT_{\leftarrow p(a)}$ for $P_1 \cup \{G_0\}$ is shown in Figure 1, where ∞ represents an infinite extension. Note that to expand the node N_1, we build a subsidiary SLDNF*-tree $T_{N_2:\leftarrow p(f(a))}$. Since $T_{N_2:\leftarrow p(f(a))}$ neither contains a success leaf nor finitely fails (i.e. not all of its leaf nodes are marked as $\Box_f$), N_1 is the last node of $T_{N_0:\leftarrow p(a)}$, marked as **LAST**. We see that $GT_{\leftarrow p(a)}$ is infinite, although all of its SLDNF*-trees are finite.

Example 2.2. Consider the following general logic program and top goal.

$$\begin{array}{lll} P_2: & p \leftarrow \neg q. & C_{p_1} \\ & q. & C_{q_1} \\ & q \leftarrow q. & C_{q_2} \\ G_0: & \leftarrow p. & \end{array}$$

The generalized SLDNF-tree $GT_{\leftarrow p}$ for $P_2 \cup \{G_0\}$ is depicted in Figure 2 (a). $GT_{\leftarrow p}$ consists of two SLDNF*-trees, $T_{N_0:\leftarrow p}$ and $T_{N_2:\leftarrow q}$, which are constructed as follows. Initially, $T_{N_0:\leftarrow p}$ has only the root node $N_0 :\leftarrow p$. Expanding the root node against the clause C_{p_1} leads to the child node $N_1 :\leftarrow \neg q$. We then build a subsidiary SLDNF*-tree $T_{N_2:\leftarrow q}$ for $P_2 \cup \{\leftarrow q\}$ via the depth-first, left-most control strategy, where the expansion stops right after the node N_3 is marked as **LAST**. Since $T_{N_2:\leftarrow q}$ has a success leaf, N_1 gets a failure child node N_5. $T_{N_0:\leftarrow p}$ is then completed.

For the purpose of comparison, the standard SLDNF-trees for $P_2 \cup \{\leftarrow p\}$ are shown in Figure 2 (b). Note that Figure 2 (a) is finite, whereas Figure 2 (b) is not.

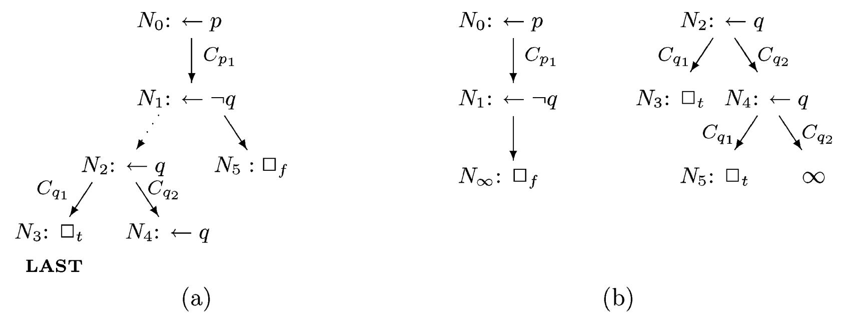

Fig. 2. The generalized SLDNF-tree $GT_{\leftarrow p}$ (a) and its two corresponding standard SLDNF-trees (b).

3. CHARACTERIZING AN INFINITE GENERALIZED SLDNF-DERIVATION

In this section we establish a necessary and sufficient condition for an infinite generalized SLDNF-derivation. We begin by introducing a few concepts.

Definition 3.1. Let T be a term or an atom and S be a string that consists of all predicate symbols, function symbols, constants and variables in T, which is obtained by reading these symbols sequentially from left to right. The *symbol string* of T, denoted S_T, is the string S with every variable replaced by $\mathcal{X}$.

For instance, let $T_1 = a$, $T_2 = f(X, g(X, f(a, Y)))$ and $T_3 = [X, a]$. Then $S_{T_1} = a$, $S_{T_2} = f\mathcal{X}g\mathcal{X}fa\mathcal{X}$ and $S_{T_3} = [\mathcal{X}|[a|[]]]$. Note that $[X, a]$ is a simplified representation for the list $[X|[a|[]]]$.

Definition 3.2. Let S_{T_1} and S_{T_2} be two symbol strings. S_{T_1} is a *projection* of S_{T_2}, denoted $S_{T_1} \subseteq_{proj} S_{T_2}$, if S_{T_1} is obtained from S_{T_2} by removing zero or more elements.

For example, $a\mathcal{X}\mathcal{X}bc \subseteq_{proj} fa\mathcal{X}e\mathcal{X}b\mathcal{X}cd$. It is easy to see that the relation $\subseteq_{proj}$ is reflexive and transitive. That is, for any symbol strings S_1, S_2 and S_3, we have $S_1 \subseteq_{proj} S_1$, and that $S_1 \subseteq_{proj} S_2$ and $S_2 \subseteq_{proj} S_3$ implies $S_1 \subseteq_{proj} S_3$.

Definition 3.3. Let $A_1 = p(.)$ and $A_2 = p(.)$ be two atoms. A_1 is said to *loop into* A_2, denoted $A_1 \rightsquigarrow_{loop} A_2$, if $S_{A_1} \subseteq_{proj} S_{A_2}$. Let $N_i : G_i$ and $N_j : G_j$ be two nodes in a generalized SLDNF-derivation with $L_i^1 \prec_{anc} L_j^1$ and $L_i^1 \rightsquigarrow_{loop} L_j^1$. Then G_j is called a *loop goal* of G_i.

The following result is immediate.

THEOREM 3.4

(1) *The relation* $\rightsquigarrow_{loop}$ *is reflexive and transitive.*
(2) *If* $A_1 \rightsquigarrow_{loop} A_2$ *then* $|A_1| \leq |A_2|$.
(3) *If* G_3 *is a loop goal of* G_2 *that is a loop goal of* G_1 *then* G_3 *is a loop goal of* G_1.

Observe that since a logic program has only a finite number of clauses, an infinite generalized SLDNF-derivation results from repeatedly applying the same set of clauses, which leads to infinite repetition of selected variant subgoals or infinite repetition of selected subgoals with recursive increase in term size. By recursive increase of term size of a subgoal A from a subgoal B we mean that A is B with a few function/constant/variable symbols added and possibly with some variables changed to different variables. Such crucial dynamic characteristics of an infinite generalized SLDNF-derivation are captured by loop goals, as shown by the following principal theorem.

THEOREM 3.5. *D is an infinite generalized SLDNF-derivation if and only if it is of the form*

$$N_0 : G_0 \Rightarrow \cdots N_{g_1} : G_{g_1} \Rightarrow \cdots N_{g_2} : G_{g_2} \Rightarrow \cdots N_{g_3} : G_{g_3} \Rightarrow \cdots$$

such that for any $j \geq 1$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}.

We need Higman's Lemma to prove this theorem.[2]

LEMMA 3.6 (HIGMAN'S LEMMA [HIGMAN 1952; BOL 1991]). *Let $\{A_i\}_{i=1}^{\infty}$ be an infinite sequence of strings over a finite alphabet Σ. Then for some i and $k > i$, $A_i \subseteq_{proj} A_k$*

The following result follows from Lemma 3.6.

LEMMA 3.7. *Let $\{A_i\}_{i=1}^{\infty}$ be an infinite sequence of strings over a finite alphabet Σ. Then there is an infinite increasing integer sequence $\{n_i\}_{i=1}^{\infty}$ such that for all i $A_{n_i} \subseteq_{proj} A_{n_{i+1}}$.*

PROOF.[3] Suppose this is not true. Let us take a finite maximal subsequence

$$A_{n_1} \subseteq_{proj} A_{n_2} \subseteq_{proj} \cdots \subseteq_{proj} A_{n_{k_1}}$$

The subsequence is maximal in the sense that for no $i > n_{k_1}$ do we have $A_{n_{k_1}} \subseteq_{proj} A_i$. We know that such a subsequence with length at least 2 must exist from Lemma 3.6 and the assumption that the assertion of the lemma does not hold for the sequence $\{A_i\}_{i=1}^{\infty}$. Now look at the elements of the original sequence with indices larger than n_{k_1} and take another such finite maximal subsequence from them. Continuing in this way, we get infinitely many such maximal subsequences. Let $\{A_{n_{k_i}}\}_{i=1}^{\infty}$ be the sequence of last elements of the maximal subsequences. By Lemma 3.6, this sequence has two elements, $A_{n_{k_i}}$ and $A_{n_{k_j}}$ with $n_{k_i} < n_{k_j}$, such that $A_{n_{k_i}} \subseteq_{proj} A_{n_{k_j}}$. This contradicts the assumption that $A_{n_{k_i}}$ is the last element of some finite maximal subsequence. □

The following lemma is needed to prove Theorem 3.5.

LEMMA 3.8. *Let D be an infinite generalized SLDNF-derivation. Then there are infinitely many goals $G_{g_1}, G_{g_2}, \ldots$ in D such that for any $j \geq 1$, $L_j^1 \prec_{anc} L_{j+1}^1$.*

PROOF. Let D be of the form

$$N_0 : G_0 \Rightarrow N_1 : G_1 \Rightarrow \cdots \Rightarrow N_i : G_i \Rightarrow N_{i+1} : G_{i+1} \Rightarrow \cdots$$

[2]It is one of the anonymous reviewers who brought this lemma to our attention.
[3]This proof is suggested by an anonymous reviewer.

Consider derivation steps like $N_i : G_i \xrightarrow{C} N_{i+1} : G_{i+1} \cdots \triangleright N_{i+2} : G_{i+2}$, where L_i^1 is a positive subgoal and $L_{i+1}^1 = \neg A$ a negative subgoal. So $L_{i+2}^1 = A$. We see that both L_i^1 and L_{i+1}^1 need the proof of L_{i+2}^1. Moreover, given L_{i+2}^1 the provability of L_i^1 does not depend on L_{i+1}^1. Since L_{i+1}^1 has no descendant subgoals, removing it would affect neither the provability nor the ancestor-descendant relationships of other subgoals in D. Therefore, we delete L_{i+1}^1 by marking N_{i+1} with #.

For each derivation step $N_i : G_i \xrightarrow{C} N_{i+1} : G_{i+1}$, where L_i^1 is a positive subgoal and $C = A \leftarrow B_1, \ldots, B_n$ such that $A\theta = L_i^1\theta$ under an mgu θ, we do the following:

(1) If $n = 0$, which means L_i^1 is proved at this step, mark node N_i with #.
(2) Otherwise, the proof of L_i^1 needs the proof of $B_j\theta$ ($j = 1, \ldots, n$). If all descendant nodes of N_i in D have been marked with #, which means that all $B_j\theta$ have been proved at some steps in D, mark node N_i with #.

Note that the root node N_0 will never be marked with #, for otherwise G_0 would have been proved and D should have ended at a success or failure leaf. After the above marking process, let D become

$$N_0 : G_0 \Rightarrow \cdots \Rightarrow N_{i_1} : G_{i_1} \Rightarrow \cdots \Rightarrow N_{i_2} : G_{i_2} \Rightarrow \cdots \Rightarrow N_{i_k} : G_{i_k} \Rightarrow \cdots$$

where all nodes except $N_0, N_{i_1}, N_{i_2}, \ldots, N_{i_k}, \ldots$ are marked with #. Since we use the depth-first, left-most control strategy, for any $j \geq 0$ the proof of $L_{i_j}^1$ needs the proof of $L_{i_{j+1}}^1$ (let $i_0 = 0$), for otherwise N_{i_j} would have been marked with #. That is, $L_{i_j}^1$ is an ancestor subgoal of $L_{i_{j+1}}^1$. Moreover, D must contain an infinite number of such nodes because if $N_{i_k} : G_{i_k}$ was the last one, which means that all nodes after N_{i_k} were marked with #, then $L_{i_k}^1$ would be proved, so that N_{i_k} should be marked with #, a contradiction. □

We are ready to prove Theorem 3.5.

PROOF (PROOF OF THEOREM 3.5). ($\Longleftarrow$) Straightforward.

($\Longrightarrow$) By Lemma 3.8, D contains an infinite sequence of selected subgoals $H_1 = \{L_{j_i}^1\}_{i=1}^{\infty}$ such that for any i $L_{j_i}^1 \prec_{anc} L_{j_{i+1}}^1$. Since any logic program has only a finite number of predicate symbols, H_1 must have an infinite subsequence $H_2 = \{L_{k_i}^1\}_{i=1}^{\infty}$ such that all $L_{k_i}^1$ have the same predicate symbol, say p. We now show that H_2 has an infinite subsequence $\{L_{g_i}^1\}_{i=1}^{\infty}$ such that for any i $L_{g_i}^1 \rightsquigarrow_{loop} L_{g_{i+1}}^1$.

Let T be the (finite) set of all constant and function symbols in the logic program and let $\Sigma = T \cup \{\mathcal{X}\}$. Then the symbol string $S_{L_{k_i}^1}$ of each $L_{k_i}^1$ in H_2 is a string over Σ that begins with p. These symbol strings constitute an infinite sequence $\{pA_i\}_{i=1}^{\infty}$ with each A_i being a substring. By Lemma 3.7 there is an infinite increasing integer sequence $\{n_i\}_{i=1}^{\infty}$ such that for any i $pA_{n_i} \subseteq_{proj} pA_{n_{i+1}}$. Therefore, H_2 has an infinite subsequence $H_3 = \{L_{g_i}^1\}_{i=1}^{\infty}$ with $S_{L_{g_i}^1} = pA_{n_i}$ being the symbol string of $L_{g_i}^1$. That is, for any i $S_{L_{g_i}^1} \subseteq_{proj} S_{L_{g_{i+1}}^1}$. Thus, for any i $L_{g_i}^1 \rightsquigarrow_{loop} L_{g_{i+1}}^1$. □

4. CHARACTERIZING TERMINATION OF GENERAL LOGIC PROGRAMS

In Schreye and Decorte [1993], a generic definition of termination of logic programs is presented as follows.

Definition 4.1 ([*Schreye and Decorte* 1993]). Let P be a general logic program, S_Q a set of queries and S_R a set of selection rules. P is terminating with respect to S_Q and S_R if for each query Q_i in S_Q and for each selection rule R_j in S_R, all SLDNF-trees for $P \cup \{\leftarrow Q_i\}$ via R_j are finite.

Observe that the above definition considers finite SLDNF-trees for termination. That is, P is terminating with respect to Q_i only if all (complete) SLDNF-trees for $P \cup \{\leftarrow Q_i\}$ are finite. This does not seem to apply to Prolog where there exist cases in which P is terminating with respect to Q_i and R_j, although some (complete) SLDNF-trees for $P \cup \{\leftarrow Q_i\}$ are infinite. Example 2.2 gives such an illustration, where Prolog terminates with a negative answer to the top goal G_0.

In view of the above observation, we present the following slightly different definition of termination based on a generalized SLDNF-tree.

Definition 4.2. Let P be a general logic program, S_Q a finite set of queries and R the depth-first, left-most control strategy. P is terminating with respect to S_Q and R if for each query Q_i in S_Q, the generalized SLDNF-tree for $P \cup \{\leftarrow Q_i\}$ via R is finite.

The above definition implies that P is terminating with respect to S_Q and R if and only if there is no infinite generalized SLDNF-derivation in any generalized SLDNF-tree $GT_{\leftarrow Q_i}$. This obviously applies to Prolog. We then have the following immediate result from Theorem 3.5, which characterizes termination of a general logic program.

THEOREM 4.3. *P is terminating with respect to S_Q and R if and only if for each query Q_i in S_Q there is no generalized SLDNF-derivation in $GT_{\leftarrow Q_i}$ of the form*

$$N_0 : G_0 \Rightarrow \cdots N_{g_1} : G_{g_1} \Rightarrow \cdots N_{g_2} : G_{g_2} \Rightarrow \cdots N_{g_3} : G_{g_3} \Rightarrow \cdots$$

such that for any $j \geq 1$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}.

5. RELATED WORK

Concerning termination analysis, we refer the reader to the articles of Decorte, De Schreye and Vandecasteele [Schreye and Decorte 1993; Decorte et al. 1999] for a comprehensive bibliography. Most existing termination analysis techniques are static approaches, which only make use of the syntactic structure of the source code of a logic program to establish some well-founded conditions/constraints that, when satisfied, yield a termination proof. Since non-termination is caused by an infinite generalized SLDNF-derivation, which contains some essential dynamic characteristics that are hard to capture in a static way, static approaches appear to be less precise than a dynamic one. For example, it is difficult to apply a static approach to prove the termination of program P_2 in Example 2.2 with respect to a query pattern p.

The concept of generalized SLDNF-trees is the basis of our approach. There are several new definitions of SLDNF-trees presented in the literature, such as that of Apt and Doets [1994], Kunen [1989], or Martelli and Tricomi [1992]. Generalized SLDNF-trees have two distinct features as compared to these definitions. First, the ancestor-descendent relation is explicitly expressed (using ancestor lists) in a generalized SLDNF-tree, which is essential in identifying loop goals. Second, a ground negative subgoal $\neg A$ at a node N_i in a SLDNF*-tree $T_{N_r:G_r}$ is formulated in the same way as in Prolog, i.e. (1) the subsidiary SLDNF*-tree $T_{N_{i+1}:\leftarrow A}$ for the subgoal terminates at the first success leaf, and (2) $\neg A$ succeeds if all branches of $T_{N_{i+1}:\leftarrow A}$ end with a failure leaf and fails if $T_{N_{i+1}:\leftarrow A}$ has a success leaf. When $T_{N_{i+1}:\leftarrow A}$ goes into an infinite extension, the node N_i is treated as the last node of $T_{N_r:G_r}$, which can be finitely generated. As a result, a generalized SLDNF-tree exists for any general logic programs.

Our work is also related to loop checking—another research topic in logic programming that focuses on detecting and eliminating infinite loops. Informally, a derivation

$$N_0 : G_0 \Rightarrow N_1 : G_1 \Rightarrow \cdots \Rightarrow N_i : G_i \Rightarrow \cdots \Rightarrow N_k : G_k \Rightarrow \cdots$$

is said to step into a loop at a node $N_k : G_k$ if there is a node $N_i : G_i$ ($0 \leq i < k$) in the derivation such that G_i and G_k are *sufficiently similar*. Many mechanisms related to loop checking have been presented in the literature (e.g. Bol et al. [1991] and Shen et al. [2001]). However, most of them apply only to *SLD-derivations* for positive logic programs and thus cannot deal with infinite recursions through negation like that in Figures 1 or 2.

Loop goals are defined on a generalized SLDNF-derivation for general logic programs and can be used to define the sufficiently similar goals in loop checking. For such an application, they play a role similar to *expanded variants* as defined in Shen et al. [2001]. Informally, expanded variants are variants except that some terms may grow bigger. However, expanded variants have at least three disadvantages as compared to loop goals: their definition is less intuitive, their computation is more expensive, and they are not transitive in the sense that A being an expanded variant of B that is an expanded variant of C does not necessarily imply A is an expanded variant of C.

6. CONCLUSIONS AND FUTURE WORK

We have presented an approach to characterizing termination of general logic programs by making use of dynamic features. A concept of generalized SLDNF-trees is introduced, a necessary and sufficient condition for infinite generalized SLDNF-derivations is established, and a new characterization of termination of a general logic program is developed.

We have recently developed an algorithm for automatically predicting termination of general logic programs based on the characterization established in this article. The algorithm identifies the most-likely non-terminating programs. Let P be a general logic program, S_Q a set of queries and R the depth-first, left-most control strategy. P is said to be *most-likely* non-terminating with respect to S_Q and R if for some query Q_i in S_Q, there is a generalized SLDNF-derivation with a few (e.g. two or three) loop goals. Our experiments show that for most representative general logic programs we have collected in the literature, they are not terminating with respect to S_Q and R if and only if they are most-likely non-terminating with respect to S_Q and R. This algorithm can be incorporated into Prolog as a debugging tool, which would provide the users with valuable debugging information for them to understand the causes of non-termination.

Tabled logic programming is receiving increasing attention in the community of logic programming (e.g. [Chen and Warren 1996; Shen et al. 2002]). Verbaeten, De Schreye and Sagonas [Verbaeten et al. 2001] recently exploited termination proofs for positive logic programs with tabling. For future research, we are considering extending the work of the current article to deal with general logic programs with tabling.

ACKNOWLEDGMENTS

We are particularly grateful to Danny De Schreye, Krzysztof Apt, Jan Willem Klop and the three anonymous referees for their constructive comments on our work and valuable suggestions for its improvement. One anonymous reviewer brought the Higman's lemma to our attention. Another anonymous reviewer suggested the proof of Lemma 3.7 and presented us a different yet interesting proof to the Higman's lemma based on an idea from Dershowitz's paper [Dershowitz 1987].

REFERENCES

Apt, K. R. and Doets, K. 1994. A new definition of sldnf-resolution. *J. Logic Program. 18*, 177–190.

Apt, K. R. and Pedreschi, D. 1993. Reasoning about termination of pure prolog programs. *Information and Computation 106*, 109–157.

Bezem, M. 1992. Characterizing termination of logic programs with level mapping. *J. Logic Program. 15*, 1/2, 79–98.

Bol, R. N. 1991. Loop checking in logic programming. Ph.D. thesis, The University of Amsterdam, Amsterdam.

Bol, R. N., Apt, K. R., and Klop, J. W. 1991. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science 86*, 1, 35–79.

Bossi, A., Cocco, N., and Fabris, M. 1994. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science 124*, 1, 297–328.

Brodsky, A. and Sagiv, Y. 1991. Inference of inequality constraints in logic programs. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, Denver, 227–240.

Chan, D. 1988. Constructive negation based on the completed database. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*. MIT Press, Seattle, 111–125.

Chen, W. D. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *J. ACM 43*, 1, 20–74.

Clark, K. L. 1978. Negation as failure. In *Logic and Databases*, H. Gallaire and J. Minker, Eds. Plenum, New York, 293–322.

Decorte, S., Schreye, D. D., and Fabris, M. 1993. Automatic inference of norms: A missing link in automatic termination analysis. In *Proceedings of the 1993 International Symposium on Logic Programming*. MIT Press, Vancouver, Canada, 420–436.

Decorte, S., Schreye, D. D., and Vandecasteele, H. 1999. Constraint-based termination analysis of logic programs. *ACM Trans. Program. Lang. Syst. 21*, 6, 1137–1195.

Dershowitz, N. 1987. Termination of rewriting. *J. Symb. Computation 3*, 69–116.

Higman, G. 1952. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society 3*, 2, 326–336.

ISLAB. 1998. *SICStus Prolog User's Manual*. Intelligent Systems Laboratory, Swedish Institute of Computer Science, Available from http://www.sics.se/sicstus/docs/3.7.1/html/sicstus_toc.html.

Kunen, K. 1989. Signed data dependencies in logic programming. *J. Logic Program. 7*, 231–246.

Lindenstrauss, N. and Sagiv, Y. 1997. Automatic termination analysis of logic programs. In *Proceedings of the Fourteenth International Conference on Logic Programming*. MIT Press, Leuven, Belgium, 63–77.

Lloyd, J. W. 1987. *Foundations of Logic Programming*. Springer-Verlag, Berlin.

Martelli, M. and Tricomi, C. 1992. A new sldnf-tree. *Information Processing Letters 43*, 2, 57–62.

Martin, J. C., King, A., and Soper, P. 1997. Typed norms for typed logic programs. In *Proceedings of the 6th International Workshop on Logic Programming Synthesis and Transformation*. Springer, Stockholm, Sweden, 224–238.

Plumer, L. 1990a. *Termination Proofs for Logic Programs*. Lecture Notes in Computer Science 446, Springer-Verlag, Berlin.

Plumer, L. 1990b. Termination proofs for logic programs based on predicate inequalities. In *Proceedings of the Seventh International Conference on Logic Programming*. MIT Press, Cambridge, MA, 634–648.

Schreye, D. D. and Decorte, S. 1993. Termination of logic programs: the never-ending story. *J. Logic Program. 19*, 20, 199–260.

Schreye, D. D. and Verschaetse, K. 1995. Deriving linear size relations for logic programs by abstract interpretation. *New Generation Computing 13*, 2, 117–154.

Schreye, D. D., Verschaetse, K., and Bruynooghe, M. 1992. A framework for analyzing the termination of definite logic programs with respect to call patterns. In *Proceedings of the International Conference on Fifth Generation Computer Systems*. IOS Press, Tokyo, Japan, 481–488.

Shen, Y. D., Yuan, L. Y., and You, J. H. 2001. Loop checks for logic programs with functions. *Theoretical Computer Science 266*, 1/2, 441–461.

Shen, Y. D., Yuan, L. Y., and You, J. H. 2002. Slt-resolution for the well-founded semantics. *J. Autom. Reason. 28*, 1, 53–97.

Ullman, J. D. and Gelder, A. V. 1988. Efficient tests for top-down termination of logical rules. *J. ACM 35*, 2, 345–373.

Verbaeten, S., Schreye, D. D., and Sagonas, K. 2001. Termination proofs for logic programs with tabling. *ACM Trans. Computational Logic 2*, 1, 57–92.

Verschaetse, K. 1992. Static termination analysis for definite horn clause programs. Ph.D. thesis, Department of Computer Science, K. U. Leuven, Available at http://www.cs.kuleuven.ac.be/lpai.

Received June 2000; revised August 2001 and December 2001; accepted April 2002