TLP **9** (6): 751–780, 2009. © Cambridge University Press 2009
doi:10.1017/S1471068409990068 First published online 16 June 2009

Termination prediction for general logic programs

YI-DONG SHEN
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
(*e-mail:* ydshen@ios.ac.cn)

DANNY DE SCHREYE and DEAN VOETS
Department of Computer Science, Celestijnenlaan 200 A, B-3001 Heverlee, Belgium
(*e-mail:* {danny.deschreye, dean.voets}@cs.kuleuven.ac.be)

submitted 10 October 2007; revised 11 September 2008, 3 April 2009; accepted 12 May 2009

Abstract

We present a heuristic framework for attacking the undecidable termination problem of logic programs, as an alternative to current termination/nontermination proof approaches. We introduce an idea of termination prediction, which predicts termination of a logic program in case that neither a termination nor a non-termination proof is applicable. We establish a necessary and sufficient characterization of infinite (generalized) SLDNF-derivations with arbitrary (concrete or moded) queries, and develop an algorithm that predicts termination of general logic programs with arbitrary nonfloundering queries. We have implemented a termination prediction tool and obtained quite satisfactory experimental results. Except for five programs which break the experiment time limit, our prediction is 100% correct for all 296 benchmark programs of the Termination Competition 2007, of which 18 programs cannot be proved by any of the existing state-of-the-art analyzers like AProVE07, NTI, Polytool, and TALP.

KEYWORDS: Logic programming, termination analysis, loop checking, moded queries, termination prediction

1 Introduction

Termination is a fundamental problem in logic programming with SLDNF-resolution as the query evaluation mechanism (Clark 1978; Lloyd 1987), which has been extensively studied in the literature (see, e.g., Schreye and Decorte 1993 for a survey and some recent papers Apt and Pedreschi 1993; Lindenstrauss and Sagiv 1997; Decorte *et al.* 1999; Mesnard and Neumerkel 2001; Genaim and Codish 2005; Payet and Mesnard 2006; Schneider-Kamp *et al.* 2006; Bruynooghe *et al.* 2007). Since the termination problem is undecidable, existing algorithms/tools either focus on computing sufficient termination conditions which once satisfied, lead to a positive conclusion *terminating* (Arts and Zantema 1995; Marchiori 1996a; Lindenstrauss *et al.* 1997; Ohlebusch *et al.* 2000; Dershowttz *et al.* 2001; Mesnard and Neumerkel

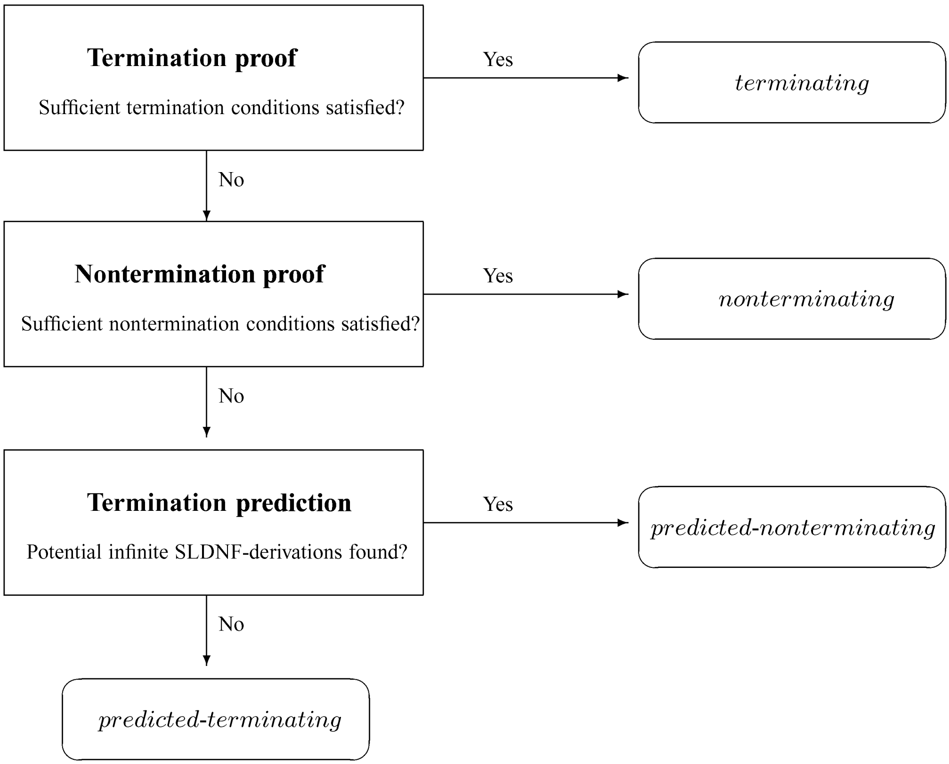

Fig. 1. A framework for handling the termination problem.

2001; Bossi *et al.* 2002; Genaim and Codish 2005; Mesnard and Bagnara 2005; Giesl *et al.* 2006; Schneider-Kamp *et al.* 2006), or on computing sufficient non-termination conditions which lead to a negative conclusion *nonterminating* (Payet 2006; Payet and Mesnard 2006). For convenience, we call the former computation a *termination proof*, and the latter a *non-termination proof*. Due to the nature of undecidability, there must be situations in which neither a termination proof nor a non-termination proof can apply; i.e., no sufficient termination/non-termination conditions are satisfied so that the user would get no conclusion (see the results of the Termination Competition 2007 which are available at http://www.lri.fr/~marche/termination-competition/2007). We observe that in such a situation, it is particularly useful to compute a heuristic conclusion indicating likely termination or likely non-termination, which guides the user to continue to improve his program towards termination. To the best of our knowledge, however, there is no existing heuristic approach available. The goal of the current paper is then to develop such a heuristic framework.

We propose an idea of *termination prediction*, as depicted in Figure 1. In the case that neither a termination nor a non-termination proof is applicable, we appeal to a heuristic algorithm to predict possible termination or non-termination. The prediction applies to general logic programs with concrete or moded queries.

We develop a framework for predicting termination of general logic programs with arbitrary (i.e., concrete or moded) queries. The basic idea is that we establish a characterization of infinite (generalized) SLDNF-derivations with arbitrary queries. Then based on the characterization, we design a complete loop-checking mechanism, which cuts all infinite SLDNF-derivations. Given a logic program and a query, we evaluate the query by applying SLDNF-resolution while performing loop checking. If the query evaluation proceeds without encountering potential infinite derivations, we predict *terminating* for this query; otherwise we predict *nonterminating*.

The core of our termination prediction is a characterization of infinite SLDNF-derivations with arbitrary queries. In Shen *et al.* (2003), a characterization is established for general logic programs with concrete queries. This is far from enough for termination prediction; a characterization of infinite SLDNF-derivations for moded queries is required. Moded queries are the most commonly used query form in static termination analysis. A moded query contains (abstract) atoms like $p(\mathcal{I}, T)$ where T is a term (i.e., a constant, variable, or function) and $\mathcal{I}$ is an input mode. An *input mode* stands for an arbitrary ground (i.e., variable-free) term, so that to prove that a logic program terminates for a moded query $p(\mathcal{I}, T)$ is to prove that the program terminates for any (concrete) query $p(t, T)$ where t is a ground term.

It is nontrivial to characterize infinite SLDNF-derivations with moded queries. The first challenge we must address is how to formulate an SLDNF-derivation for a moded query Q_0, as the standard SLDNF-resolution is only for concrete queries (Clark 1978; Lloyd 1987). We will introduce a framework called a *moded-query forest*, which consists of all (generalized) SLDNF-trees rooted at an instance of Q_0 (the instance is Q_0 with each input mode replaced by a ground term). An SLDNF-derivation for Q_0 is then defined over the moded-query forest such that a logic program P terminates for Q_0 if and only if the moded-query forest contains no infinite SLDNF-derivations.

A moded-query forest may have an infinite number of SLDNF-trees, so it is infeasible for us to predict termination of a logic program by traversing the moded-query forest. To handle this challenge, we will introduce a novel compact approximation for a moded-query forest, called a *moded generalized SLDNF-tree*. The key idea is to treat an input mode as a special meta-variable in the way that during query evaluation, it can be substituted by a constant or function, but cannot be substituted by an ordinary variable. As a result, SLDNF-derivations for a moded query can be constructed in the same way as the ones for a concrete query. A characterization of infinite SLDNF-derivations for moded queries is then established in terms of some key properties of a moded generalized SLDNF-tree.

We have implemented a termination prediction tool and obtained quite satisfactory experimental results. Except for five programs which break the experiment time limit, our prediction is 100% correct for all 296 benchmark programs of the Termination Competition 2007, of which 18 programs cannot be proved by any of the existing state-of-the-art analyzers like AProVE07, NTI, Polytool, and TALP.

The paper is organized as follows. Section 2 reviews some basic concepts including generalized SLDNF-trees. Sections 3 and 4 present a characterization of infinite SLDNF-derivations for concrete and moded queries, respectively. Section 5 introduces a new loop-checking mechanism, and based on it develops an algorithm that predicts termination of general logic programs with arbitrary queries. The termination prediction method is illustrated with representative examples including ones borrowed from the Termination Competition 2007. Section 6 describes the implementation of our termination prediction algorithm and presents experimental results over the programs of the Termination Competition 2007. Section 7 mentions related work, and Section 8 concludes.

2 Preliminaries

We assume that the reader is familiar with standard terminology of logic programs, in particular with SLDNF-resolution, as described in Lloyd (1987). Variables begin with a capital letter X, Y, Z, U, V, or I, and predicate, function, and constant symbols with a lower case letter. A term is a constant, a variable, or a function of the form $f(T_1, \ldots, T_m)$ where f is a function symbol and each T_i is a term. For simplicity, we use $\overline{T}$ to denote a tuple of terms $T_1, \ldots, T_m$. An atom is of the form $p(\overline{T})$ where p is a predicate symbol. Let A be an atom/term. The size of A, denoted by $|A|$, is the number of occurrences of function symbols, variables, and constants in A. Two atoms are called *variants* if they are the same up to variable renaming. A literal is an atom A or the negation $\neg A$ of A.

A (general) logic program P is a finite set of clauses of the form $A \leftarrow L_1, \ldots, L_n$, where A is an atom and each L_i is a literal. Throughout the paper, we consider only Herbrand models. The Herbrand universe and Herbrand base of P are denoted by $HU(P)$ and $HB(P)$, respectively.

A goal G_i is a headless clause $\leftarrow L_1, \ldots, L_n$ where each literal L_j is called a subgoal. The goal $G_0 = \leftarrow Q_0$ for a query Q_0 is called a top goal. Without loss of generality, we assume that Q_0 consists only of one atom. Q_0 is a *moded query* if some arguments of Q_0 are input modes (in this case, Q_0 is called an *abstract* atom); otherwise, it is a *concrete query*. An input mode always begins with a letter $\mathscr{I}$.

Let P be a logic program and G_0 a top goal. G_0 is evaluated by building a *generalized SLDNF-tree* GT_{G_0} as defined in Shen *et al.* (2003), in which each node is represented by $N_i : G_i$ where N_i is the name of the node and G_i is a goal attached to the node. We do not reproduce the definition of a generalized SLDNF-tree. Roughly speaking, GT_{G_0} is the set of standard SLDNF-trees for $P \cup \{G_0\}$ augmented with an ancestor–descendant relation on their subgoals. Let L_i and L_j be the selected subgoals at two nodes N_i and N_j, respectively. L_i is an *ancestor* of L_j, denoted by $L_i \prec_{anc} L_j$, if the proof of L_i goes via the proof of L_j. Throughout the paper, we choose to use the best-known *depth-first, left-most* control strategy, as is used in Prolog, to select nodes/goals and subgoals (it can be adapted to any other fixed control strategies). So by the *selected subgoal* in each node $N_i : \leftarrow L_1, \ldots, L_n$, we refer to the left-most subgoal L_1.

Recall that in SLDNF-resolution, let $L_i = \neg A$ be a ground negative subgoal selected at N_i, then (by the negation-as-failure rule Clark 1978) a subsidiary child SLDNF-tree $T_{N_{i+1}:\leftarrow A}$ rooted at $N_{i+1} : \leftarrow A$ will be built to solve A. In a generalized SLDNF-tree GT_{G_0}, such parent and child SLDNF-trees are connected from N_i to

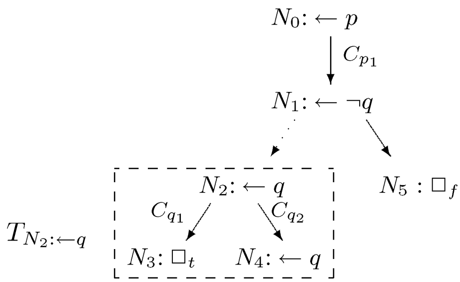

Fig. 2. The generalized SLDNF-tree GT_{G_0} of P_0.

N_{i+1} via a dotted edge "$\cdots\triangleright$" (called a *negation arc*), and A at N_{i+1} inherits all ancestors of L_i at N_i. Therefore, a path of a generalized SLDNF-tree may come across several SLDNF-trees through dotted edges. Any such path starting at the root node $N_0 : G_0$ of GT_{G_0} is called a *generalized SLDNF-derivation.*

We do not consider *floundering* queries; i.e., we assume that no nonground negative subgoals are selected at any node of a generalized SLDNF-tree (see Shen *et al.* 2003).

Another feature of a generalized SLDNF-tree GT_{G_0} is that each subsidiary child SLDNF-tree $T_{N_{i+1}:\leftarrow A}$ in GT_{G_0} terminates (i.e., stops expanding its nodes) at the first success leaf. The intuition behind this is that it is absolutely unnecessary to exhaust the remaining branches because they would never generate any new answers for A (since A is ground). In fact, Prolog executes the same pruning by using a *cut* operator to skip the remaining branches once the first success leaf is generated (e.g., see SICStus Prolog at http://www.sics.se /sicstus/docs/latest4/pdf/sicstus.pdf). To illustrate, consider the following logic program and top goal:

$$
\begin{array}{lll}
P_0 : & p \leftarrow \neg q. & C_{p_1} \\
 & q. & C_{q_1} \\
 & q \leftarrow q. & C_{q_2} \\
G_0 : & \leftarrow p. &
\end{array}
$$

The generalized SLDNF-tree GT_{G_0} for $P_0 \cup \{G_0\}$ is depicted in Figure 2. Note that the subsidiary child SLDNF-tree $T_{N_2:\leftarrow q}$ terminates at the first success leaf N_3, leaving N_4 not further expanded. As a result, all generalized SLDNF-derivations in GT_{G_0} are finite.

For simplicity, in the following sections by a derivation or SLDNF-derivation we refer to a generalized SLDNF-derivation. Moreover, for any node $N_i : G_i$ we use L_i^1 to refer to the selected subgoal in G_i.

A derivation step is denoted by $N_i : G_i \Rightarrow_{C,\theta_i} N_{i+1} : G_{i+1}$ meaning that applying a clause C to G_i produces $N_{i+1} : G_{i+1}$, where G_{i+1} is the resolvent of C and G_i on L_i^1 with the mgu (most general unifier) θ_i. Here, for a substitution of two variables, X in L_i^1 and Y in (the head of) C, we always use X to substitute for Y. When no confusion would occur, we may omit the mgu θ_i when writing a derivation step.

3 A characterization of infinite SLDNF-derivations for concrete queries

In this section, we review the characterization of infinite derivations with concrete queries presented in Shen *et al.* (2003).

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 by S_T, is the string S with every variable replaced by $\mathscr{X}$.

For instance, let $T_1 = a$ and $T_2 = f(X, g(X, f(a, Y)))$. Then $S_{T_1} = a$ and $S_{T_2} = f\mathscr{X}g\mathscr{X}fa\mathscr{X}$.

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 by $S_{T_1} \subseteq_{proj} S_{T_2}$, if S_{T_1} is obtained from S_{T_2} by removing zero or more elements.

Definition 3.3
Let A_1 and A_2 be two atoms (positive subgoals) with the same predicate symbol. A_1 is said to *loop into* A_2, denoted by $A_1 \rightsquigarrow_{loop} A_2$, if $S_{A_1} \subseteq_{proj} S_{A_2}$. Let $N_i : G_i$ and $N_j : G_j$ be two nodes in a 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.

Observe that if $A_1 \rightsquigarrow_{loop} A_2$ then $|A_1| \leqslant |A_2|$, and that 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. Since a logic program has only a finite number of clauses, an infinite derivation results from repeatedly applying the same set of clauses, which leads to either 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 derivation are captured by loop goals. The following result is proved in Shen *et al.* (2003).

Theorem 3.1
Let $G_0 = \leftarrow Q_0$ be a top goal with Q_0 a concrete query. Any infinite derivation D in GT_{G_0} contains an infinite sequence of goals $G_0, \ldots, G_{g_1}, \ldots, G_{g_2}, \ldots$ such that for any $j \geqslant 1$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}.

Put another way, Theorem 3.1 states that any infinite derivation D in GT_{G_0} is of the form

$$N_0 : G_0 \Rightarrow_{C_0} \cdots \ N_{g_1} : G_{g_1} \Rightarrow_{C_1} \cdots \ N_{g_2} : G_{g_2} \Rightarrow_{C_2} \cdots \ N_{g_3} : G_{g_3} \Rightarrow_{C_3} \cdots,$$

where for any $j \geqslant 1$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}. This provides a necessary and sufficient characterization of an infinite generalized SLDNF-derivation with a concrete query.

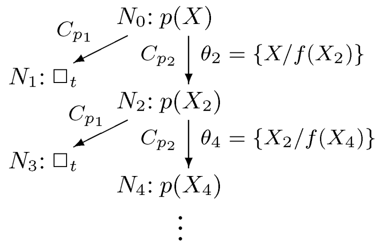

Fig. 3. The generalized SLDNF-tree $GT_{\leftarrow p(X)}$ of P_1 for a concrete query $p(X)$.

Example 3.1
Consider the following logic program:

$$\begin{array}{lll} P_1: & p(a). & C_{p_1} \\ & p(f(X)) \leftarrow p(X). & C_{p_2} \end{array}$$

The generalized SLDNF-tree $GT_{\leftarrow p(X)}$ for a concrete query $p(X)$ is shown in Figure 3, where for simplicity the symbol $\leftarrow$ in each goal is omitted. Note that $GT_{\leftarrow p(X)}$ has an infinite derivation

$$N_0 : p(X) \Rightarrow_{C_{p_2}} N_2 : p(X_2) \Rightarrow_{C_{p_2}} N_4 : p(X_4) \Rightarrow_{C_{p_2}} \cdots,$$

where for any $j \geqslant 0$, $G_{2(j+1)}$ is a loop goal of G_{2j}.

4 A characterization of infinite SLDNF-derivations for moded queries

We first define generalized SLDNF-derivations for moded queries by introducing a framework called moded-query forests.

Definition 4.1
Let P be a logic program and $Q_0 = p(\mathscr{I}_1, \ldots, \mathscr{I}_m, T_1, \ldots, T_n)$ a moded query. The *moded-query forest* of P for Q_0, denoted by MF_{Q_0}, consists of all generalized SLDNF-trees for $P \cup \{G_0\}$, where $G_0 =\leftarrow p(t_1, \ldots, t_m, T_1, \ldots, T_n)$ with each t_i being a ground term from $HU(P)$. A (*generalized SLDNF-*) *derivation for the moded query* Q_0 is a derivation in any generalized SLDNF-tree of MF_{Q_0}.

Therefore, a logic program P terminates for a moded query Q_0 if and only if there is no infinite derivation for Q_0 if and only if MF_{Q_0} has no infinite derivation.

Example 4.1
Consider the logic program P_1 again. We have $HU(P_1) = \{a, f(a), f(f(a)), \ldots\}$. Let $p(\mathscr{I})$ be a moded query. The moded-query forest $MF_{p(\mathscr{I})}$ consists of generalized SLDNF-trees $GT_{\leftarrow p(a)}$, $GT_{\leftarrow p(f(a))}$, etc., as shown in Figure 4. Note that $MF_{p(\mathscr{I})}$ has an infinite number of generalized SLDNF-trees. However, any individual tree, GT_{G_0} with $G_0 =\leftarrow p(\underbrace{f(f(\ldots f}_{n\ items}(a)\ldots)))$ $(n \geqslant 0)$, is finite. $MF_{p(\mathscr{I})}$ contains no infinite derivation, thus P_1 terminates for $p(\mathscr{I})$.

$$GT_{\leftarrow p(a)}: \begin{array}{c} N_0\colon p(a) \\ C_{p_1}\big\downarrow \\ N_1\colon \Box_t \end{array} \qquad GT_{\leftarrow p(f(a))}: \begin{array}{c} N_0\colon p(f(a)) \\ C_{p_2}\big\downarrow \\ N_2\colon p(a) \\ C_{p_1}\big\downarrow \\ N_3\colon \Box_t \end{array} \qquad \cdots$$

Fig. 4. The moded-query forest $MF_{p(\mathscr{I})}$ of P_1 for a moded query $p(\mathscr{I})$.

In a moded-query forest, all input modes are instantiated into ground terms in $HU(P)$. When $HU(P)$ is infinite, the moded-query forest would contain infinitely many generalized SLDNF-trees. This means that it is infeasible to build a moded-query forest to represent the derivations for a moded query. An alternative yet ideal way is to directly apply SLDNF-resolution to evaluate input modes and build a compact generalized SLDNF-tree for a moded query. Unfortunately, SLDNF-resolution accepts only terms as arguments of a top goal; an input mode $\mathscr{I}$ is not directly evaluable.

Since an input mode stands for an arbitrary ground term, i.e., it can be any term from $HU(P)$, during query evaluation it can be instantiated to any term except variable (note that a ground term cannot be substituted by a variable). This suggests that we may approximate the effect of an input mode $\mathscr{I}$ by treating it as a special (meta-) variable I in the way that in SLDNF-derivations, I can be substituted by a constant or function, but cannot be substituted by an ordinary variable. Therefore, when doing unification of a special variable I and a variable X, we always substitute I for X.

Definition 4.2
Let P be a logic program and $Q_0 = p(\mathscr{I}_1, \ldots, \mathscr{I}_m, T_1, \ldots, T_n)$ a moded query. The *moded generalized SLDNF-tree* of P for Q_0, denoted by MT_{Q_0}, is defined to be the generalized SLDNF-tree GT_{G_0} for $P \cup \{G_0\}$, where $G_0 = \leftarrow p(I_1, \ldots, I_m, T_1, \ldots, T_n)$ with each I_i being a distinct special variable not occurring in any T_j. The special variables $I_1, \ldots, I_m$ for the input modes $\mathscr{I}_1, \ldots, \mathscr{I}_m$ are called *input mode variables* (or *input variables*).

In a moded generalized SLDNF-tree, an input variable I may be substituted by either a constant t or a function $f(\overline{T})$. It will not be substituted by any noninput variable. If I is substituted by $f(\overline{T})$, all variables in $\overline{T}$ are also called input variables (thus are treated as special variables).

In this paper, we do not consider *floundering moded* queries; i.e., we assume that no negative subgoals containing either ordinary or input variables are selected at any node of a moded generalized SLDNF-tree.

Definition 4.3
Let P be a logic program, $Q_0 = p(\mathscr{I}_1, \ldots, \mathscr{I}_m, T_1, \ldots, T_n)$ a moded query, and $G_0 = \leftarrow p(I_1, \ldots, I_m, T_1, \ldots, T_n)$. Let D be a derivation in the moded generalized SLDNF-tree MT_{Q_0}. A *moded instance* of D is a derivation obtained from D by first instantiating all input variables at the root node $N_0 : G_0$ with an mgu $\theta = \{I_1/t_1, \ldots, I_m/t_m\}$,

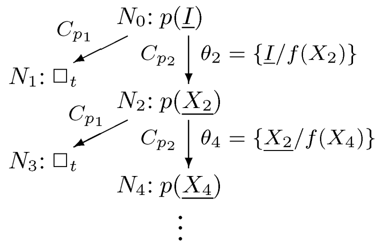

Fig. 5. The moded generalized SLDNF-tree $MT_{p(\mathscr{I})}$ of P_1 for a moded query $p(\mathscr{I})$.

where each $t_i \in HU(P)$, then passing the instantiation θ down to the other nodes of D.

Example 4.2

Consider the logic program P_1 again. Let $Q_0 = p(\mathscr{I})$ be a moded query and $G_0 = \leftarrow p(I)$. The moded generalized SLDNF-tree MT_{Q_0} is GT_{G_0} as depicted in Figure 5, where all input variables are underlined. Since I is an input variable, X_2 is an input variable, too (due to the mgu θ_2). For the same reason, all X_{2i} are input variables ($i > 0$).

Consider the following infinite derivation D in MT_{Q_0}:

$$N_0 : p(I) \Rightarrow_{C_{p_2}} N_2 : p(X_2) \Rightarrow_{C_{p_2}} N_4 : p(X_4) \Rightarrow_{C_{p_2}} \cdots .$$

By instantiating the input variable I at N_0 with different ground terms from $HU(P_1)$ and passing the instantiation θ down to the other nodes of D, we can obtain different moded instances from D. For example, instantiating I to a (i.e., $\theta = \{I/a\}$) yields the moded instance

$$N_0 : p(a).$$

Instantiating I to $f(a)$ (i.e., $\theta = \{I/f(a)\}$) yields the moded instance

$$N_0 : p(f(a)) \Rightarrow_{C_{p_2}} N_2 : p(a).$$

And, instantiating I to $f(f(a))$ (i.e., $\theta = \{I/f(f(a))\}$) yields the moded instance

$$N_0 : p(f(f(a))) \Rightarrow_{C_{p_2}} N_2 : p(f(a)) \Rightarrow_{C_{p_2}} N_4 : p(a.)$$

Observe that a moded instance of a derivation D in MT_{Q_0} is a derivation in $GT_{G_0\theta}$, where $G_0\theta = \leftarrow p(t_1, \ldots, t_m, T_1, \ldots, T_n)$ with each t_i being a ground term from $HU(P)$. By Definition 4.1, $GT_{G_0\theta}$ is in the moded-query forest MF_{Q_0}. This means that any moded instance of a derivation in MT_{Q_0} is a derivation for Q_0 in MF_{Q_0}. For instance, all moded instances illustrated in Example 4.2 are derivations in the moded-query forest MF_{Q_0} of Figure 4.

Theorem 4.1

Let MF_{Q_0} and MT_{Q_0} be the moded-query forest and the moded generalized SLDNF-tree of P for Q_0, respectively. If MF_{Q_0} has an infinite derivation D', MT_{Q_0} has an infinite derivation D with D' as a moded instance.

Proof

Let $Q_0 = p(\mathscr{I}_1,\ldots,\mathscr{I}_m,T_1,\ldots,T_n)$. Then, the root node of D' is $N_0 :\leftarrow p(t_1,\ldots,t_m, T_1,\ldots,T_n)$ with each $t_i \in HU(P)$, and the root node of MT_{Q_0} is $N_0 :\leftarrow p(I_1,\ldots,I_m, T_1,\ldots,T_n)$ with each I_i being an input variable not occurring in any T_j. Note that the former is an instance of the latter with the mgu $\theta = \{I_1/t_1,\ldots,I_m/t_m\}$. Let D' be of the form

$$N_0 :\leftarrow p(t_1,\ldots,t_m,T_1,\ldots,T_n) \Rightarrow_{C_0} N_1 : G'_1 \cdots \Rightarrow_{C_i} N_{i+1} : G'_{i+1} \cdots .$$

MT_{Q_0} must have a derivation D of the form

$$N_0 :\leftarrow p(I_1,\ldots,I_m,T_1,\ldots,T_n) \Rightarrow_{C_0} N_1 : G_1 \cdots \Rightarrow_{C_i} N_{i+1} : G_{i+1} \cdots ,$$

such that each $G'_i = G_i\theta$, since for any $i \geqslant 0$ and any clause C_i in P, if G'_i can unify with C_i, so can G_i with C_i. Note that when the selected subgoal at some G'_i is a negative ground literal, by the assumption that Q_0 is nonfloundering, we have the same selected literal at G_i. We then have the proof. $\square$

Our goal is to establish a characterization of infinite derivations for a moded query such that the converse of Theorem 4.1 is true under some conditions.

Consider the infinite derivation in Figure 5 again. The input variable I is substituted by $f(X_2)$; X_2 is then substituted by $f(X_4), \ldots$. This produces an infinite chain of substitutions for I of the form $I/f(X_2), X_2/f(X_4), \ldots$. The following lemma shows that infinite derivations containing such an infinite chain of substitutions have no infinite moded instances.

Lemma 4.2

If a derivation D in a moded generalized SLDNF-tree MT_{Q_0} is infinite but none of its moded instances is infinite, then there is an input variable I such that D contains an infinite chain of substitutions for I of the form

$$I/f_1(\ldots,Y_1,\ldots),\ldots,Y_1/f_2(\ldots,Y_2,\ldots),\ldots,Y_{i-1}/f_i(\ldots,Y_i,\ldots),\ldots \quad (1)$$

(some f_is would be the same).

Proof

We distinguish the following four types of substitution chains for an input variable I in D:

(1) $X_1/I,\ldots,X_m/I$ or $X_1/I,\ldots,X_i/I, \ldots$ That is, I is never substituted by any terms.
(2) $X_1/I,\ldots,X_m/I, I/t$ where t is a ground term. That is, I is substituted by a ground term.
(3) $X_1/I,\ldots,X_m/I, I/f_1(\ldots,Y_1,\ldots),\ldots,Y_1/f_2(\ldots,Y_2,\ldots),\ldots,Y_{n-1}/f_n(\ldots,Y_n,\ldots),\ldots$ where $f_n(\ldots,Y_n,\ldots)$ is the last nonground function in the substitution chain for I in D. In this case, I is recursively substituted by a finite number of functions.
(4) $X_1/I,\ldots,X_m/I, I/f_1(\ldots,Y_1,\ldots),\ldots,Y_1/f_2(\ldots,Y_2,\ldots),\ldots,Y_{i-1}/f_i(\ldots,Y_i,\ldots),\ldots$ In this case, I is recursively substituted by an infinite number of functions.

For type 4, D retains its infinite extension for whatever ground term we replace I with. For type 4, D retains its infinite extension when we use t to replace I. To sum up, for any input variable I whose substitution chain is of type 4 or of type 4, there is a ground term t such that replacing I with t does not affect the infinite extension of D. In this case, replacing I in D with t leads to an infinite derivation less general than D.

For type 4, note that all variables appearing in the $f_i(.)$s are input variables. Since $f_n(\ldots, Y_n, \ldots)$ is the last nonground function in the substitution chain for I in D, the substitution chain for every variable Y_n in $f_n(\ldots, Y_n, \ldots)$ is either of type 4 or of type 4. Therefore, we can replace each Y_n with an appropriate ground term t_n without affecting the infinite extension of D. After this replacement, D becomes D_n and $f_n(\ldots, Y_n, \ldots)$ becomes a ground term $f_n(\ldots, t_n, \ldots)$. Now $f_{n-1}(\ldots, Y_{n-1}, \ldots)$ is the last nonground function in the substitution chain for I in D_n. Repeating the above replacement recursively, we will obtain an infinite derivation D_1, which is D with all variables in the $f_i(.)$s replaced with a ground term. Assume $f_1(\ldots, Y_1, \ldots)$ becomes a ground term t in D_1. Then the substitution chain for I in D_1 is of type 4. So replacing I with t in D_1 leads to an infinite derivation D_0.

The above constructive proof shows that if the substitution chains for all input variables in D are of type 4, 4, or 4, then D must have an infinite moded instance. Since D has no infinite moded instance, there must exist an input variable I whose substitution chain in D is of type 4. That is, I is recursively substituted by an infinite number of functions. Note that some f_is would be the same because a logic program has only a finite number of function symbols. This concludes the proof. □

We are ready to introduce the following principal result:

Theorem 4.3

Let MF_{Q_0} and MT_{Q_0} be the moded-query forest and the moded generalized SLDNF-tree of P for Q_0, respectively. MF_{Q_0} has an infinite derivation if and only if MT_{Q_0} has an infinite derivation D of the form

$$N_0 : G_0 \Rightarrow_{C_0} \cdots \; N_{g_1} : G_{g_1} \Rightarrow_{C_1} \cdots \; N_{g_2} : G_{g_2} \Rightarrow_{C_2} \cdots \; N_{g_3} : G_{g_3} \Rightarrow_{C_3} \cdots, \qquad (2)$$

where (i) for any $j \geqslant 1$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}, and (ii) for no input variable I, D contains an infinite chain of substitutions for I of form (1).

Proof

($\Longrightarrow$) Assume MF_{Q_0} has an infinite derivation D'. By Theorem 4.1, GT_{G_0} has an infinite derivation D with D' as a moded instance. By Theorem 3.1, D is of form (2) and satisfies condition (i).

Assume, on the contrary, that D does not satisfy condition (ii). That is, for some input variable I, D contains an infinite chain of substitutions for I of the form

$$I/f_1(\ldots, Y_1, \ldots), \ldots, Y_1/f_2(\ldots, Y_2, \ldots), \ldots, Y_{i-1}/f_i(\ldots, Y_i, \ldots), \ldots.$$

Note that for whatever ground term t we assign to I, this chain can be instantiated at most as long in length as the following one:

$$t/f_1(\ldots, t_1, \ldots), \ldots, t_1/f_2(\ldots, t_2, \ldots), \ldots, t_k/f_{k+1}(\ldots, Y_{k+1}, \ldots),$$

where $k = |t|$, t_is are ground terms and $|t_k| = 1$. This means that replacing I with any ground term t leads to a finite moded instance of D. Therefore, D has no infinite moded instance in MF_{Q_0}, a contradiction.

($\Longleftarrow$) Assume, on the contrary, that MF_{Q_0} has no infinite derivation. By Lemma 4.2, we reach a contradiction to condition (ii). $\square$

Theorem 4.3 provides a necessary and sufficient characterization of an infinite generalized SLDNF-derivation for a moded query. Note that it coincides with Theorem 3.1 when Q_0 is a concrete query, where $MF_{Q_0} = MT_{Q_0}$ and condition (ii) is always true.

The following corollary is immediate to this theorem.

Corollary 4.4
A logic program P terminates for a moded query Q_0 if and only if the moded generalized SLDNF-tree MT_{Q_0} has no infinite derivation of form (2) satisfying conditions (i) and (ii) of Theorem 4.3.

We use simple yet typical examples to illustrate the proposed characterization of infinite SLDNF-derivations with moded queries.

Example 4.3
Consider the moded generalized SLDNF-tree MT_{Q_0} in Figure 5. It has only one infinite derivation, which satisfies condition (i) of Theorem 4.3 where for each $j \geqslant 0$, N_{g_j} in Theorem 4.3 corresponds to N_{2j} in Figure 5. However, the chain of substitutions for I in this derivation violates condition (ii). This means that MF_{Q_0} contains no infinite derivations; therefore, there is no infinite derivation for the moded query $p(\mathcal{I})$. As a result, P_1 terminates for $p(\mathcal{I})$.

Example 4.4
Consider the *append* program:

$$
\begin{array}{lll}
P_2: & append([], X, X). & C_{a_1} \\
 & append([X|Y], U, [X|Z]) \leftarrow append(Y, U, Z). & C_{a_2}
\end{array}
$$

Let us choose the following three simplest moded queries:

$$
\begin{aligned}
Q_0^1 &= append(\mathcal{I}, V_2, V_3), \\
Q_0^2 &= append(V_1, \mathcal{I}, V_3), \\
Q_0^3 &= append(V_1, V_2, \mathcal{I}).
\end{aligned}
$$

Since applying clause C_{a_1} produces only leaf nodes, for simplicity we ignore it when depicting moded generalized SLDNF-trees. The three moded generalized SLDNF-trees $MT_{Q_0^1}$, $MT_{Q_0^2}$, and $MT_{Q_0^3}$ are shown in Figures 6(a), 6(b), and 6(c), respectively. Note that all the derivations are infinite and satisfy condition (i) of Theorem 4.3, where for each $j \geqslant 0$, N_{g_j} in Theorem 4.3 corresponds to N_j in Figure 6. Apparently, the chains of substitutions for I in the derivations of $MT_{Q_0^1}$ and $MT_{Q_0^3}$ violate condition (ii) of Theorem 4.3. $MF_{Q_0^1}$ and $MF_{Q_0^3}$ contain no infinite derivation and thus there exists no infinite derivation for the moded queries Q_0^1 and Q_0^3. Therefore, P_2 terminates for Q_0^1 and Q_0^3. However, the derivation in $MT_{Q_0^2}$ satisfies condition (ii), thus there exist infinite derivations for the moded query Q_0^2. P_2 does not terminate for Q_0^2.

$$\begin{array}{l}
N_0\colon append(\underline{I}, V_2, V_3) \\
\quad C_{a_2} \Big\downarrow \begin{array}{l} \theta_0 = \{\underline{I}/[X|Y], \\ U/V_2, V_3/[X|Z]\} \end{array} \\
N_1\colon append(\underline{Y}, V_2, Z) \\
\quad C_{a_2} \Big\downarrow \begin{array}{l} \theta_1 = \{\underline{Y}/[X_1|Y_1], \\ U_1/V_2, Z/[X_1|Z_1]\} \end{array} \\
N_2\colon append(\underline{Y_1}, V_2, Z_1)
\end{array}$$

(a)

$$\begin{array}{l}
N_0\colon append(V_1, \underline{I}, V_3) \\
\quad C_{a_2} \Big\downarrow \begin{array}{l} \theta_0 = \{V_1/[X|Y], \\ U/\underline{I}, V_3/[X|Z]\} \end{array} \\
N_1\colon append(Y, \underline{I}, Z) \\
\quad C_{a_2} \Big\downarrow \begin{array}{l} \theta_1 = \{Y/[X_1|Y_1], \\ U_1/\underline{I}, Z/[X_1|Z_1]\} \end{array} \\
N_2\colon append(Y_1, \underline{I}, Z_1)
\end{array}$$

(b)

$$\begin{array}{l}
N_0\colon append(V_1, V_2, \underline{I}) \\
\quad C_{a_2} \Big\downarrow \begin{array}{l} \theta_0 = \{V_1/[X|Y], \\ U/V_2, \underline{I}/[X|Z]\} \end{array} \\
N_1\colon append(Y, V_2, \underline{Z}) \\
\quad C_{a_2} \Big\downarrow \begin{array}{l} \theta_1 = \{Y/[X_1|Y_1], \\ U_1/V_2, \underline{Z}/[X_1|Z_1]\} \end{array} \\
N_2\colon append(Y_1, V_2, \underline{Z_1})
\end{array}$$

(c)

Fig. 6. Three moded generalized SLDNF-trees: (a) $MT_{Q_0^1}$, (b) $MT_{Q_0^2}$, and (c) $MT_{Q_0^3}$.

Let $pred(P)$ be the set of predicate symbols in P. Define

$$\begin{aligned} MQ(P) = \{p(\overline{T}) \mid\ & p \text{ is an } n\text{-ary predicate symbol in } pred(P), \\ & \text{and } \overline{T} \text{ consists of } m > 0 \text{ input modes and } n - m \text{ variables}\}. \end{aligned}$$

Note that $MQ(P)$ contains all most general moded queries of P in the sense that any moded query of P is an instance of some query in $MQ(P)$. Since $pred(P)$ is finite, $MQ(P)$ is finite. Therefore, it is immediate that P terminates for all moded queries if and only if it terminates for each moded query in $MQ(P)$.

Theorem 4.5
Let $Q_1 = p(\overline{T_1})$ and $Q_2 = p(\overline{T_2})$ be two moded queries in $MQ(P)$, where all input modes of Q_1 occur in Q_2. If there is no infinite derivation for Q_1, there is no infinite derivation for Q_2.

Proof
Note that we consider only nonfloundering queries by assuming that no negative subgoals containing either ordinary or input variables are selected at any node of a moded generalized SLDNF-tree. Then, for any concrete query Q, that there is no infinite derivation for Q implies there is no infinite derivation for any instance of Q.

For ease of presentation, let $Q_1 = p(\mathscr{I}_1, \ldots, \mathscr{I}_l, X_{l+1}, \ldots, X_n)$ and $Q_2 = p(\mathscr{I}_1, \ldots, \mathscr{I}_m, X_{m+1}, \ldots, X_n)$ with $l < m$.

Assume that there is no infinite derivation for Q_1, then, there is no infinite derivation for any query $Q = p(t_1, \ldots, t_l, X_{l+1}, \ldots, X_n)$, where each t_i is a ground term from $HU(P)$. Then, there is no infinite derivation for any query $Q' = p(t_1, \ldots, t_l, s_{l+1}, \ldots, s_m, X_{m+1}, \ldots, X_n)$, where each t_i is a ground term from $HU(P)$ and each s_i an instance of X_i. Since all X_is are variables, there is no infinite derivation for any query $Q'' = p(t_1, \ldots, t_l, t_{l+1}, \ldots, t_m, X_{m+1}, \ldots, X_n)$, where each t_i is a ground term from $HU(P)$. That is, there is no infinite derivation for Q_2. □

Applying this theorem, we can conclude that P_2 in Example 4.4 terminates for all moded queries in $MQ(P_2)$ except Q_0^2.

5 An algorithm for predicting termination of logic programs

We develop an algorithm for predicting termination of logic programs based on the necessary and sufficient characterization of an infinite generalized SLDNF-derivation (Theorem 4.3 and Corollary 4.4). We begin by introducing a loop-checking mechanism.

A loop-checking mechanism, or more formally a *loop check* (Bol *et al.* 1991), defines conditions for us to cut a possibly infinite derivation at some node. By cutting a derivation at a node N we mean removing all descendants of N. Informally, a loop check is said to be *weakly sound* if for any generalized SLDNF-tree GT_{G_0}, GT_{G_0} having a success derivation before cut implies it has a success derivation after cut; it is said to be *complete* if it cuts all infinite derivations in GT_{G_0}. An ideal loop check cuts all infinite derivations while retaining success derivations. Unfortunately, as shown by Bol *et al.* (1991), there exists no loop check that is both weakly sound and complete. In this paper, we focus on complete loop checks, because we want to apply them to predict termination of logic programs.

Definition 5.1
Given a repetition number $r \geqslant 2$, *LP-check* is defined as follows: Any derivation D in a generalized SLDNF-tree is cut at a node N_{g_r} if D has a prefix of the form

$$N_0 : G_0 \Rightarrow_{C_0} \cdots\ N_{g_1} : G_{g_1} \Rightarrow_{C_k} \cdots\ N_{g_2} : G_{g_2} \Rightarrow_{C_k} \cdots\ N_{g_r} : G_{g_r} \Rightarrow_{C_k}, \tag{3}$$

such that (a) for any $j < r$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}, and (b) for all $j \leqslant r$, the clause C_k applied to G_{g_j} is the same. C_k is then called a *looping clause.*

LP-check predicts infinite derivations from prefixes of derivations based on the characterization of Theorem 3.1 (or condition (i) of Theorem 4.3). The repetition number r specifies the minimum number of loop goals appearing in the prefixes. It does not appear appropriate to choose $r < 2$, as that may lead to many finite derivations being wrongly cut. Although there is no mathematical mechanism available for choosing this repetition number (since the termination problem is undecidable), in many situations it suffices to choose $r = 3$ for a correct prediction of infinite derivations. For instance, choosing $r = 3$ we are able to obtain correct predictions for all benchmark programs of the Termination Competition 2007 (see Section 6).

LP-check applies to any generalized SLDNF-trees including moded generalized SLDNF-trees.

Theorem 5.1
LP-check is a complete loop check.

Proof
Let D be an infinite derivation in GT_{G_0}. By Theorem 3.1, D is of the form

$$N_0 : G_0 \Rightarrow_{C_0} \cdots\ N_{f_1} : G_{f_1} \Rightarrow_{C_1} \cdots\ N_{f_2} : G_{f_2} \Rightarrow_{C_2} \cdots,$$

such that for any $i \geqslant 1$, $G_{f_{i+1}}$ is a loop goal of G_{f_i}. Since a logic program has only a finite number of clauses, there must be a (looping) clause C_k being repeatedly applied at infinitely many nodes $N_{g_1} : G_{g_1}, N_{g_2} : G_{g_2}, \ldots$, where for each $j \geqslant 1$, $g_j \in \{f_1, f_2, \ldots\}$. Then for any $r > 0$, D has a partial derivation of form (3). So D will be cut at node $N_{g_r} : G_{g_r}$. This shows that any infinite derivation can be cut by LP-check. That is, LP-check is a complete loop check. $\square$

Example 5.1
Let us choose $r = 3$ and consider the infinite derivation D depicted in Figure 5. $p(X_4)$ at N_4 is a loop goal of $p(X_2)$ at N_2 that is a loop goal of $p(I)$ at N_0. Moreover, the same clause C_{p_2} is applied at the three nodes. D satisfies the conditions of LP-check and is cut at node N_4.

Recall that to prove that a logic program P terminates for a moded query $Q_0 = p(\mathscr{I}_1, \ldots, \mathscr{I}_m, T_1, \ldots, T_n)$ is to prove that P terminates for any query $p(t_1, \ldots, t_m, T_2, \ldots, T_n)$, where each t_i is a ground term. This can be reformulated in terms of a moded-query forest; that is, P terminates for Q_0 if and only if MF_{Q_0} has no infinite derivation. Then, Corollary 4.4 shows that P terminates for Q_0 if and only if the moded generalized SLDNF-tree MT_{Q_0} has no infinite derivation D of form (2) satisfying the two conditions (i) and (ii) of Theorem 4.3. Although this characterization cannot be directly used for automated termination test because it requires generating infinite derivations in MT_{Q_0}, it can be used along with LP-check to predict termination, as LP-check is able to guess if a partial derivation would extend to an infinite one. Before describing our prediction algorithm with this idea, we introduce one more condition following Definition 5.1.

Definition 5.2
Let D be a derivation with a prefix of form (3). The prefix of D is said to have the *term-size decrease* property if for any i with $0 < i < r$, there is a substitution $X/f(\ldots Y \ldots)$ between N_{g_i} and $N_{g_{i+1}}$, where X is an input variable and Y (an ordinary or input variable) appears in the selected subgoal of $G_{g_{i+1}}$.

Theorem 5.2
Let D be a derivation such that for all $r \geqslant 2$, D has a prefix of form (3), which has the term-size decrease property. D contains an infinite chain of substitutions of form (1) for some input variable I at the root node of D.

Proof
Due to the term-size decrease property of the prefix of D which holds for all $r \geqslant 2$, D contains an infinite number of substitutions of the form $X/f(\ldots)$, where X is an input variable. Assume, on the contrary, that D does not contain such an infinite chain of form (1). Let M be the longest length of substitutions of form (1) for each input variable I at the root node of D. Note that each input variable can be substituted only by a constant or function. For each substitution $X/f(\ldots)$ with X an input variable, assume $f(\ldots)$ contains at most N variables (i.e., it introduces at most N new input variables). Then, D contains at most $K \times (N^0 + N^1 + \cdots + N^M)$ substitutions of the form $X/f(\ldots)$, where K is the number of input variables at the root node of D and X is an input variable. This contradicts the condition that D contains an infinite number of such substitutions. $\square$

LP-check and the term-size decrease property approximate conditions (i) and (ii) of Theorem 4.3, respectively. So, we can guess an infinite extension (2) from a prefix (3) by combining the two mechanisms, as described in the following algorithm:

Algorithm 5.1 (Predicting termination of a logic program)
Input: A logic program P, a (concrete or moded) query Q_0, and a repetition number $r \geqslant 2$ ($r = 3$ is recommended).
Output: *terminating*, *predicted-terminating*, or *predicted-nonterminating*.
Method: Apply the following procedure.
procedure TPoLP(P, Q_0, r)
{

(1) Initially, set $L = 0$. Construct the moded generalized SLDNF-tree MT_{Q_0} of P for Q_0 in the way that whenever a prefix D_x of the form

$$N_0 : G_0 \Rightarrow_{C_0} \cdots\ N_{g_1} : G_{g_1} \Rightarrow_{C_k} \cdots\ N_{g_2} : G_{g_2} \Rightarrow_{C_k} \cdots\ N_{g_r} : G_{g_r} \Rightarrow_{C_k}$$

is produced which satisfies conditions (a) and (b) of LP-check, if D_x does not have the term-size decrease property then goto step 5.1; else set $L = 1$ and extend D_x from the node N_{g_r} with the looping clause C_k skipped.

(2) Return *terminating* if $L = 0$; otherwise, return *predicted-terminating*.

(3) Return *predicted-nonterminating*.

}

Starting from the root node $N_0 : G_0$, we generate derivations of a moded generalized SLDNF-tree MT_{Q_0} step by step. If a prefix D_x of form (3) is generated which satisfies conditions (a) and (b) of LP-check, then by Theorem 3.1 D_x is very likely to extend infinitely in MT_{Q_0} (via the looping clause C_k). By Theorem 4.1, however, the extension of D_x may not have infinite moded instances in MF_{Q_0}. So in this case, we further check if D_x has the term-size decrease property. If not, by Theorem 4.3 D_x is very likely to have moded instances that extend infinitely in MF_{Q_0}. Algorithm 5.1 then predicts nonterminating for Q_0 by returning an answer *predicted-nonterminating*. If D_x has the term-size decrease property, however, we continue to extend D_x from N_{g_r} by skipping the clause C_k (i.e., the derivation via C_k is cut at N_{g_r} by LP-check).

When the answer is not *predicted-nonterminating*, we distinguish between two cases: (1) $L = 0$. This shows that no derivation was cut by LP-check during the construction of MT_{Q_0}. Algorithm 5.1 concludes terminating for Q_0 by returning an answer *terminating*. (2) $L = 1$. This means that some derivations were cut by LP-check, all of which have the term-size decrease property. Algorithm 5.1 then predicts terminating for Q_0 by returning an answer *predicted-terminating*.

Note that for a concrete query Q_0, no derivation has the term-size decrease property. Therefore, Algorithm 5.1 returns *predicted-nonterminating* for Q_0 once a prefix of a derivation satisfying the conditions of LP-check is generated.

We prove the termination property of Algorithm 5.1.

Proposition 5.3
For any logic program P, concrete/moded query Q_0, and repetition number r, the procedure TPoLP(P, Q_0, r) terminates.

Proof

The procedure TPoLP (Termination Prediction of Logic Programs) constructs MT_{Q_0} while applying LP-check to cut possible infinite derivations. Since LP-check is a complete loop check, it cuts all infinite derivations at some depth. This means that MT_{Q_0} after cut by LP-check is finite. So, TPoLP(P, Q_0, r) terminates. □

Algorithm 5.1 yields a heuristic answer, *predicted-terminating* or *predicted-nonterminating*, or an exact answer *terminating*, as shown by the following theorem:

Theorem 5.4

A logic program P terminates for a query Q_0 if Algorithm 5.1 returns *terminating*.

Proof

If Algorithm 5.1 returns *terminating*, no derivations were cut by LP-check, so the moded generalized SLDNF-tree MT_{Q_0} for Q_0 is finite. By Corollary 4.4, the logic program P terminates for the query Q_0. □

In the following examples, we choose a repetition number $r = 3$.

Example 5.2

Consider Figure 5. Since the prefix D_x between N_0 and N_4 satisfies the conditions of LP-check, Algorithm 5.1 concludes that the derivation may extend infinitely in MT_{Q_0}. It then checks the term-size decrease property to see if D_x has moded instances that would extend infinitely in MF_{Q_0}. Clearly, D_x has the term-size decrease property. So Algorithm 5.1 skips C_{p_2} at N_4 (the branch is cut by LP-check). Consequently, Algorithm 5.1 predicts terminating for $p(\mathscr{I})$ by returning an answer *predicted-terminating*. This prediction is correct (see Example 4.3).

Example 5.3

Consider Figure 6. All the derivations starting at N_0 and ending at N_2 satisfy the conditions of LP-check, so they are cut at N_2. Since the derivations in $MT_{Q_0^1}$ and $MT_{Q_0^3}$ have the term-size decrease property, Algorithm 5.1 returns *predicted-terminating* for Q_0^1 and Q_0^3. Since the derivation in $MT_{Q_0^2}$ does not have the term-size decrease property, Algorithm 5.1 returns *predicted-nonterminating* for Q_0^2. These predictions are all correct (see Example 4.4).

Example 5.4

Consider the following logic program P_3:

$$mult(s(X), Y, Z) \leftarrow mult(X, Y, U), add(U, Y, Z). \qquad C_{m_1}$$
$$mult(0, Y, 0). \qquad C_{m_2}$$
$$add(s(X), Y, s(Z)) \leftarrow add(X, Y, Z). \qquad C_{a_1}$$
$$add(0, Y, Y). \qquad C_{a_2}$$

$MQ(P_3)$ consists of 14 moded queries, seven for predicate *mult* and 7 for predicate *add*. Applying Algorithm 5.1 yields the following result: (1) P_3 is *predicted-terminating* for all moded queries to *add* except $add(V_1, \mathscr{I}_2, V_3)$ for which P_3 is *predicted-nonterminating*, and (2) P_3 is *predicted-terminating* for $mult(\mathscr{I}_1, \mathscr{I}_2, V_3)$ and $mult(\mathscr{I}_1, \mathscr{I}_2, \mathscr{I}_3)$, but is *predicted-nonterminating* for the remaining moded

$$\begin{array}{l}
N_0\colon mult(\underline{I}, V_2, V_3) \\
\quad C_{m_1} \downarrow \ \theta_0 = \{\underline{I}/s(X_1), Y_1/V_2, Z_1/V_3\} \\
N_1\colon mult(\underline{X_1}, V_2, U_1), add(U_1, V_2, V_3) \\
\quad C_{m_1} \downarrow \ \theta_1 = \{\underline{X_1}/s(X_2), Y_2/V_2, Z_2/U_1\} \\
N_2\colon mult(\underline{X_2}, V_2, U_2), add(U_2, V_2, U_1), \\
\qquad\qquad add(U_1, V_2, V_3) \\
\quad C_{m_2} \downarrow \ \theta_2 = \{\underline{X_2}/0, Y_3/V_2, U_2/0\} \\
N_3\colon add(0, V_2, U_1), add(U_1, V_2, V_3) \\
\quad C_{a_2} \downarrow \ \theta_3 = \{Y_4/V_2, U_1/V_2\} \\
N_4\colon add(V_2, V_2, V_3) \\
\quad C_{a_1} \downarrow \ \theta_4 = \{V_2/s(X_3), V_3/s(Z_3)\} \\
N_5\colon add(X_3, s(X_3), Z_3) \\
\quad C_{a_1} \downarrow \ \theta_5 = \{X_3/s(X_4), Z_3/s(Z_4)\} \\
N_6\colon add(X_4, s(s(X_4)), Z_4)
\end{array}$$

(a)

$$\begin{array}{l}
N_0\colon mult(\underline{I_1}, \underline{I_2}, V_3) \\
\quad C_{m_1} \downarrow \ \theta_0 = \{\underline{I_1}/s(X_1), Y_1/\underline{I_2}, Z_1/V_3\} \\
N_1\colon mult(\underline{X_1}, \underline{I_2}, U_1), add(U_1, \underline{I_2}, V_3) \\
\quad C_{m_1} \downarrow \ \theta_1 = \{\underline{X_1}/s(X_2), Y_2/\underline{I_2}, Z_2/U_1\} \\
N_2\colon mult(\underline{X_2}, \underline{I_2}, U_2), add(U_2, \underline{I_2}, U_1), \\
\qquad\qquad add(U_1, \underline{I_2}, V_3) \\
\quad C_{m_2} \downarrow \ \theta_2 = \{\underline{X_2}/0, Y_3/\underline{I_2}, U_2/0\} \\
N_3\colon add(0, \underline{I_2}, U_1), add(U_1, \underline{I_2}, V_3) \\
\quad C_{a_2} \downarrow \ \theta_3 = \{Y_4/\underline{I_2}, U_1/\underline{I_2}\} \\
N_4\colon add(\underline{I_2}, \underline{I_2}, V_3) \\
\quad C_{a_1} \downarrow \ \theta_4 = \{\underline{I_2}/s(X_3), V_3/s(Z_3)\} \\
N_5\colon add(\underline{X_3}, s(\underline{X_3}), Z_3) \\
\quad C_{a_1} \downarrow \ \theta_5 = \{\underline{X_3}/s(X_4), Z_3/s(Z_4)\} \\
N_6\colon add(\underline{X_4}, s(s(\underline{X_4})), Z_4)
\end{array}$$

(b)

Fig. 7. Two moded generalized SLDNF-trees of P_3 generated by Algorithm 5.1.

queries to *mult*. For illustration, we depict two moded generalized SLDNF-trees for $mult(\mathscr{I}, V_2, V_3)$ and $mult(\mathscr{I}_1, \mathscr{I}_2, V_3)$, as shown in Figures 7(a) and 7(b), respectively. In the two moded generalized SLDNF-trees, the prefix from N_0 down to N_2 satisfies the conditions of LP-check and has the term-size decrease property, so clause C_{m_1} is skipped when expanding N_2. When the derivation is extended to N_6, the conditions of LP-check are satisfied again, where G_6 is a loop goal of G_5 that is a loop goal of G_4. Since the derivation for $mult(\mathscr{I}, V_2, V_3)$ (Figure 7(a)) does not have the term-size decrease property, Algorithm 5.1 returns an answer, *predicted-nonterminating*, for this moded query. The derivation for $mult(\mathscr{I}_1, \mathscr{I}_2, V_3)$ (Figure 7(b)) has the term-size decrease property, so clause C_{a_1} is skipped when expanding N_6. For simplicity, we omitted all derivations leading to a success leaf. Because all derivations satisfying the conditions of LP-check have the term-size decrease property, Algorithm 5.1 ends with an answer, *predicted-terminating*, for $mult(\mathscr{I}_1, \mathscr{I}_2, V_3)$. It is then immediately inferred by Theorem 4.5 that P_3 is *predicted-terminating* for $mult(\mathscr{I}_1, \mathscr{I}_2, \mathscr{I}_3)$. It is not difficult to verify that all these predictions are correct.

AProVE07 (Giesl *et al.* 2006), NTI (Payet 2006; Payet and Mesnard 2006), Polytool (Nguyen and Schreye 2005; Nguyen *et al.* 2006), and TALP (Ohlebusch *et al.* 2000) are four well-known state-of-the-art analyzers. NTI proves nontermination, while the others prove termination. The Termination Competition 2007 (http://www.lri.fr/~marche/termination-competition/2007) reports their latest performance. We borrow three representative logic programs from the competition website to further demonstrate the effectiveness of our termination prediction.

Example 5.5

Consider the following logic program coming from the Termination Competition 2007 with Problem id *LP/talp/apt - subset1* and difficulty rating 100%. AProVE07,

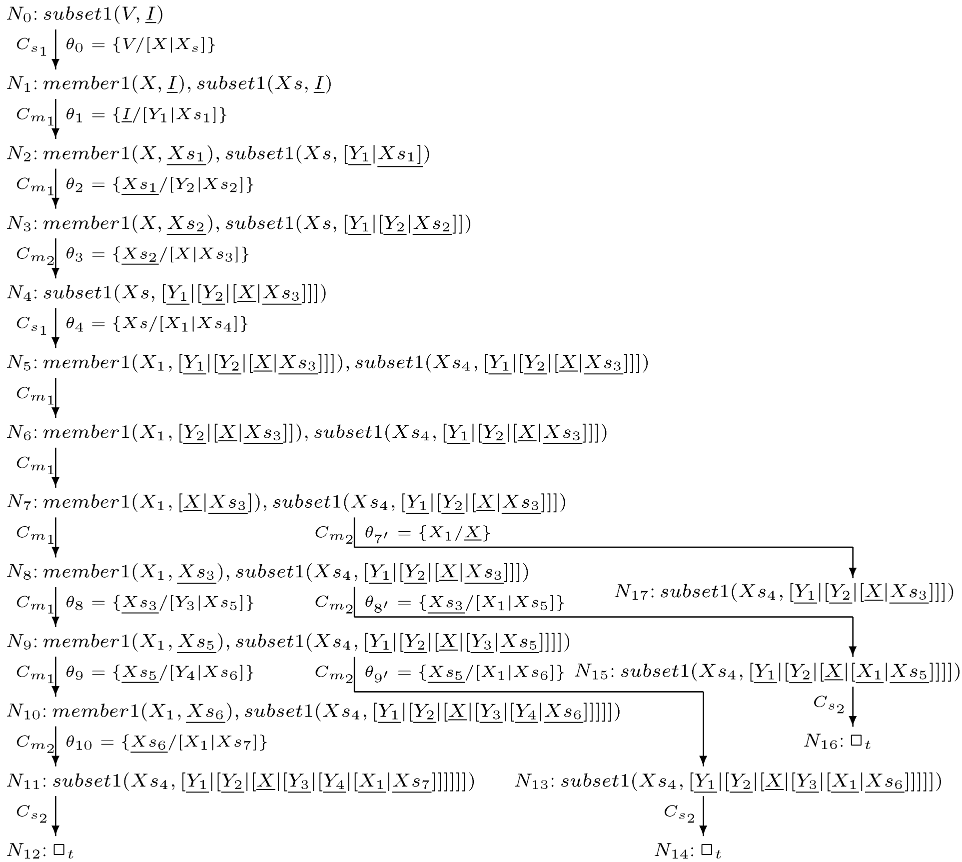

Fig. 8. The moded generalized SLDNF-tree of P_4 generated by Algorithm 5.1.

NTI, Polytool, and TALP all failed to prove/disprove its termination by yielding an answer 'don't know' in the competition.

$$
\begin{array}{lll}
P_4: & member1(X, [Y|Xs]) \leftarrow member1(X, Xs). & C_{m_1} \\
 & member1(X, [X|Xs]). & C_{m_2} \\
 & subset1([X|Xs], Ys) :- member1(X, Ys), subset1(Xs, Ys). & C_{s_1} \\
 & subset1([], Ys). & C_{s_2}
\end{array}
$$

Query Mode: $subset1(o, i)$.

The query mode $subset1(o, i)$ means that the second argument of any query must be a ground term, while the first one can be an arbitrary term. Then, to prove the termination property of P_4 with this query mode is to prove the termination for the moded query $Q_0 = subset1(V, \mathscr{I})$. Applying Algorithm 5.1 generates a moded generalized SLDNF-tree as shown in Figure 8. The prefix from N_0 down to N_3 satisfies the conditions of LP-check and has the term-size decrease property, so clause C_{m_1} is skipped when expanding N_3. When the derivation is extended to N_{10}, the conditions of LP-check are satisfied again, where G_{10} is a loop goal of G_9 that

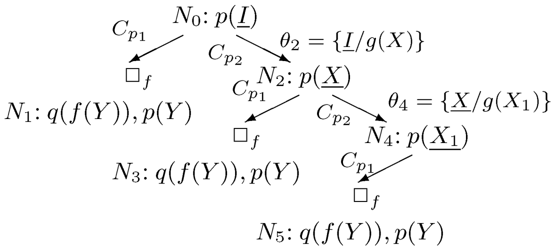

Fig. 9. The moded generalized SLDNF-tree of P_5 generated by Algorithm 5.1.

is a loop goal of G_8. Since the derivation has the term-size decrease property, N_{10} is expanded by C_{m_2}.

At N_{11} (resp. N_{13} and N_{15}), the derivation satisfies the conditions of LP-check and has the term-size decrease property, where G_{11} (resp. N_{13} and N_{15}) is a loop goal of G_4 that is a loop goal of G_0. Therefore, N_{11} (resp. N_{13} and N_{15}) is expanded by C_{s_2}. When the derivation is extended to N_{17}, the conditions of LP-check are satisfied, where G_{17} is a loop goal of G_4 that is a loop goal of G_0, but the term-size decrease condition is violated. Algorithm 5.1 stops immediately with an answer, *predicted-nonterminating*, for the query Q_0. It is easy to verify that this prediction is correct.

Example 5.6
Consider another logic program in the Termination Competition 2007 with Problem id *LP/SGST06 - incomplete* and difficulty rating 75%. Polytool succeeded to prove its termination, while AProVE07, NTI, and TALP failed.

$$\begin{array}{llr} P_5: & p(X) \leftarrow q(f(Y)), p(Y). & C_{p_1} \\ & p(g(X)) \leftarrow p(X). & C_{p_2} \\ & q(g(Y)). & C_{q_1} \end{array}$$

Query Mode: $p(i)$.

To prove the termination property of P_5 with this query mode is to prove the termination for the moded query $Q_0 = p(\mathscr{I})$. Applying Algorithm 5.1 generates a moded generalized SLDNF-tree as shown in Figure 9. The prefix from N_0 down to N_4 satisfies the conditions of LP-check and has the term-size decrease property, so clause C_{p_2} is skipped when expanding N_4. Algorithm 5.1 yields an answer *predicted-terminating* for the query Q_0. This prediction is correct.

Example 5.7
Consider a third logic program from the Termination Competition 2007 with Problem id *LP/SGST06 - incomplete2* and difficulty rating 75%. In contrast to Example 5.6, for this program AProVE07 succeeded to prove its termination, while Polytool, NTI, and TALP failed.

$$\begin{array}{llr} P_6: & f(X) \leftarrow g(s(s(s(X)))). & C_{f_1} \\ & f(s(X)) \leftarrow f(X). & C_{f_2} \\ & g(s(s(s(s(X))))) \leftarrow f(X). & C_{g_1} \end{array}$$

Query Mode: $f(i)$.

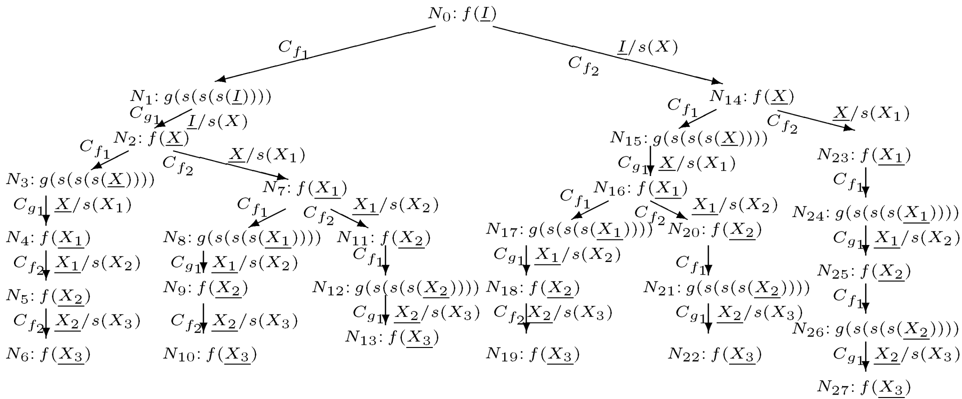

Fig. 10. The moded generalized SLDNF-tree of P_6 generated by Algorithm 5.1.

To prove the termination property of P_6 with this query mode is to prove the termination for the moded query $Q_0 = f(\mathscr{I})$. Applying Algorithm 5.1 generates a moded generalized SLDNF-tree as shown in Figure 10. C_{f_1} and/or C_{f_2} is skipped at $N_4, N_5, N_6, N_9, N_{10}$, N_{11}, N_{13}, N_{18}, $N_{19}, N_{20}, N_{22}, N_{23}, N_{25}$, and N_{27}, due to the occurrence of the following prefixes which satisfy both the conditions of LP-check and the term-size decrease condition:

(1), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} \cdots\ N_2 : f(\underline{X}) \Rightarrow_{C_{f_1}} \cdots\ N_4 : f(\underline{X_1}) \Rightarrow_{C_{f_1}}$
(2), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} \cdots\ N_2 : f(\underline{X}) \Rightarrow_{C_{f_1}} \cdots\ N_5 : f(\underline{X_2}) \Rightarrow_{C_{f_1}}$
(3), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} \cdots\ N_2 : f(\underline{X}) \Rightarrow_{C_{f_1}} \cdots\ N_6 : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
(4), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} \cdots\ N_4 : f(\underline{X_1}) \Rightarrow_{C_{f_2}} N_5 : f(\underline{X_2}) \Rightarrow_{C_{f_2}} N_6 : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
(5), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} \cdots\ N_7 : f(\underline{X_1}) \Rightarrow_{C_{f_1}} \cdots\ N_9 : f(\underline{X_2}) \Rightarrow_{C_{f_1}}$
(6), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} \cdots\ N_7 : f(\underline{X_1}) \Rightarrow_{C_{f_1}} \cdots\ N_{10} : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
(7), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} \cdots\ N_2 : f(\underline{X}) \Rightarrow_{C_{f_2}} \cdots\ N_9 : f(\underline{X_2}) \Rightarrow_{C_{f_2}} N_{10} : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
(8), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} \cdots\ N_{11} : f(\underline{X_2}) \Rightarrow_{C_{f_1}} \cdots\ N_{13} : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
(9), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} \cdots\ N_2 : f(\underline{X}) \Rightarrow_{C_{f_2}} N_7 : f(\underline{X_1}) \Rightarrow_{C_{f_2}} \cdots\ N_{13} : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
(10), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} \cdots\ N_2 : f(\underline{X}) \Rightarrow_{C_{f_2}} N_7 : f(\underline{X_1}) \Rightarrow_{C_{f_2}} N_{11} : f(\underline{X_2}) \Rightarrow_{C_{f_2}}$
(11), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_1}} \cdots\ N_{16} : f(\underline{X_1}) \Rightarrow_{C_{f_1}} \cdots\ N_{18} : f(\underline{X_2}) \Rightarrow_{C_{f_1}}$
(12), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_1}} \cdots\ N_{16} : f(\underline{X_1}) \Rightarrow_{C_{f_1}} \cdots\ N_{19} : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
(13), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} \cdots\ N_{18} : f(\underline{X_2}) \Rightarrow_{C_{f_2}} N_{19} : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
(14), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_1}} \cdots\ N_{20} : f(\underline{X_2}) \Rightarrow_{C_{f_1}} \cdots\ N_{22} : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
(15), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} \cdots\ N_{16} : f(\underline{X_1}) \Rightarrow_{C_{f_2}} \cdots\ N_{22} : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
(16), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} \cdots\ N_{16} : f(\underline{X_1}) \Rightarrow_{C_{f_2}} N_{20} : f(\underline{X_2}) \Rightarrow_{C_{f_2}}$
(17), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} \cdots\ N_{23} : f(\underline{X_1}) \Rightarrow_{C_{f_1}} \cdots\ N_{25} : f(\underline{X_2}) \Rightarrow_{C_{f_1}} \cdots\ N_{27} : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
(18), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_2}} \cdots\ N_{27} : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
(19), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_2}} \cdots\ N_{25} : f(\underline{X_2}) \Rightarrow_{C_{f_2}}$
(20), $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_2}} \cdots\ N_{23} : f(\underline{X_1}) \Rightarrow_{C_{f_2}}$

Since there is no derivation satisfying the conditions of LP-check while violating the term-size decrease condition, Algorithm 5.1 ends with an answer *predicted-terminating* for the query Q_0. This again is a correct prediction.

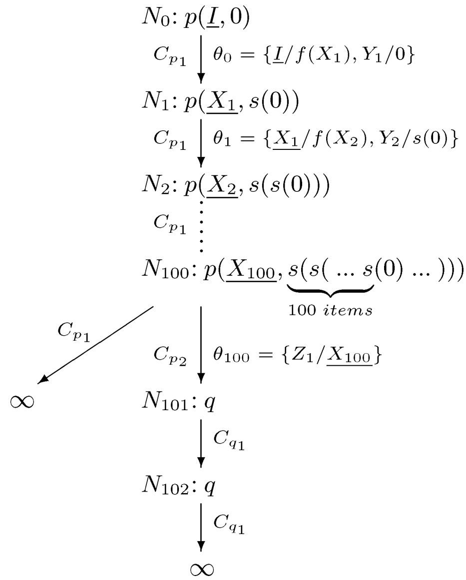

Fig. 11. The moded generalized SLDNF-tree of P_7 with a moded query $p(\mathscr{I}, 0)$.

Choosing $r = 3$ for Algorithm 5.1, we are able to obtain correct predictions for all benchmark programs of the Termination Competition 2007 (see Section 6). However, we should remark that due to the undecidability of the termination problem, there exist cases that choosing $r = 3$ will lead to an incorrect prediction. Consider the following carefully crafted logic program:

$$
\begin{array}{llr}
P_7: & p(f(X), Y) \leftarrow p(X, s(Y)). & C_{p_1} \\
& p(Z, \underbrace{s(s(}_{100\ items}(0)\ldots))) \leftarrow q. & C_{p_2} \\
& q \leftarrow q. & C_{q_1}
\end{array}
$$

P_7 does not terminate for a moded query $Q_0 = p(\mathscr{I}, 0)$, as there is an infinite derivation (see Figure 11)

$$N_0 : p(\underline{I}, 0) \Rightarrow_{C_{p_1}} \cdots \; N_{101} : q \Rightarrow_{C_{q_1}} N_{102} : q \Rightarrow_{C_{q_1}} \cdots$$

which satisfies conditions (i) and (ii) of Theorem 4.3, where for any $j \geqslant 101$, G_{j+1} is a loop goal of G_j. Note that for any repetition number r with $3 \leqslant r \leqslant 100$, the prefix ending at N_{r-1} satisfies both the conditions of LP-check and the term-size decrease property, where for any j with $0 \leqslant j < r - 1$, G_{j+1} is a loop goal of G_j. However, for any $r > 100$, a prefix ending at N_{100+r} will be encountered, which satisfies the conditions of LP-check but violates the term-size decrease condition, where for any j with $101 \leqslant j < 100 + r$, G_{j+1} is a loop goal of G_j. Therefore, Algorithm 5.1 will return *predicted-terminating* for Q_0 unless r is set above 100.

The following result shows that choosing a sufficiently large repetition number guarantees the correct prediction for nonterminating programs.

Theorem 5.5
Let P be a logic program and Q be a query such that P is nonterminating for Q. There always exists a number R such that Algorithm 5.1 with any repetition number $r \geqslant R$ produces the answer *predicted-nonterminating*.

Proof
Let us assume the contrary. That is, we assume that for any number N, there exists a larger number r such that Algorithm 5.1 for P with query Q and repetition number r produces the answer *predicted-terminating*. This means that for all $r \geqslant 2$ the prefix of form (3) of each infinite branch D in the moded generalized SLDNF-tree MT_Q satisfies the term-size decrease property. According to Theorem 5.2, D has an infinite chain of substitutions of form (1) for some input variable I at Q. This means that D does not satisfy condition (ii) of Theorem 4.3. However, since P is nonterminating for Q, by Corollary 4.4 MT_Q has at least one infinite branch of form (2) satisfying conditions (i) and (ii) of Theorem 4.3. We then have a contradiction and thus conclude the proof. □

The same result applies for any concrete query Q. That is, there always exists a number R such that Algorithm 5.1 with any $r \geqslant R$ produces the answer *terminating* or *predicted-terminating* when P is terminating for Q. The proof for this is simple. When P is terminating for a concrete query Q, the (moded) generalized SLDNF-tree for Q is finite. Let R be the number of nodes of the longest branch in the tree. For any $r \geqslant R$, Algorithm 5.1 will produce the answer *terminating* or *predicted-terminating*, since no branch will be cut by LP-check.

However, whether the above claim holds for any moded query Q when P is terminating for Q remains an interesting open problem.

6 Experimental results

We have evaluated our termination prediction technique on a benchmark of 301 Prolog programs. In this section, we first describe the benchmark and our experimental results using a straightforward implementation of Algorithm 5.1. Then, we define a pruning technique to reduce the size of moded generalized SLDNF-trees generated for our prediction. Finally, we make a comparison between the state-of-the-art termination and nontermination analyzers and our termination prediction tool.

Our benchmark consists of 301 programs with moded queries from the Termination Competition 2007 (http://www.lri.fr/˜marche/termination-competition/2007). Only 23 programs of the competition are omitted because they contain nonlogical operations such as arithmetics (for most of these programs neither termination nor nontermination could be shown by any of the tools in the competition). The benchmark contains 244 terminating programs and 57 nonterminating ones. The most accurate termination analyzer of the competition, AProVE (Giesl *et al.* 2006), proves termination of 238 benchmark programs. The nontermination analyzer NTI (Payet 2006; Payet and Mesnard 2006) proves nontermination of 42 programs. Because the prediction does not produce a termination or nontermination proof,

Table 1. Prediction with different repetition numbers

	$r = 2$	$r = 3$	$r = 4$
Correct predictions	291	271	234
Wrong predictions	7	0	0
Out of time/memory	3	30	67
Average time (s)	1.7	24.9	59.3

our goal is to outperform the analyzers of the competition by providing a higher number of correct predictions.

We implemented our tool, TPoLP: Termination Prediction of Logic Programs, in SWI-prolog (http://www.swi-prolog.org). TPoLP is freely available from http://www.cs.kuleuven.be/~dean. The moded generalized SLDNF-tree is generated following Algorithm 5.1. It is initialized with the moded query and extended until all branches are cut or a timeout occurs. To improve the efficiency of the analysis, a number of optimizations were implemented, such as constant time access to the nodes and the arcs of the derivations. The experiments have been performed using SWI-Prolog 5.6.40 (http://www.swi-prolog.org), on an Intel Core2 Duo 2,33GHz, 2 Gb RAM.

Table 1 gives an overview of the predictions with repetition numbers $r = 2$, $r = 3$, and $r = 4$. As we mentioned earlier, $r = 2$ does not suffice because some of the predictions are wrong and we want high reliable predictions. When r is set above two, all predictions made for the benchmark are correct. This shows that in practice, there is no need to increase the repetition number any further.

When we increase the repetition number, the cost of prediction increases as well. Table 1 shows that for $r = 3$, about 10% of the programs break the time limit of 4 min, and for $r = 4$, about 20% break the limit.

The component of the algorithm taking most of the time differs from program to program. When a lot of branches are cut by LP-check, constructing the LP cuts is usually the bottleneck. For programs with a low amount of LP cuts, most of the time is spent on constructing the SLDNF-derivations. Some of the derivations count more than a million nodes. To overcome such performance issues, we implemented the following pruning technique on loop goals:

Definition 6.1 (*Pruning variants*)
Let G_2 be a loop goal of G_1 for which the selected subgoals are variants. Then, all clauses that have already been applied at G_2 are skipped at G_1 during backtracking.

The idea of this pruning is simple. For loop goals with variant selected subgoals, applying the nonlooping clauses to them will generate the same derivations below them with the same termination properties. Therefore, the derivations already generated below G_2 need not be regenerated at G_1 during backtracking.

For the sake of efficiency, in our implementation we determine variants by checking that they have the same symbol string.

Consider Example 5.7 again. When the above pruning mechanism is applied, Algorithm 5.1 will simplify the moded generalized SLDNF-tree of Figure 10 into

Table 2. The effect of pruning

	No pruning	Pruning variants	Pruning loop goals
Correct predictions	271	296	297
Wrong predictions	0	0	3
Out of time/memory	30	5	1
Average time (s)	24.9	4.4	0.05

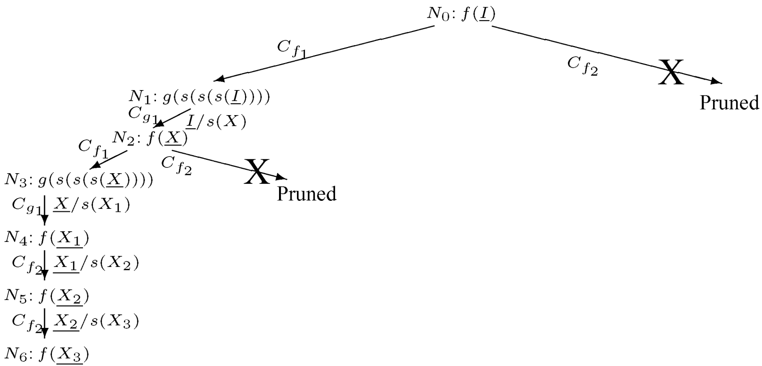

Fig. 12. Figure 10 is simplified with pruning.

Figure 12. The pruning takes place at N_2 and N_0, where G_4 is a loop goal of G_2 that is a loop goal of G_0 and their selected subgoals are variants.

A stronger version of the above pruning mechanism can be obtained by removing the condition in Definition 6.1: for which the selected subgoals are variants. That is, we do not require the selected subgoals of loop goals to be variants. We call this version *Pruning loop goals.*

Table 2 gives an overview of our predictions with $r = 3$ as the repetition number in the cases of no pruning, pruning variants, and pruning loop goals. The table shows that pruning is a good tradeoff between the accuracy and the efficiency of the prediction. When applying the variants pruning mechanism the size of the derivations drops considerably, while all predictions for the benchmark are still correct. Due to the pruning, more than 98% of the predictions finish within the time limit. Applying the loop goals pruning mechanism leads to a greater reduction in the size of derivations. However, in this case we sacrifice accuracy: three nonterminating programs are predicted to be terminating.

Table 3 gives a comparison between our predictions (with $r = 3$ and the variants pruning mechanism) and the proving results of the state-of-the-art termination and nontermination analyzers. Note that our tool, TPoLP, is the only tool which analyzes both for termination and nontermination of logic programs. The results are very encouraging. We correctly predict the termination property of all benchmark programs except for five programs which broke the time limit. It is also worth noticing that for all programs of the benchmark, either an existing analyzer finds a termination or nontermination proof, or a correct prediction is made by our tool.

Table 3. Comparison between TPoLP and the existing analyzers.

		Termination/nontermination proof			
	TPoLP prediction	AProVE	NTI	Polytool	TALP
Answer *Terminating* (244)	239	238	0	206	164
Answer *Nonterminating* (57)	57	0	42	0	0

This shows that our prediction tool can be a very useful addition to any termination or nontermination analyzer.

7 Related work

Most existing approaches to the termination problem are *norm*- or *level mapping-based* in the sense that they perform termination analysis by building from the source code of a logic program some well-founded termination conditions/constraints in terms of norms (i.e., term sizes of atoms of clauses), level mappings, interargument size relations, and/or instantiation dependencies, which when solved, yield a termination proof (see, e.g., Schreye and Decorte 1993 for a survey and more recent papers Apt and Pedreschi 1993; Marchiroi 1996a; Lidenstrauss and Sagiv 1997; Decorte *et al.* 1999; Mesnard and Neumerkel 2001; Bossi *et al.* 2002; Genaim and Codish 2005; Bruynooghe *et al.* 2007). Another main stream is *transformational* approaches, which transform a logic program into a term rewriting system (TRS) and then analyze the termination property of the resulting TRS instead (Aguzzi and Modigliani 1993; Arts and Zantema 1995; Marchiori 1996b; van Raamsdonk 1997; Rao *et al.* 1998; Ohlebusch *et al.* 2000; Giesl *et al.* 2006; Schneider-Kamp *et al.* 2006). All of these approaches are used for a termination proof; i.e., they compute sufficient termination conditions which once satisfied, lead to a positive conclusion *terminating*. Recently, Payet (2006) and Payet and Mesnard (2006) propose an approach to computing sufficient nontermination conditions which when satisfied lead to a negative conclusion *nonterminating*. A majority of these termination/nontermination proof approaches apply only to positive logic programs.

Our approach presented in this paper differs significantly from existing termination analysis approaches. First, we do not make a termination proof, nor do we make a nontermination proof. Instead, we make a termination prediction (see Figure 1)—a heuristic approach to attacking the undecidable termination problem. Second, we do not rely on static norms or level mappings, nor do we transform a logic program to a term rewriting system. Instead, we focus on detecting infinite SLDNF-derivations with the understanding that a logic program is terminating for a query if and only if there is no infinite SLDNF-derivation with the query. We have established a necessary and sufficient characterization of infinite SLDNF-derivations with arbitrary (concrete or moded) queries, introduced a new loop-checking mechanism, and developed an algorithm that predicts termination of general logic programs with arbitrary queries by identifying potential infinite SLDNF-derivations. Since the algorithm implements the necessary and sufficient conditions (the characterization) of an infinite SLDNF-derivation, its prediction is very effective. Our experimental results show that except for five programs which break the time limit, our prediction is 100% correct for all 296 benchmark programs of the Termination Competition 2007, of which 18 programs cannot be proved by any of the existing state-of-the-art analyzers like AProVE07 (Giesl *et al.* 2006), NTI (Payet 2006; Payet and Mesnard 2006), Polytool (Nguyen and Schreye 2005; Nguyen *et al.* 2006), and TALP (Ohlebusch *et al.* 2000).

Our termination prediction approach uses a loop-checking mechanism (a loop check) to implement a characterization of infinite SLDNF-derivations. Well-known loop checks include VA-check (Gelder 1987; Bol *et al.* 1991), OS-check (Bruynooghe *et al.* 1992; Sahlin 1993; Martens and Schreye 1996), and VAF-checks (Shen 1997; Shen *et al.* 2001). All apply to positive logic programs. In particular, VA-check applies to function-free logic programs, where an infinite derivation is characterized by a sequence of selected *variant subgoals*. OS-check identifies an infinite derivation with a sequence of selected subgoals with the same predicate symbol *whose sizes do not decrease*. VAF-checks take a sequence of selected *expanded variant subgoals* as major characteristics of an infinite derivation. Expanded variant subgoals are variant subgoals except that some terms may grow bigger. In this paper, a new loop-check mechanism, LP-check, is introduced in which an infinite derivation is identified with a sequence of *loop goals*. Most importantly, enhancing LP-check with the term-size decrease property leads to the first loop check for moded queries.

8 Conclusion and future work

We have presented a heuristic framework for attacking the undecidable termination problem of logic programs, as an alternative to current termination/nontermination proof approaches. We introduced an idea of termination prediction, established a necessary and sufficient characterization of infinite SLDNF-derivations with arbitrary (concrete or moded) queries, built a new loop-checking mechanism, and developed an algorithm that predicts termination of general logic programs with arbitrary queries. We have implemented a termination prediction tool, TPoLP, and obtained quite satisfactory experimental results. Except for five programs which break the experiment time limit, our prediction is 100% correct for all 296 benchmark programs of the Termination Competition 2007.

Our prediction approach can be used standalone, e.g., it may be incorporated into Prolog as a termination debugging tool; or it is used along with some termination/nontermination proof tools (see the framework in Figure 1).

Limitations of the current prediction approach include that it cannot handle floundering queries and programs with nonlogical operators. To avoid floundering, we assume that no negative subgoals containing either ordinary or input variables are selected at any node of a moded generalized SLDNF-tree (violation of the assumption can easily be checked in the course of constructing generalized SLDNF-trees). This assumption seems able to be relaxed by allowing input variables to occur in selected negative subgoals. This makes us able to predict termination of programs like

$$P: \quad p(X) \leftarrow \neg q(X).$$
$$q(a) \leftarrow q(a).$$

which is nonterminating for the moded query $p(\mathcal{I})$.

Our future work includes further improvement of the prediction efficiency of TPoLP. As shown in Table 2, there are five benchmark programs breaking our experiment time limit. We are also considering extensions of the proposed termination prediction to typed queries (Bruynooghe *et al.* 2007) and to logic programs with tabling (Chen and Warren 1996; Verbaeten *et al.* 2001; Shen *et al.* 2004).

Acknowledgements

We would like to thank the anonymous referees for their constructive comments and suggestions that helped us improve this work. Yi-Dong Shen is supported in part by NSFC grants 60673103, 60721061, and 60833001, and by the National High-tech R&D Program (863 Program). Dean Voets is supported by the Flemish Fund for Scientific Research—FWO-project G0561-08.

References

Aguzzi, G. and Modigliani, U. 1993. Proving termination of logic programs by transforming them into equivalent term rewriting systems. In *Proceedings of the 13th FST and TCS*, Bombay, India. Lecture Notes in Computer Science 761, Springer, 114–124.

Apt, K. R. and Pedreschi, D. 1993. Reasoning about termination of pure prolog programs. *Information and Computation 106*(1), 109–157.

Arts, T. and Zantema, H. 1995. Termination of logic programs using semantic unification. In *Proceedings of the 5th International Symposium on Logic-based Program Synthesis and Transformation*, Utrecht, the Netherlands. Lecture Notes in Computer Science 1048, Springer, 219–233.

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., Rossi, S. and Etalle, S. 2002. On modular termination proofs of general logic programs. *Theory and Practice of Logic Programming 2*(3), 263–291.

Bruynooghe, M., Codish, M., Gallagher, J., Genaim, S. and Vanhoof, W. 2007. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems 29*(2), 10.

Bruynooghe, M., Schreye, D. D. and Martens, B. 1992. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing 11*(1), 47–79.

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

Clark, K. L. 1978. Negation as failure. In *Logic and Databases*, Ed. H. Gallaire and J. Minker, Perseus Publishing, 293–322.

Decorte, S., Schreye, D. D. and Vandecasteele, H. 1999. Constraint-based termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems 21*(6), 1137–1195.

Dershowttz, N., Lindenstrauss, N., Sagiv, Y. and Serebrenik, A. 2001. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing 12*(1/2), 117–156.

Gelder, A. V. 1987. Efficient loop detection in prolog. *Journal of Logic Programming 4*, 23–31.

Genaim, S. and Codish, M. 2005. Inferring termination conditions for logic programs using backwards analysis. *Theory and Practice of Logic Programming* 5(1/2), 75–91.

Giesl, J., Schneider-Kamp, P. and Thiemann, R. 2006. Aprove 1.2: Automatic termination proofs in the dp framework. In *Proceedings of the 3rd IJCAR*, Seattle, WA, USA. Lecture Notes in Computer Science 4130, Springer, 281–286.

Lindenstrauss, N. and Sagiv, Y. 1997. Automatic termination analysis of logic programs. In *Proceedings of the 14th ICLP*, Leuven, Belgium. MIT Press, 63–77.

Lindenstrauss, N., Sagiv, Y. and Serebrenik, A. 1997. Termilog: A system for checking termination of queries to logic programs. In *Proceedings of the 9th CAV*, Haifa, Israel. Lecture Notes in Computer Science 1254, Springer, 444–447.

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

Marchiori, E. 1996a. Practical methods for proving termination of general logic programs. *Journal of Artificial Intelligence Research 4*, 179–208.

Marchiori, M. 1996b. Proving existential termination of normal logic programs. In *Proceedings of the 5th AMAST*, Munich, Germany. Lecture Notes in Computer Science 1101, Springer, 375–390.

Martens, B. and Schreye, D. D. 1996. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming 28*(2), 89–146.

Mesnard, F. and Bagnara, R. 2005. cti: A constraint-based termination inference tool for iso-prolog. *Theory and Practice of Logic Programming* 5(1/2), 243–257.

Mesnard, F. and Neumerkel, U. 2001. Applying static analysis techniques for inferring termination conditions of logic programs. In *Proceedings of the 8th International Symposium on Static Analysis*, Paris, France. Lecture Notes in Computer Science 2126, Springer, 93–110.

Nguyen, M. T. and Schreye, D. D. 2005. Polynomial interpretations as a basis for termination analysis of logic programs. In *Proceedings of the 21st ICLP*, Sitges, Spain. Lecture Notes in Computer Science 3668, Springer, 311–326.

Nguyen, M. T., Schreye, D. D., Giesl, J. and Schneider-Kamp, P. 2006. *Polytool: Proving Termination Automatically Based on Polynomial Interpretations*. Technical Report, Department of Computer Science, K.U. Leuven, Belgium.

Ohlebusch, E., Claves, C., and Marche, C. 2000. Talp: A tool for the termination analysis of logic programs. In *Proceedings of the 11th RTA*, Norwich, UK. Lecture Notes in Computer Science 1833, Springer, 270–273.

Payet, E. 2006. Detecting non-termination of term rewriting systems using an unfolding operator. In *Proc. of the 16th International Symposium on Logic-Based Program Synthesis and Transformation*, Venice, Italy. Lecture Notes in Computer Science 4407, Springer, 194–209.

Payet, E. and Mesnard, F. 2006. Non-termination inference of logic programs. *ACM Transactions on Programming Languages and Systems 28*(2), 256–289.

Rao, M. K., Kapur, D. and Shyamasundar, R. 1998. Transformational methodology for proving termination of logic programs. *Journal of Logic Programming 34*(1), 1–42.

Sahlin, D. 1993. Mixtus: An automatic partial evaluator for full prolog. *New Generation Computing 12*(1), 7–51.

Schneider-Kamp, P., Giesl, J., Serebrenik, A. and Thiemann, R. 2006. Automated termination analysis for logic programs by term rewriting. In *Proceedings of the 16th*

International Symposium on Logic-based Program Synthesis and Transformation, Venice, Italy. Lecture Notes in Computer Science 4407, Springer, 177–193.

Schreye, D. D. and Decorte, S. 1993. Termination of logic programs: The never-ending story. *Journal of Logic Programming 19/20*(1), 199–260.

Shen, Y. D. 1997. An extended variant of atoms loop check for positive logic programs. *New Generation Computing 15*(2), 187–204.

Shen, Y. D., You, J. H. and Yuan, L. Y. 2004. Enhancing global sls-resolution with loop cutting and tabling mechanisms. *Theoretical Computer Science 328*(3), 271–287.

Shen, Y. D., You, J. H., Yuan, L. Y., Shen, S. P. and Yang, Q. 2003. A dynamic approach to characterizing termination of general logic programs. *ACM Transactions on Compututational Logic 4*(4), 417–430.

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.

van Raamsdonk, F. 1997. Translating logic programs into conditional rewriting systems. In *Proceedings of the 14th ICLP*, Leuven, Belgium. MIT Press, 168–182.

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