A new approach to non-termination analysis of Logic Programs

Dean Voets
Danny De Schreye

Report CW 537, March 2009



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A new approach to non-termination analysis of Logic Programs

Dean Voets
Danny De Schreye

Report CW 537, March 2009

Department of Computer Science, K.U.Leuven

Abstract

In this paper, we present a new approach to non-termination analysis of logic programs based on moded SLDNF-resolution. Moded SLDNF-resolution is a symbolic execution for moded goals developed for termination prediction. To prove non-termination, we use a complete loop checker to create a finite symbolic derivation tree of a logic program and moded query. Then, we check if this derivation tree contains an infinite loop, using a new non-termination condition. We implemented this approach and tested it on the benchmark from the Termination Competition of 2007. The results are very satisfactory: our tool is able to prove non-termination and construct non-terminating queries for all non-terminating benchmark programs, and thus, significantly improves on the results of the only other non-termination analyzer, *NTI*.

Keywords : non-termination analysis, program analysis.

1 Introduction

One of the central concerns of declarative programming, in particular of Logic Programming, is that the use of a declarative programming style in a declarative programming language leads to less error-prone, more understandable and better maintainable programs. However, it is well-known that a declarative programming style also results in less efficient computations, and in the extreme case, in non-terminating computations. The latter problem has received considerable attention within the community. Much research has been done on termination analysis, loop detection and, more recently, non-termination analysis.

Among these areas, termination analysis has by far received most attention. [13] presents a survey of the extensive amount of work up till 1994. But, most of the more powerful approaches and techniques have been introduced in the last decade: the constraint-based approach to termination analysis [5], the local approaches [4], the use of types in termination analysis [2], powerful transformational approaches [12], termination inference [7], and the porting of TRS-techniques to the LP-context [14], [9], [8].

A new concern in this research is the precision of the termination analysis. Since termination is undecidable in general, only sufficient conditions for terminations are verified. It is important to have a good understanding of the precision of these techniques: do they actually capture most of the terminating computations?

With respect to the other two approaches, loop detection and non-termination analysis, there is often confusion concerning their relation. Because both approaches use similar techniques, their distinguishing features and aims are not always well understood. Loop detection is a run-time technique. It aims to cut infinite derivations for a concrete query at run-time. One of the possible approaches to achieve this is tabulation [11]. For an extensive overview and comparison of different loop checking algorithms, we refer to [1]. Non-termination analysis is a compile-time approach. It aims to prove that a certain class of queries will result in non-terminating computations for at least some of the queries in the considered class. Typically, non-termination analysis is performed for classes of queries described in terms of modes (or types). One of the key concerns of non-termination analysis is to address the important issue of precision analysis of termination analysis. A termination analysis can be shown to be precise by proving the that the class of queries for which termination could not be proven is actually non-terminating. This has been one of the main goals and achievements of the only non-termination analyzer developed up till now, NTI[10].

Very recently, yet another, fourth approach to the problem has been introduced: termination and non-termination prediction [15]. In this approach, techniques developed in loop-detection are lifted to classes of moded queries to allow a prediction of the termination behavior of these queries. Although the predictions do not take the form of formal proofs, experiments show that they can be extremely precise. Moreover, for non-termination prediction, it has been proven that by increasing a parameter in the analysis, the repetition number, in the limit, the prediction is always correct.

Our work has been inspired both by the work on termination/non-termination prediction and by NTI. We propose a new non-termination analysis. It reuses the analysis scheme proposed in [15] to produce a finite representation of the computation for a model query, given some logic program. We introduce a new non-termination condition expressed in terms of this finite representation of the computation. We prove its correctness and extend it in several ways.

It turns out that our characterization of non-terminating computations is more precise than that of NTI. We have implemented the technique and performed extensive experiments with it on the basis of the benchmark of the termination analysis competition of 2007[1]. The experiments show that our technique has a 100% success-rate on this benchmark, outperforming the only competiting approach, NTI.

The paper is organized as follows. In the next section we introduce some preliminaries. In section 3, we present our conditions implying non-termination and show that we are able to derive classes of non-terminating queries. In Section 4, we present our experimental evaluation and we compare our analyzer with the non-termination inference tool NTI [10]. Finally, Section 5 concludes this paper.

2 Preliminaries

In the next section we introduce some preliminaries concerning the symbolic derivation tree used to prove non-termination. First, we introduce moded SLDNF-trees as defined in [15]. These trees represent the derivation trees of all concrete queries corresponding to a moded query. Then, we introduce complete loop checks for these SLDNF-trees and introduce *LP-check* [15], a loop check for moded SLDNF-resolution introduced for termination prediction.

2.1 Moded generalized SLDNF-trees

We assume the reader is familiar with standard terminology of logic programs, in particular with SLDNF-resolution, as described in [6]. Variables are denoted by character strings beginning with a capital letter. Predicates, functions and constant symbols are denoted by character strings beginning with a lower case letter. A term is a constant, a variable, or a function of the form $f(t_1, ..., t_m)$ where f is a function symbol and each t_i is a term. We denote the set of terms constructible from a program P, by $Term_P$. An atom is of the form $p(t_1, ..., t_m)$ where p is a predicate symbol. Two atoms are called *variants* if they are equal 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 $A \leftarrow L_1, ..., L_n$, where A is an atom and each L_i is a literal. A goal G_i is a headless clause $\leftarrow L_1, ..., L_n$. A query, Q, is a conjunction of literals $L_1, \ldots, L_n$. The goal, $G_0 = \leftarrow Q$, for a query Q is called a *top goal*. Without loss of generality, we assume that Q consists only of one atom. Q is a *moded* query if some arguments of Q are input modes, otherwise, it is a *concrete* query.

[1] www.lri.fr/~marche/termination-competition/

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 [16], 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. 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 literals. Let L_i and L_j be the selected literals at two nodes N_i and N_j, respectively. L_i is an *ancestor* of L_j, denoted $L_i \prec_{anc} L_j$, 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 goals and literals. So by the *selected literal* in each node $N_i :\leftarrow L_1, ..., L_n$, we refer to the left-most literal L_1.

Recall that in SLDNF-resolution, let $L_i = \neg A$ be a ground negative literal selected at N_i, then, by the negation-as-failure rule [3], a subsidiary child SLDNF-tree 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 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 a 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 non-ground negative literals are selected at any node of a generalized SLDNF-tree (see [16]).

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 literal in G_i.

A derivation step is denoted by $N_i : G_i \Rightarrow_C N_{i+1} : G_{i+1}$, meaning that applying a clause C to G_i produces $N_{i+1} : G_{i+1}$.

As we are interested in proving non-termination of moded queries, the definition of a generalized SLDNF-tree is extended for moded goals. Note that an input mode stands for an arbitrary ground term, i.e. it can be any variable-free term of $Term_P$. This suggests that we may approximate the effect of an input mode, by treating it as a special variable I, in such a way that in SLDNF-derivations I can be substituted by a constant or function, but cannot be substituted by an ordinary variable. Therefore, when unifying a special variable I and a variable X, we always substitute I for X. In the remainder of the paper, we will denote a special variable by underlining the variable's name.

Definition 1. *Let P be a logic program and $Q = p(\mathcal{I}_1, ..., \mathcal{I}_m, T_1, ..., T_n)$ a moded query. The* ***moded generalized SLDNF-tree*** *of P for Q, is defined to be the generalized SLDNF-tree GT_{G_0} for $P \cup \{\leftarrow p(\underline{I_1}, ..., \underline{I_m}, T_1, ..., T_n)\}$, with each $\underline{I_i}$ being a distinct special variable not occurring in any T_j. The special variables $\underline{I_1}, ..., \underline{I_m}$ are called* ***input variables****.* □

In a moded generalized SLDNF-tree, an input variable $\underline{I}$ may be substituted by either a constant or a function $f(t_1, \ldots, t_n)$. If $\underline{I}$ is substituted by $f(t_1, \ldots, t_n)$, all variables in $t_1, \ldots, t_n$ are also called input variables and treated as special variables. We refer to Figure 1(a) for an illustration of a (part of) a moded generalized SLDNF-tree. The figure also illustrates a loop check.

A moded atom A corresponds to a set of concrete atoms, called the *denotation* of A. Let $\underline{I_1}, ..., \underline{I_n}$ be all input variables occurring in A. Let $t_1, \ldots, t_n \in Term_P$. $A(t_1 \rightarrow \underline{I_1}, \ldots, t_n \rightarrow \underline{I_n})$ denotes the concrete atom obtained by replacing the input variables $\underline{I_1}, ..., \underline{I_n}$ by the terms $t_1, \ldots, t_n$.

Definition 2. *Let A be an atom with $\underline{I_1}, \ldots, \underline{I_n}$ as its input variables. The **denotation** of A is*

$$Den(A) = \big\{A(t_1 \rightarrow \underline{I_1}, \ldots, t_n \rightarrow \underline{I_n}) \mid t_i \in Term_P,\ t_i \text{ ground}\big\}. \qquad \square$$

This concept can be adapted to moded goals in a straightforward way. Note that the denotation of a concrete atom is a singleton containing the atom itself.

2.2 Loop Checking

A complete loop check for moded goals cuts all infinite branches in a moded generalized SLDNF-tree.

Definition 3. *A loop check L is **complete** w.r.t. moded SLDNF-resolution if for every logic program P and moded query Q, every infinite derivation of P for Q is cut by L.* $\square$

Many simple complete loop checks can be constructed, for example a bound on the number of times a certain predicate occurs in a derivation. However, only one loop check for moded SLDNF-resolution is discussed in the literature, *LP-check* [15]. LP-check is a complete loop check developed for termination prediction.

In [15], it is proven that every infinite derivation contains an infinite chain of *loop goals*. These are goals satisfying some conditions on the selected literals. A clause is cut by LP-check if a prefix of such a chain is encountered.

Definition 4. *Let A be a moded atom, the **symbol string** of A, S_A, is the string obtained by reading all predicate symbols, function symbols, constants and variables in A, from left to right, with the variables replaced by $\mathcal{X}$.*

*A symbol string S_{A_1} is a **projection** of S_{A_2}, denoted $S_{A_1} \subseteq_{proj} S_{A_2}$, if S_{T_1} is obtained from S_{T_2} by removing zero or more elements.* $\square$

Example 1. Let $T_1 = a$ and $T_2 = f(X, g(X, f(a, \underline{I})))$. Then, $S_{T_1} = a$, $S_{T_2} = f\mathcal{X}g\mathcal{X}fa\mathcal{X}$ and $S_{T_1} \subseteq_{proj} S_{T_2}$. $\square$

Definition 5. *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 $S_{L_i^1} \subseteq_{proj} S_{L_j^1}$. Then, G_j is called a **loop goal** of G_i.* $\square$

LP-check uses a *repetition number* defining how long the chain of loop goals can become before it is cut by LP-check.

Definition 6. *Given a repetition number $r \geq 2$, **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*

$$N_0 : G_0 \Rightarrow_{C_0} \ldots N_{g_1} : G_{g_1} \Rightarrow_{C_k} \ldots N_{g_2} : G_{g_2} \Rightarrow_{C_k} \ldots N_{g_r} : G_{g_r} \Rightarrow_{C_k} \qquad (1)$$

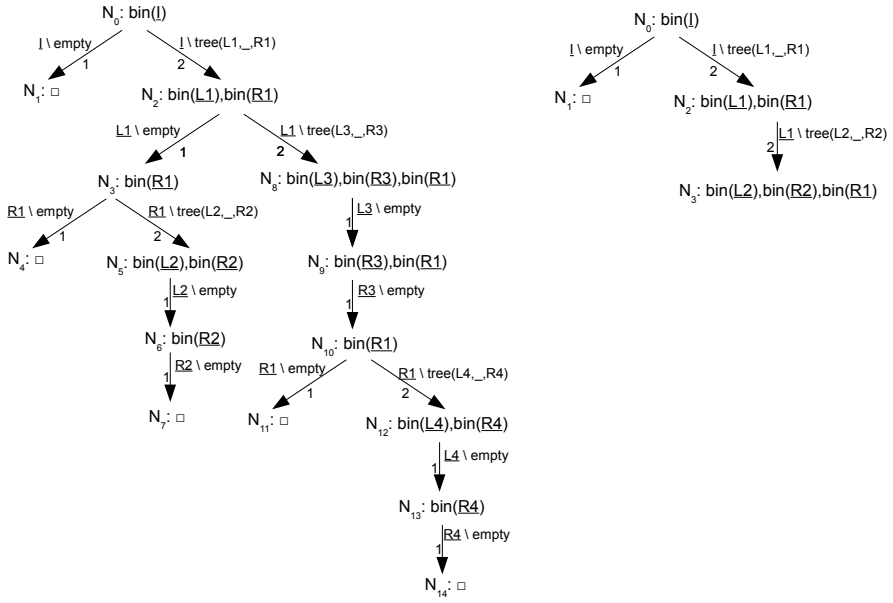
such that (a) for any $j < r$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}, and (b) for all $j \leq r$, the clause C_k applied to G_{g_j} is the same. The prefix is called an ***LP-cut****, the nodes $N_{g_1}, \ldots, N_{g_r}$ are called the nodes of the LP-cut.* □

Because LP-check is a rather expensive loop check, a variant on LP-check is defined in [15]: *LP-check with pruning*. This loop check reduces the amount of redundant branches by pruning clauses if they are already applied to an ancestor or descendant with a variant as a selected literal. We illustrate these loop checks with the *binary tree* program.

Example 2. The following program succeeds if the argument of the top goal represents a binary tree.

```
bin(empty).
bin(tree(L,_,R)):- bin(L), bin(R).
```

Figures 1(a) and 1(b) show moded SLDNF-trees constructed using LP-check and LP-check with pruning, respectively, for the *binary tree* program with $bin(\underline{I})$ as a query and 3 as a repetition number.



(a) LP-check (b) LP-check with pruning

Fig. 1. Two loop checks for moded SLDNF-resolution

In the SLDNF-tree constructed by LP-check, as shown in Figure 1(a), clause 2 is cut at node N_5 because of LP-cut: $N_0 \Rightarrow_{C_2} \ldots N_3 \Rightarrow_{C_2} N_5 \Rightarrow_{C_2}$. Similarly, LP-check cuts clause 2 at nodes: N_6, N_8, N_9, N_{12} and N_{13}.

The SLDNF-tree constructed by LP-check with pruning, depicted in Figure 1(b), is much smaller. Clause 1 is cut at nodes N_2 and N_3, because that clause is

applied to the selected literal of N_0, which is both an ancestor and a variant of the selected literals of nodes N_2 and N_3. At node N_3, clause 2 is cut by LP-check with pruning because of LP-cut: $N_0 \Rightarrow_{C_2} N_2 \Rightarrow_{C_2} N_3 \Rightarrow_{C_2}$. □

In [15], a technique to predict the termination behavior of a logic program and moded query, using LP-check or LP-check with pruning, is introduced. Every time a clause is cut, one checks a condition on the selected literals occuring in the derivation. If this condition fails, the LP-cut is predicted to identify an infinite loop. It is proven that any non-terminating program and moded query is predicted to be non-terminating if the repetition number is sufficiently high.

3 A new non-termination condition

In this section, we present a new non-termination analysis technique for general logic programs with moded queries. We consider a program *non-terminating* w.r.t. a moded query, if the denotation of the query contains at least one concrete query that has an infinite branch in its generalized SLDNF-tree.

3.1 The moded more general relation

To prove non-termination, we prove that a path between two nodes N_b and N_e in a moded SLDNF-derivation can be repeated infinitely. To find such a path, we check three properties. Because the rules in the path must be applicable independent of the values of the input variables, no substitutions on the input variables may occur in the path from N_b down to N_e. Because this path should be a loop, L_b^1 must be an ancestor of L_e^1. Finally, a special more general relation for moded atoms must hold between L_b^1 and L_e^1. We will show that these three conditions imply non-termination.

A moded atom A is *moded more general* than a moded atom B, if any atom in the denotation of A is more general than some atom in the denotation of B.

Definition 7. *A moded atom A is **moded more general** than a moded atom B w.r.t. program P, $A \rhd B$, iff:*

$$\forall I \in Den(A), \exists I' \in Den(B) : I \textit{ is more general than } I' \qquad \square$$

We illustrate this moded more general relation with some small examples.

Example 3. The following relations hold w.r.t. the *binary tree* program:

- $bin(X) \rhd bin(\underline{I})$
  The denotation of $bin(X)$ only contains the atom itself, which is more general than any atom in the denotation of $bin(\underline{I})$, e.g. $bin(empty)$.
- $bin(tree(tree(\underline{In}, Xn), Y)) \rhd bin(tree(\underline{I}, tree(X, empty)))$
  For example, if $\underline{In} = empty$, then $\underline{I} = tree(empty, empty)$ yields an atom in the denotation that satisfies the more general relation. □

Because the denotation of a moded atom is in general infinite, we cannot check this property for every atom in the denotation. However, there is a syntactic sufficient condition to check if the moded more general relation holds between two given moded atoms A and B. The condition is based on a particular form of unifiability of the atoms.

We introduce the following notations. Let $InVar_P$ be the set of input variables and Var_P the set of normal variables. To every $\underline{I} \in InVar_P$ we associate a fresh normal variable I. Let $Term_P^+$ denote the set of all terms constructible in the underlying language of P augmented with the variables $\{I \mid \underline{I} \in InVar_P\}$.

Proposition 1. *Let A and B be two moded atoms. Let A_1 and B_1 be renamings of these atoms such that they have no shared variables. Let A_2 and B_2 denote variants of A_1 and B_1 in which every input variable $\underline{I}$ is replaced by I. Let $N_1^a, \ldots, N_n^a$ be a subset of the normal variables in A_2 and $I_1^b, \ldots, I_m^b$ be the fresh variables associated to the input variables in B_2.*

If A_2 and B_2 are unifiable with a substitution $\gamma = \{N_1^a \setminus t_1, \ldots, N_n^a \setminus t_n, I_1^b \setminus t_1^+, \ldots, I_m^b \setminus t_m^+\}$ with $t_1, \ldots, t_n \in Term_P$ and $t_1^+, \ldots, t_m^+ \in Term_P^+$, then A is moded more general then B. □

Proof. Let $\alpha = \{N_1^a \setminus t_1, \ldots, N_n^a \setminus t_n\}$ and $\beta = \{I_1^b \setminus t_1^+, \ldots, I_m^b \setminus t_m^+\}$. Because $I_1^b, \ldots, I_m^b$ can not occur in $t_1, \ldots, t_n$, $\gamma = \beta \circ \alpha$, and by unifiability, $A_2\alpha\beta = B_2\alpha\beta$. Moreover, since B_2 does not contain $N_1^a, \ldots, N_n^a$, $B_2\alpha\beta = B_2\beta$, and since $A_2\alpha$ does not contain $I_1^b, \ldots, I_m^b$, $A_2\alpha\beta = A_2\alpha$. Thus, $A_2\alpha = B_2\beta$.

Let A_c be an element of $Den(A_1)$. Then, there exists a substitution $\psi = \{I_1^a \setminus s_1, \ldots, I_k^a \setminus s_k\}$, where $\underline{I_1^a}, \ldots, \underline{I_k^a}$ are all input variables of A_1, $s_1, \ldots, s_k \in Term_P$ and $s_1, \ldots, s_k$ are ground, such that $A_c = A_2\psi$.

Now consider the atom $B_c = B_2\beta\psi$. First, $B_c \in Den(B_1)$. This is because β replaces all I_j^b of B_2 by terms t_j^+. These terms t_j^+ may contain variables I_l^a of A_2, but these are all substituted to ordinary ground terms $s_l \in Term_P$ by ψ.

Finally, $A_c\alpha = A_2\psi\alpha = A_2\alpha\psi = B_2\beta\psi = B_c$. Note that $A_2\psi\alpha = A_2\alpha\psi$ because no s_i of ψ can contain a variable N_j^a of α, nor can any t_i of α contain a variable I_j^a of ψ. Thus A_c is more general than an element of $Den(B_1)$. □

Example 4. The moded atoms of the last example are already variable disjunct. To check if the moded more general relation holds, we have to check if the atoms are unifiable with a substitution of the correct forms.

- $bin(X) = bin(I)$ with substitution: $\{I \setminus X\}$
- $bin(tree(tree(In, Xn), Y)) = bin(tree(I, tree(X, empty)))$ with substitution: $\{I \setminus tree(I, Xn), Y \setminus tree(X, empty)\}$ □

3.2 Non-termination of moded more general loop

If a moded SLDNF-derivation contains a path without substitutions on input variables, such that the ancestor relation and the moded more general relation hold between the first and last selected literal in that path, we call this path a *moded more general loop*. We will show that a moded more general loop implies non-termination.

Definition 8. *In a moded SLDNF-derivation D, nodes $N_i : G_i$ and $N_j : G_j$ are a* ***moded more general loop****, $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$, iff:*

- *No substitutions on input variables occur in the path from N_i down to N_j.*
- $L_i^1 \prec_{anc} L_j^1$.
- $L_j^1 \rhd L_i^1$ □

Note that when no confusion can occur, we may omit writing the goal in the moded more general loop.

A moded more general loop, $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$, corresponds to an infinite loop for every concrete goal in the denotation of G_i.

Theorem 1 (Sufficiency of the moded more general loop). *Let $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$ be a moded more general loop in a moded SLDNF-derivation D of program P and moded query I. The sequence of clauses from N_i down to N_j, $\langle C_1, \ldots, C_n \rangle$, can be repeated infinitely often for any goal in $Den(G_i)$.* □

Proof. Because L_i^1 is an ancestor of L_j^1, only the selected literal of N_i influences if the sequence of clauses can be repeated infinitely often.

Because no substitutions on input variables occur in the path from N_i down to N_j, $\langle C_1, \ldots, C_n \rangle$ is applicable to any atom in $Den(L_i^1)$. Obviously, this path is also applicable to any atom A, which is more general than some atom B in $Den(L_i^1)$. Furthermore, after applying $\langle C_1, \ldots, C_n \rangle$ to A, the resulting selected literal is more general than the selected literal after applying $\langle C_1, \ldots, C_n \rangle$ to B.

As $L_j^1 \rhd L_i^1$, any atom in $Den(L_j^1)$ is more general than some atom in $Den(L_i^1)$.

Therefore, let S be the union of $Den(L_i^1)$ and all more general atoms. Then, $\langle C_1, \ldots, C_n \rangle$ is applicable to any atom of S, and after applying these clauses, the selected literal of the resulting goal is again an atom of S. Thus, this sequence of clauses is infinitely often applicable to elements of S. □

We will illustrate this non-termination condition with our running example.

Example 5 (Non-termination proof of binary tree). Let us revisit Example 1 with a query $bin(X)$. The SLDNF-tree constructed by LP-check for this program and query is almost the same as in Figure 1(a), the only difference is that the input variables are replaced by ordinary variables.

The constructed SLDNF-tree satisfies the conditions of Definition 8, so $N_0 \overset{mmg}{\rightarrow} N_2$ is a moded more general loop. Therefore, non-termination of this example is proven by Theorem 1. □

Observe that Theorem 1 can straightforwardly be generalized to conclude non-termination for any goal that is more general than an element of $Den(G_i)$.

3.3 Input-generalizations

Our experimental evaluation (see Section 4) shows that for many non-terminating programs, non-termination can be proven using the moded more general loop. But, the next example shows that there is room for further improvement.

Example 6 (Termination behavior of $flat$).

```
flat(niltree, nil).
flat(tree(X, niltree, XS), cons(X, YS)) :- flat(XS, YS).
flat(tree(X, tree(Y, YS1, YS2), XS), ZS) :-
        flat(tree(Y, YS1, tree(X, YS2, XS)), ZS).
```

This program, $flat$, flattens a binary tree into a list denoted with the *cons* notation. To flatten the tree, the program repeatedly moves one element from the left to the right subtree until the left subtree is empty. When the left subtree is empty, we proceed by processing the right subtree. If the first argument of the top level query is a variable, this program loops w.r.t. the thirth clause.

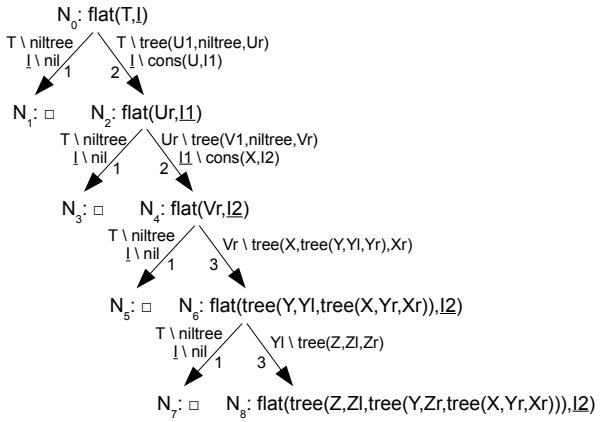


Fig. 2. Moded generalized SLDNF-tree with LP-check of $flat$

Figure 2 shows the moded generalized SLDNF-tree constructed for moded query $flat(T, \underline{I})$ using LP-check with repetition number 3. No nodes in the derivations satisfy Definition 8. The reason for this is that we replace a variable by a compound term when applying the thirth clause. □

To prove non-termination for programs such as $flat$, we define an *input-generalization*. This input-generalization is such that proving non-termination of an input-generalized goal implies non-termination of the original goal.

Definition 9. *Let A be a moded atom. We say that A^α is an input-generalization of A, if there exist terms $t_1, \ldots, t_n$ in A and fresh input variables $\underline{I_1}, \ldots, \underline{I_n}$ such that $A^\alpha = A(\underline{I_1} \rightarrow t_1, \ldots, \underline{I_n} \rightarrow t_n)$.* □

Example 7 (Input generalizations).
- $bin(tree(\underline{I}, \underline{I_1}))$ is an input-generalization of $bin(tree(\underline{I}, tree(X, empty)))$
- $bin(\underline{I_2})$ is an input-generalization of $bin(tree(\underline{I}, \underline{I_1}))$ □

To check if a path is non-terminating w.r.t. an input-generalized goal, we define an *input-generalized derivation*. This derivation is constructed by applying a path in a given derivation to the input-generalized selected literal of the first node in the path.

Definition 10. *Let D be a moded SLDNF-derivation $N_i, \ldots, N_j$, such that $L_i^1 \prec_{anc} L_j^1$. Let $\langle C_1, \ldots, C_n \rangle$ be the sequence of clauses applied from N_i down to N_j and let A be an input-generalization of L_i^1.*

*The **input-generalized derivation** D' for A, is constructed by applying the sequence of clauses $\langle C_1, \ldots, C_n \rangle$ to A. The **input-generalized nodes** N_i^α and N_j^α are the top and bottom nodes of D', respectively.* □

Next, we prove that non-termination of the input-generalized derivation implies non-termination of the original goal. First we introduce two lemmas.

Lemma 1. *Let A^α be an input generalization of A, then $A \rhd A^\alpha$.* □

Proof. Let $\underline{I_1}, \ldots, \underline{I_n}$ be the input variables of A and $\underline{I_{n+1}}, \ldots, \underline{I_m}$ be the new introduced input variables in A^α. For every concrete atom A_c in $Den(A)$, $\underline{I_1}, \ldots, \underline{I_n}$ are replaced by ground terms. To construct an atom A_c^α of $Den(A^\alpha)$, for which A_c is more general then A_c^α, one replaces $\underline{I_1}, \ldots, \underline{I_n}$ by the same values as in A_c and $\underline{I_{n+1}}, \ldots, \underline{I_m}$ by instances of the corresponding terms in A_c. □

Lemma 2. *Let A and B be atoms such that $A \rhd B$ and let every atom in $Den(B)$ be non-terminating w.r.t. a program P, then, every atom in $Den(A)$ is non-terminating w.r.t. P.* □

Proof. Every atom of $Den(A)$ is more general than a non-terminating atom. □

Corollary 1 (Non-termination with input-generalization). *Let $N_i : G_i$ and $N_j : G_j$ be nodes in a derivation D of program P for moded query I, such that $L_i^1 \prec_{anc} L_j^1$, and let N_i^α and N_j^α be the input-generalized nodes in the input-generalized derivation D' of N_i and N_j for A.*

If $N_i^\alpha \overset{mmg}{\rightarrow} N_j^\alpha$, then every concrete goal in the denotation of G_i is non-terminating w.r.t. program P. □

Proof. Follows from Theorem 1 and the two previous lemmas. □

We illustrate these input-generalizations by revisiting the $flat$ example.

Example 8 (Non-termination of $flat$). To prove non-termination, we generalize node N_6 to $flat(tree(Y, Yl, \underline{In}), \underline{I2})$, by changing the subterm $tree(X, Yr, Xr)$ to a new input variable $\underline{In}$.

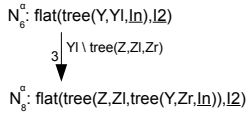


Fig. 3. Input-generalized SLDNF-derivation of $flat$

Figure 3, shows the input-generalized moded SLDNF-derivation for $flat(tree(Y, Yl, \underline{In}), \underline{I2})$. This derivation is a moded more general loop: $N_6^\alpha \overset{mmg}{\rightarrow} N_8^\alpha$. Therefore, non-termination of the program $flat$ w.r.t. the concrete goals in the denotation of the goal of N_6 is proven by Corollary 1. □

3.4 Constructing classes of non-terminating queries

As stated in the introduction, a non-termination analyzer can be a valuable tool for a programmer. In order to realize this, it is important that the analyzer constructs classes of non-terminating queries, so that the programmer can correct the source of non-termination by tracing the computation.

Theorem 1 and Corollary 1 prove that we are able to derive classes of non-terminating concrete goals w.r.t. some program. However, we wanted to derive classes of non-terminating concrete queries corresponding to a given top-level moded query.

Note that a derivation in a moded SLDNF-tree is applicable to some concrete queries in the denotation of the moded query. In order for a branch to be applicable to a specific concrete query, the actual values of the input variables need to be such that the substitutions on the input variables in the path from the root to the first node of the moded more general loop succeed. By applying all substitutions in the derivation, we derive classes of top level queries, for which all goals in the denotation are non-terminating.

We illustrate this with an example.

Example 9 (Flatten tree continued). Figure 2 shows the constructed derivation for $flat$ with the query $flat(T, \underline{I})$.

In Example 8, we proved that every goal in the denotation of $flat(tree(Y, Yl, tree(X, Yr, Xr)), \underline{I2})$ at node N_6, is non-terminating w.r.t. the program. However, not every element of the denotation of the query can follow the path from the root N_0 down to N_6. The following substitution on input variables occur between nodes N_0 and N_6:

- $\underline{I} \setminus cons(U, I1)$
- $\underline{I1} \setminus cons(X, I2)$

When we apply these substitutions to the query, we get a new query: $flat(T, cons(\underline{U}, cons(\underline{X}, \underline{I2})))$. Every element of its denotation can reach node N_6 and is therefore non-terminating w.r.t. $flat$. □

4 Experimental evaluation

To evaluate our approach to non-termination analysis, we implemented a non-termination analyzer $P2P$, $from\ Prediction\ to\ Proof$, based on Corollary 1. We tested $P2P$ on a benchmark of 48 non-terminating pure logic programs. First, we describe our non-termination analyzer and the benchmark. Then, we compare our tool with non-termination inference tool NTI [10].

4.1 *P2P: from Prediction to Proof*

We implemented $P2P$ in SWI-prolog[2]. $P2P$ is freely available[3] and consists of two components. The first component predicts the termination behavior of a

[2] Homepage of SWI-prolog: www.swi-prolog.org/
[3] Available at www.cs.kuleuven.be/~dean/p2p.html

logic program. The second component checks if the moded SLDNF-tree contains a moded more general loop or if an input-generalized derivation contains a moded more general loop.

To predict the termination behavior of a logic program, we use $TPoLP$, the implementation of the termination prediction technique. $TPoLP$ is freely available[4]. This tool offers the choice between LP-check and LP-check with pruning.

If non-termination is predicted by $TPoLP$, the second component tries to prove non-termination in the path between the pairs of nodes of the LP-cut for which non-termination is predicted.

4.2 Benchmark of Termination Problems

Our benchmark consists of the non-terminating pure logic programs from the termination competition of 2007. The benchmark and the results from the tools that participated in the competition are available[5]. The benchmark of the termination competition contains around 300 logic programs and moded queries representing different challenges in termination and non-termination analysis. A few programs from the competition are omitted because they contain non-logical operations such as arithmetics. The competition benchmark contains some doubles. These doubles are also omitted. Our benchmark contains 48 non-terminating programs. All programs contain between 2 and 15 clauses, except for binary4, which contains 41 clauses. The only other non-termination analyzer, NTI [10], proves non-termination for about 95% of the benchmark programs.

Table 1 shows our experimental evaluation on this benchmark using LP-check with pruning, with 4 as a repetition number. The result of our tool is given in the column $P2P$, V denotes that non-termination is proven while X denotes that no non-termination proof was found. The result of NTI is given in the column NTI. The columns *Size* and *Time* show the size in the number of nodes of the SLDNF-tree and the analysis time in seconds, respectively.

The results are very satisfactory. For all programs in the benchmark, non-termination is proven and a class of non-terminating queries can be constructed. The analyzer is very efficient. Any benchmark program is analyzed in less as a second and the memory use never exceeds a few megabytes.

As stated, these experiments have been performed using 4 as a repetition number. When we use 3 as a repetition number, our tool fails to prove non-termination of programs $pl7.6.2.a$ and $pl7.6.2.b$. When using 2 as repetition number, proving non-termination fails for about 25% of the benchmark programs.

4.3 Comparison with *NTI*

NTI To infer non-terminating queries, NTI first transforms a given program into a binary program using *binary unfoldings* and then compares the head and

[4] Available at www.cs.kuleuven.be/~dean/termination_prediction.html
[5] Available at www.lri.fr/~marche/termination-competition/

Name program	$P2P$	Size	Time	NTI	Name program	$P2P$	Size	Time	NTI
ackermann-ioi	V	9	0.33	V	permutation-fb	V	22	0.26	V
bad sublist	V	33	0.29	V	pl1.1	V	8	0.25	V
binary4	V	12	0.27	V	pl3.1.1	V	12	0.30	V
delete-bff	V	13	0.31	V	pl3.5.6	V	13	0.31	V
der-fb	V	22	0.29	V	pl4.0.1-oooi	V	33	0.27	V
doublehalfpred	V	38	0.28	V	pl4.5.2	V	481	0.36	V
example4-2	V	4	0.23	V	pl4.5.3a	V	10	0.29	V
flatlength-fbf	V	14	0.23	V	pl4.5.3b	V	10	0.24	V
flatlength-ffb	V	19	0.23	V	pl4.5.3c	V	11	0.27	V
flat-oi	V	9	0.26	X	pl5.2.2	V	59	0.27	V
frontier-fb	V	12	0.27	V	pl7.6.2.a	V	39	0.27	X
ifdiv	V	19	0.29	V	pl7.6.2.b	V	45	0.33	X
in-bf	V	18	0.29	V	quicksort-fb	V	72	0.26	V
inorder-fb	V	4	0.27	V	quicksort-oi	V	74	0.26	V
insert-bff	V	22	0.29	V	reverse-fb	V	9	0.32	V
log2a-oi	V	35	0.25	V	select-bff	V	8	0.32	V
log2b-oi	V	29	0.28	V	slowsort-fb	V	123	0.27	V
mapcolor	V	23	0.31	V	slowsort-oi	V	26	0.26	V
member-bf	V	8	0.27	V	sublist-bf	V	30	0.21	V
mergesort	V	171	0.28	V	subset-bf	V	21	0.23	V
mergesort-oi	V	54	0.28	V	subset-fb	V	14	0.26	V
mergesort_variant	V	15	0.23	V	suffix-bf	V	9	0.25	V
minimum-fb	V	8	0.29	V	transpose2	V	6	0.28	V
naive reverse-fb	V	8	0.37	V	tree_member-bf	V	12	0.28	V

Table 1. Benchmark of non-terminating pure logic programs.

body of the clauses in the binary program with a special more general relation. If this special more general relation holds, a non-terminating query is inferred.

The binary unfolding of a program represent the calls made during program execution. Thus, it corresponds to comparing the selected literals in our symbolic computation. The binary unfolding of a program can be computed using a fix-point operator. Both the binary unfolding of a program and the selected literals in the moded SLDNF-resolution give a finite description of the calls made in the program. In the remainder of this comparison we will consider both approaches to be equally strong.

If the binary unfolding of a program contains a clause for which the body is more general as the head, non-termination is proven. For example, the clause $a(s(X)) : -a(X)$, is an infinite loop for every atom that is more general than $a(s(X))$. The more general relation is extended with *derivation neutral filters* on some arguments of a predicate. We will explain this using an example. For a clause $a(s(X)) : -a(s(s(X)))$, the more general than relation is not satisfied. However, this clause loops if the first argument is an instance of $s(X)$. Such a condition is called a derivation neutral filter. A binary clause is looping if each argument is either replaced by a more general one or satisfies some derivation neutral filter. NTI only considers *variable independent filters*, filters that are not dependant on the names of the variables. The arguments at positions where a filter is used, are arguments that become more specific and thus correspond to positions with input variables. Therefore, if NTI infers non-termination using filters on some argument positions, the moded query with input variables on those positions, is non-terminating.

Comparison As stated, moded SLDNF-resolution and binary unfoldings are comparable in use and strength. But, we argue without proof that the moded more general relation is more precise on identifying infinite loops as the more general relation with filters, by giving two classes of programs where NTI fails to prove non-termination and our approach may succeed.

The first class of programs where NTI fails is due to the condition that filters have to be variable independent. Variable independent filters fail to express that two argument positions must contain the same element. An example of such a clause is $a(X, X) : -a(s(X), s(X))$. The second class of programs are programs where, in one argument position, a subterm is replaced by a more general subterm while another one is replaced by a more specific subterm. For example the argument in $a(f(X, s(Y))) : -a(f(s(X), Y))$. Such an argument position neither becomes more general nor can it satisfy a derivation neutral filter. Because in our approach the moded more general relation is defined on atoms rather than arguments, our approach does not have this restriction.

Benchmark results Table 1 shows that NTI fails to prove non-termination of 3 programs. The results on the termination competition of 2007 were worse for NTI, as we have rewritten some programs that NTI could not parse.

The program $flat-oi$ cannot be proven to be non-terminating using NTI because every non-terminating query contains both an input and ordinary variable in the first argument position. The other two programs for which non-termination cannot be proven with NTI fail because every non-terminating query contains different argument positions with a shared input variable.

5 Conclusion

We introduced a new approach to non-termination analysis of logic programs that proves non-termination based on a finite, symbolic derivation tree of a logic program and a moded query. These derivation trees can be constructed using moded SLDNF-resolution and *LP-check* as defined in [15]. We have defined a condition proving non-termination given such a finite moded SLDNF-tree.

This approach has been implemented in a non-termination analyzer $P2P$. We evaluated $P2P$ on a benchmark of 48 non-terminating logic programs. The results are very satisfactory. Our tool is able to prove non-termination and construct non-terminating queries for all benchmark programs.

We compared our tool with NTI [10], the only other non-termination analyzer for logic programs. We have shown that our approach improves on the results of NTI and that we are able to prove non-termination of new classes of programs.

References

1. R. N. Bol. *Loop checking in logic programming.* CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1995.

2. M. Bruynooghe, M. Codish, J. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis through combination of type based norms. *ACM Transactions on Programming Languages and Systems*, 29(2):10, 2007.
3. Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.
4. Michael Codish. Proving termination with (boolean) satisfaction. In A. King, editor, *LOPSTR 2007*, volume 4915 of *LNCS*, pages 1–7, 2008.
5. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 21(6):1137–1195, 1999.
6. John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
7. Frédéric Mesnard and Roberto Bagnara. Cti: A constraint-based termination inference tool for iso-prolog. *TPLP*, 5(1-2):243–257, 2005.
8. M. T. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination analysis of logic programs based on dependency graphs. pages 8–22, 2008.
9. Manh Thang Nguyen and Danny De Schreye. Polytool: Proving termination automatically based on polynomial interpretations. In Germán Puebla, editor, *LOPSTR*, volume 4407 of *LNCS*, pages 210–218. Springer, 2006.
10. Étienne Payet and Frédéric Mesnard. Nontermination inference of logic programs. *ACM Transactions on Programming Languages and Systems*, 28(2):256–289, 2006.
11. I. V. Ramakrishnan, P. Rao, K. F. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *J. Log. Program.*, 38(1):31–54, 1999.
12. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination analysis for logic programs by term rewriting. In *LOPSTR '06*, volume 4407 of *Lecture Notes in Computer Science*, pages 177–193, Heidelberg, 2007. Springer.
13. Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *J. Log. Program.*, 19/20:199–260, 1994.
14. Danny De Schreye and Alexander Serebrenik. Acceptability with general orderings. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2407 of *LNCS*, pages 187–210. Springer, 2002.
15. Yi-Dong Shen, Danny De Schreye, and Dean Voets. Termination prediction for general logic programs: Submitted. Report CW 536, 2008.
16. Yi-Dong Shen, Jia-Huai You, Li-Yan Yuan, Samuel S. P. Shen, and Qiang Yang. A dynamic approach to characterizing termination of general logic programs. *ACM Trans. Comput. Log.*, 4(4):417–430, 2003.