*TLP* **11** (4–5): 521–536, 2011. © Cambridge University Press 2011
doi:10.1017/S1471068411000159


# *Non-termination analysis of logic programs with integer arithmetics*

DEAN VOETS* and DANNY DE SCHREYE
*Department of Computer Science, K.U. Leuven, Belgium*
*Celestijnenlaan 200A, 3001 Heverlee, Belgium*
(*e-mail:* {Dean.Voets, Danny.DeSchreye}@cs.kuleuven.be)

**Abstract**

In the past years, analyzers have been introduced to detect classes of non-terminating queries for definite logic programs. Although these non-termination analyzers have shown to be rather precise, their applicability on real-life Prolog programs is limited because most Prolog programs use non-logical features. As a first step towards the analysis of Prolog programs, this paper presents a non-termination condition for Logic Programs containing integer arithmetics. The analyzer is based on our non-termination analyzer presented at International Logic Programming Conference (ICLP) 2009. The analysis starts from a class of queries and infers a subclass of non-terminating ones. In first phase, we ignore the outcome (success or failure) of the arithmetic operations, assuming success of all arithmetic calls. In second phase, we characterize successful arithmetic calls as a constraint problem, the solution of which determines the non-terminating queries.

*KEYWORDS*: non-termination analysis, numerical computation, constraint-based approach

## 1 Introduction

The problem of proving termination has been studied extensively in Logic Programming. Since the early works on termination analysis in Logic Programming, see, or example, (De Schreye and Decorte 1994), there has been a continuing interest from the community on this topic. Lots of in-language and transformational tools have been developed, for instance, (Giesl *et al.* 2006) and Nguyen *et al.* (to appear), and since 2004, there has been an annual Termination Competition[1] to compare the current analyzers on the basis of an extensive database of logic programs.

In contrast with termination analysis, the dual problem to detect non-terminating classes of queries is a fairly new topic. The development of first and the most well-known non-termination analyzer, *NTI* (Payet and Mesnard 2006), was motivated by difficulties in obtaining precision results for termination analyzers. As the halting problem is undecidable, one way of demonstrating the precision of a termination

* Supported by the Fund for Scientific Research - FWO-project G0561-08.
[1] Results are available at http://termcomp.uibk.ac.at/

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

analyzer is with a non-termination analyzer. For $NTI$ it had been already shown that for many examples one can partition queries in terminating and non-terminating. $NTI$ compares the consecutive calls in program using binary unfoldings and proves non-termination by comparing the head and body of these binary clauses with a special more general relation.

Recently, in a joint work with Shen (Shen *et al.* 2009), we integrated loop checking into termination analysis, yielding a very accurate technique to predict the termination behavior for classes of queries described using modes. Classes of queries are represented as *moded queries*. A moded query consists of a query and a label, input or output, for each variable in the query. These moded queries are then evaluated with a *moded Selective Linear Definite* (SLD)-*tree* obtained by applying clauses to the partially instantiated query and propagating the labels. In order to guarantee a finite analysis, this moded SLD-tree is constructed using a complete loop check. After evaluating the moded query, the analysis predicts the termination behavior of the program for the considered queries based on the labels and substitutions in the moded SLD-tree.

Motivated by the elegance of this approach and the accuracy of the predictions, our research focused on defining a non-termination condition based on these moded queries. In Voets and De Schreye (2009), we introduced a non-termination condition identifying paths in a moded SLD-tree that can be often repeated infinitely. This approach was implemented in a system called $P2P$, which proved more accurate than $NTI$ on the benchmark of the termination competition. An evaluation of the classes of queries not handled by current approaches lead to considerable improvements in our non-termination analysis. These improvements have been presented by Voets and De Schreye (to appear) and implemented in $pTNT$ analyzer.

Both termination and non-termination analyzers have been rather successful in analyzing the termination behavior of definite logic programs, but only a few termination analyzers, e.g., Serebrenik and De Schreye (2001), and none of the non-termination analyzers handle nonlogical features such as arithmetics or cuts, typically used in practical Prolog programs. In this paper, we introduce a technique for proving non-termination of logic programs containing a subset of the built-in predicates for integer arithmetic, commonly found in Prolog implementations.

Given a program, containing integer arithmetics, and a class of queries, described using modes, we infer a subset of these queries for which we prove existential non-termination (i.e., the derivation tree for these queries contains an infinite path). The inference and proof are done in two phases. In the first phase, non-termination of the logic part of the program is proven by assuming that all comparisons between integer expressions succeed. We will show that only a minor adaption of our technique presented by Voets and De Schreye (2009) is needed to achieve this. In the second phase, given the moded query, integer arguments are identified and constraints over these arguments are formulated such that solutions for these constraints correspond to non-terminating queries.

The paper is structured as follows. In Section 2, we introduce some preliminaries concerning logic programs and integer arithmetics and present the symbolic derivation trees used to abstract the computation. In Section 3, we introduce

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

our non-termination condition for programs containing integer arithmetics. In Section 4, we describe our prototype analyzer and some results. Finally, we conclude in Section 5.

## 2 Preliminaries

### 2.1 Logic programming

We assume the reader is familiar with standard terminology of logic programs, in particular with SLD resolution as described by Lloyd (1987). Variables are denoted by strings beginning with a capital letter. Predicates, functions, and constant symbols are denoted by strings beginning with a lower case letter. We denote the set of terms constructible from a program $P$ by $Term_P$. Two atoms are called *variants* if they are equal up to variable renaming. Atom $A$ is *more general* than atom $B$, and $B$ is an *instance* of $A$ if there exists a substitution $\theta$ such that $A\theta = B$.

We restrict our attention to definite logic programs. A logic program $P$ is a finite set of clauses of the form $H \leftarrow A_1, \ldots, A_n$, where $H$ and each $A_i$ are atoms. A goal $G_i$ is a headless clause $\leftarrow A_1, \ldots, A_n$. A top goal is also called a query. Without loss of generality, we assume that a query contains only one atom.

Let $P$ be a logic program and $G_0$ be a goal. $G_0$ is evaluated by building a *generalized SLD-tree* as defined in Shen *et al.* (2009), 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. Throughout this paper, we choose to use the best-known *depth-first, left-most* control strategy, as is used in Prolog, to select goals and atoms. So by the *selected atom* in each node $N_i :\leftarrow A_1, \ldots, A_n$, we refer to the left-most atom $A_1$. For any node $N_i : G_i$, we use $A_i^1$ to refer to the selected atom in $G_i$. Let $A_i^1$ and $A_j^1$ be the selected atoms at two nodes $N_i$ and $N_j$, respectively. $A_i^1$ is an *ancestor* of $A_j^1$ if the proof of $A_i^1$ goes through the proof of $A_j^1$.

A derivation step is denoted by $N_i : G_i \Longrightarrow_C N_{i+1} : G_{i+1}$, meaning that applying a clause $C$ to $G_i$ produces $N_{i+1} : G_{i+1}$. Any path of such derivation steps starting at the root node $N_0 : G_0$ is called a *generalized SLD derivation.*

### 2.2 Integer arithmetics

Prolog implementations contain special purpose predicates for handling integer arithmetics. Examples are $is/2, \geqslant /2, =:= /2, \ldots$

*Definition 1*
An expression $Expr$ is an *integer expression* if it can be constructed by the following recursive definition.

$$Expr = z \in \mathbb{Z} \mid -Expr \mid Expr + Expr \mid Expr - Expr \mid Expr * Expr. \qquad \square$$

An atom "V is Expr", with $V$ a free variable and $Expr$ an integer expression, is called an *integer constructor*. An atom $Expr1 \circ Expr2$ is called an *integer condition* if $Expr1$ and $Expr2$ are integer expressions and $\circ \in \{>, >=, =<, <, =:=, =/=\}$.

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

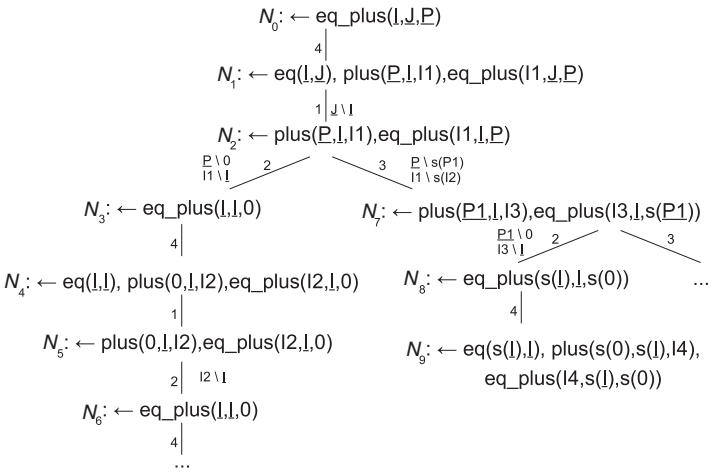



Fig. 1. Moded SLD-tree *eq_plus*.

### 2.3 Moded SLD-trees and loop checking

In Voets and De Schreye (2009), classes of queries are represented as *moded queries*. Moded queries are partially instantiated queries, in which variables can be labeled as *input*. Variables labeled as input are called *input variables* and represent arbitrary ground terms. In order to indicate that a variable is labeled as input, the name of the variable is underlined. A query in which no variable is labeled as input is called a *concrete query*. A set of concrete queries represented by a moded query $Q$ is called the *denotation* of $Q$.

*Definition 2*
Let $Q$ be a query and $\{\underline{I_1}, \ldots, \underline{I_n}\}$ its set of input variables. The *denotation* of $Q$, $Den(Q)$, is defined as

$$Den(Q) = \{Q\{\underline{I_1} \setminus t_1, \ldots, \underline{I_n} \setminus t_n\} \mid t_i \in Term_P, t_i \text{ is ground}, 1 \leqslant i \leqslant n\}. \quad \square$$

Note that the denotation of a concrete query is a singleton containing the query itself. Denotations of moded goals and atoms are defined similarly.

A moded query $\leftarrow Q$ is evaluated by constructing a moded SLD-tree, representing the derivations of the queries in $Den(\leftarrow Q)$. This moded SLD-tree is constructed by applying SLD resolution to the query and propagating the labels. An input variable $\underline{I}$ can be unified with any term $t \in Term_P$. After unifying $\underline{I}$ and $t$, all variables of $t$ will also be considered input.

*Example 1*
Figure 1 shows the moded SLD-tree of the program *eq_plus* for the moded query $\leftarrow eq_plus(\underline{I}, \underline{J}, \underline{P})$. This program is non-terminating for any query in $Den(\leftarrow eq_plus(\underline{I}, \underline{I}, 0))$ and fails for all other queries in $Den(\leftarrow eq_plus(\underline{I}, \underline{J}, \underline{P}))$. A query fails if its derivation tree is finite, with no path ending with an empty goal:

```
eq_plus(I,J,P):- eq(I,J), plus(P,I,In), eq_plus(In,J,P).
eq(A,A).     plus(0,B,B).     plus(s(A),B,s(C)):- plus(A,B,C).
```

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

Substitutions on input variables express conditions for the clause to be applicable. The edge from nodes $N_2$ to $N_3$ shows that clause two is applicable if the concrete term denoted by $\underline{P}$ can be unified with 0. The substitution, $I1 \setminus \underline{I}$, shows that applying this clause unifies $I1$ with the term corresponding to $\underline{I}$.

Every derivation in a moded SLD-tree for a query $\leftarrow Q$ corresponds to a concrete derivation for a subclass of $Den(\leftarrow Q)$. The subclass of queries for which a derivation to node $N_i$ is applicable is obtained by applying all substitutions on input variables from nodes $N_0$ to $N_i$. Our condition presented by Voets and De Schreye (2009) proves non-termination for every query for which the derivation to node $N_3$ is applicable. The substitutions on input variables in the derivation to node $N_3$ are $\underline{J} \setminus \underline{I}$ and $\underline{P} \setminus 0$. Applying these to the query proves non-termination for the queries in $Den(\leftarrow eq_plus(\underline{I}, \underline{I}, 0))$. □

As in the example, moded SLD-trees are usually infinite. In order to obtain a finite analysis, a complete loop check is applied during the construction of the tree. As in our previous works (for instance in Voets and De Schreye (2009) and Voets and De Schreye (to appear)), we use complete loop check *LP-check* (Shen *et al.* 2009). Without proof, we state that this loop check can also be used for moded SLD-trees and refer to Shen *et al.* (2003) for more information.

*Example 2*
In Figure 1, LP-check cuts clause 4 at node $N_6$ and clause 3 at node $N_7$. □

Combined with the loop check, a moded SLD-tree can be considered a light-weight alternative to an abstract interpretation of mode analysis.

## 3 Non-termination analysis for programs with integer arithmetics

In this section, we introduce a non-termination condition for programs containing integer arithmetics. In order to abstract computations for the considered queries, the moded SLD-tree of Shen *et al.* (2009) is used, with some modifications to handle integer constructors and integer conditions. LP-check ensures finiteness of the tree and detects paths that may correspond to infinite loops. For every such path, two analyses are combined to identify classes of non-terminating queries.

In the first phase, an adaption of the non-termination condition presented by Voets and De Schreye (2009) detects a class of queries such that each query is non-terminating or fails due to the evaluation of an integer condition such as $> /2$. This class of queries is a moded query with an additional integer label for variables representing unknown integers. In the second phase, the class of queries is restricted to a class of non-terminating queries by formulating additional constraints on the integer variables of the moded query. In order to prove that this class of non-terminating queries is not empty, these constraints over unknown integers are transformed to constraints over the natural numbers and solved by applying well-known techniques from termination analysis. Then we try to solve these constraints by transforming them to constraints over natural numbers and applying well-known techniques on them.

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

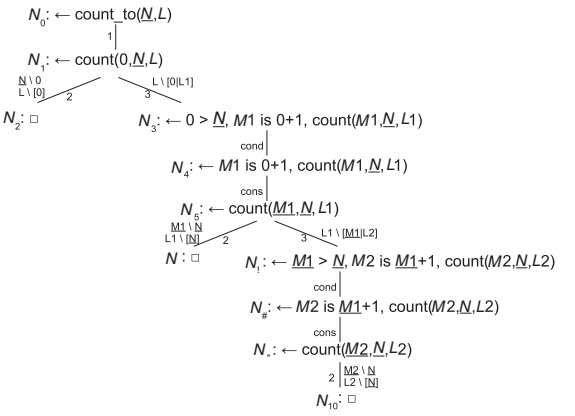



Fig. 2. Moded SLD-tree *count_to*.

### 3.1 Moded SLD-tree for programs with integer arithmetics

The first step of the extension is rather straightforward. The extensions to the moded SLD-tree of Shen *et al.* (2009) are limited to the introduction of the label *integer variable* and additional transitions to handle integer constructors and integer conditions. Integer variables are also input variables and will also be represented by underlining the name of the variable. An integer constructor, i.e., $is/2$, is applicable if the first argument is a free variable and the second argument is an integer expression. The application of an integer constructor labels the free variable as an integer variable. An integer condition, e.g., $\geqslant /2$, is applicable if both arguments are integer expressions. As integer variables denote unknown integers, integer expressions are allowed to contain integer variables. Applications of integer constructors and integer conditions in the moded SLD-tree are denoted by derivation steps $N_i : G_i \Longrightarrow_{cons} N_{i+1} : G_{i+1}$ and $N_i : G_i \Longrightarrow_{cond} N_{i+1} : G_{i+1}$, respectively.

*Example 3*

The following program, *count_to*, is a faulty implementation of a predicate generating the list starting from 0 up to a given number. The considered class of queries is represented by the moded query $\leftarrow count_to(\underline{N}, L)$ with $\underline{N}$ being an integer variable.

```
count_to(N,L):- count(0,N,L).        count(N,N,[N]).
count(M,N,[M|L]):- M > N, M1 is M+1, count(M1,N,L).
```

In the last clause, the integer condition should be `M < N` instead of `M > N`. Due to this error, the program

- fails for the queries for which $\underline{N} > 0$ holds,
- succeeds for $\leftarrow count_to(0, L)$, and
- loops for the queries for which $\underline{N} < 0$ holds.

Figure 2 shows the moded SLD-tree for the considered query, constructed using LP-check, which cuts clause 3 at node $N_9$. □

Note that by ignoring the possible values for integer variables when constructing a tree, some derivations in it may not be applicable to any considered query. For

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

example, the refutations at nodes $N_6$ and $N_{10}$ in the previous example cannot be reached by the considered queries.

### 3.2 Adapting the non-termination condition

In Voets and De Schreye (2009), programs are shown to be non-terminating for a moded query by proving that a path in the moded SLD-tree can often be repeated infinitely. Such a path, from nodes $N_b$ to $N_e$, is identified based on three properties. The path should be applicable, independent from the concrete terms represented by the input variables. Therefore, the first property states that no substitutions on input variables may occur between nodes $N_b$ and $N_e$. The second property states that the selected atom of $N_b$ – i.e., $A_b^1$ – has to be an ancestor of $A_e^1$. These two properties prove that the sequence of clauses in the path from nodes $N_b$ to $N_e$ is applicable to any goal with a selected atom from $Den(A_b^1)$. Therefore, non-termination is proven by requiring that $Den(A_e^1)$ is a subset of $Den(A_b^1)$. This property can be relaxed by requiring that each atom in $Den(A_e^1)$ is more general than some atom in $Den(A_b^1)$. If this is the case, $A_e^1$ is called *moded more general* than $A_b^1$. For definite logic programs, these three properties imply non-termination.

*Definition 3*
Let $A$ and $B$ be moded atoms. $A$ is *moded more general* than $B$ if

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

*Example 4*
In Figure 1, the path from nodes $N_3$ to $N_6$ satisfies these properties. The ancestor relation holds. There are no substitutions on input variables in the path. Finally, the selected atoms are identical and therefore denote the same concrete atoms. $\square$

The following proposition provides a practical sufficient condition to verify whether the moded more general relation holds.

*Proposition 1* (*Proposition 1 of Voets and De Schreye (2009)*)
Let $A$ and $B$ be moded atoms. Let $A_1$ and $B_1$ be renamings of these atoms such that they do not share variables. $A$ is moded more general than $B$ if $A_1$ and $B_1$ are unifiable with most general unifier $\{V_1 \setminus t_1, \ldots, V_n \setminus t_n\}$, $t_i \in Term_P$, $1 \leqslant i, \leqslant n$ such that for each binding $V_i \setminus t_i$, either

- $V_i \in Var(B_1)$ and $V_i$ is labeled as input, or
- $V_i \in Var(A_1)$, $V_i$ is not labeled as input and no variable of $Var(t_i)$ is labeled as input. $\square$

As stated, we want to prove that every query in the denotation of the considered moded query is either non-terminating or terminates due to the evaluation of an integer condition. To achieve this, we need to guarantee that integer constructors are repeatedly evaluated with a free variable and an integer expression as arguments and that integer conditions are repeatedly evaluated with integer expressions as arguments. Proposition 1 already implies that the first argument of all integer constructors are free variables in the subsequent iterations of the loop.

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

In order to prove the repeated behavior on integer constructors and integer expressions stated above, the *integer-similar to* relation is defined. Intuitively, given some loop in the computation, if an atom at the end of the loop is integer-similar to an atom at the start of the loop, then it will provide the required integer expressions to the first atom. First, we introduce positions to identify subterms and a function to obtain a subterm from a given position.

*Definition 4*
Let $L$ be a list of natural numbers, called a *position*, and $A$ a moded atom or term. The function *subterm(L,A)* returns the subterm:

- if $L = [I]$ and $A = f(A_1, \ldots, A_I, A_{I+1}, \ldots, A_n)$, then $subterm(L, A) = A_I$; or
- if $L = [I|T]$ and $A = f(A_1, \ldots, A_I, A_{I+1}, \ldots, A_n)$, then $subterm(L, A) = subterm(T, A_I)$. □

An atom $A$ is integer-similar to an atom $B$ if it has integer expressions on all positions corresponding to integer expressions in $B$.

*Definition 5*
Let $A$ and $B$ be moded atoms. $A$ is integer-similar to $B$ if for every integer expression $t_B$ of $B$, with $subterm(L, B) = t_B$, there exists an integer expression $t_A$ of $A$, with $subterm(L, A) = t_A$. □

*Example 5*
- $count(0, \underline{N}, L)$ is integer-similar to $count(\underline{M}, \underline{N}, L)$.
- $count(\underline{M}, \underline{N}, L)$ is integer-similar to $count(0, \underline{N}, L)$.
- $count(\underline{M} + 1, \underline{N}, L)$ is integer-similar to $count(\underline{M}, \underline{N}, L)$.
- $count(\underline{M}, \underline{N}, L)$ is not integer-similar to $count(\underline{M} + 1, \underline{N}, L)$.

Note that the last one is a counter-example because $count(\underline{M}+1, \underline{N}, L)$ has integer expressions on $[1, 1]$ and $[1, 2]$, while $count(\underline{M}, \underline{N}, L)$ does not have any subterms on these positions. □

*Theorem 1*
Let $N_b$ and $N_e$ be nodes in a moded SLD-tree for a moded query $Q$. Let $Q'$ be the moded atom obtained by applying to $Q$ all substitutions on input variables from nodes $N_0$ to $N_b$. Every query in $Den(Q')$ is either non-terminating or terminates due to the evaluation of an integer condition if the following properties hold:

- $A_b^1$ is an ancestor of $A_e^1$.
- No substitutions on input variables occur from $N_b$ to $N_e$.
- $A_e^1$ is moded more general than $A_b^1$.
- $A_e^1$ is integer-similar to $A_b^1$. □

*Example 6*
The path between nodes $N_5$ and $N_9$ in Figure 2 satisfies the conditions of Theorem 1. There are no substitutions on input variables from nodes $N_0$ to $N_5$ and thus, every query in $Den(\leftarrow count_to(\underline{N}, L))$ is either non-terminating or fails due to the evaluation of an integer condition. Note that although $\leftarrow count_to(0, L)$ has a succeeding derivation to node $N_2$, its derivation to node $N_9$ fails due to the integer condition $0 > \underline{N}$. □

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

In order to verify the last property automatically, we strengthen Proposition 1 to imply both the moded more general relation and the integer-similar to relation.

*Proposition 2*
Let $A$ and $B$ be moded atoms. Let $A_1$ and $B_1$ be renamings of these atoms such that they do not share variables. $A$ is moded more general than $B$ and $A$ is integer-similar to $B$ if $A_1$ and $B_1$ are unifiable with most general unifier $\{V_1 \setminus t_1, \ldots, V_n \setminus t_n\}$ such that for each binding $V_i \setminus t_i$, $1 \leqslant i \leqslant n$, either

- $V_i \in Var(B_1)$ and $V_i$ is labeled as integer and $t_i$ is an integer expression, or
- $V_i \in Var(B_1)$ and $V_i$ is labeled as input but not as integer variable, or
- $V_i \in Var(A_1)$, $V_i$ is not labeled as input, no variable of $Var(t_i)$ is labeled as input and $t_i$ does not contain integers. □

*Example 7*
As the selected atoms of nodes $N_5$ and $N_9$ in Figure 2 are variants, Proposition 2 holds. □

### *3.3 Generating the constraints on the integers of the query*

In this subsection, we introduce the constraints on the integer variables of the moded query, identifying values for which all integer conditions in the considered derivations succeed. These constraints consist of reachability constraints, identifying queries for which the derivation up to the last node is applicable, and an implication proving that the integer conditions will also succeed in the following iterations.

*Example 8*
As a first example, we introduce the constraints for the path between nodes $N_5$ and $N_9$ in the moded SLD-tree of *count_to* in Figure 2. For this path, Theorem 1 holds and thus every query denoted by $\leftarrow count_to(\underline{N}, L)$ is either non-terminating or terminates due to an integer condition.

In order to restrict the class of considered queries to those for which the derivation to node $N_9$ is applicable, all integer conditions in the derivation are expressed in terms of the integers of the query, yielding $0 > \underline{N}$ and $0 + 1 > \underline{N}$.

For this program and considered class of queries, the condition $0 > \underline{N}$ implies that the derivation is applicable up to node $N_9$. The following implication states that if the condition of node $N_7$ holds for any two values $M$ and $N$, then it also holds for the values of the next iteration.

$$\forall M, N \in \mathbb{Z} : M > N \Longrightarrow M + 1 > N$$

This implication is correct and thus proves non-termination for the considered queries if the precondition holds in the first iteration. This is the case for all queries in $Den(\leftarrow count_to(\underline{N}, L))$ with $0 > \underline{N}$, because the value corresponding to $M$ in the first iteration is 0 and the value corresponding to $N$ is $\underline{N}$. This proves non-termination of all considered queries for which $0 > \underline{N}$. □

In the following example, applicability of the derivation does not imply non-termination. To detect a class of non-terminating queries, a domain constraint is added to pre- and post-conditions of the implication.

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

*Example 9*

```
constants(I,J):- I =:= 2, In is J*2, Jn is I-J, constants(In,Jn).
```

The clause in *constants* is applicable to any goal with $constants(2, \underline{J})$ as a selected atom, with $\underline{J}$ an integer variable. As the first argument in the next iteration is the value corresponding to $\underline{J} * 2$, only goals with the selected atom $constants(2, 1)$ are non-terminating for this program.

As applicability of the derivation does not imply non-termination, a similar implication as in the previous example is false, $\forall I, J \in \mathbb{Z} : I = 2 \Longrightarrow J * 2 = 2$. To overcome this, a constraint is added to pre- and post-conditions of this implication, restricting the considered values of $\underline{J}$ to an unknown set of integers, called its *domain*,

$$\exists Dom_j \subset \mathbb{Z}, \forall I, J \in \mathbb{Z} : I = 2, J \in Dom_j \Longrightarrow J * 2 = 2, I - J \in Dom_j.$$

The resulting implication is true for $Dom_j = \{1\}$. By requiring that the considered moded query satisfies both the reachability constraint and the additional constraint in the precondition, the non-terminating query $\leftarrow constants(2, 1)$ is obtained. □

All information needed to construct these constraints can be obtained from the moded SLD-tree.

*Definition 6*

Let $C$ be an integer condition or expression and $N_i$ and $N_j$ two nodes in a moded SLD-tree $D$. Let $Cons$ be the set of all integer constructors occurring as selected atom in a node $N_p$ $(i \leqslant p \leqslant j)$ in $D$.

The function $apply_cons(C, N_i, N_j)$ returns the integer condition or expression obtained by exhaustively applying $\underline{I} \setminus Expr$ to $C$, for any $\underline{I}$ *is* $Expr \in Cons$. □

The constraints guaranteeing a derivation to $N_j$ to be applicable can be obtained using $apply_cons(Cond, N_0, N_i)$ for any integer condition $Cond$ in node $N_i$ in the considered derivation. For a path from nodes $N_b$ to $N_e$, the precondition of the implication is obtained using $apply_cons(Cond, N_b, N_i)$, for each condition $Cond$ in a node $N_i$ between nodes $N_b$ and $N_e$ and universally quantifying the integer variables of $N_b$.

*Example 10*

The derivation to node $N_9$ in Figure 2 contains integer conditions in nodes $N_3$ and $N_7$. These are expressed on the integer variable of the query, $\underline{N}$, using *apply_cons*:

- $apply_cons(0 > \underline{N}, N_0, N_3) = 0 > \underline{N}$.
- $apply_cons(\underline{M1} > \underline{N}, N_0, N_7) = 0 + 1 > \underline{N}$.

In order to obtain the precondition of the implication, the integer condition in node $N_7$ is expressed in terms of integer variables of node $N_5$:

- $apply_cons(\underline{M1} > \underline{N}, N_5, N_7) = \underline{M1} > \underline{N}$.

Universally quantifying these variables yields the precondition. □

In order to obtain the consequence of the implication for a path from nodes $N_b$ to $N_e$, one first replaces the integer variables of node $N_b$ in the precondition by the corresponding integer variables of node $N_e$. Then *apply_cons* is used to express the consequence in terms of the values in the previous iteration.

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

*Definition 7*

Let LHS be the precondition of an implication consisting of integer conditions and constraints of the form $I \in Dom_I$. Let $N_i$ and $N_j$ be two nodes in a moded SLD derivation such that all integer variables in LHS are in $A_i^1$ and let $\underline{I_1}, \ldots, \underline{I_n}$ be all integer variables of $A_i^1$.

If there exist subterms of $A_j^1$, $t_1, \ldots, t_n$, such that $\forall L : subterm(L, A_i^1) = \underline{I_p} \Longrightarrow subterm(L, A_j^1) = t_p, 1 \leqslant p \leqslant n$, then $replace(LHS, N_i, N_j)$ is obtained by applying $\{\underline{I_1} \setminus t_1, \ldots, \underline{I_n} \setminus t_n\}$ to all constraints in LHS. □

*Example 11*

In Example 10, we generated the precondition of the implication, $\underline{M1} > \underline{N}$. To obtain the consequence, $replace(\underline{M1} > \underline{N}, N_5, N_9)$ is applied, yielding $\underline{M2} > \underline{N}$. Then the integer variable of node $N_9$, i.e., $\underline{M_2}$, is expressed in terms of the integer variables of node $N_5$ using $apply_cons(\underline{M2} > \underline{N}, N_5, N_9) = \underline{M_1} + 1 > \underline{N}$.

Adding domains to pre- and post-conditions yields the desired implication:

$$\exists Dom_N, Dom_{M1} \subset \mathbb{Z}, \forall N, M1 \in \mathbb{Z} : M1 > N,\ N \in Dom_N,\ M1 \in Dom_{M1} \Longrightarrow$$
$$M1 + 1 > N,\ N \in Dom_N,\ M1 + 1 \in Dom_M.$$ □

Adding these constraints to the class of queries detected by Theorem 1, yields a class of non-terminating queries.

### *3.4 Proving that the constraints on integers are solvable*

The previous subsection introduced constraints, implying that all integer conditions in a considered derivation succeed. In this subsection, we introduce a technique to check solutions for these constraints by using a constraint-based approach. Symbolic coefficients represent values for the integers in the query and domains in the implication, for which the considered path is a loop. After these coefficients are introduced, the implication is transformed into a set of equivalent implications over natural numbers. These implications can then be solved automatically in the constraint-based approach, based on Proposition 3 given by Nguyen *et al.* (to appear).

*Proposition 3* (*Proposition 3 of Nguyen et al. (to appear)*)

Let *prem* be a polynomial over $n$ variables and *conc* a polynomial over 1 variable, both with natural coefficients, where *conc* is not a constant. Moreover, let $p_1, \ldots, p_{n+1}, q_1, \ldots, q_{n+1}$ be arbitrary polynomials with integer coefficients[2] over variables $\overline{X}$. If

$$\forall \overline{X} \in \mathbb{N} : conc(p_{n+1}) - conc(q_{n+1}) - prem(p_1, \ldots, p_n) + prem(q_1, \ldots, q_n) \geqslant 0$$

is valid, then $\forall \overline{X} \in \mathbb{N} : p_1 \geqslant q_1, \ldots, p_n \geqslant q_n \Longrightarrow p_{n+1} \geqslant q_{n+1}$ is also valid. □

[2] Proposition 3 in Nguyen *et al.* (to appear) uses natural coefficients, but the proposition also holds for polynomials with integer coefficients.

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

### 3.4.1 Introducing the symbolic coefficients

In order to represent half-open domains in the implication by symbolic coefficients, the domains are described by two symbolic coefficients, one upper or lower limit and another for the direction. Constraints of the form $Exp \in Dom_I$ in the implication, are replaced by constraints of the form $d_I * Exp \geqslant d_I * c_I$ with $d_I$ either 1 or $-1$, describing the domain $\{c_I, c_I - 1, \ldots\}$ for $d_I = -1$ and $\{c_I, c_I + 1, \ldots\}$ for $d = 1$. The values to be inferred for the integers of the query should satisfy the precondition of the implication. Of course, the symbolic coefficients $c_I$ should also be consistent with the values of the integers in the query.

*Example 12*

In Example 8, we introduced constraints on the integer variable $\underline{N}$, $0 > \underline{N}$, and $0 + 1 > \underline{N}$, proving non-termination for queries in $Den(\leftarrow count_to(\underline{N}, L))$. By convention, we denote the symbolic coefficients as constants. For the integer variable $\underline{N}$, we introduce the symbolic coefficient $n$.

The implication introduced in Example 8, for the path from nodes $N_5$ to $N_9$ in Figure 2, does not contain constraints on the domains. When adding these constraints to pre- and post-conditions, we obtain the following implication

$$\forall M, N \in \mathbb{Z} : \; M > N, \; N \in Dom_N, \; M \in Dom_M \Longrightarrow \\ M + 1 > N, \; N \in Dom_N, \; M + 1 \in Dom_M.$$

Representing these domains by symbolic coefficients yields the following implication:

$$\forall M, N \in \mathbb{Z} : \; M > N, \; d_N * N \geqslant d_N * c_N, \; d_M * M \geqslant d_M * c_M \Longrightarrow \\ M + 1 > N, \; d_N * N \geqslant d_N * c_N, \; d_M * (M + 1) \geqslant d_M * c_M.$$

In order to guarantee that the precondition succeeds for the considered derivation, $c_M$ and $c_N$ are required to be the values for $\underline{M}$ and $\underline{N}$ in node $N_5$. Combining these constraints implies non-termination for the query $\leftarrow count_to(n, L)$, for which the following constraints are satisfied with some unknown integers $c_N, c_M, d_N$, and $d_M$:

(1) $0 > n, \; 0 + 1 > n$ to guarantee applicability of the derivation.
(2) $c_N = n, \; c_M = 0 + 1$ to guarantee that the precondition holds.
(3) $d_N = 1 \vee d_N = -1, \; d_M = 1 \vee d_M = -1$.
(4) $\forall M, N \in \mathbb{Z} : M > N, d_N * N \geqslant d_N * c_N, d_M * M \geqslant d_M * c_M \Longrightarrow$
$M + 1 > N, d_N * N \geqslant d_N * c_N, d_M * (M + 1) \geqslant d_M * c_M$ to prove that the condition often succeeds infinitely.

Due to the implication, $d_M$ has to be 1. $d_N$ can be either 1 or $-1$. □

To be able to infer singleton domains, we allow the constant describing of the direction of the interval to be 0. If in such a constant $d_I$ is zero, the constraints on the domain are satisfied trivially because they simplify to $0 \geqslant 0$. To guarantee that the domain is indeed a singleton when $d_I$ is inferred to be zero, a constraint of the form $(1 - d_I^2)Exp = (1 - d_I^2) * c_I$ is added to the postcondition for every constraint $d_I * I \geqslant d_I * c_I$. This constraint is trivially satisfied for half-open domains and proves that $\{c_I\}$ is the domain in the case that $d_I = 0$.

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

*Example 13*
In Example 9, we introduced constraints on the integer variables $\underline{I}$ and $\underline{J}$, proving non-termination for queries in $Den(\leftarrow constants(\underline{I}, \underline{J}))$. Introducing symbolic coefficient $i$ and $j$ for the integers of the query and for the domains of $\underline{I}$ and $\underline{J}$, yields the following constraints:

(1) $i = 2$ to guarantee applicability of the derivation.
(2) $c_I = i,\ c_J = j$ to guarantee that the precondition holds.
(3) $d_I \leqslant 1,\ d_I \geqslant -1,\ d_J \leqslant 1,\ d_J \geqslant -1$.
(4) $\forall I, J \in \mathbb{Z} : I = 2,\ d_I * I \geqslant d_I * c_I,\ d_J * J \geqslant d_J * c_J \Longrightarrow$
$J * 2 = 2,\ d_I * (J * 2) \geqslant d_I * c_I, (1 - d_I^2) * (J * 2) = (1 - d_I^2) * c_I,$
$d_J * (I - J) \geqslant d_J * c_J, (1 - d_J^2) * (I - J) = (1 - d_J^2) * c_J.$

The implication in (4) can only be satisfied when $d_J$ is equal to zero. □

### 3.4.2 *To implications over the natural numbers*

The symbolic coefficients to be inferred, which represent the domains, allow to transform the implication over $\mathbb{Z}$ to an equivalent implication over $\mathbb{N}$:

- For $d_I = 1$, any integer in $\{c_I,\ c_I + 1,\ \ldots\}$ that satisfies the precondition is in $\{c_I + d_I * N \mid N \in \mathbb{N}\}$.
- For $d_I = -1$, any integer in $\{c_I,\ c_I - 1,\ \ldots\}$ that satisfies the precondition is in $\{c_I + d_I * N \mid N \in \mathbb{N}\}$
- For $d_I = 0$, any integer in $\{c_I\}$ that satisfies the precondition is in $\{c_I + d_I * N \mid N \in \mathbb{N}\}$.

Therefore, we obtain an equivalent implication over the natural numbers by replacing each integer $I$ by its corresponding expression $c_I + d_I * N$ and replacing the universal quantifier over $I$ by a quantifier over $N$.

### 3.4.3 *Automation by a translation to diophantine constraints*

In order to solve the resulting constraints, we use the approach given by Nguyen *et al.* (to appear). Constraints of the form $A =:= B$ in the implication are replaced by the conjunction $A \geqslant B,\ B \geqslant A$. Constraints of the form $A = / = B$ yield two disjunctive cases: one obtained by replacing $= / =$ in pre- and post-conditions by $>$ and another obtained by replacing it by $<$. The other conditions – i.e., $>, <$ and $\leqslant$ – are transformed into $\geqslant$-constraints in an obvious way. Implications with only one consequence are obtained by creating one implication for each consequence, with the precondition of the original implication.

The resulting implications allow to apply Proposition 3. These inequalities of the form, $p \geqslant 0$, are then transformed into a set of *diophantine constraints*, i.e., constraints without universally quantified variables, by requiring that all coefficients of $p$ are non-negative. As proposed by Nguyen *et al.* (to appear), the resulting diophantine constraints are then transformed into a SAT-problem. The constraints are then proven to have solutions by a SAT solver by inferring one possible solution.

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

Table 1. *An overview of the experiments*

| | Linear-class, 3 bits | Linear-class, 4 bits | Max2-class, 3 bits | Max2-class, 4 bits |
|---|---|---|---|---|
| count_to | + | + | + | + |
| constants | + | + | + | *OS* |
| int1 | + | + | + | + |
| int2 | + | + | + | + |
| int3 | + | + | + | *OS* |
| int4 | + | + | + | *OS* |
| int5 | + | + | + | *OS* |
| int6 | + | + | + | *OS* |
| int7 | + | + | *OS* | *OS* |
| int8 | + | + | *OS* | *OS* |
| int9 | − | + | *OS* | *OS* |
| int10 | − | − | + | *OS* |
| int11 | − | + | − | *OS* |
| int12 | − | + | − | *OS* |
| int13 | + | + | + | + |
| int14 | + | + | + | *OS* |

## 4 Evaluation

We have implemented our analysis and integrated it within our existing non-termination analyzer *pTNT*. The analyzer can be downloaded from http://www.cs.kuleuven.be/~dean/iclp2011.html We tested our analysis on a benchmark of 16 programs similar to those in the paper. These programs are also available online. In order to solve the resulting SAT-problem, MiniSat (Een and Srensson 2003) is used.

We experimented with different bit-sizes in the translation to SAT and different classes of functions for the *prem* functions in Proposition 3. As *conc* functions, the identity function was used. Table 1 shows the results for the considered settings, + denotes that non-termination is proven successfully, − denotes that non-termination could not be proven, and *OS* denotes that the computation went out of stack. The considered settings are 3 and 4 as bit-sizes and *linear* and *max*2 as forms for the symbolic *prem*-functions. The *linear* class is a weighted sum of each argument. The *max*2 class contains a weighted term for each multiplication of two arguments. The analysis time is between 1 and 20 seconds for all programs and settings.

Table 1 shows non-termination can be proven for any program of the benchmark when choosing the right combination of parameters, but no setting succeeds in proving non-termination for all programs. Programs *int*9 and *int*12 require a constant that cannot be represented with bit-size 3. Linear prem-functions cannot prove non-termination for *int*10. However, the setting withbit-size 4 and *max*2 as class of *prem*-function usually fails, because these settings cause an exponential increase in memory use during SAT-translation.

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

## 5 Conclusion

In this paper, we introduced a technique to detect classes of non-terminating queries for logic programs with integer arithmetic. The analysis starts with a given program and class queries, specified using modes, and detects subclasses of non-terminating queries. First, the derivations for the given class of queries are abstracted by building a moded SLD-tree (Shen *et al.* 2009) with additional transitions to handle integer arithmetic. Then this moded SLD-tree is used to detect subclasses of non-terminating queries in two phases. In the first phase, we ignore the conditions over integers, e.g., $> /2$, and detect paths in the moded SLD-tree that correspond to infinite derivations if all conditions on integers in those derivations succeed. For every such path, the corresponding subclass of queries is generated. In the second phase, the obtained classes of queries are restricted to classes of non-terminating queries, by formulating constraints implying that all conditions on integers will succeed. These constraints are then solved by transforming them into a SAT problem.

We implemented this approach in our non-termination analyzer $pTNT$ and evaluated it on a small benchmark of non-terminating Prolog programs with integer arithmetic. The evaluation shows that the proposed technique is rather powerful, and the parameters in the transformation to SAT must be chosen carefully to avoid excessive memory use. For future work, we plan to improve the efficiency by using SMT solvers.

## Acknowledgements

We thank the referees for their useful and constructive comments.

## References

De Schreye, D. and Decorte, S. 1994. Termination of logic programs: The never-ending story. *Journal of Logic Programming 19/20*, 199–260.

En, N. and Srensso n, N. 2003. An extensible sat-solver. In *SAT*, E. Giunchiglia and A. Tacchella, Eds. Lecture Notes in Computer Science (LNCS), vol. 2919. Springer, New York, 502–518.

Giesl, J., Schneider-kamp, P. and Thiemann, R. 2006. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *Third International Joint Conference on Automated Reasoning* (*IJCAR 06*). Springer-Verlag, Berlin, Germany, 281–286.

Lloyd, J. W. 1987. *Foundations of Logic Programming,* 2nd ed. Springer, New York.

Nguyen, M. T., De Schreye, D., Giesl, J. and Schneider-Kamp, P. 2011. Polytool: Polynomial interpretations as a basis for termination analysis of logic programs. *Theory and Practice of Logic Programming*, 11(1), 33–63.

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

Serebrenik, A. and De Schreye, D. 2001. Inference of termination conditions for numerical loops in prolog. In *International Conferences on Logic for Programming, Artificial Intelligence and Reasoning* (*LPAR*), R. Nieuwenhuis and A. Voronkov, Eds. Lecture Notes in Computer Science (LNCS), vol. 2250. Springer, New York, 654–668.

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press

Shen, Y.-D., De Schreye, D. and Voets, D. 2009. Termination prediction for general logic programs. *Theory and Practice of Logic Programming 9*(6), 751–780.

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

Voets, D. and De Schreye, D. 2009. A new approach to non-termination analysis of logic programs. In *International Logic Programming Conference* (*ICLP*), P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science (LNCS), vol. 5649. Springer, New York, 220–234.

Voets, D. and De Schreye, D. 2010. Non-termination analysis of logic programs using types. In *Proceedings of Logic Program Synthesis and Transformation* (*LOPSTR*). M. Alpuente, Ed. Lecture Notes in Computer Science (LNCS), vol. 6564. Springer, New York, 133–148.

https://doi.org/10.1017/S1471068411000159 Published online by Cambridge University Press