

Inference of termination conditions for numerical loops

Alexander Serebrenik
Danny De Schreye

Report CW 308, May 2001



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Inference of termination conditions for numerical loops

Alexander Serebrenik

Danny De Schreye

Report CW 308, May 2001

Department of Computer Science, K.U.Leuven

Abstract

We present a new approach to termination analysis of numerical computations in logic programs. Traditional approaches fail to analyse them due to non well-foundedness of the integers. We present a technique that allows to overcome these difficulties. Our approach is based on transforming a program in way that allows integrating and extending techniques originally developed for analysis of numerical computations in the framework of query-mapping pairs with the well-known framework of acceptability. Such an integration not only contributes to the understanding of termination behaviour of numerical computations, but also allows to perform a correct analysis of such computations automatically, thus, extending previous work on a constraints-based approach to termination. In the last section of the paper we discuss possible extensions of the technique, including incorporating general term orderings.

Keywords : termination analysis, numerical computation, program transformation.

1 Introduction

Numerical computations form an essential part of almost any real-world program. Clearly, in order for a termination analyser to be of practical use it should contain a mechanism for inferring termination of such computations. However, this topic attracted less attention of the research community. In this work we concentrate on automatic termination inference for logic programs depending on numerical computations. Dershowitz *et al.* [10] showed that termination of general numerical computations, for instance on floating point numbers, may be contr-intuitive, i.e., the actually observed behaviour does not necessary coincide with the theoretically expected one. Thus, we restrict ourselves to integer computations only.

While discussing termination of integer computations the following question should be asked: what conditions on the queries should be assumed, such that the queries will terminate. We refer to this question as to *termination inference problem*. We illustrate this notion with the following example:

Example 1.

$$p(X) \leftarrow X < 7, X1 \text{ is } X + 1, p(X1).$$

This program terminates for queries $p(X)$, for all integer values of X . Thus, the answer for the termination inference problem is the condition “*true*”. \square

This example also hints why the traditional approaches to termination analysis fail to prove termination of this example. These approaches are mostly based on the notion of *level mapping*, that is a function from the set of all possible atoms to the natural numbers, and should decrease while traversing the rules. In our case, such a level mapping should depend on X , but X can be negative as well!

Two approaches for solving this problem are possible. First, once can change the definition of the level mapping to map atoms to integers. However, integers are, in general, not well-founded. Thus, to prove termination one should prove that the mapping is to some well-founded subset of integers. In the example above $(-\infty, 7)$ forms such a subset with an ordering \succ , such that $x \succ y$ if $x < y$, with respect to the usual ordering on integers.

The second approach, that we present in the paper, does not require changing the definition of level mapping. Indeed, the level mapping as required exists. It maps $p(X)$ to $7 - X$ if $X < 7$ and to 0 otherwise.

This level mapping decreases while traversing the rule, i.e., the size of $p(X)$, $7 - X$, is greater than the size of $p(X1)$, $6 - X$, thus, proving termination. We present a transformation that allows to define such a level mappings in an automatic way. The transformation presented allows to incorporate techniques of [10], such as level mapping inference, in the well-known framework of the acceptability with respect to a set [7, 8]. This integration provides not only a better understanding of termination behaviour of integer computations, but also the possibility to perform the analysis automatically as in Decorte *et al.* [9].

The rest of the paper is organised as following. After making some preliminary remarks we present in Section 3 our transformation—first by means of an example, then more formally. In Section 4 we discuss more practical issues and present the algorithm, implementing the termination inference. In Section 5 we discuss further extensions, such as proving termination of programs depending in numerical computations as well as the symbolic ones. Then we review the related work and conclude.

2 Preliminaries

2.1 Logic Programming

We follow the standard notation for terms and atoms. A *query* is a finite sequence of atoms. Given an atom A , $rel(A)$ denotes the predicate occurring in A . $Term_P$ and $Atom_P$ denote, respectively, sets of all terms and atoms that can be constructed from the language underlying P . The extended Herbrand Universe U_P^E (the extended Herbrand base B_P^E) is a quotient set of $Term_P$ ($Atom_P$) modulo the variant relation.

We refer to an SLD-tree constructed using the left-to-right selection rule of Prolog, as an LD-tree. We will say that a goal G *LD-terminates* for a program P , if the LD-tree for (P, G) is finite.

The following definition is borrowed from [1].

Definition 1. *Let P be a program and p, q be predicates occurring in it.*

- *We say that p refers to q in P if there is a clause in P that uses p in its head and q in its body.*
- *We say that p depends on q in P and write $p \sqsupseteq q$, if (p, q) is in the transitive, reflexive closure of the relation refers to.*
- *We say that p and q are mutually recursive and write $p \simeq q$, if $p \sqsupseteq q$ and $q \sqsupseteq p$. We also write $p \sqsubset q$ when $p \sqsupseteq q$ and $q \not\sqsupseteq p$.*

2.2 Termination analysis

In this subsection we recall some basic notions, related to termination analysis. A *level mapping* is a function $|\cdot|: B_P^E \rightarrow \mathcal{N}$, where \mathcal{N} is the set of the natural numbers.

The following definition generalises the notion of acceptability with respect to a set [7, 8] by extending it to mutual recursion, using the standard notion of mutual recursion [1].

Definition 2. *Let S be a set of atomic queries and P a definite program. P is acceptable with respect to S if there exists a level mapping $|\cdot|$ such that*

- for any $A \in \text{Call}(P, S)$
- for any clause $A' \leftarrow B_1, \dots, B_n$ in P , such that $\text{mgu}(A, A') = \theta$ exists,
- for any atom B_i , such that $\text{rel}(B_i) \simeq \text{rel}(A)$
- for any computed answer substitution σ for $\leftarrow (B_1, \dots, B_{i-1})\theta$:

$$|A| > |B_i\theta\sigma|$$

The following proposition characterises LD-termination in terms of acceptability.

Theorem 1. (cf. [7]) *Let P be a program. P is acceptable with respect to a set of atomic queries S if and only if P is LD-terminating for all queries in S .*

We also need to introduce the notion of interargument relations.

Definition 3. [9] *Let P be a definite program, p/n a predicate in P . An interargument relation for p/n is a relation $R_p \subseteq \mathcal{N}^n$. R_p is a valid interargument relation for p/n with respect to a norm $\|\cdot\|$ if and only if for every $p(t_1, \dots, t_n) \in \text{Atom}_P$: if $P \models p(t_1, \dots, t_n)$ then $(\|t_1\|, \dots, \|t_n\|) \in R_p$.*

To characterise program transformations Bossi and Cocco [4] introduced the following notion for a program P and a query Q . $\mathcal{M}[[P]](Q) =$

$$\begin{aligned} & \{\sigma \mid \text{there is a successful LD-derivation of } Q \text{ and } P \text{ with c.a.s. } \sigma\} \\ & \cup \{\perp \mid \text{there is an infinite LD-derivation of } Q \text{ and } P\} \end{aligned}$$

3 Methodology

In this section we introduce our methodology using a simple example. In the subsequent section we formalise it and discuss different extensions.

Our first example generates an oscillating sequence like $-2, 4, -16, \dots$ and stops if the generated value is greater than 1000 or smaller than -1000 . The treatment is done first on the intuitive level.

Example 2. We are interested in proving termination of the set of queries $S = \{p(X) \mid X \text{ is an integer}\}$ with respect to the following program.

$$\begin{aligned} p(X) &\leftarrow X > 1, X < 1000, X1 \text{ is } -X * X, p(X1). \\ p(X) &\leftarrow X < -1, X > -1000, X1 \text{ is } X * X, p(X1). \end{aligned}$$

The direct attempt to define the level mapping of $p(X)$ as X fails, since X can be positive as well as negative. Thus, a more complex level mapping should be defined. We start with some observations.

The first clause is applicable if $1 < X < 1000$, the second one, if $-1000 < X < -1$. Observe that termination of $p(X)$ for $X \leq -1000$, $-1 \leq X \leq 1$ or $X \geq 1000$ is trivial. Moreover, if the first clause is applied, then for the recursive call $p(X1)$ it holds that $-1000 < X1 < 1$. Similarly, if the second clause is applied, then for the recursive call $p(X1)$ it holds that $1 < X1 < 1000$. We use this observation and replace a predicate p with two new predicates $p^{1 < X < 1000}$ and $p^{-1000 < X < -1}$, such that $p^{1 < X < 1000}$ is called if $p(X)$ is called and $0 < X < 1000$ holds and $p^{-1000 < X < -1}$ is called if $p(X)$ is called and $-1000 < X < -1$ holds. The following program is obtained:

$$\begin{aligned} p^{1 < X < 1000}(X) &\leftarrow X > 1, X < 1000, X1 \text{ is } -X * X, p^{-1000 < X < -1}(X1). \\ p^{-1000 < X < -1}(X) &\leftarrow X < -1, X > -1000, X1 \text{ is } X * X, p^{1 < X < 1000}(X1). \end{aligned}$$

Now we can define two *different* level mappings, one for atoms of $p^{1 < X < 1000}$ and another one for atoms of $p^{-1000 < X < -1}$. Let $| p^{1 < X < 1000}(X) | = 1000 - X$ and let $| p^{-1000 < X < -1}(X) | = 1000 + X$. We verify that the transformed program is acceptable with respect to $S' = \{p^{1 < X < 1000}(X) \mid 1 < X < 1000\} \cup \{p^{-1000 < X < -1}(X) \mid -1000 < X < -1\}$ via the specified level mappings. This will imply termination of the transformed program with respect to these queries, and thus, termination of the original program with respect to S .

Indeed, consider first queries of the form $p^{1 < X < 1000}(n)$ for $1 < n < 1000$. The only clause that its head can be unified with this query is the first clause and the only atom of a predicate mutually recursive

with $p^{1 < X < 1000}(X)$ is $p^{-1000 < X < -1}(m)$. Then, the following should hold: $|p^{1 < X < 1000}(n)| > |p^{-1000 < X < -1}(m)|$, i.e., $1000 - n > 1000 + m$. Recall that $n > 1$ and $m = -n^2$, thus, $1000 - n > 1000 - n^2$ which is true for $n > 1$. Similarly, for queries of the form $p^{-1000 < X < -1}(n)$, the acceptability condition is reduced to $1000 + n > 1000 - n^2$ which is true for $n < -1$. \square

The intuitive presentation above hints to the major issues to be discussed in the following sections: how the cases as above can be extracted from the program, and how, given the cases extracted, the program should be transformed? Before discussing the answers to these questions we present some basic notions.

3.1 Basic notions

In this section we formally introduce some notions that the further analysis will be based on. Recall that the aim of our analysis, given a predicate and a query, is to find a sufficient condition for termination of this query with respect to this program. Thus, we need to define a notion of a termination condition. To do so we start with a number of auxiliary definitions.

Definition 4. *Let p be a predicate of arity n . Then, $\$1^p, \dots, \n^p are called argument positions denominators.*

If the predicate is clear from the context the superscripts will be omitted.

Definition 5. *Let P be a program, S be a set of queries. An argument position i of a predicate p is called integer argument position, if for every $p(t_1, \dots, t_n) \in \text{Call}(P, S)$, t_i is an integer.*

Argument positions denominators corresponding to integer argument positions will be called integer argument positions denominators.

An *integer inequality* is an atom of one of the following forms $\text{Exp1} > \text{Exp2}$, $\text{Exp1} < \text{Exp2}$, $\text{Exp1} \geq \text{Exp2}$ or $\text{Exp1} \leq \text{Exp2}$, where Exp1 and Exp2 are numerical expressions, i.e., are constructed from integers, variables and the four operations of arithmetics. A *symbolic inequality over the arguments of a predicate p* is constructed similarly to an integer inequality. However, instead of variables, integer argument positions denominators are used.

Example 3. $X > 0$ and $Y \leq X + 5$ are integer inequalities. Given a predicate p of arity 3, having only integer argument positions $\$1^p > 0$ and $\$2^p \leq \$1^p + \$3^p$ are symbolic inequalities over the arguments of p . \square

Disjunctions of conjunctions based on integer inequalities are called *integer conditions*. Similarly, propositional calculus formulae based on symbolic inequalities over the arguments of a same predicate are called *symbolic conditions over the integer arguments of this predicate*.

Example 4. $X > 0 \wedge Y \leq X + 5$ is an integer condition. Given a predicate p as above $\$1^p > 0 \wedge \$2^p \leq \$1^p + \3^p is a symbolic condition over the integer arguments of p . \square

Definition 6. Let $p(t_1, \dots, t_n)$ be an atom and let c_p be a symbolic condition over the arguments of p . An instance of the condition with respect to an atom, denoted $c_p(p(t_1, \dots, t_n))$, is obtained by replacing the argument positions denominators with the corresponding arguments, i.e., $\$i^p$ with t_i .

Example 5. Let $p(X, Y, 5)$ be an atom and let c_p be $(\$1^p > 0) \wedge (\$2^p \leq \$1^p + \$3^p)$. Then, $c_p(p(X, Y, 5))$ is the integer conjunction $(X > 0) \wedge (Y \leq X + 5)$. \square

Now we are ready to define *termination condition* formally.

Definition 7. Let P be a program, and Q be a query. A symbolic condition $c_{\text{rel}(Q)}$ is a termination condition for Q if given that $c_{\text{rel}(Q)}(Q)$ holds, Q left-terminates with respect to P .

A termination condition for Example 2 is *true*, i.e., the query $\leftarrow p(X)$, terminates with respect to the program for all integer X . Obviously this is not always the case.

Example 6. Consider the following program.

$$\begin{aligned} q(X) &\leftarrow X > 0, X \leq 5, q(X). \\ q(X) &\leftarrow X > -5. \end{aligned}$$

This program terminates for queries $\leftarrow p(n)$, such that $n \leq -5$, since no rule is applicable, or $-5 < n \leq 0 \vee n > 5$, since repeated rule application in this case is finite. Thus, a termination condition for the goal $\leftarrow q(X)$ is $\$1 \leq 0 \vee \$1 > 5$. \square

3.2 Types

In this subsection we discuss inferring what values integer arguments can take during traversal of the rules, i.e., the “case analysis” performed

in Example 2. This information is crucial for defining level-mappings. Example 2 provides already the underlying intuition—“cases” are types, i.e., calls of the predicate p^c are identical to the calls of the predicate p , where c holds for its arguments. More formally we define a notion of *set of adornments*, later we specify when it is *guard-tuned* and we show how such a guard-tuned set of adornments can be constructed.

Definition 8. *Let p be a predicate, and let c_1, \dots, c_n be symbolic conditions over the integer arguments of p . The set $\mathcal{A}_p = \{c_1, \dots, c_n\}$ is called set of adornments for p if for all i, j such that $1 \leq i < j \leq n$, $c_i \wedge c_j = \text{false}$ and $\bigvee_{i=1}^n c_i = \text{true}$.*

A set of adornments partitions the domain for (some of) the integer variables of the predicate.

Example 7. Let P be as in Example 2. The following are examples of sets of adornments: $\{\$1 \leq 100, \$1 > 100\}$ and $\{(\$1 \leq -1000) \vee (-1 \leq \$1 \leq 1) \vee (\$1 \geq 1000), -1000 < \$1 < -1, 1 < \$1 < 1000\}$. \square

3.3 Program transformation

The next question that should be answered is how the program should be transformed given a set of adornments. After this transformation $p^c(X_1, \dots, X_n)$ will behave with respect to the transformed program exactly as $p(X_1, \dots, X_n)$ does, for all calls that satisfy the condition c . Intuitively, we replace each call to the predicate p in the original program by a number of possible calls in the transformed one. To define a transformation formally we introduce the following definition:

Let $H \leftarrow B_1, \dots, B_n$ be a rule. B_1, \dots, B_i , $1 \leq i \leq n$, is called *prefix of the rule*, if for all j , $1 \leq j \leq i$, B_j is an integer inequality and the only variables appearing in its arguments are variables of H . B_1, \dots, B_i is called *the maximal prefix* of the rule, if it is a prefix and B_1, \dots, B_i, B_{i+1} is not a prefix.

Observe, that since a prefix constrains only variables appearing in the head of a clause there exists a symbolic condition over the arguments of the predicate of the head, such that the prefix is its instance with respect to the head. Note, that in general, this symbolic condition is not necessarily unique.

Example 8. Consider the following program: $p(X, Y, Y) \leftarrow Y > 5$. The only prefix of this rule is $Y > 5$. There are two symbolic conditions over the arguments of p , $\$2 > 5$ and $\$3 > 5$, such that $Y > 5$ is their instance with respect to $p(X, Y, Y)$. \square

The following notion, borrowed from [10], guarantees uniqueness of such symbolic conditions. In this case we say that the symbolic condition *corresponds* to the prefix.

Definition 9. [10] A rule $H \leftarrow B_1, \dots, B_n$ is called partially normalised if all integer argument positions in H are occupied by distinct variables¹.

We will also say that a program P is partially normalised if all the rules in P are partially normalised. After integer argument positions are identified a program can be easily rewritten to partially normalised form.

Now we are ready to present the transformation formally.

Definition 10. Let P be a program and let p be a predicate in it. Let $\mathcal{A} = \bigcup_{q \in P} \mathcal{A}_q$ be a set of possible adornments for P . Then, the program P^a , called adorned with respect to p , is obtained by two steps as following:

1. For every rule r in P and for every subgoal $q(t_1, \dots, t_n)$, $p \simeq q$, in r
 - For every $A \in \mathcal{A}_q$
 - Replace $q(t_1, \dots, t_n)$ by $q^A(t_1, \dots, t_n)$.
2. For every newly obtained rule r
 - Are adornments and inequalities in the body of r consistent? *
 - If not—reject the rule.
 - If r defines some q , $q \simeq p$
 - Get as adornments of the head of r all $A \in \mathcal{A}_q$, that are consistent with comparisons of the maximal prefix of r and adornments of the body of r .

Example 9. Continue Example 2. The sets of adornments presented in Example 7 are used. With the first set of adornments in Example 7 we obtain the program:

$$\begin{aligned}
p^{\$1 \leq 100}(X) &\leftarrow X > 1, X < 1000, X1 \text{ is } - X * X, p^{\$1 \leq 100}(X1). \\
p^{\$1 > 100}(X) &\leftarrow X > 1, X < 1000, X1 \text{ is } - X * X, p^{\$1 \leq 100}(X1). \\
p^{\$1 \leq 100}(X) &\leftarrow X < -1, X > -1000, X1 \text{ is } X * X, p^{\$1 \leq 100}(X1). \\
p^{\$1 \leq 100}(X) &\leftarrow X < -1, X > -1000, X1 \text{ is } X * X, p^{\$1 > 100}(X1).
\end{aligned}$$

If the second set of adornments is used, the following program is obtained:

$$\begin{aligned}
p^{1 < \$1 < 1000}(X) &\leftarrow X > 1, X < 1000, X1 \text{ is } - X * X, \\
&p^{-1000 < \$1 < -1}(X1).
\end{aligned}$$

¹ If such a rule has only integer arguments Apt *et al.* [2] call it *homogeneous*.

$$\begin{aligned}
p^{1 < \$1 < 1000}(X) &\leftarrow X > 1, X < 1000, X1 \text{ is } - X * X, \\
& p^{(\$1 \leq -1000) \vee (-1 \leq \$1 \leq 1) \vee (\$1 \geq 1000)}(X1). \\
p^{-1000 < \$1 < -1}(X) &\leftarrow X < -1, X > -1000, X1 \text{ is } X * X, \\
& p^{1 < \$1 < 1000}(X1). \\
p^{-1000 < \$1 < -1}(X) &\leftarrow X < -1, X > -1000, X1 \text{ is } X * X, \\
& p^{(\$1 \leq -1000) \vee (-1 \leq \$1 \leq 1) \vee (\$1 \geq 1000)}(X1).
\end{aligned}$$

□

Correctness of the transformation should be proved. First of all, finiteness of the number of clauses, the number of subgoals in a clause and the number of elements in an adornment ensures that the transformation always terminates. Second, we need to prove that the transformation preserves termination.

Adorning clauses introduces new predicates. This means that the query Q gives rise to a number of different queries. Clearly, termination of all of these queries with respect to P^a is equivalent to termination of Q with respect to P^a augmented by a set of the clauses, such that for every $p \simeq \text{rel}(Q)$ and for every $A \in \mathcal{A}_p$ the clause $p(X_1, \dots, X_n) \leftarrow p^A(X_1, \dots, X_n)$ is added. We call this extended program P^{ag} .

Lemma 1. *Let P be a program, and let Q be a query. Let P^{ag} be a program obtained as described above. Then, $\mathcal{M}[[P^{ag}]](Q) \subseteq \mathcal{M}[[P]](Q)$.*

Proof. Proof is done similarly to [13]. Replace each call to adorned predicate p^A in the body of clauses originating from P^a by the corresponding call to p . Call the obtained program P^{agw} . Since $P^{ag} \setminus P^a$ has a clause for every adornment in \mathcal{A}_p , every path in the LD-tree of Q w.r.t. P^{ag} has a corresponding path in the LD-tree of Q w.r.t. P^{agw} . Thus, $\mathcal{M}[[P^{ag}]](Q) \subseteq \mathcal{M}[[P^{agw}]](Q)$.

Similarly, every path in the LD-tree of Q w.r.t. P^{agw} has a corresponding path in the LD-tree of Q w.r.t. P . Indeed, every call to p on the path of the LD-tree of Q w.r.t. P^{agw} is followed by calls to all adorned versions of p via rules originating from $P^{ag} \setminus P^a$, and subsequently to the rules of those predicates. However, in P these rules are directly defining P . Thus, $\mathcal{M}[[P^{agw}]](Q) \subseteq \mathcal{M}[[P]](Q)$.

We conclude, that $\mathcal{M}[[P^{ag}]](Q) \subseteq \mathcal{M}[[P]](Q)$. ■

The second direction of the containment depends on the consistency check strategy applied at the point marked by $*$ in the definition of P^a .

Example 10. Let Q be $p(X)$ and let P be the following program

$$p(X) \leftarrow X > 0, q(X), X < 0. \quad q(X) \leftarrow X > 0, p(X).$$

Predicates p and q are mutually recursive. Thus, both of them should be adorned. Let \mathcal{A}_p be $\{\$1 > 0, \$1 \leq 0\}$ and \mathcal{A}_q be $\mathcal{A}_q = \{\$1 > 0, \$1 \leq 0\}$. The following program is obtained after the first step of the adorning process.

$$\begin{aligned} p(X) &\leftarrow X > 0, q^{\$1>0}(X), X < 0. & q(X) &\leftarrow X > 0, p^{\$1>0}(X). \\ p(X) &\leftarrow X > 0, q^{\$1\leq 0}(X), X < 0. & q(X) &\leftarrow X > 0, p^{\$1\leq 0}(X). \end{aligned}$$

The second step of the adorning process should infer adornments for the heads of the clauses, possibly rejecting the inconsistent ones. If the inference technique is eager, i.e., tries to use all the information it has in the body constraints and adornments of body subgoals, a program consisting only of one rule, namely $q^{\$1>0}(X) \leftarrow X > 0, p^{\$1>0}(X)$, is obtained. Other clauses are rejected because inconsistency of the set of built-in comparisons and adornments applied to the corresponding atoms is discovered. Thus, the extended program is the following one:

$$\begin{aligned} q^{\$1>0}(X) &\leftarrow X > 0, p^{\$1>0}(X). \\ q(X) &\leftarrow q^{\$1>0}(X). & p(X) &\leftarrow p^{\$1>0}(X). \\ q(X) &\leftarrow q^{\$1\leq 0}(X). & p(X) &\leftarrow p^{\$1\leq 0}(X). \end{aligned}$$

The query $p(X)$ terminates with respect to this program, while it does not terminate with respect to the original one. This example shows that eager inference technique can actually improve termination.

In order termination to be preserved a weaker inference engine should be used. For example, one can use an inference technique that considers inequalities only of the maximal prefix. If this technique is used, the following program is obtained after extension:

$$\begin{aligned} p^{\$1>0}(X) &\leftarrow X > 0, q^{\$1>0}(X), X < 0. \\ q^{\$1>0}(X) &\leftarrow X > 0, p^{\$1>0}(X). \\ q(X) &\leftarrow q^{\$1>0}(X). & p(X) &\leftarrow p^{\$1>0}(X). \\ q(X) &\leftarrow q^{\$1\leq 0}(X). & p(X) &\leftarrow p^{\$1\leq 0}(X). \end{aligned}$$

The query $p(X)$ does not terminate with respect to this program, just as it does not terminate with respect to the original one. The following lemma shows that if this weaker inference technique is used, termination is preserved. \square

Observe that it is known that unfolding preserves computed answer substitutions [3, 4], thus in order to prove the second direction of the containment we have to prove that termination is preserved. Unlike the previous lemma that can be established for an arbitrary set of symbolic conditions, used as adornments, this lemma holds only sets of adornments as defined in Definition 8.

Lemma 2. *Let P be a program, and let Q be a query. Let P^{ag} be a program obtained as described above with respect to a set of adornments and consistency checking with respect to maximal prefixes. Then, $\mathcal{M}[[P]](Q) \subseteq \mathcal{M}[[P^{ag}]](Q)$.*

Proof. Assume that Q terminates w.r.t. P^{ag} and does not terminate w.r.t. P . Let $H \leftarrow B_1, \dots, B_n$ be a clause in P , such that Q can be unified with H and the application of this rule starts an infinite branch in the LD-tree.

In the resolution w.r.t. P^{ag} the only clauses to be applied to resolve with Q are those of $P^{ag} \setminus P^a$. This will reduce the query to queries of the form $q^A(t_1, \dots, t_k)$, where q^A are adorned versions of the predicate q of Q and t_1, \dots, t_k are arguments of Q .

First of all, we prove that for some adornment of q there exists an adorned variant of $H \leftarrow B_1, \dots, B_n$ exists in P^{ag} . For the sake of contradiction assume that this does not hold, i.e., there is no possible adornment of atoms of predicates mutually recursive with $rel(H)$ that is consistent with comparisons of the maximal prefix of the clause and with one of the possible adornments for $rel(H)$. This verbal description above can be rewritten in the following way:

$$\begin{aligned} false &= A_1 \& C \& B_{11} \& \dots \& B_{n1} \\ &\dots \\ false &= A_n \& C \& B_{1m_1} \& \dots \& B_{nm_n} \end{aligned}$$

where A_1, \dots, A_n are all possible adornments for $rel(H)$, C is a conjunction of comparisons of the maximal prefix of the clause and B_{11}, \dots, B_{nm_n} are all possible adornments for the atoms of predicates mutually recursive with $rel(H)$ and appearing in the body of the clause. Disjunction of these conjunctions is, on the one hand, *false* and on the other hand, C (since \mathcal{A}_p is complete). Thus, $C = false$.

Observe, that comparisons of the maximal prefix are not affected by the transformation. Thus, the body of the clause above starts with a sequence of inconsistent comparisons. Since comparisons are part of the maximal prefix, inconsistency will be discovered before any atom other than a comparison is reached. Thus, any application of this clause will

cause a failure and it cannot start an infinite branch of the LD-tree. Thus, the adorned version of $H \leftarrow B_1, \dots, B_n$ exists in P^{ag} . Let $H' \leftarrow B'_1, \dots, B'_n$ be this adorned version, where B'_i denotes either an adorned version of B_i , if B_i was adorned, and is identical to B_i otherwise.

Let B'_j be the first (while going from left to right) adorned atom in the clause body. Since transformation does not affect the preceding body atoms the corresponding queries are identical w.r.t. P and w.r.t. P^{ag} . Let Q'_j be a query corresponding to B'_j , and let Q_j be a query corresponding to Q_j . We have to prove that Q_j terminates.

Assume that Q_j does not terminate. Let $G \leftarrow \dots$ be a clause in P , such that Q_j is resolved with on the infinite branch of the LD-tree. By the previous claim there is an adorned version of this clause that belongs to P^{ag} . If there is no adorned version of this clause with the adornment of Q'_j then by reasoning similar to above one can conclude that this adornment is inconsistent with the comparisons of the maximal prefix. Since those are not affected by the transformation and are identical in the clause of P and in the corresponding clause of P^{ag} . Thus, any application of this clause will cause a failure and it cannot continue an infinite branch of the LD-tree. This means, that there exists an adorned variant of the clause, such that its head can be unified with Q'_j . Since computed answer substitutions of queries with respect to P and of P^{ag} are identical (follows from the fact that unfolding preserves computed answer substitutions [11]) the same reasoning can be done for any of the subsequent calls and clauses, i.e., we will mimic the resolution started by Q_j w.r.t. P by a resolution that is started by Q'_j w.r.t. P^{ag} . Since any resolution of Q'_j w.r.t. P^{ag} is finite, contradiction to the assumption is obtained.

Since computed answers are preserved the same claim can be proved also for the queries, originating from other atoms that B_j , thus, completing the proof. ■

The following theorem summarises lemmas above, for the case of maximal prefixes.

Theorem 2. *Let P be a program, let Q be a query and let \mathcal{A} be a set of adornments. Let P^{ag} be a program obtained as described above with respect to \mathcal{A} . Then, $\mathcal{M}[P](Q) = \mathcal{M}[P^{ag}](Q)$.*

This theorem has two corollaries. The first one establishes a relation between the termination condition and the adornments and the second one allows to reason on the termination with respect to the original program P .

Corollary 1. *Let P be a program, let Q be a query and let \mathcal{A} be a set of adornments. Let*

$$A = \{a \mid a \in \mathcal{A}, \text{ for all } q \text{ rel}(Q)^a \sqsupseteq q, q \text{ is not recursive in } P^a\}.$$

Then $\bigvee_{a \in A}$ is a termination condition for P .

Example 11. Continue Example 9. In the program obtained with respect to the second set of adornments, predicate $p^{(\$1 \leq -1000) \vee (-1 \leq \$1 \leq 1) \vee (\$1 \geq 1000)}$ satisfies the Corollary. Thus, $(\$1 \leq -1000) \vee (-1 \leq \$1 \leq 1) \vee (\$1 \geq 1000)$ is a termination condition for $p(X)$ with respect to the program presented in Example 2. \square

Theorem 2 implies that a program P is LD-terminating with respect to all queries in a set of atomic queries S if and only if P^{ag} , constructed as above, is acceptable with respect to S . However, the later one is equivalent to acceptability of P^a with respect to $\{q^A(t_1, \dots, t_n) \mid q(t_1, \dots, t_n) \in S, A \in \mathcal{A}_q\}$.

Corollary 2. *Let P be a program, let S be a set of atomic queries and let $\mathcal{A} = \bigcup_{Q \in (S), q \approx \text{rel}(Q)} \mathcal{A}_q$ be a set of adornments. Let P^a be obtained with respect to \mathcal{A} . Then, P is LD-terminating with respect to all queries in S if and only if P^a is acceptable with respect to $\{q^A(t_1, \dots, t_n) \mid p(t_1, \dots, t_n) \in S, A \in \mathcal{A}_q\}$.*

This corollary allows to complete the termination proof for Example 2.

Example 12. The transformed program P^a (with respect to $\{-1000 < \$1 < -1, 1 < \$1 < 1000, (\$1 \leq -1000) \vee (-1 \leq \$1 \leq 1) \vee (\$1 \geq 1000)\}$) is presented in Example 9. We prove acceptability of P^a with respect to the set $S = \{p^{1 < \$1 < 1000}(X), p^{-1000 < \$1 < -1}(X), p^{(\$1 \leq -1000) \vee (-1 \leq \$1 \leq 1) \vee (\$1 \geq 1000)}(X)\}$. Then, $S = \text{Call}(P^a, S)$. Let $|\cdot|$ be the level mapping, defined as follows:

$$\begin{aligned} |p^{-1000 < \$1 < -1}(X)| &= \begin{cases} 1000 + X & \text{if } -1000 < X < -1 \\ 0 & \text{otherwise} \end{cases} \\ |p^{1 < \$1 < 1000}(X)| &= \begin{cases} 1000 - X & \text{if } 1 < X < 1000 \\ 0 & \text{otherwise} \end{cases} \\ |p^{(\$1 \leq -1000) \vee (-1 \leq \$1 \leq 1) \vee (\$1 \geq 1000)}(X)| &= 0 \end{aligned}$$

We are not going to prove completely that P^a is acceptable with respect to S via $|\cdot|$, but restrict ourselves only for one call, $p^{1 < \$1 < 1000}(X)$.

There are two clauses—the first and the second one—such that their heads can be unified with $p^{1 < \$1 < 1000}(X)$. The second clause is not recursive and the condition holds vacuously. The first clause is recursive and the acceptability requires $1000 - X > 1000 + X1$, where $X > 1$, $X < 1000$ and $X1 = -X^2$. Substituting the last equality and simplifying one gets $X^2 > X$, that is true for $X > 1$. Other calls are solved similarly. \square

4 Practical issues

In the previous section we have shown the transformation that allows reasoning on termination of the numerical computations in the framework of acceptability with respect to the set. In this section we discuss a number of practical issues to be considered for an automated termination analysis.

4.1 Guard-tuned sets of adornments

In Example 7 we have seen two different sets of adornments. Both of them are valid according to Definition 8. However, $\{-1000 < \$1 < -1, 1 < \$1 < 1000, (\$1 \leq -1000) \vee (-1 \leq \$1 \leq 1) \vee (\$1 \geq 1000)\}$ is in some sense preferable to $\{\$1 \leq 100, \$1 > 100\}$. There is a number of reasons to prefer the first set to the second one. First of all, it has a *declarative reading*: the sets that are constructed are related to the constraints in the bodies of the clauses and in fact express conditions that, when satisfied, allow to traverse the rule. Second, comparing the two adorned programs in Example 9 one might observe that the second program has two mutually recursive predicates, connected by two clauses, while the first program has not only this connection, but also self-loop on one of the predicates.

Intuitively, a set of adornments of a predicate p is *guard-tuned* if for every adornment a in it and every clause c of the program defining p the conjunction of the maximal prefix of c and a is either *false* or the conjunction is identical to a . We will see that the set of adornments we preferred in the discussion above is guard-tuned, while the second set is not.

Definition 11. *Let P be a partially normalised program, let p be a predicate in P , and let \mathcal{A}_p be a set of adornments for p . We say that \mathcal{A}_p is guard-tuned if for every $A \in \mathcal{A}_p$ and for every rule $r \in P$ with the symbolic condition c corresponding to the maximal prefix of r holds that either $c \wedge A = \text{false}$ or $c \wedge A = A$.*

Example 13. Consider the sets of adornments presented in Example 7. The first set of adornments is not guard-tuned while the second one is guard-tuned. \square

4.2 How to construct a guard-tuned set of adornments?

In this subsection we present a technique allowing to construct a guard-tuned set of adornments for a predicate p given a program P . To do so, recall once more Examples 7 and 13. They suggest two ways of constructing such a set. The first one is: given a program P collect the symbolic conditions, corresponding to the maximal prefixes of the rules defining p (we denote this set \mathcal{C}_p) and add the completion of the constructed disjunction. Unfortunately, this set of conditions is not necessarily a set of adornments and if so, it is not necessary guard-tuned.

Example 14. Consider the following program.

$$\begin{aligned} r(X) &\leftarrow X > 5. \\ r(X) &\leftarrow X > 10, r(X). \end{aligned}$$

Two sets of symbolic conditions can be constructed in the way described above: $\{r^{\$1 \leq 5}, r^{\$1 > 5}, r^{\$1 > 10}\}$ which is not a set of adornments and $\{r^{\$1 \leq 5}, r^{\$1 > 5}\}$ which is not guard-tuned due to the second rule of the program. \square

Thus, we are going to use a different approach. Once more, we start by finding \mathcal{C}_p . Let $\mathcal{C}_p = c_1, \dots, c_n$, then let \mathcal{A}_p be the set of conjunctions of c_i 's and their negations. We claim that the constructed set is always a guard-tuned set of adornments. Before stating this formally, consider once more Example 14.

Example 15. In this case, the symbolic conditions corresponding to the maximal prefixes of the rules are $\$1 > 5$ and $\$1 > 10$. Thus, the adornments are: $\$1 > 5 \wedge \$1 > 10, \$1 \leq 5 \wedge \$1 > 10, \$1 > 5 \wedge \$1 \leq 10, \$1 \leq 5 \wedge \$1 \leq 10$. After simplifying and removing the inconsistent conjuncts: $\$1 > 10, \$1 > 5 \wedge \$1 \leq 10, \$1 \leq 5$. \square

Lemma 3. *Let P be a program, p be a predicate in P and \mathcal{A}_p be constructed as described. Then \mathcal{A}_p is a guard-tuned set of adornments.*

Proof. The proof is immediate by checking the definitions \blacksquare

4.3 How to define a level mapping?

One of the questions that should be answered is how the level mappings should be generated automatically. The problem with defining level mappings is that they should reflect changes on possibly negative integer arguments, on the one hand, and remain non-negative, on the other. We also like to remain in the framework of level mappings on atoms defined as linear combinations of sizes of their arguments.

We are going to solve this problem by defining different level mappings for different adorned versions of the predicate. The major observation underlying the technique presented in this subsection is that if $\$1 > \2 appears in the adornment of a recursive clause, then for each call to this adorned predicate $\$1 - \2 will be positive, and thus, can be used for defining a level mapping. More formally:

Definition 12. Let $p^{E_1 \rho E_2}$ be an adorned predicate, where E_1 and E_2 are expressions and $\rho \in \{>, \geq\}$. The primitive level mapping is defined as:

$$\begin{cases} (E_1 - E_2)(t_1, \dots, t_n) & \text{if } E_1(t_1, \dots, t_n) \rho E_2(t_1, \dots, t_n) \\ 0 & \text{otherwise} \end{cases}$$

In most of the examples more than one conjunct will appear in the adornment. In this case the level mapping is defined as a linear combination of primitive level mappings corresponding to the conjuncts. Some of the conjuncts may actually be disjunctions—they are ignored, since disjunctions can be introduced only by the fact the some rule *cannot* be applied.

Definition 13. Let p^c be an adorned predicate, The natural level mapping is defined as:

$$|p^c(t_1, \dots, t_n)| = \sum_{E_1 \rho E_2 \in c} c_{E_1 \rho E_2} |p^{E_1 \rho E_2}(t_1, \dots, t_n)|^{PI},$$

where c 's are natural coefficients, E_1 and E_2 are expressions and $\rho \in \{>, \geq\}$.

Example 16. The level mappings used in Example 12 are natural level mappings with the following coefficients: $c_{\$1 > 1} = 0$, $c_{\$1 < 1000} = 1$, $c_{\$1 < -1} = 0$, $c_{\$1 > -1000} = 1$. For $p^{(\$1 \leq -1000) \vee (-1 \leq \$1 \leq 1) \vee (\$1 \geq 1000)}$ the definition holds trivially. \square

Technique developed by Decorte *et al.* [9] allows to define symbolic counterparts of the level mappings and to infer the actual values of the coefficients by solving a system of constraints.

4.4 Putting it all together and inferring termination constraints

In this section we show how the steps studied so far can be combined to an algorithm that allows to infer termination conditions. Intuitively, one starts with a termination condition initialised to be *true*, i.e., assuming that query Q terminates with respect to a program P for all possible values of integer arguments. Constraints are constructed similarly to [9]. If these can be satisfied without imposing any additional constraints on the integer variables, stop and report termination for the condition constructed so far. If these impose some constraints involving a new integer variable, repeat the process. If neither of the cases hold, stop and report possibility of non-termination.

Any other technique proving termination and being able not only to claim “termination can be proved” or “termination cannot be proved” but also providing in the latter case some constraint that, if satisfied, implies termination can be used instead of [9]. The algorithm is presented in Figure 1.

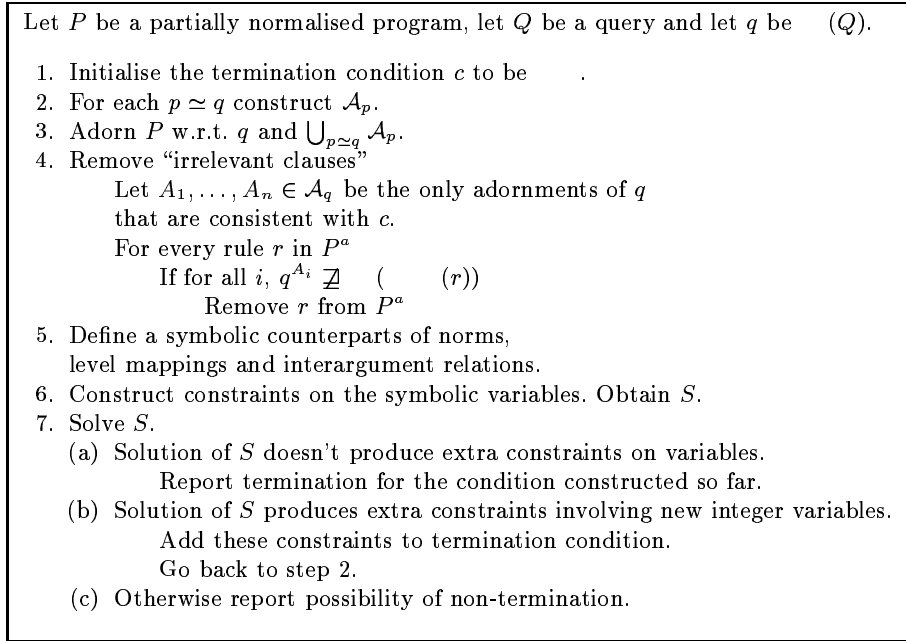


Fig. 1. Termination Inference Algorithm

Example 17. Consider the following program.

$$q(X, Y) \leftarrow X > Y, Z \text{ is } X - Y, Y1 \text{ is } Y + 1, q(Z, Y1).$$

We are interested in finding values of X and Y such that $q(X, Y)$ terminates. The first step of our algorithm is inferring the sets of adornments. There is only one inequality in the clause, i.e., $X > Y$. The corresponding symbolic constraint is $\$1 > \2 . Thus, the inferred adornment is $\{\$1 > \$2, \$1 \leq \$2\}$.

The adorned version of this program is

$$q^{\$1 > \$2}(X, Y) \leftarrow X > Y, Z \text{ is } X - Y, Y1 \text{ is } Y + 1, q^{\$1 > \$2}(Z, Y1).$$

$$q^{\$1 > \$2}(X, Y) \leftarrow X > Y, Z \text{ is } X - Y, Y1 \text{ is } Y + 1, q^{\$1 \leq \$2}(Z, Y1).$$

There is no clause defining $q^{\$1 \leq \$2}$. By Corollary 1, $\$1 \leq \2 is a termination condition. The recursive clause is analysed further. The level mapping is

$$|q^{\$1 > \$2}(X, Y)| = c_{\$1 > \$2} * \begin{cases} X - Y & \text{if } X > Y \\ 0 & \text{otherwise} \end{cases}$$

The acceptability decrease implies (see [9]):

$$c_{\$1 > \$2}(X - Y) > c_{\$1 > \$2}(X - Y) - c_{\$1 > \$2}Y,$$

that is $c_{\$1 > \$2}Y > 0$. Since $c_{\$1 > \$2} \geq 0$, $Y > 0$ should hold. Now we restart the whole process with respect to $Y > 0$. The following adorned program is obtained:

$$q^{\$1 > \$2, \$2 > 0}(X, Y) \leftarrow X > Y, Z \text{ is } X - Y, Y1 \text{ is } Y + 1, q^{\$1 > \$2, \$2 > 0}(Z, Y1)$$

$$q^{\$1 > \$2, \$2 \leq 0}(X, Y) \leftarrow X > Y, Z \text{ is } X - Y, Y1 \text{ is } Y + 1, q^{\$1 > \$2, \$2 \leq 0}(Z, Y1)$$

$$q^{\$1 > \$2, \$2 \leq 0}(X, Y) \leftarrow X > Y, Z \text{ is } X - Y, Y1 \text{ is } Y + 1, q^{\$1 \leq \$2, \$2 \leq 0}(Z, Y1)$$

Our assumption is that $\$2 > 0$. The second and the third clauses are “irrelevant” with respect to it. Thus, the only clause that should be analysed is the first one. The level mapping is thus, redefined as

$$|q^{\$1 > \$2}(X, Y)| = c_{\$1 > \$2} * \begin{cases} X - Y & \text{if } X > Y \\ 0 & \text{otherwise} \end{cases} + c_{\$2 > 0} * \begin{cases} Y & \text{if } Y > 0 \\ 0 & \text{otherwise} \end{cases}$$

Acceptability decreases imply that

$$c_{\$1 > \$2}(X - Y) + c_{\$2 > 0}Y > c_{\$1 > \$2}((X - Y) - Y) + c_{\$2 > 0}(Y + 1),$$

i.e., $c_{\$1 > \$2} Y > c_{\$2 > 0}$. Since Y is assumed to be positive this can be satisfied by taking $c_{\$1 > \$2} = 1$ and $c_{\$2 > 0} = 0$. This solution does not impose additional constraints on integer variables. Thus, the analysis terminates reporting $\$1 \leq \$2 \vee (\$1 > \$2 \wedge \$2 > 0)$ as a termination condition. \square

In order to prove correctness of this algorithm we have to prove its termination and partial correctness, i.e., that the symbolic condition constructed is a termination condition. Termination of the algorithm follows from termination of its steps discussed earlier and from the finiteness of the number of integer variables, restricting a number of possible jumps. Partial correctness follows from the correctness of transformations and the corresponding result of [9]. Observe that after removing “irrelevant clauses” the non-terminating (for some query) program might become terminating for it. However, this transformation expresses the meaning of termination condition: if c holds, clauses defining predicates such that their adornments are inconsistent with c can not be unified with Q . Observe also, that at the first traversal of the algorithm ($c = true$), if P does not have an unreachable code, this step does not change P^a .

5 Further extensions

In this section we discuss possible extensions of the algorithm presented above. First of all we re-consider inference of adornments, then we discuss integrating termination analysis of numerical and symbolic computations.

5.1 Once more about the inference of adornments

The set of adornments \mathcal{A}_p , inferred in Subsection 4.2 may sometimes be too weak for inferring precise termination conditions, as the following example illustrates.

Example 18. Consider the following program:

$$p(X, Y) \leftarrow X < 0, Y1 \text{ is } Y + 1, p(Y1, X).$$

The maximal prefix of the rule above is $X < 0$, thus, $\mathcal{A}_p = \{\$1 < 0, \$1 \geq 0\}$. The only termination condition that will be found is $\$1 \geq 0$, while the precise termination condition is $\$1 \geq 0 \vee (\$1 < 0 \wedge \$2 \geq -1)$. \square

The problem occurred due to the fact that \mathcal{A}_p restricts only *some* subset of integer argument positions, while for the termination proof information on integer arguments outside of this subset may be needed. Thus, we need to infer information on some variables, given some information on some other variables.

Definition 14. Let P be a program, let p be a predicate in P , let C_q be a set of symbolic constraints over the integer argument positions of q , and $C = \cup_{q \in P} C_q$. A symbolic constraint c over the integer argument positions of p is called an extension of C if exists $r \in P$, defining p , such that some integer argument position denominator appearing in c does not appear in C_p , and c is implied by some $c_q \in C_q$ for the recursive subgoals and some interargument relations for the non-recursive ones.

Let C be a set of symbolic constraints over the integer argument positions of p and let $\varphi(C)$ be $C \cup \{c \mid c \text{ is an extension of } C\}$. Define the set of adornments for p as $\{c'_1 \wedge \dots \wedge c'_n \mid c'_i \in \varphi^*(C_p) \text{ or } \neg c'_i \in \varphi^*(C_p)\}$, where $\varphi^*(C)$ is a fixpoint of powers of φ and C_p is defined as in Subsection 4.2.

Example 19. Consider once more Example 18. Symbolic comparison $\$1 < 0 \wedge \$2 < -1$ is the only extension of $C_p = \{\$1 < 0\}$, i.e., $\varphi(C_p) = \{\$1 < 0, \$1 < 0 \wedge \$2 < -1\}$. All integer argument positions denominators already appear in $\varphi(C_p)$. Thus, $\varphi^*(C_p) = \varphi(C_p)$ and the set of adornments is $\{\$1 < 0 \wedge \$2 < -1, (\$1 < 0 \wedge \$2 \geq -1) \vee \$1 \geq 0\}$. \square

An alternative approach to propagating such an information was suggested in [10]. To capture interaction between the variables a graph was constructed with integer argument positions as vertices and a “can influence”-relation as edges. This allows to propagate the existing adornments but not to infer the new ones and thus, is less precise than the approach presented in this subsection.

5.2 Integrating numerical and symbolic computation

As already claimed in the Introduction numerical computations form an essential part of almost any real-world program. Sometimes, such computation is “pure numerical”, that is does not involve any reasoning on the symbolic level, such as in the examples above. However, sometimes numerical computation is interleaved with a symbolic one as illustrated by the following example, collecting leaves of a tree with a variable branching factor, being a common data structure in natural language processing [15].

Example 20.

$$\begin{aligned} \text{collect}(X, [X|L], L) &\leftarrow \text{atomic}(X). \\ \text{collect}(T, L0, L) &\leftarrow \text{compound}(T), \text{functor}(T, _, A), \\ &\quad \text{process}(T, 0, A, L0, L). \\ \text{process}(_, A, A, L, L). \end{aligned}$$

$$\begin{aligned} & \text{process}(T, I, A, L0, L2) \leftarrow I < A, I1 \text{ is } I + 1, \text{arg}(I1, T, \text{Arg}), \\ & \text{collect}(\text{Arg}, L0, L1), \text{process}(T, I1, A, L1, L2). \end{aligned}$$

To prove termination of queries $\{\text{collect}(t, v, [])\}$, where t is a tree and v is a variable, the following three decreases should be shown: between a call to *collect* and a call to *process* in the second clause, between a call to *process* and a call to *collect* in the fourth one and between two calls to *process* in the same clause. The first and the second decreases can be shown only by a symbolic level mapping, the third one—only by the numerical approach. \square

Thus, our goal is to *combine* the existing symbolic approaches with the numerical one presented so far. One of the possible ways to do so is to combine two level mappings, $|\cdot|_1$ and $|\cdot|_2$, for example, by mapping each atom $A \in B_P^E$ to a pair of natural numbers $(|A|_1, |A|_2)$. Then an ordering relation on the atoms can be defined, based on the lexicographic ordering of such pairs. Well-foundedness of the natural numbers implies that this ordering is well-founded and the framework of term-acceptability [18], allows to reason on termination of programs in terms of decreases on such orderings.

Example 21. Continue Example 20. Let $\varphi : B_P^E \rightarrow (\mathcal{N} \cup \mathcal{N}^2)$ be a following mapping: $\varphi(\text{collect}(t, l0, l)) = \|t\|$, $\varphi(\text{process}(t, i, a, l0, l)) = (\|t\|, a - i)$ where $\|\cdot\|$ is a term-size norm. Then, the three decreases are satisfied with respect to $>$, such that $A_1 > A_2$ if and only if $\varphi(A_1) \succ \varphi(A_2)$, where \succ is the lexicographic ordering on $\mathcal{N} \cup \mathcal{N}^2$. \square

This integrated approach allows to analyse correctly examples such as *ground*, *unify*, *numbervars* [19] and Example 6.12 in [10]. Moreover, some numerical examples, such as Ackermann's function, that cannot be analysed by extending [9] due to the limitations of level mappings defined as linear combinations, can be analysed by the integrated approach.

6 Conclusion

Termination of numerical computations was studied by a number of authors [1, 2, 10]. Apt *et al.* [2] provided a declarative semantics, so called Θ -semantics, for Prolog programs with first-order built-in, including arithmetic operations. In this framework the property of strong termination, i.e., finiteness of all LD-trees for all possible goals, was completely characterised based on appropriately tuned notion of acceptability. This approach provides important theoretical results, but seems to be difficult

to integrate in automatic tools. In [1] it is claimed that an unchanged acceptability condition can be applied to programs in pure Prolog with arithmetic by defining the level mappings on ground atoms with the arithmetic relation to be zero. This approach ignores the actual computation, and thus, its applicability is restricted to programs using some arithmetics but not really relying on them, such as *quicksort*. Moreover, as Example 17 illustrates there are many programs that terminate only for *some* queries. Alternatively, Dershowitz *et al.* [10] extended the query-mapping pairs formalism of [12] to deal with numerical computations. However, this approach inherited the disadvantages of [12], such as high computational price, as well.

More research has been done on termination analysis for constraint logic programs [5, 14, 16, 17]. Since numerical computations in Prolog should be written in a way that allows a system to verify their satisfiability we can see numerical computations of Prolog as an *ideal constraint system*. Thus, all the results obtained for ideal constraints systems can be applied. Unfortunately, the research was either oriented towards theoretical characterisations [16, 17] or restricted to domains isomorphic to \mathcal{N} , such as trees and terms [14].

In a contrast to the approach of [10] that was restricted to verifying termination, we presented a methodology for *inferring* termination conditions. It is not clear whether and how [10] can be extended to infer such conditions. A main contribution of this work to the theoretical understanding of termination of numerical computations is in situating them in the well-known framework of acceptability and allowing integration with the existing approaches to termination of symbolic computations. The methodology presented can be integrated in automatic termination analysers, such as [9].

The kernel technique is powerful enough to analyse correctly examples such as *gcd* and *mod* [10], all examples appearing in the dedicated to arithmetic Chapter 8 of [19], also being a superset of arithmetical examples appearing in [1]. Moreover, our approach gains its power from the underlying framework of [9] and thus, allows to prove termination of some examples that cannot be analysed correctly by [10], similar to *confused delete* [6, 9]. The extended technique, presented in Section 5, allows to analyse correctly examples such as Ackermann's function, *ground*, *unify*, *numbervars* [19] and Example 6.12 in [10].

As a future work we consider a complete implementation of the algorithm. Due to the use of the constraint solving techniques we expect it both to be powerful and highly efficient.

References

1. K. R. Apt. *From Logic Programming to Prolog*. Prentice-Hall Int. Series in Computer Science. Prentice Hall, 1997.
2. K. R. Apt, E. Marchiori, and C. Palamidessi. A declarative approach for first-order built-in's in prolog. *Applicable Algebra in Engineering, Communication and Computation*, 5(3/4):159–191, 1994.
3. A. Bossi and N. Cocco. Basic transformation operations which preserve computed answer substitutions of logic programs. *J. Logic Programming*, 16:47–87, May 1993.
4. A. Bossi and N. Cocco. Preserving universal termination through unfold/fold. In G. Levi and M. Rodríguez-Artalejo, editors, *Algebraic and Logic Programming*, pages 269–286. Springer Verlag, 1994. LNCS 850.
5. L. Colussi, E. Marchiori, and M. Marchiori. On termination of constraint logic programs. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming - CP'95*, pages 431–448. Springer Verlag, 1995. LNCS 976.
6. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *J. Logic Programming*, 19/20:199–260, May/July 1994.
7. D. De Schreye, K. Verschaeftse, and M. Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. In I. Staff, editor, *Proc. of the Int. Conf. on Fifth Generation Computer Systems.*, pages 481–488. IOS Press, 1992.
8. S. Decorte and D. De Schreye. Termination analysis: some practical properties of the norm and level mapping space. In J. Jaffar, editor, *Proc. of the 1998 Joint Int. Conf. and Symp. on Logic Programming*, pages 235–249. MIT Press, June 1998.
9. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1137–1195, November 1999.
10. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Appl. Algebr. Eng. Commun. Comput.*, 2001. accepted.
11. T. Kawamura and T. Kanamori. Preservation of stronger equivalence in unfold/fold transformation. *Theoretical Computer Science*, 75(1&2):139–156, Sept. 1990.
12. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In L. Naish, editor, *Proc. of the Fourteenth Int. Conf. on Logic Programming*, pages 63–77. MIT Press, July 1997.
13. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Unfolding mystery of the *mergesort*. In N. Fuchs, editor, *Proc. of the Seventh Int. Workshop on Logic Program Synthesis and Transformation*. Springer Verlag, 1998. LNCS 1463.
14. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In M. Maher, editor, *Proc. JICSLP'96*, pages 7–21. The MIT Press, 1996.
15. C. Pollard and I. A. Sag. *Head-driven Phrase Structure Grammar*. The University of Chicago Press, 1994.
16. S. Ruggieri. Termination of constraint logic programs. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97*, pages 838–848. Springer Verlag, 1997. LNCS 1256.
17. S. Ruggieri. *Verification and validation of logic programs*. PhD thesis, Università di Pisa, 1999.

18. A. Serebrenik and D. De Schreye. Non-transformational termination analysis of logic programs, based on general term-orderings. In K.-K. Lau, editor, *Pre-Proceedings of Tenth International Workshop on Logic-based Program Synthesis and Transformation, 2000*, pages 45–54. University of Manchester, 2000. Department of Computer Science, Univ. of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1, URL : <http://www.cs.man.ac.uk/cstechrep/titles00.html>.
19. L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1994.