TLP **4** (5 & 6): 719–751, 2004. © 2004 Cambridge University Press
DOI: 10.1017/S1471068404002042 Printed in the United Kingdom

Inference of termination conditions for numerical loops in Prolog

ALEXANDER SEREBRENIK and DANNY DE SCHREYE
Department of Computer Science, K.U. Leuven, Celestijnenlaan 200A, B-3001, Heverlee, Belgium
(*e-mail:* {Alexander.Serebrenik,Danny.DeSchreye}@cs.kuleuven.ac.be)

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 overcoming these difficulties. Our approach is based on transforming a program in a 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 us to perform a correct analysis of such computations automatically, by extending previous work on a constraint-based approach to termination. Finally, we discuss possible extensions of the technique, including incorporating general term orderings.

KEYWORDS: termination analysis, numerical computation

1 Introduction

One of the important aspects in verifying the correctness of logic programs (as well as functional programs and term rewrite systems) is verification of termination. Due to the declarative formulation of programs, the danger of non-termination may be increased. As a result, termination analysis has received considerable attention in logic programming (Apt *et al.*, 1994; Bossi *et al.*, 2002; Bruynooghe *et al.*, 2002; Codish and Taboch, 1999; Colussi *et al.*, 1995; Decorte *et al.*, 1999; Dershowitz *et al.*, 2001; Genaim *et al.*, 2002; Lindenstrauss and Sagiv, 1997; Mesnard, 1996; Mesnard and Ruggieri, 2003; Plümer, 1991; Ruggieri, 1997; Verbaeten *et al.*, 2001).

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 paper we concentrate on automatic termination inference for logic programs depending on numerical computations.

Dershowitz *et al.* (2001) showed that termination of general numerical computations, for instance on floating point numbers, may be counter-intuitive, i.e. the observed behaviour does not necessarily coincide with the theoretically expected

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

one. Moreover, as the following program shows, similar results can be obtained even if the built-in predicates of the underlying language are restricted to include "greater than" and multiplication only.

Example 1

Consider the following program, that given a positive number x results in a sequence of calls $p(0.25x), p((0.25)^2x), \ldots$

$$p(X) \leftarrow X > 0, X1 \ \ is \ \ X * 0.25, p(X1).$$

If we reason purely in terms of real numbers, we might expect that the computation started by $p(1.0)$ will be infinite. However, in practice the goal above terminates with respect to this program, since there exists k, such that $(0.25)^k$ is small enough for the comparison $(0.25)^k > 0$ to fail. □

We discuss these issues in detail in Serebrenik and De Schreye (2002). In the current paper we avoid these complications by restricting to integer computations only.

Next, we illustrate the termination problem for integer computations with the following example:

Example 2

Consider the following program:

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

This program terminates for queries $p(X)$, for all integer values of X. □

Most of the existing automated approaches to termination analysis for logic programs (Codish and Taboch, 1999; Lindenstrauss and Sagiv, 1997; Mesnard and Neumerkel, 2001; Ohlebusch, 2001) fail to prove termination for such examples. The reason is that they are most often based on the notion of a *level mapping*, that is, a function from the set of all possible atoms to the natural numbers, which should decrease while traversing the rules. Usually level mappings are defined to depend on the structure of terms and to ignore constants, making the analysis of Example 2 impossible.

Of course, this can be easily repaired, by considering level mappings that map each natural number to itself. In fact, the kernels of two termination analysers for logic programs, namely cTI (Mesnard, 1996; Mesnard and Neumerkel, 2001) and TerminWeb (Codish and Taboch, 1999), rely on abstracting logic programs to CLP($\mathcal{N}$) programs, and use the identity level mapping on $\mathcal{N}$ in the analysis of the abstract versions of the programs[1].

[1] We thank the anonymous referees for pointing this link to related work out to us.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Note however that this is insufficient for the analysis of Example 2. In fact, there remain two problems. First, the program in Example 2 is defined on a (potentially negative) integer argument. This means that we need a level mapping which is different from the identity function.

Two approaches for solving this problem are possible. First, one can change the definition of the level mapping to map atoms to integers. However, integers are not well-founded. 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. Continuing this line of thought one might consider mapping atoms to more general well-founded domains. In fact, already in the early days of program analysis (Floyd, 1967; Katz and Manna, 1975) general well-founded domains were discussed. However, the growing importance of automatic termination analysers and requirements of robustness and efficiency stimulated researchers to look for more specific instances of well-founded domains, such as natural numbers in logic programming and terms in term-rewriting systems.

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 $p(X)$, $7 - X$ for $X < 7$, is greater than the size of $p(X1)$, $6 - X$ for $X < 7$ and 0 for $X \geqslant 7$, thus, proving termination.

A second problem with approaches based on the identity function, as the level mapping used on CLP($\mathcal{N}$), is that, even if the program in Example 2 would have been defined on natural values of X only, they would still not be able to prove termination. The reason is that the natural argument increases under the standard ordering of the natural numbers. Such bounded increases (be it of structure-sizes or of numerical values) are not dealt with by standard termination analysers. Note that the two approaches presented above also solve this second problem.

The main contribution of this paper is that we provide a transformation – similar to multiple specialisation (Winsborough, 1992) – that allows us to define level mappings of the form illustrated in the second approach above in an automatic way. To do so, we incorporate techniques of Dershowitz *et al.* (2001), such as level mapping inference, in the framework of the acceptability with respect to a set (De Schreye *et al.*, 1992; Decorte and De Schreye, 1998). 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.* (1999).

Moreover, we also be somewhat more general than Decorte *et al.* (1999), by studying the problem of termination inference, rather than termination verification. More precisely, we shall be inferring conditions that, if imposed on the queries, will ensure that the queries will terminate. Inference of termination conditions was studied elsewhere (Mesnard, 1996; Mesnard and Neumerkel, 2001; Genaim and Codish, 2001). Unlike termination conditions inferred by these approaches, stated in terms of groundedness of arguments, our technique produces conditions based on domains of the arguments, as shown in Example 3.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Example 3
Extend the program of Example 2 with the following clause:

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

This extended program terminates for $X \leqslant 7$ and this is the condition we will infer. □

The rest of the paper is organised as follows. 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. Section 5 contains the results of an experimental evaluation of the method. In Section 6 we discuss further extensions, such as proving termination of programs depending on numerical computations as well as symbolic ones. We summarise our contribution in Section 7, review related work and conclude.

2 Preliminaries

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. $Atom_P$ ($Term_P$) denotes the set of all atoms (terms) that can be constructed from the language underlying P. The extended Herbrand Base B_P^E (the extended Herbrand Universe U_P^E) is the quotient set of $Atom_P$ ($Term_P$) modulo the variant relation. An SLD-tree constructed using the left-to-right selection rule of Prolog is called an LD-tree. A goal G *LD-terminates* for a program P, if the LD-tree for (P, G) is finite.

The following definition is similar to Definition 6.30 (Apt, 1997).

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.
- p *depends on* q *in* P and write $p \sqsupseteq q$, if (p, q) is in the transitive closure of the relation *refers to*.
- p *and* q *are mutually recursive* and write $p \simeq q$, if $p \sqsupseteq q$ and $q \sqsupseteq p$.

The only difference between our definition and the one by Apt (1997) is that we require the relation $\sqsupseteq$ to be the *transitive* closure of the relation *refers to*, while Apt (1997) requires it to be *transitive, reflexive* closure. Using our definition we call a predicate p *recursive* if $p \simeq p$ holds.

We recall some basic notions, related to termination analysis. A *level mapping* is a function $|\cdot| : B_P^E \to \mathcal{N}$, where $\mathcal{N}$ is the set of the naturals. Similarly, a *norm* is a function $\|\cdot\| : U_P^E \to \mathcal{N}$.

We study termination of programs with respect to sets of queries. The following notion is one of the most basic notions in this framework.

Definition 2
Let P be a definite program and S be a set of atomic queries. The *call set*, $Call(P, S)$, is the set of all atoms A from the extended Herbrand Base B_P^E, such that a variant

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

of A is a selected atom in some derivation for $P \cup \{\leftarrow Q\}$, for some $Q \in S$ and under the left-to-right selection rule.

The following definition (Serebrenik and De Schreye, 2001) generalises the notion of acceptability with respect to a set (De Schreye *et al.*, 1992; Decorte and De Schreye, 1998) by extending it to mutual recursion.

Definition 3
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 $A \in Call(P,S)$
- for any clause $A' \leftarrow B_1,\ldots,B_n$ in P, such that $\mathrm{mgu}(A,A') = \theta$ exists,
- for any atom B_i, such that $rel(B_i) \simeq rel(A)$ and for any computed answer substitution σ for $\leftarrow (B_1,\ldots,B_{i-1})\theta$ holds that

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

De Schreye *et al.* (1992) characterise LD-termination in terms of acceptability.

Theorem 1 (*cf. De Schreye* et al. *(1992)*)
Let P be a definite 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 notions of rigidity and of interargument relations. Given a norm $\|\cdot\|$ and a term t, Bossi *et al.* (1991) call t *rigid* with respect to $\|\cdot\|$ if for any substitution σ, $\|t\sigma\| = \|t\|$. Observe that ground terms are rigid with respect to all norms. The notion of rigidity is obviously extensible to atoms and level mappings. Interargument relations have initially been studied by Ullman and van Gelder (1988), Plümer (1991) and Verschaetse and De Schreye (1991). In this paper we use the definition of Decorte *et al.* (1999).

Definition 4
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 $p(t_1,\ldots,t_n) \in Atom_P$ if $P \models p(t_1,\ldots,t_n)$ then $(\|t_1\|,\ldots,\|t_n\|) \in R_p$.

Combining the notions of rigidity, acceptability and interargument relations allows us to reason on termination completely at the clause level.

Theorem 2 (*rigid acceptability (cf. Decorte* et al. *(1999))*)
Let S be a set of atomic queries and P a definite program. Let $\|\cdot\|$ be a norm and, for each predicate p in P, let R_p be a valid interargument relation for p with respect to $\|\cdot\|$. If there exists a level mapping $|\cdot|$ which is rigid on $Call(P,S)$ such that

- for any clause $H \leftarrow B_1,\ldots,B_n \in P$, and
- for any atom B_i in its body such that $rel(B_i) \simeq rel(H)$,
- for substitution θ such that the arguments of the atoms in $(B_1,\ldots,B_{i-1})\theta$ all satisfy their associated interargument relations $R_{B_1},\ldots,R_{B_{i-1}}$:

$$|H\theta| > |B_i\theta|$$

then P is acceptable with respect to S.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

3 Methodology

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

Computing a query with respect to the following example results in a sequence of calls with oscillating arguments like $p(-2), p(4), p(-16), \ldots$ and stops if the argument is greater than 1000 or smaller than -1000. The treatment is done first on the intuitive level.

Example 4
We are interested in proving termination of the set of queries $\{p(z) \mid z \text{ is an integer}\}$ with respect to the following program:

$$p(X) \leftarrow X > 1, X < 1000, X1 \; is \; -X * X, p(X1).$$
$$p(X) \leftarrow X < -1, X > -1000, X1 \; is \; X * X, p(X1).$$

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 $-1000 < X < -1$. Thus, termination of $p(X)$ for $X \leqslant -1000$, $-1 \leqslant X \leqslant 1$ or $X \geqslant 1000$ is trivial. Moreover, if the first clause is applied and $1 < X < 1000$ holds, then either $-1000 < X1 < -1$ or $X1 \leqslant -1000 \vee -1 \leqslant X1 \leqslant 1 \vee X1 \geqslant 1000$ should hold. Similarly, if the second clause is applied and $-1000 < X < 1$ holds, either $1 < X1 < 1000$ or $X1 \leqslant -1000 \vee -1 \leqslant X1 \leqslant 1 \vee X1 \geqslant 1000$ should hold.

We use this observation and split the domain of the argument of p, denoted p_1, in three parts as following:

a $\quad 1 < p_1 < 1000$
b $\quad -1000 < p_1 < -1$
c $\quad p_1 \leqslant -1000 \vee -1 \leqslant p_1 \leqslant 1 \vee p_1 \geqslant 1000$

Next we replace the predicate p with three new predicates p^{a}, p^{b} and p^{c}. We add conditions before the calls to p to ensure that p^{a} is called if $p(X)$ is called and $1 < X < 1000$ holds, p^{b} is called if $p(X)$ is called and $-1000 < X < -1$ holds and p^{c} is called if $p(X)$ is called and $X \leqslant -1000 \vee -1 \leqslant X \leqslant 1 \vee X \geqslant 1000$ holds. The following program is obtained:

$$p^{\mathsf{a}}(X) \leftarrow X > 1, X < 1000, X1 \; is \; -X * X,$$
$$\quad -1000 < X1, X1 < -1, p^{\mathsf{b}}(X1).$$
$$p^{\mathsf{a}}(X) \leftarrow X > 1, X < 1000, X1 \; is \; -X * X,$$
$$\quad (X1 \leqslant -1000; (-1 \leqslant X1, X1 \leqslant 1); X1 \geqslant 1000), p^{\mathsf{c}}(X1).$$
$$p^{\mathsf{b}}(X) \leftarrow X < -1, X > -1000, X1 \; is \; X * X,$$
$$\quad 1 < X1, X1 < 1000, p^{\mathsf{a}}(X1).$$
$$p^{\mathsf{b}}(X) \leftarrow X < -1, X > -1000, X1 \; is \; X * X,$$
$$\quad (X1 \leqslant -1000; (-1 \leqslant X1, X1 \leqslant 1); X1 \geqslant 1000), p^{\mathsf{c}}(X1).$$

Observe that the transformation we performed is a form of multiple specialisation, well-known in the context of abstract interpretation (Winsborough, 1992).

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Now we define three *different* level mappings, one for atoms of p^a, another one for atoms of p^b and the last one for atoms of p^c. Let

$$
\begin{aligned}
| p^a(n) | &= \begin{cases} 1000 - n & \text{if } 1 < n < 1000 \\ 0 & \text{otherwise} \end{cases} \\
| p^b(n) | &= \begin{cases} 1000 + n & \text{if } -1000 < n < -1 \\ 0 & \text{otherwise} \end{cases} \\
| p^c(n) | &= 0
\end{aligned}
$$

We verify acceptability of the transformed program with respect to $\{p^a(n) \mid 1 < n < 1000\} \cup \{p^b(n) \mid -1000 < n < -1\}$ via the specified level mappings. This implies termination of the transformed program with respect to these queries, and thus, termination of the original program with respect to $\{p(z) \mid z \text{ is an integer}\}$.

For the sake of brevity we discuss only queries of the form $p^a(n)$ for $1 < n < 1000$. Heads of the first and the second clauses can be unified with this query, however, the second clause does not contain calls to predicates mutually recursive with p^a and the only such atom in the first clause is $p^b(m)$, where $m = -n^2$. Then, $| p^a(n) | > | p^b(m) |$ should hold, i.e., $1000 - n > 1000 + m$, that is $1000 - n > 1000 - n^2$ ($n > 1$ and $m = -n^2$), which is true for $n > 1$.

For queries of the form $p^b(n)$, the acceptability condition is reduced to $1000 + n > 1000 - n^2$ which is true for $n < -1$. □

The intuitive presentation above hints at the main issues to be discussed in the following sections: how the cases such as those above can be extracted from the program, and how given the extracted cases, 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 further analysis will be based on. Recall that the aim of our analysis is to find, given a predicate and a query, a sufficient condition for termination of this query with respect to this program. Thus, we need to define a notion of a termination condition. We start with a number of auxiliary definitions.

Given a predicate p, p_i denotes the ith argument of p and is called *argument position denominator*.

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 $p(t_1, \ldots, t_n) \in Call(P, S)$, t_i is an integer.

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

An *integer inequality* is an atom of one of the following forms $Exp1 > Exp2$, $Exp1 < Exp2$,$Exp1 \geqslant Exp2$ or $Exp1 \leqslant Exp2$, where $Exp1$ and $Exp2$ are constructed from integers, variables and the four operations of arithmetics on integers. A *symbolic inequality over the arguments of a predicate p* is constructed similarly to an integer

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

inequality. However, instead of variables, integer argument positions denominators are used.

Example 5
$X > 0$ and $Y \leqslant X + 5$ are integer inequalities. Given a predicate p of arity 3, having only integer argument positions, $p_1 > 0$ and $p_2 \leqslant p_1 + p_3$ are symbolic inequalities over the arguments of p. □

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

Definition 6
Let $p(t_1, \ldots, t_n)$ 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 $c_p(p(t_1, \ldots, t_n))$, is obtained by replacing the argument positions denominators with the corresponding arguments, i.e. p_i with t_i.

Example 6
Let $p(X, Y, 5)$ be an atom and let c_p be a symbolic condition $(p_1 > 0) \wedge (p_2 \leqslant p_1 + p_3)$. Then, $c_p(p(X, Y, 5))$ is $(X > 0) \wedge (Y \leqslant X + 5)$. □

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

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

For any integer z a termination condition for $p(z)$ with respect to Examples 2 and 4 is *true*, i.e. for any integer z, $p(z)$ terminates with respect to these programs. Clearly, more than one termination condition is possible for a given query with respect to a given program. For example, termination conditions for $p(5)$ with respect to Example 3, are among others, *true*, $p_1 \leqslant 7$ and $p_1 > 0$. Analogously, *false*, $p_1 \leqslant 7$, $p_1 < 10$ are termination conditions for $p(11)$ with respect to Example 3. It should also be noted that a disjunction of two termination conditions is always a termination condition.

Similarly to Theorems 1 and 2 we would like to consider termination with respect to sets of atomic queries. Therefore we extend the notion of termination condition to a set of queries. This, however, is meaningful only if all the queries of the set have the same predicate. We call such a set *single predicate set of atomic queries*. For a single predicate set of atomic queries S, $rel(S)$ denotes the predicate of the queries of the set.

Definition 8
Let P be a program, and S be a single predicate set of atomic queries. A symbolic condition $c_{rel(S)}$ is a *termination condition* for S if $c_{rel(S)}$ is a termination condition for all $Q \in S$.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

From the discussion above it follows that a termination condition for $S = \{p(z) \mid z \text{ is an integer}\}$ with respect to Examples 2 and 4 is *true*. This is not the case for Example 3, since termination is observed only for some queries of S, namely $p(z)$, such that $z \leqslant 7$. Thus, $p_1 \leqslant 7$ is a termination condition for S with respect to Example 3.

We discuss now inferring what values integer arguments can take during traversal of the rules, i.e. the "case analysis" performed in Example 4. It provides already the underlying intuition—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 a *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 9
Let p be a predicate. The set $\mathscr{A}_p = \{c_1, \ldots, c_n\}$ of symbolic conditions over the integer arguments of p is called *set of adornments for* p if $\bigvee_{i=1}^{n} c_i = true$ and for all i, j such that $1 \leqslant i < j \leqslant n$, $c_i \wedge c_j = false$.

A set of adornments partitions the domain for (some of) the integer variables of the predicate. Similarly to Example 4, in the examples to come, elements of a set of adornments are denoted $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$

Example 7
Example 4, continued. The following are examples of sets of adornments:

$$\{\mathsf{a}, \mathsf{b}, \mathsf{c}\} \quad \begin{array}{l} \text{where } \mathsf{a} \text{ is } 1 < p_1 < 1000, \mathsf{b} \text{ is } -1000 < p_1 < -1 \\ \text{and } \mathsf{c} \text{ is } p_1 \leqslant -1000 \vee -1 \leqslant p_1 \leqslant 1 \vee p_1 \geqslant 1000. \end{array}$$

and

$$\{\mathsf{d}, \mathsf{e}\} \text{ where } \mathsf{d} \text{ is } p_1 \leqslant 100 \text{ and } \mathsf{e} \text{ is } p_1 > 100.$$

□

In the next section we are going to present a transformation, related to the multiple specialisation technique. To define it formally we introduce the following definition:

Definition 10
Let $H \leftarrow B_1, \ldots, B_n$ be a rule. $B_1, \ldots, B_i$, is called *an integer prefix of the rule*, if for all j, $1 \leqslant j \leqslant i \leqslant n$, B_j is an integer inequality and the only variables in its arguments are variables of H. $B_1, \ldots, B_i$ is called *the maximal integer prefix* of the rule, if it is an integer prefix and $B_1, \ldots, B_i, B_{i+1}$ is not an integer prefix.

Since an integer 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 integer prefix is its instance with respect to the head. In general, this symbolic condition is not necessarily unique.

To guarantee the uniqueness of such symbolic conditions we require integer argument positions in the heads of the rules to be occupied by distinct variables. For the sake of simplicity we assume *all* argument positions in the heads of the rules to be occupied by distinct variables. Apt *et al.* (1994) call such a rule *homogeneous*. Analogously, a logic program is called homogeneous if all its clauses

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

are homogeneous. Programs can be easily rewritten to a homogeneous form (see Apt *et al.* (1994)). In the following we assume that all programs are homogeneous.

3.2 Program transformation

The next question that should be answered is how the program should be transformed given a set of adornments. After this transformation $p^c(X_1,\ldots,X_n)$ will behave with respect to the transformed program exactly as $p(X_1,\ldots,X_n)$ 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.

Given a program P and a set of possible adornments $\mathscr{A} = \bigcup_{p\in P} \mathscr{A}_p$, the transformation is performed in a number of steps. Below we use Example 2 as a running example to illustrate the different steps. Recall that it consists of only one clause

$$p(X) \leftarrow X < 7, X1 \; is \; X + 1, p(X1).$$

As set of adornments we use

$$\mathscr{A}_p = \{\mathsf{a}, \mathsf{b}\}, \text{ where } \mathsf{a} \text{ is } p_1 < 7 \text{ and } \mathsf{b} \text{ is } p_1 \geqslant 7.$$

1. For each clause r in P and for each call $p(t_1,\ldots,t_n)$ to a recursive predicate p occurring in r add $\bigvee_{c\in\mathscr{A}_p} c(p(t_1,\ldots,t_n))$ before $p(t_1,\ldots,t_n)$. By Definition 9 the disjunction is *true*, thus, the transformed program is equivalent to the original one. In the example, the clause is transformed to

$$p(X) \leftarrow X < 7, X1 \; is \; X + 1, (X1 < 7 \; ; \; X1 \geqslant 7), p(X1).$$

2. For each clause, such that the head of the clause, say $p(t_1,\ldots,t_n)$, has a recursive predicate p, add $\bigvee_{c\in\mathscr{A}_p} c(p(t_1,\ldots,t_n))$ as the first subgoal in its body. As for the previous step, the introduced call is equivalent to *true*, so that the transformation is obviously correct. In the example, we obtain:

$$\begin{aligned} p(X) \leftarrow\; & (X < 7 \; ; \; X \geqslant 7), X < 7, \\ & X1 \; is \; X + 1, (X1 < 7 \; ; \; X1 \geqslant 7), p(X1). \end{aligned}$$

3. Next, moving to an alternative procedural interpretation of disjunction, for each clause in which we introduced a disjunction in one of the previous two steps, and for each such introduced disjunction $\bigvee_{c\in\mathscr{A}_p} c(p(t_1,\ldots,t_n))$ we split these disjunctions, introducing a separate clause for each disjunct. Thus, we apply the transformation

$$H \leftarrow B_1,\ldots,(A_1 \; ; \; \ldots \; ; \; A_k),\ldots,B_n.$$

to

$$\begin{aligned} H &\leftarrow B_1,\ldots,A_1,\ldots,B_n. \\ H &\leftarrow B_1,\ldots,A_2,\ldots,B_n. \\ &\vdots \\ H &\leftarrow B_1,\ldots,A_k,\ldots,B_n. \end{aligned}$$

to each disjunction introduced in steps 1 and 2.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

For our running example, we obtain four clauses:

$$p(X) \leftarrow X < 7, X < 7, X1 \; is \; X + 1, X1 < 7, p(X1).$$
$$p(X) \leftarrow X < 7, X < 7, X1 \; is \; X + 1, X1 \geqslant 7, p(X1).$$
$$p(X) \leftarrow X \geqslant 7, X < 7, X1 \; is \; X + 1, X1 < 7, p(X1).$$
$$p(X) \leftarrow X \geqslant 7, X < 7, X1 \; is \; X + 1, X1 \geqslant 7, p(X1).$$

Note that, although this transformation is logically correct, it is *not* correct for Prolog programs with non-logical features. For instance, in the presence of "cut", it may produce a different computed answer set. Also, in the context of "read" or "write" calls, the procedural behaviour may become very different. However, for purely logical programs with integer computations, both the declarative semantics and the computed answer semantics are preserved. Likewise, the termination properties are also preserved. Indeed, the transformation described can be seen as a repeated unfolding of ; using the following clauses:

$$;(X, Y) \leftarrow X.$$
$$;(X, Y) \leftarrow Y.$$

It is well-known that unfolding cannot introduce infinite derivations (Bossi and Cocco, 1994). On the other hand, an infinite derivation of the original program can be easily mimicked by the transformed program.

From here on we will restrict our attention to purely logical programs, augmented with integer arithmetic. To prepare the next step in the transformation, note that, in the program resulting from step 3, for each rule r and for each recursive predicate p:

- if a call $p(t_1, \ldots, t_n)$ occurs in r, then it is immediately preceded by some $c(p(t_1, \ldots, t_n))$,
- if an atom $p(t_1, \ldots, t_n)$ occurs as the head of r, then it is immediately followed by some $c(p(t_1, \ldots, t_n))$.

Moreover, since the elements $\mathscr{A}_p$ partition the domain (see Definition 9), conjuncts like $c_i(p(t_1, \ldots, t_n)), p(t_1, \ldots, t_n)$ and $c_j(p(t_1, \ldots, t_n)), p(t_1, \ldots, t_n)$ for $i \neq j$, are mutually exclusive, as well as the initial parts of the rules, like

$$p(t_1, \ldots, t_n) \leftarrow c_i(p(t_1, \ldots, t_n)) \text{ and } p(t_1, \ldots, t_n) \leftarrow c_j(p(t_1, \ldots, t_n)),$$

$i \neq j$. This means that we can now safely rename the different cases apart.

4. Replace each occurrence of $c(p(t_1, \ldots, t_n)), p(t_1, \ldots, t_n)$ in the body of the clause with $c(p(t_1, \ldots, t_n)), p^c(t_1, \ldots, t_n)$ and each occurrence of a rule

$$p(t_1, \ldots, t_n) \leftarrow c(p(t_1, \ldots, t_n)), B_1, \ldots, B_n$$

with a corresponding rule

$$p^c(t_1, \ldots, t_n) \leftarrow c(p(t_1, \ldots, t_n)), B_1, \ldots, B_n.$$

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

In our example we get:

$$p^{\mathsf{a}}(X) \leftarrow X < 7, X < 7, X1 \; is \; X + 1, X1 < 7, p^{\mathsf{a}}(X1).$$
$$p^{\mathsf{a}}(X) \leftarrow X < 7, X < 7, X1 \; is \; X + 1, X1 \geqslant 7, p^{\mathsf{b}}(X1).$$
$$p^{\mathsf{b}}(X) \leftarrow X \geqslant 7, X < 7, X1 \; is \; X + 1, X1 < 7, p^{\mathsf{a}}(X1).$$
$$p^{\mathsf{b}}(X) \leftarrow X \geqslant 7, X < 7, X1 \; is \; X + 1, X1 \geqslant 7, p^{\mathsf{b}}(X1).$$

Because of the arguments presented above, the renaming is obviously correct, in the sense that the LD-trees that exist for the given program and for the renamed program are identical, except for the names of the predicates and for a number of failing 1-step derivations (due to entering clauses that fail in their guard in the given program). As a result, both the semantics (up to renaming) and the termination behaviour of the program are preserved.

5. Remove all rules with a maximal integer prefix which is inconsistent, and remove from the bodies of the remaining clauses all subgoals that are preceded by an inconsistent conjunction of inequalities. In the example, both rules defining p^{b} are eliminated and we obtain:

$$p^{\mathsf{a}}(X) \leftarrow X < 7, X < 7, X1 \; is \; X + 1, X1 < 7, p^{\mathsf{a}}(X1).$$
$$p^{\mathsf{a}}(X) \leftarrow X < 7, X < 7, X1 \; is \; X + 1, X1 \geqslant 7, p^{\mathsf{b}}(X1).$$

Performing this step requires verifying the consistency of a set of constraints, a task that might be computationally expensive. Depending on the constraints the implementation of our technique is supposed to deal with, the programmer can either opt for more restricted but potentially faster solvers, such as linear rational solver (Holzbaur, 1995), or for more powerful but potentially slower ones, such as mixed integer programminging solver (ILOG, 2001).

6. Replace each rule

$$p^c(t_1, \ldots, t_n) \leftarrow c(p(t_1, \ldots, t_n)), B_1, \ldots, B_n$$

by a rule

$$p^c(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_n.$$

In the example we obtain:

$$p^{\mathsf{a}}(X) \leftarrow X < 7, X1 \; is \; X + 1, X1 < 7, p^{\mathsf{a}}(X1).$$
$$p^{\mathsf{a}}(X) \leftarrow X < 7, X1 \; is \; X + 1, X1 \geqslant 7, p^{\mathsf{b}}(X1).$$

which is the *adorned* program, $P^{\mathscr{A}}$ ($P^{\{\mathsf{a},\mathsf{b}\}}$ in our case). Note that this last step is only correct if we also transform the set of original queries. Namely, given a single predicate set of original atomic queries S for P and a set of adornments $\mathscr{A} = \bigcup_{p \in P} \mathscr{A}_p$, the corresponding set of queries considered for $P^{\mathscr{A}}$ is $S^{\mathscr{A}} = \{c_1(Q) \wedge Q^{c_1}, \ldots, c_n(Q) \wedge Q^{c_n} \mid Q \in S, \{c_1, \ldots, c_n\} = \mathscr{A}_{rel(Q)}\}$, where Q^c denotes $p^c(t_1, \ldots, t_n)$ if Q is $p(t_1, \ldots, t_n)$. In our running example the set of queries is $\{z < 7 \wedge p^{\mathsf{a}}(z), z \geqslant 7 \wedge p^{\mathsf{b}}(z) \mid z \text{ is an integer}\}$.

Before stating our results formally we illustrate the transformation by a second example.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Example 8
Example 4, continued. With the first set of adornments from Example 7 we obtain $P^{\{a,b,c\}}$:

$$p^a(X) \leftarrow X > 1, X < 1000, X1 \text{ is } -X * X, -1000 < X1, X1 < -1, p^b(X1).$$
$$p^a(X) \leftarrow X > 1, X < 1000, X1 \text{ is } -X * X, (X1 \leqslant -1000; (-1 \leqslant X1, X1 \leqslant 1); X1 \geqslant 1000), p^c(X1).$$
$$p^b(X) \leftarrow X < -1, X > -1000, X1 \text{ is } X * X, 1 < X1, X1 < 1000, p^a(X1).$$
$$p^b(X) \leftarrow X < -1, X > -1000, X1 \text{ is } X * X, (X1 \leqslant -1000; (-1 \leqslant X1, X1 \leqslant 1); X1 \geqslant 1000), p^c(X1).$$

The set of queries to be considered is

$$\{ \; z > 1 \wedge z < 1000 \wedge p^a(z), \; z > -1000 \wedge z < -1 \wedge p^b(z), \; (z \leqslant -1000 \vee (z \geqslant -1 \wedge z \leqslant 1) \vee z \geqslant 1000) \wedge p^c(z) \mid z \text{ is an integer} \; \}$$

If the second set of adornments is used, the program $P^{\{d,e\}}$ is obtained:

$$p^d(X) \leftarrow X > 1, X < 1000, X1 \text{ is } -X * X, X1 \leqslant 100, p^d(X1).$$
$$p^e(X) \leftarrow X > 1, X < 1000, X1 \text{ is } -X * X, X1 \leqslant 100, p^d(X1).$$
$$p^d(X) \leftarrow X < -1, X > -1000, X1 \text{ is } X * X, X1 \leqslant 100, p^d(X1).$$
$$p^d(X) \leftarrow X < -1, X > -1000, X1 \text{ is } X * X, X1 > 100, p^e(X1).$$

Analogously, the following is the set of the corresponding queries

$$\{z \leqslant 100 \wedge p^d(z), z > 100 \wedge p^e(z) \mid z \text{ is an integer}\}$$

□

Formally, the following lemma holds:

Lemma 1
Let P be a definite pure logical program with integer computations, let Q be an atomic query, let $\mathscr{A} = \cup \mathscr{A}_p$ be a set of adornments and let c be an adornment in $\mathscr{A}_{rel(Q)}$. Let $P^{\mathscr{A}}$ be a program obtained as described above with respect to $\mathscr{A}$. Then, c is a termination condition for Q with respect to P if and only if $P^{\mathscr{A}}$ LD-terminates with respect to $c(Q) \wedge Q^c$.

Proof
The construction of $P^{\mathscr{A}}$ implies that the LD-tree of $c(Q) \wedge Q^c$ with respect to $P^{\mathscr{A}}$ is isomorphic to the LD-tree of $c(Q) \wedge Q$ with respect to P, implying the theorem. □

Again, in practice we do not prove termination of a single query, but of a single predicate set of queries. Furthermore, recalling that a disjunction of

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

termination conditions is a termination condition itself we can generalise our lemma to disjunctions of adornments. Taking these two considerations into account, the following theorem holds.

Theorem 3
Let P be a definite pure logical program with integer computations, let S be a single predicate set of atomic queries, let $\mathscr{A} = \cup \mathscr{A}_p$ be a set of adornments and let $c_1, \ldots, c_n$ be adornments in $\mathscr{A}_{rel(S)}$. Let $P^{\mathscr{A}}$ be a program obtained as described above with respect to $\mathscr{A}$. Then, $c_1 \vee \ldots \vee c_n$ is a termination condition for S with respect to P if and only if $P^{\mathscr{A}}$ LD-terminates with respect to $\{c_1(Q) \wedge Q^{c_1}, \ldots, c_n(Q) \wedge Q^{c_n} \mid Q \in S\}$.

Proof
Immediately from Lemma 1 and the preceding observations. □

The goal of the transformation presented is, given a program and a partition of the domain, to generate a program having separate clauses for each one of the cases. Clearly, this may (and usually will) increase the number of clauses. Each clause can be replaced by maximum c^{n+1} new clauses, were c is a number of adornments and n is a number of recursive body subgoals. Thus, the size of the transformed program doesn't exceed

$$r \times c^{n+1}, \tag{1}$$

where r is the size of the original one. This may seem a problematically large increase, however, the number of recursive body atoms (depending on numerical arguments) in numerical programs is usually small.

Since the transformation preserves termination, acceptability of the transformed program implies termination of the original program. In the next section we will see that having separate clauses for different cases allows us to define less sophisticated level-mappings for proving termination. Such level-mappings can be constructed automatically, and thus, play a key role in automation of the approach.

4 Generating adornments, level mappings and termination constraints

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 a set of queries. In this section we discuss how adornments can be generated, how level mappings can be proposed and which termination conditions finally turn up.

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 9. However, recalling $P^{\{a,b,c\}}$ and $P^{\{d,e\}}$ as shown in Example 8, we conclude that $\{a, b, c\}$ is in some sense preferable to $\{d, e\}$. Observe that $P^{\{d,e\}}$ does not only have two mutually recursive predicates, as $P^{\{a,b,c\}}$ does, but also self-loops on one of the predicates. To distinguish between "better" and "worse" sets of adornments we define *guard-tuned* sets of adornments.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Intuitively, a set of adornments of a predicate p is *guard-tuned* if it is based on "subcases" of maximal integer prefixes.

Definition 11
Let P be a homogeneous program, let p be a predicate in P. A set of adornments $\mathscr{A}_p$ is called *guard-tuned* if for every $A \in \mathscr{A}_p$ and for every rule $r \in P$, defining p, with the symbolic condition c corresponding to its maximal integer prefix, either $c \wedge A = \mathit{false}$ or $c \wedge A = A$ holds.

Example 9
The first set of adornments, presented in Example 7, is guard-tuned while the second one is not guard-tuned. □

Examples 7 and 9 suggest the following way of constructing a guard-tuned set of adornments. First, we collect the symbolic conditions, corresponding to the maximal integer prefixes of the rules defining a predicate p (we denote this set $\mathscr{C}_p$). Let $\mathscr{C}_p$ be $\{c_1, \ldots, c_n\}$. Then we define $\mathscr{A}_p$ to be the set of all conjunctions $\wedge_{i=1}^{n} d_i$, where d_i is either c_i or $\neg c_i$. Computing $\mathscr{A}_p$ might be exponential in the number of elements of $\mathscr{C}_p$, i.e. in the number of maximal integer prefixes. The number of integer prefixes is bounded by the number of clauses. Thus, recalling (1), the upper bound on the size of the transformed program is $r \times 2^{r(n+1)}$, i.e. it is exponential in the number of clauses r and in a number of recursive subgoals n. However, again our experience suggests that numerical parts of real-world programs are usually relatively small and depend on one or two different integer prefixes. Analogously, clauses having more than two recursive body subgoals are highly exceptional. Therefore, we conclude that in practice the size of the transformed program is not problematic.

We claim that the constructed set $\mathscr{A}_p$ is always a guard-tuned set of adornments. Before stating this formally, consider the following example.

Example 10
Consider the following program.

$$r(X) \leftarrow X > 5.$$
$$r(X) \leftarrow X > 10, r(X).$$

Then, $\mathscr{C}_r = \{r_1 > 5, r_1 > 10\}$. The following conjunctions can be constructed from the elements of $\mathscr{C}_p$ and their negations: $\{r_1 > 5 \wedge r_1 > 10,\ r_1 > 5 \wedge \neg(r_1 > 10),\ \neg(r_1 > 5) \wedge r_1 > 10,\ \neg(r_1 > 5) \wedge \neg(r_1 > 10)\}$. After simplifying and removing inconsistencies $\mathscr{A}_r = \{r_1 > 10, r_1 > 5 \wedge r_1 \leqslant 10, r_1 \leqslant 5\}$. □

Lemma 2
Let P be a program, p be a predicate in P and $\mathscr{A}_p$ be constructed as described. Then $\mathscr{A}_p$ is a guard-tuned set of adornments.

Proof
The proof is done by checking the definitions.

1. Let $a_1, a_2 \in \mathscr{A}_p$ and $a_1 \neq a_2$. Then, there exists $c_i \in \mathscr{C}_p$, such that $a_1 = d_1 \wedge \ldots \wedge c_i \wedge \ldots \wedge d_n$ and $a_2 = d'_1 \wedge \ldots \wedge \neg c_i \wedge \ldots \wedge d'_n$. Thus, $a_1 \wedge a_2 = \mathit{false}$.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

2. By definition of $\mathscr{A}_p$, $\vee_{a_i \in \mathscr{A}_p} a_i = true$. Thus, $\mathscr{A}_p$ is a set of adornments.
3. Let $a \in \mathscr{A}_p$ be an adornment and let c be a symbolic condition corresponding to the maximal integer prefix of a rule. By definition of $\mathscr{C}_p$, $c \in \mathscr{C}_p$. Thus, either c is one of the conjuncts of a or $\neg c$ is one of the conjuncts of a. In the first case, $c \wedge a = a$. In the second case $c \wedge a = c \wedge (\neg c) = false$. Therefore, $\mathscr{A}_p$ is a guard-tuned set of adornments.

□

From here on we assume that all sets of adornments are guard-tuned.

4.2 How to define a level mapping

One of the questions that should be answered is how the level mappings should be generated automatically. Clearly, one cannot expect automatically defined level mappings to be powerful enough to prove termination of all terminating examples. In general we cannot hope but for a good guess.

The problem with level mappings is that they should reflect changes on possibly negative arguments and remain non-negative at the same time. We also like to remain in the framework of level mappings on atoms defined as linear combinations of sizes of their arguments (Bossi *et al.*, 1994). We 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 $p_1 > p_2$ appears in the adornment of a recursive clause, then for each call to this adorned predicate $p_1 - p_2$ will be positive, and thus, can be used for defining a level mapping. On the other hand, $p_1 < p_2$ can always be interpreted as $p_2 > p_1$. These observations form a basis for definition of a *primitive level mapping*.

Definition 12
Let p^c be an adorned predicate. The *primitive level mapping*, $|\cdot|^{pr}$, is defined as

- if c is $E_1 \ \rho \ E_2$, where E_1 and E_2 are expressions and ρ is either $>$ or $\geqslant$ then

$$|\ p^c(t_1,\ldots,t_n)\ |^{pr} = \begin{cases} (E_1 - E_2)(t_1,\ldots,t_n) & \text{if } E_1(t_1,\ldots,t_n)\ \rho\ E_2(t_1,\ldots,t_n) \\ 0 & \text{otherwise} \end{cases}$$

- if c is $E_1 \ \rho \ E_2$, where E_1 and E_2 are expressions and ρ is either $<$ or $\leqslant$ then

$$|\ p^c(t_1,\ldots,t_n)\ |^{pr} = \begin{cases} (E_2 - E_1)(t_1,\ldots,t_n) & \text{if } E_1(t_1,\ldots,t_n)\ \rho\ E_2(t_1,\ldots,t_n) \\ 0 & \text{otherwise} \end{cases}$$

- otherwise,

$$|\ p^c(t_1,\ldots,t_n)\ |^{pr} = 0.$$

If more than one conjunct appears in the adornment, the level mapping is defined as a linear combination of primitive level mappings corresponding to the conjuncts.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Definition 13
Let $p^{c_1 \wedge \ldots \wedge c_n}$ be an adorned predicate such that each c_i is $E_1^i \rho^i E_2^i$ for some expressions E_1^i and E_2^i and ρ^i is either $>$ or $\geqslant$. Let $w_{c_1}, \ldots, w_{c_n}$ be natural numbers. Then, a level mapping $|\cdot|$ satisfying

$$| p^{c_1 \wedge \ldots \wedge c_n}(t_1, \ldots, t_n) | = \sum_i w_{c_i} | p^{c_i}(t_1, \ldots, t_n) |^{pr},$$

is called a *natural level mapping*.

Example 11
The level mappings used in Example 4 are natural level mappings such that $w_{p_1>1} = w_{p_1<-1} = 0$, $w_{p_1<1000} = w_{p_1>-1000} = 1$. We have seen that these level mappings are powerful enough to prove termination. □

The definition of natural level mapping implies that if c is a disjunction, it is ignored. The reason for doing so is that disjunctions are introduced only as negations of symbolic constraints corresponding to maximal integer prefixes of the rules. Thus, they signify that some rule *cannot* be applied, and can be ignored.

Example 12
Example 4, continued. Recalling that c denotes $p_1 \leqslant -1000 \vee -1 \leqslant p_1 \leqslant 1 \vee p_1 \geqslant 1000$ the following holds for any integer n, $| p^{\mathsf{c}}(n) |^{pr} = 0$. □

Of course, if the original program already contains disjunctions of numerical constraints, then we transform it in a preprocessing to eliminate the disjunctions.

As the following example illustrates, natural level mappings gain their power from the fact that the set of adornments used is guard-tuned.

Example 13
In Example 7 we have seen two different sets of adornments. We have seen in Example 4 that if a guard-tuned set of adornments is chosen the natural level mapping is powerful enough to prove acceptability and, thus, termination. If a non guard-tuned set of adornments is chosen, the second program of Example 8 is obtained. Then, a following natural level mapping is defined (for some natural numbers $w_{p_1 \leqslant 100}$ and $w_{p_1>100}$):

$$| p^{\mathsf{d}}(X) | = w_{p_1 \leqslant 100} * \begin{cases} 100 - X & \text{if } X \leqslant 100 \\ 0 & \text{otherwise} \end{cases}$$

$$| p^{\mathsf{e}}(X) | = w_{p_1>100} * \begin{cases} X - 100 & \text{if } X > 100 \\ 0 & \text{otherwise} \end{cases}$$

Consider the following clause.

$$p^{\mathsf{d}}(X) \leftarrow X > 1, X < 1000, X1 \textit{ is } -X * X, X1 \leqslant 100, p^{\mathsf{d}}(X1).$$

To prove acceptability we have to show that the size of the call to $p^{\mathsf{d}}(X)$ is greater than the size of the corresponding call to $p^{\mathsf{d}}(X1)$. If the first argument x at the call to $p^{\mathsf{d}}(X)$ is greater than 1 and less than 10, the acceptability decrease requires $w_{p_1 \leqslant 100}(100 - x) > w_{p_1 \leqslant 100}(100 + x^2)$, contradicting $x > 1$ and $w_{p_1 \leqslant 100}$ being a natural number. Thus, acceptability cannot be proved with natural level mappings. □

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

The approach of Decorte *et al.* (1999) defines symbolic counterparts of the level mappings and infers the values of the coefficients by solving a system of constraints. Intuitively, instead of considering w_{c_i}'s as given coefficients, they are regarded as variables. More formally, similarly to Decorte *et al.* (1999), we introduce the following notion.

Definition 14
Let $p^{c_1 \wedge \ldots \wedge c_n}$ be an adorned predicate. A *symbolic counterpart of a natural level mapping* is an expression:

$$| p^{c_1 \wedge \ldots \wedge c_n}(t_1, \ldots, t_n) |^s = \sum_i W_{c_i} | p^{c_i}(t_1, \ldots, t_n) |^{pr},$$

where the W_{c_i}'s are symbols, associated to a predicate $p^{c_1 \wedge \ldots \wedge c_n}$.

The intuition behind the symbolic counterpart of a natural level mapping is that natural level mappings are instances of it. Therefore, we also require $W_c \geqslant 0$ to hold for any constraint c.

Example 14
Example 4, continued. Recalling that a stands for $1 < p_1 < 1000$, a symbolic counterpart of a natural level mapping for $p^{\mathsf{a}}(n)$ is $W_{p_1>1}(n-1) + W_{p_1<1000}(1000 - n)$. □

To verify the rigid acceptability condition (Theorem 2) interargument relations may be required as well. Interargument relations are usually represented as saying that a weighted sum of sizes of some arguments (with respect to a given norm) is greater or equal to a weighted sum of sizes of other arguments (e.g. see Plümer (1990)). In the numerical case these sizes should be replaced with expressions as used in Definition 12. Observe that for simpler examples no interargument relations are needed. Symbolic counterparts of norms and interargument relations can be defined analogously to Definition 14. In the next subsection we discuss how the symbolic counterparts are used to infer termination conditions.

4.3 Inferring termination constraints

In this section, we combine the steps studied so far into an algorithm that infers termination conditions. The program transformation, described in Section 3.2, implies that a trivial termination condition can be computed as a disjunction of the adornments corresponding to the predicates that can be completely unfolded, i.e., to the predicates that do not depend directly or indirectly on recursive predicates. More formally we can draw the following corollary from Theorem 3:

Corollary 1
Let P be a program, let S be a single predicate set of atomic queries and let $\mathscr{A}$ be a set of adornments for $rel(S)$. Let $A = \{c \mid c \in \mathscr{A}, \text{for all } q \text{ such that } rel(S)^c \sqsupseteq q : q \text{ is not recursive in} P^{\mathscr{A}}\}$. Then $\bigvee_{c \in A} c$ is a termination condition for S with respect to P.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Let P be a homogeneous program, let S be a single predicate set of atomic queries and let q be $rel(S)$.

1. For each $p \simeq q$ construct a guard-tuned set $\mathscr{A}_p$. (Section 4.1)
2. Adorn P with respect to q and $\bigcup_{p \simeq q} \mathscr{A}_p$. (Section 3.2)
3. Let $A = \{c \mid c \in \mathscr{A}_q, \textit{for all } p \textit{ such that } q^c \sqsupseteq p \ : \ p \textit{ is not recursive in } P^{\mathscr{A}}\}$. Let $cond_1 = \bigvee_{c \in A} c$. Let $cond_2 = \bigvee_{c \in \mathscr{A}_q, c \notin A} c$.
4. Let S' be $\{c(Q) \wedge Q^c \mid c \in \mathscr{A}_q, c \notin A, c(Q) \wedge Q^c \in S^{\mathscr{A}}\}$.
5. Define the symbolic counterparts of level mappings and interargument relations. (Section 4.2)
6. Let Σ be a set of constraints on the symbolic variables, following from rigid acceptability of S' with respect to $P^{\mathscr{A}}$ and validity of interargument relations.
7. Solve Σ with respect to the symbolic variables.

(a) Solution of Σ doesn't produce extra constraints on variables.

Report termination for *true*.

(b) Solution of Σ produces extra constraints on integer variables, appearing in the heads of the clauses.

Conjunct these constraints to termination condition $cond_2$.

Report termination for $cond_1 \vee cond_2$.

(c) There is no solution or integer variables, constrained by the solution of Σ, do not appear in the heads of the clauses

Report termination for $cond_1$.

Fig. 1. Termination inference algorithm.

Proof

By definition of A, for all $Q \in S$, $P^{\mathscr{A}}$ LD-terminates with respect to $\{c(Q) \wedge Q^c \mid c \in A\}$. Thus, by Theorem 3, $\bigvee_{c \in A} c$ is a termination condition for S with respect to P. □

Example 15

The termination condition constructed according to Corollary 1 for Example 4 is c, i.e., $(p_1 \leqslant -1000) \vee (-1 \leqslant p_1 \leqslant 1) \vee (p_1 \geqslant 1000)$. □

In general, the termination condition is constructed as a disjunction of two conditions: $cond_1$ for non-recursive cases, according to Corollary 1, and $cond_2$, for recursive cases. The later condition is initialised to be $\neg cond_1$ and further refined by adding constraints obtained from the rigid acceptability condition, as in (Decorte *et al.*, 1999)[2]. The algorithm is sketched in Figure 1.

Termination inference is inspired by the constraints-based approach of Decorte *et al.* (1999). Similarly to their work we start by constructing symbolic counterparts of the level mappings (Definition 14) and interargument relations, and construct conditions following from rigid acceptability (Theorem 2) and validity of interargument relations. Unlike their work in our case no rigidity constraints are needed

[2] Any other technique proving termination and able to provide some constraint that, if satisfied, implies termination can be used instead of Decorte *et al.* (1999).

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

(since integer arguments are ground and obviously rigid) and norms are fixed. Thus, the constraints system turns out to be simpler and better suited for automation. Finally, the conditions constructed are solved with respect to the symbolic variables (W_{c_i}'s). In Example 16 below we are going to see that rigid acceptability will be implied by the following constraint (Σ):

$$W_{q_1>q_2}(X-Y)>W_{q_1>q_2}((X-Y)-Y),$$

that is $W_{q_1>q_2}Y>0$ should hold. In both approaches $W_{q_1>q_2}\geqslant 0$ is required to hold. At this point the approach of Decorte *et al.* (1999), interpreting Y as a norm of arguments (i.e. Y is a natural number), will conclude $W_{q_1>q_2}>0$. In our case, we do not know *a priori* that Y is a natural number. Therefore, we would infer from Σ that $Y>0$ and $W_{q_1>q_2}>0$.

In general, given the system of constraints inferred by Decorte *et al.* (1999), we distinguish between the following cases:

- There is no solution. We report $cond_1$ as a termination condition (Corollary 1). Observe that when the algorithm reports the termination condition to be *false*, it suspects the possibility of non-termination.
- There is a solution for any values of integer variables. Namely, there are natural level mappings and interargument relations that prove termination of the program for *any* values of integer variables. Termination condition in this case is, thus, *true*.
- There is a solution for some values of integer variables. In other words, the solution constrains integer variables appearing in the clauses. Two cases can be distinguished:
  - — Integer variables constrained appear in the heads of the clauses. Then, constraints on these variables can be regarded as constraints on the arguments of the queries posed. In this case termination can be shown if these constraints are fulfilled.
  - — Integer variables constrained do not appear in the heads of the clauses. In this case our methodology is too weak to obtain some information implying termination of the queries. The best we can do is to report termination for $cond_1$.

Example 16

Consider the following program.

$$q(X,Y)\leftarrow X>Y,Z\ is\ X-Y,q(Z,Y).$$

We look for integer values of X and Y such that $q(X,Y)$ terminates. First, the algorithm infers adornments. In our case $\{\mathsf{a},\mathsf{b}\}$ are inferred, such that a denotes $q_1>q_2$ and b denotes $q_1\leqslant q_2$.

The adorned version of this program is

$$q^{\mathsf{a}}(X,Y)\leftarrow X>Y,Z\ is\ X-Y,Z>Y,q^{\mathsf{a}}(Z,Y).$$
$$q^{\mathsf{a}}(X,Y)\leftarrow X>Y,Z\ is\ X-Y,Z\leqslant Y,q^{\mathsf{b}}(Z,Y).$$

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

The corresponding set of queries is

$$\{x > y \wedge q^{\mathsf{a}}(x, y), x \leqslant y \wedge q^{\mathsf{b}}(x, y) \mid x, y \text{ are integers}\}.$$

There is no clause defining q^{b}. By Corollary 1, b, i.e., $q_1 \leqslant q_2$ is a termination condition. This is the one we denoted $cond_1$. The termination condition for q^{a}, denoted $cond_2$, is initialised to be $q_1 > q_2$. The symbolic counterpart of a natural level mapping is

$$| q^{\mathsf{a}}(X, Y) | = W_{q_1 > q_2} * \begin{cases} X - Y & \text{if } X > Y \\ 0 & \text{otherwise} \end{cases}$$

The set of constraints Σ implied by rigid acceptability is:

$$W_{q_1 > q_2}(X - Y) > W_{q_1 > q_2}((X - Y) - Y), \tag{2}$$

that is $W_{q_1 > q_2} Y > 0$ should hold. Since $W_{q_1 > q_2} \geqslant 0$, $Y > 0$ and $W_{q_1 > q_2} > 0$ should hold. Variable Y appears in the head of the clause, i.e. $Y > 0$ can be viewed as a constraint on the query. We update $cond_2$ to be $(q_1 > q_2) \wedge (q_2 > 0)$ and report termination for $q_1 \leqslant q_2 \vee (q_1 > q_2 \wedge q_2 > 0)$. □

Formally, the following theorem holds.

Theorem 4
Let P be a homogeneous pure logical program with integer computation, let S be a single predicate set of atomic queries and let `Algo` be the algorithm presented in Figure 1. Then the following holds:

- `Algo`(P, S) terminates;
- Let c be a symbolic condition returned by `Algo`(P, S). Then c is a termination condition for S.

Proof

- Termination of `Algo`(P, S) follows from termination of its steps. Termination of steps 1 and 2 follows from the presentation of these transformations in Sections 4.1 and 3.2, respectively. Termination of steps 3–7 is obvious.
- Partial correctness follows from the correctness of transformations and the corresponding result of Decorte *et al.* (1999). Correctness of step 1 is established by Lemma 2, of step 2 by Theorem 3. For step 4 observe that termination for queries in $S^{\mathscr{A}} \backslash S'$ is obvious by choice of A. Correctness of steps 6 and 7 follows from the corresponding result of (Decorte *et al.*, 1999).

□

In Example 16 the termination condition inferred by our algorithm was optimal, i.e. any other termination condition implies it. However, undecidability of the termination problem implies that no automatic tool can always guarantee optimality.

Example 17
Consider the following program.

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

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Table 1. *Examples of Sterling and Shapiro*

Ref	Queries	Time
8.1	greatest_common_divisor(i, i, v)	0.03
8.2	factorial(i, v)	0.02
8.3	factorial(i, v)	0.03
8.4	factorial(i, v)	0.03
8.5	between(i, i, v)	0.03
8.6a	sumlist(li, v)	0.00
8.6b	sumlist(li, v)	0.00
8.7a	inner_product(li, li, v)	0.00
8.7b	inner_product(li, li, v)	0.01
8.8	area(lp, v)	0.03
8.9	maxlist(li, v)	0.02
8.10	length(v, li)	0.01
8.11	length(li, v)	0.01
8.12	range(i, i, v)	0.03

We would like to study termination of this program with respect to $\{q(z_1, z_2) \mid z_1, z_2 \text{ are integers}\}$. Our algorithm infers the following termination condition: $q_1 \leqslant q_2 \vee (q_1 > q_2 \wedge q_2 \geqslant 0)$. This is a correct termination condition, but it is not optimal as $q(z_1, z_2)$ terminates, in fact, for all values of z_1 and z_2, i.e., the optimal termination condition is *true*. □

5 Experimental evaluation

The algorithm presented in Figure 1 was integrated in the system implementing the constraint-based approach of Decorte *et al.* (1999). As a preliminary step of our analysis, given a program and a set of atomic queries, the call set has to be computed. To do so, the type inference technique of Janssens and Bruynooghe (1992) was used. We opted for a very simple type inference technique that provides us only with information whether some argument is integer or not. More refined analysis can be used. For instance, the technique presented in Janssens *et al.* (1994) would have allowed us to know whether some numerical argument belongs to a certain interval. Alternatively, the integer intervals domain of Cousot and Cousot (1976, 1977) might have been used.

We have tested our system on a number of examples. First, we considered examples from two textbooks chapters dedicated to programming with arithmetic, namely, Chapter 8 of Sterling and Shapiro (1994) and Chapter 9 of Apt (1997). These results are summarised in Tables 1 and 2, respectively. We can prove termination of all the examples presented for all possible values of the integer arguments, that is, the termination condition inferred is *true*. Next, we've collected a number of programs from different sources. Table 3 presents timings and results for these programs. Again, termination of almost all programs can be shown for all possible values of the integer arguments. We believe that the reason for this is that most textbooks authors prefer to write programs ensuring termination. Finally, Table 4

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Table 2. *Examples of Apt*

Name	Queries	Time
between	between(*i*, *i*, *v*)	0.02
delete	delete (*i*, *i*, *v*)	0.04
factorial	fact(*i*, *v*)	0.01
in_tree	in_tree(*i*, *t*)	0.01
insert	insert(*i*, *t*, *v*)	0.01
length1	length(*li*, *v*)	0.00
maximum	maximum(*li*, *v*)	0.00
ordered	ordered(*li*)	0.01
quicksort	qs(*li*, *v*)	0.10
quicksort_acc	qs_acc(*li*, *v*)	0.10
quicksort_dl	qs_dl(*li*, *v*)	0.13
search_tree	is_search_tree(*t*)	0.06
tree_minimum	minimum(*t*, *v*)	0.01

Table 3. *Various examples*

Name	Ref	Queries	Time	T
dldf	(Bratko, 1986)	depthfirst2(*c*, *v*, *i*)	0.03	T
exp	(Coelho and Cotta, 1988)	exp(*i*, *i*, *v*)	0.07	N+
fib	(Bueno *et al*., 1994)	fib(*i*, *v*)	0.16	T
fib	(O'Keefe, 1990)	fib(*i*, *v*)	0.05	T*
forwardfib	(Bratko, 1986)	fib3(*i*, *v*)	0.02	T
money	(Bueno *et al*., 1994)	money(*v*, *v*, *v*, *v*, *v*, *v*, *v*, *v*)	0.20	T
oscillate	Example 4	p(*i*)	0.07	T
p32	(Hett, 2001)	gcd(*i*, *i*, *v*)	0.03	T
p33	(Hett, 2001)	coprime(*i*, *i*)	0.05	T
p34	(Hett, 2001)	totient_phi(*i*, *v*)	0.14	T
primes	(Clocksin and Mellish, 1981)	primes(*i*, *v*)	0.08	T
pythag	(Clocksin and Mellish, 1981)	pythag(*v*, *v*, *v*)	0.05	N+
r	(Dershowitz *et al*., 2001)	r(*i*, *v*)	0.01	T
triangle	(McDonald and Yazdani, 1990)	triangle(*i*, *v*)	0.03	N+

demonstrates some of the termination conditions inferred by our system. We can summarise our results by saying that the system turned out to be powerful enough to analyse correctly a broad spectrum of programs, while the time spent on the analysis never exceeded 0.20 seconds. In fact, for 90% of the programs results were obtained in 0.10 seconds or less.

The core part of the implementation was done in SICStus Prolog (SICS, 2002), type inference of Janssens and Bruynooghe (Janssens and Bruynooghe, 1992) was implemented in MasterProLog (IT Masters, 2000). Tests were performed on SUN SPARC Ultra-60, model 2360. The SPECint_95 and SPECfp_95 ratings for this machine are 16.10 and 29.50, respectively.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Table 4. *Examples of inferring termination conditions*

Name	Ref	Queries	Time	Condition
q	Example 16	q(*i*, *i*)	0.04	$q_1 \leqslant q_2 \vee (q_1 > q_2 \wedge q_2 > 0)$
q	Example 17	q(*i*, *i*)	0.05	$q_1 \leqslant q_2 \vee (q_1 > q_2 \wedge q_2 \geqslant 0)$
gcd	(Bratko, 1986)	*gcd*(*i*,*i*,*v*)	0.10	$q_1 = q_2 \vee (q_1 > q_2 \wedge q_2 \geqslant 1)$

In Tables 1–4 the following abbreviations are used:

- Ref: reference to the program;
- Name: name of the program;
- Queries: single predicate set of atomic queries of interest, where the arguments are denoted
  - — *c*, if the argument is a character;
  - — *i*, if the argument is an integer;
  - — *li*, if the argument is a list of integers;
  - — *lp*, if the argument is a list of pairs of integers;
  - — *t*, if the argument is a binary tree, containing integers;
  - — *v*, if the argument is a variable;
- Time: time (in seconds) needed to analyse the example;
- T: termination behaviour, see further.

One of the less expected results was finding non-terminating examples in Prolog textbooks. The first one, due to Coelho and Cotta (Coelho and Cotta, 1988), should compute an *n*th power of a number:

$$exp(X, 0, 1).$$
$$exp(X, Y, Z) \leftarrow even(Y), R \text{ is } Y/2, P \text{ is } X * X, exp(P, R, Z).$$
$$exp(X, Y, Z) \leftarrow T \text{ is } Y - 1, exp(X, T, Z1), Z \text{ is } Z1 * X.$$

$$even(Y) \leftarrow R \text{ is } Y \bmod 2, R = 0.$$

The termination condition inferred by our system is *false* and indeed, this is the only termination condition possible, since for any goal G the LD-tree of this program and G is infinite. This fact is denoted in Table 3 as N+. Similarly, the fact that for any goal G the LD-tree of the program and G is finite and our system is powerful enough to discover this is denoted as T.

McDonald and Yazdani suggest the following exercise in their book (McDonald and Yazdani, 1990): write a predicate *triangle* which finds the number of balls in a triangle of base N. For example, for $N = 4$ the number of balls is $4+3+2+1 = 10$. The next program is the solution provided by the authors:

$$triangle(1, 1).$$
$$triangle(N, S) \leftarrow M \text{ is } N - 1, triangle(M, R), S \text{ is } M + R.$$

Once more, the termination condition inferred by our system is *false*, and it is the only possible one.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

O'Keefe (1990) suggested a more efficient way of calculating Fibonacci numbers performing $O(n)$ work each time it is called, unlike the version of Bueno *et al.* (1994) that performs an exponential amount of work each time.

$$
\begin{aligned}
&\mathit{fib}(1, X) \leftarrow !, X = 1.\\
&\mathit{fib}(2, X) \leftarrow !, X = 1.\\
&\mathit{fib}(N, X) \leftarrow N > 2, \mathit{fib}(2, N, 1, 1, X).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fib}(N, N, X2, _, X) \leftarrow !, X = X2.\\
&\mathit{fib}(N_0, N, X2, X1, X) \leftarrow N1 \mathit{\ is\ } N_0 + 1,\\
&\qquad X3 \mathit{\ is\ } X2 + X1, \mathit{fib}(N_1, N, X3, X2, X).
\end{aligned}
$$

Termination of goals of the form *fib*(i,v) with respect to this example depends on the cut in the first clause of *fib*/5. If it is removed and if we add at the beginning of the second clause $N_0 \neq N$ termination can be proved. This fact is denoted T* in Table 3. Note that this replacement does not affect complexity of the computation. Observe also that a more declarative way to write the program might be to add $N_0 < N$ instead of $N_0 \neq N$. However, while the latter condition can be inferred automatically from the program, it is not clear whether this is also the case for the former one.

6 Further extensions

In this section we discuss possible extensions of the algorithm presented in Section 4. First, we discuss integrating termination analysis of numerical and symbolic computations, and then show how our results can be used to improve existing termination analyses of symbolic computations, such as Mesnard (1996) and Codish and Taboch (1999).

6.1 Integrating numerical and symbolic computation

In the real-world programs numerical computations are sometimes interleaved with symbolic ones, 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 (Pollard and Sag, 1994).

Example 18

$$
\begin{aligned}
&\mathit{collect}(X, [X|L], L) \leftarrow \mathit{atomic}(X).\\
&\mathit{collect}(T, L0, L) \leftarrow \mathit{compound}(T), \mathit{functor}(T, _, A), \qquad (3)\\
&\qquad \mathit{process}(T, 0, A, L0, L).\\
&\mathit{process}(_, A, A, L, L).\\
&\mathit{process}(T, I, A, L0, L2) \leftarrow I < A, I1 \mathit{\ is\ } I + 1, \mathit{arg}(I1, T, \mathit{Arg}), \qquad (4)\\
&\qquad \mathit{collect}(\mathit{Arg}, L0, L1), \mathit{process}(T, I1, A, L1, L2).
\end{aligned}
$$

To prove termination of $\{\mathit{collect}(t, v, [])\}$, where t is a tree and v is a variable, three decreases should be shown: between a call to *collect* and a call to *process* in (3),

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

between a call to *process* and a call to *collect* in (4) and between two calls to *process* in (4). The first two can be shown only by a symbolic level mapping, the third one—only by the numerical approach. □

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 $|\cdot|_2$ by mapping each atom $A \in B_P^E$ either to a natural number $|A|_1$ or to a pair of natural numbers $(|A|_1, |A|_2)$ and prove termination by establishing decreases via orderings on $(\mathcal{N} \cup \mathcal{N}^2)$ as suggested in Serebrenik and De Schreye (2001).

Example 19

Example 18, continued. Define $\varphi : B_P^E \to (\mathcal{N} \cup \mathcal{N}^2)$ as follows: $\varphi(collect(t, l0, l)) = \|t\|$, $\varphi(process(t, i, a, l0, l)) = (\|t\|, a - i)$ where $\|\cdot\|$ is a term-size norm. The 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 defined as: $n \succ m$, if $n >_{\mathcal{N}} m$, $n \succ (n, m)$, if *true*, $(n, m_1) \succ (n, m_2)$, if $m_1 >_{\mathcal{N}} m_2$ and $(n_1, m) \succ n_2$, if $n_1 >_{\mathcal{N}} n_2$ and $>_{\mathcal{N}}$ is the usual order on the naturals. □

This integrated approach allows one to analyse correctly examples such as *ground*, *unify*, *numbervars* (Sterling and Shapiro, 1994) and Example 6.12 in Dershowitz *et al.* (2001).

6.2 Termination of symbolic computations – revised

A number of modern approaches to termination analysis of logic programs (Codish and Taboch, 1999; Mesnard, 1996; Mesnard *et al.*, 2002) abstract a program to CLP($\mathcal{N}$) and then infer termination of the original program from the corresponding property of the abstract one. However, as mentioned in the introduction, techniques used to prove termination of the numerical program are often restricted to the identity function as the level-mapping.

Example 20

Consider the following example:

$$p(X) \leftarrow append(X, _, [_, _, _, _, _, _, _]), p([_|X]).$$
$$append([], L, L).$$
$$append([H|X], Y, [H|Z]) \leftarrow append(X, Y, Z).$$

Using the list-length norm, defined as

$$\|t\| = \begin{cases} 1 + \|t'\| & \text{if } t = [h|t'] \\ 0 & \text{otherwise} \end{cases}$$

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

the following CLP($\mathcal{N}$)-abstraction can be computed:

$$p(X) \leftarrow append(X, _, 7), p(1 + X).$$
$$append(0, L, L).$$
$$append(1 + X, Y, 1 + Z) \leftarrow append(X, Y, Z).$$

Computing a model for the abstraction of *append* and transforming the clause for p as described by Mesnard (1996) the following program is obtained:

$$p(X) \leftarrow X \leqslant 7, p(1 + X).$$

Termination of this program cannot be shown by the identity function as a level-mapping. Thus, non-termination will be suspected. □

Our approach is able to bridge the gap and provide the correct analysis of Example 20. Since our results have been stated for numerical computations and not for CLP($\mathcal{N}$) minor changes in the abstraction process are required. Instead of replacing a term t in an atom a with the size of t, a fresh variable V is introduced. Then, we add a goal V *is size*(t) before a (if a is a body subgoal) or after a (if it is a head of the clause). Next, we replace t in a by V, and proceed with the transformation of Mesnard (1996).

Example 21
Example 20, continued. Applying the abstraction technique above with respect to the list-length norm the following program is obtained:

$$p(X) \leftarrow append(X, _, 7), X1 \text{ is } X + 1, p(X1).$$
$$append(0, L, L).$$
$$append(X1, Y, Z1) \leftarrow X1 \text{ is } X + 1, Z1 \text{ is } Z + 1, append(X, Y, Z).$$

After computing the models this program is transformed to:

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

Termination of $p(n)$ with respect to this program can be shown by our approach for any integer number n. This implies termination of $p(t)$ with respect to the original one for any list of finite length t. □

To summarise this discussion, we believe that integrating our technique for proving termination of numerical computations with CLP($\mathcal{N}$) abstracting methodologies of Codish and Taboch (1999), Mesnard (1996) and Mesnard *et al.* (2002) will significantly extend the class of logic programs that can be analysed automatically.

7 Conclusion

We have presented an approach to verification of termination for logic programs with integer computations. This functionality is lacking in current available termination analysers for Prolog, such as cTI (Mesnard, 1996; Mesnard and Neumerkel, 2001),

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

TerminWeb (Codish and Taboch, 1999), and TermiLog (Lindenstrauss and Sagiv, 1997; Lindenstrauss *et al.*, 1997). The main contribution of this work is threefold. First, from the theoretical perspective, our study improves the understanding of termination of numerical computations, situates them in the well-known framework of acceptability and allows integration with the existing approaches to termination of symbolic computations. Moreover, our technique can be used to strengthen the existing techniques for proving termination of symbolic computations.

Second, unlike the majority of works on termination analysis for logic programs concerned with termination verification, we go further and do inference, i.e. we infer conditions on integer arguments of the queries that imply termination. To perform the inference task we apply a methodology inspired by the constraints based approach (Decorte *et al.*, 1999), i.e. we start by symbolic counterparts of level mappings and interargument relations and infer constraints on the integer arguments from rigid acceptability condition and validity of interargument relations.

Finally, the methodology presented has been integrated in the automatic termination analyser of Decorte *et al.* (1999). It was shown that our approach is robust enough to prove termination for a wide range of numerical examples, including *gcd* and *mod* (Dershowitz *et al.*, 2001) all examples appearing in Chapter 8 of Sterling and Shapiro (1994) and those appearing in Apt (1997).

Termination of numerical computations was studied by a number of authors (Apt, 1997; Apt *et al.*, 1994; Dershowitz *et al.*, 2001). Apt *et al.* (1994) provided a declarative semantics, so called Θ-semantics, for Prolog programs with first-order built-in predicates, 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 an appropriately tuned notion of acceptability. This approach provides important theoretical results, but seems to be difficult to integrate in automatic tools. In Apt (1997), 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 arithmetic but whose termination behaviour is not dependent on their arithmetic part, such as *quicksort*. Moreover, there are many programs that terminate only for *some* queries, such as Example 16. Alternatively, Dershowitz *et al.* (2001) extended the query-mapping pairs formalism of (Lindenstrauss and Sagiv, 1997) to deal with numerical computations. However, this approach inherited the disadvantages of Lindenstrauss and Sagiv (1997), such as high computational price, inherent to this approach due to repetitive fixpoint computations. Moreover, since our approach gains its power from the underlying framework of Decorte *et al.* (1999), it allows one to prove termination of some examples that cannot be analysed correctly by Dershowitz *et al.* (2001), similar to *confused delete* (Decorte *et al.*, 1999).

More research has been done on termination analysis for constraint logic programming (Colussi *et al.*, 1995; Mesnard, 1996; Ruggieri, 1997). 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*

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

system. Thus, all the results obtained for ideal constraints systems can be applied. Unfortunately, the research was either oriented towards theoretical characterisations (Ruggieri, 1997) or restricted to domains isomorphic to $\mathcal{N}$ (Mesnard, 1996), such as trees and terms.

In a contrast to the approach of Dershowitz *et al.* (2001) that was restricted to verifying termination, we presented a methodology for *inferring* termination conditions. It is not clear whether and how Dershowitz *et al.* (2001) can be extended to infer such conditions.

Numerical computations have been also analysed in the early works on termination analysis for imperative languages (Floyd, 1967; Katz and Manna, 1975), considering, as we have already pointed out in Section 1, general well-founded domains. However, our approach to automation differs significantly from these works. Traditionally, the verification community considered automatic generation of invariants (Bjørner *et al.*, 1997), while automatic generation of ranking functions (level mappings, in the logic programming parlance) just started to emerge (Colón and Sipma 2001; Colón and Sipma 2002). The inherent restriction of the latter results is that ranking functions have to be linear. Moreover, in order to perform the analysis of larger programs, such as *mergesort*, in a reasonable amount of time, authors further restricted the ranking functions to depend on one variable only. Unlike these results, our approach doesn't suffer from such limitations.

The idea of splitting a predicate into cases was first mentioned by Ullman and Van Gelder (1988), where existence has been assumed of a preprocessor that transformed a set of clauses to the new set, in which every subgoal unifies with all of the rules for its predicate symbol. However, neither in this paper, nor in the subsequent one (Sohn and Van Gelder, 1991), the methodology proposed was presented formally. To the best of our knowledge the first formal presentation of splitting in the framework of termination analysis is due to Lindenstrauss *et al.* (1998). Unlike these results, a numerical and not a symbolic domain was considered in the current paper.

The termination condition inferred for Example 16 is *optimal*, i.e., it is implied by any other termination condition. Clearly, undecidability of the termination problem implies that no automatic tool can always guarantee optimality of the condition inferred. However, verifying if the condition inferred is optimal seems to be an interesting question, related to looping analysis (Bol, 1991; De Schreye *et al.*, 1990; Shen, 1997; Shen *et al.*, 2001; Skordev, 1997). So far, in the context of logic programming, optimality of termination conditions inferred has been studied by Mesnard *et al.* (2002) only for symbolic computations.

8 Acknowledgements

Alexander Serebrenik is supported by GOA: "LP^+: a second generation logic programming language". We are very grateful to Gerda Janssens and Vincent Englebert for making their type analysis systems available for us. Many useful suggestions and helpful comments were proposed to us by anonymous referees—we are much obliged to their careful reading.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

References

Apt, K. R. 1997. *From Logic Programming to Prolog*. Series in Computer Science. Prentice Hall.

Apt, K. R., Marchiori, E. and Palamidessi, C. 1994. A declarative approach for first-order built-in's in Prolog. *Applicable Algebra in Engineering, Communication and Computation 5,* 3/4, 159–191.

Bjørner, N., Browne, A. and Manna, Z. 1997. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science 173,* 1 (February), 49–87.

Bol, R. N. 1991. Loop checking in logic programming. PhD thesis, Universiteit van Amsterdam.

Bossi, A. and Cocco, N. 1994. Preserving universal temination through unfold/fold. In *Algebraic and Logic Programming*, G. Levi and M. Rodríguez-Artalejo, Eds. Lecture Notes in Computer Science, vol. 850. Springer Verlag, pp. 269–286.

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

Bossi, A., Cocco, N. and Fabris, M. 1991. Proving Termination of Logic Programs by Exploiting Term Properties. In *Proceedings CCPSD-TAPSOFT'91*. Lecture Notes in Computer Science, vol. 494. Springer Verlag, pp. 153–180.

Bossi, A., Cocco, N. and Fabris, M. 1994. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science 124,* 2 (February), 297–328.

Bratko, I. 1986. *Prolog programming for Artificial Intelligence*. Addison-Wesley.

Bruynooghe, M., Codish, M., Genaim, S. and Vanhoof, W. 2002. Reuse of results in termination analysis of typed logic programs. In *Static Analysis, 9th International Symposium*, M. V. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer Verlag, pp. 477–492.

Bueno, F., García de la Banda, M. J. and Hermenegildo, M. V. 1994. Effectiveness of global analysis in strict independence-based automatic parallelization. In *Logic Programming, Proceedings of the 1994 International Symposium*, M. Bruynooghe, Ed. MIT Press, pp. 320–336.

Clocksin, W. F. and Mellish, C. S. 1981. *Programming in Prolog*. Springer-Verlag.

Codish, M. and Taboch, C. 1999. A semantic basis for termination analysis of logic programs. *Journal of Logic Programming 41,* 1, 103–123.

Coelho, H. and Cotta, J. C. 1988. *Prolog by example*. Springer-Verlag.

Colón, M. A. and Sipma, H. B. 2001. Synthesis of linear ranking functions. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference*, T. Margaria and W. Yi, Eds. Lecture Notes in Computer Science, vol. 2031. Springer Verlag, pp. 67–81.

Colón, M. A. and Sipma, H. B. 2002. Practical methods for proving program termination. In *Computer Aided Verification, 14th International Conference*, E. Brinksma and K. Guldstrand Larsen, Eds. Lecture Notes in Computer Science, vol. 2404. Springer Verlag, pp. 442–454.

Colussi, L., Marchiori, E. and Marchiori, M. 1995. On termination of constraint logic programs. In *Principles and Practice of Constraint Programming - CP'95*,, U. Montanari and F. Rossi, Eds. Lecture Notes in Computer Science, vol. 976. Springer Verlag, pp. 431–448.

Cousot, P. and Cousot, R. 1976. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, pp. 106–130.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record*

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM Press, New York, NY, Los Angeles, CA, pp. 238–252.

De Schreye, D., Verschaetse, K. and Bruynooghe, M. 1990. A practical technique for detecting non-terminating queries for a restricted class of Horn clauses, using directed, weighted graphs. In *Logic Programming, Proceedings of the Seventh International Conference*, D. H. Warren and P. Szeredi, Eds. MIT Press, pp. 649–663.

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

Decorte, S. and De Schreye, D. 1998. Termination analysis: some practical properties of the norm and level mapping space. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, J. Jaffar, Ed. MIT Press, pp. 235–249.

Decorte, S., De Schreye, D. and Vandecasteele, H. 1999. Constraint-based termination analysis of logic programs. *ACM TOPLAS 21,* 6 (November), 1137–1195.

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

Floyd, R. W. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, J. Schwartz, Ed. American Mathematical Society, 19–32. Proceedings of Symposiumsia in Applied Mathematics; v. 19.

Genaim, S. and Codish, M. 2001. Inferring termination conditions for logic programs using backwards analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conferencerence, Proceedings*, R. Nieuwenhuis and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 2250. Springer-Verlag, pp. 685–694.

Genaim, S., Codish, M., Gallagher, J. and Lagoon, V. 2002. Combining norms to prove termination. In *Third International Workshop on Verification, Model Checking and Abstract Interpretation*, A. Cortesi, Ed. Lecture Notes in Computer Science, vol. 2294. Springer-Verlag, pp. 126–138.

Hett, W. 2001. P-99: Ninety-nine Prolog problems. Available at `http://www.hta-bi.bfh.ch/~hew/informatik3/prolog/p-99/`.

Holzbaur, C. 1995. OFAI CLP(Q,R) Manual. Tech. Rep. TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna.

ILOG. 2001. *ILOG Solver 5.1 User's Manual*. ILOG s.a. `http://www.ilog.com`.

IT Masters. 2000. MasterProLog Programming Environment. Available at `http://www.itmasters.com/`.

Janssens, G. and Bruynooghe, M. 1992. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming 13,* 2&3 (July), 205–258.

Janssens, G., Bruynooghe, M. and Englebert, V. 1994. Abstracting numerical values in CLP(H, N). In *Programming Language Implementation and Logic Programming, 6th International Symposiumsium, PLILP'94*, M. V. Hermenegildo and J. Penjam, Eds. Lecture Notes in Computer Science, vol. 844. Springer-Verlag, pp. 400–414.

Katz, S. and Manna, Z. 1975. A closer look at termination. *Acta Informatica 5*, 333–352.

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

Lindenstrauss, N., Sagiv, Y. and Serebrenik, A. 1997. *TermiLog*: A system for checking termination of queries to logic programs. In *Computer Aided Verification, 9th International*

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Conference, O. Grumberg, Ed. Lecture Notes in Computer Science, vol. 1254. Springer-Verlag, pp. 444–447.

Lindenstrauss, N., Sagiv, Y. and Serebrenik, A. 1998. Unfolding the mystery of *mergesort*. In *Proceedings of the 7th International Workshop on Logic Program Synthesis and Transformation*, N. Fuchs, Ed. Lecture Notes in Computer Science, vol. 1463. Springer-Verlag, pp. 206–225.

McDonald, C. and Yazdani, M. 1990. *Prolog programming: a tutorial introduction*. Artificial Intelligence Texts. Blackwell Scientific Publications.

Mesnard, F. 1996. Inferring left-terminating classes of queries for constraint logic programs. In *Proceedings of the 1996 Joint International Conference and Syposium on Logic Programming*, M. Maher, Ed. MIT Press, Cambridge, MA, USA, pp. 7–21.

Mesnard, F. and Neumerkel, U. 2001. Applying static analysis techniques for inferring termination conditions of logic programs. In *Static Analysis, 8th International Symposium, SAS 2001*, P. Cousot, Ed. Lecture Notes in Computer Science, vol. 2126. Springer-Verlag, pp. 93–110.

Mesnard, F., Payet, E. and Neumerkel, U. 2002. Dectecting optimal termination conditions of logic programs. In *Static Analysis, 8th International Symposium, SAS 2002*, P. Cousot, Ed. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, pp. 509–527.

Mesnard, F. and Ruggieri, S. 2003. On proving left termination of constraint logic programs. *ACM Transaction on Computational Logic 4,* 2, 207–259.

Ohlebusch, E. 2001. Automatic termination proofs of logic programs via rewrite systems. *Applicable Algebra in Engineering, Communication and Computing 12,* 1–2, 73–116.

O'Keefe, R. A. 1990. *The Craft of Prolog*. MIT Press, Cambridge, MA.

Plümer, L. 1990. Termination proofs for logic programs based on predicate inequalities. In *Proceedings of ICLP'90*. MIT Press, pp. 634–648.

Plümer, L. 1991. Automatic termination proofs for Prolog programs operating on nonground terms. In *International Logic Programming Symposium*. MIT Press.

Pollard, C. and Sag, I. A. 1994. *Head-driven Phrase Structure Grammar*. University of Chicago Press.

Ruggieri, S. 1997. Termination of constraint logic programs. In *Automata, Languages and Programming, 24th International Colloquium, ICALP'97*, P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, Eds. Lecture Notes in Computer Science, vol. 1256. Springer-Verlag, pp. 838–848.

Serebrenik, A. and De Schreye, D. 2001. Non-transformational termination analysis of logic programs, based on general term-orderings. In *Logic Based Program Synthesis and Transformation 10th International Workshop, Selected Papers*, K.-K. Lau, Ed. Lecture Notes in Computer Science, vol. 2042. Springer-Verlag, pp. 69–85.

Serebrenik, A. and De Schreye, D. 2002. On termination of logic programs with floating point computations. In *9th International Static Analysis Symposium*, M. V. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, pp. 151–164.

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

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

SICS. 2002. *SICStus User Manual. Version 3.10.0*. Swedish Institute of Computer Science.

Skordev, D. 1997. An abstract approach to some loop detection problems. *Fundamenta Informaticae 31,* 2, 195–212.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press

Sohn, K. and Van Gelder, A. 1991. Termination detection in logic programs using argument sizes. In *Proceedings of the Tenth ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*. ACM Press, pp. 216–226.

Sterling, L. and Shapiro, E. 1994. *The Art of Prolog*. The MIT Press, Cambridge, MA.

Ullman, J. D. and Van Gelder, A. 1988. Efficient tests for top-down termination of logical rules. *Journal of the ACM 35,* 2 (April), 345–373.

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

Verschaetse, K. and De Schreye, D. 1991. Deriving termination proofs for logic programs, using abstract procedures. In *Logic Programming, Proceedings Eighth International Conference*, K. Furukawa, Ed. MIT Press, pp. 301–315.

Winsborough, W. 1992. Multiple specialization using minimal-function graph semantics. *Journal of Logic Programming 13,* 2/3, 259–290.

https://doi.org/10.1017/S1471068404002042 Published online by Cambridge University Press