Checking Termination of Datalog with Function Symbols Through Linear Constraints

Citation for published version:

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Checking Termination of Datalog with Function Symbols Through Linear Constraints

Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna
{calautti,greco,molinaro,trubitsyna}@dimes.unical.it
DIMES, Università della Calabria, 87036 Rende (CS), Italy

Discussion Paper

Abstract. Enriching Datalog programs with function symbols makes modeling easier, increases the expressive power, and allows us to deal with infinite domains. However, this comes at a cost: common inference tasks become undecidable. To cope with this issue, recent research has focused on finding trade-offs between expressivity and decidability by identifying classes of logic programs that impose limitations on the use of function symbols but guarantee decidability of common inference tasks. Despite the significant body of work in this area, current approaches do not include many simple practical programs whose evaluation terminates. In this paper, we present the novel class of rule-bounded programs. While current techniques perform a limited analysis of how terms are propagated from an individual argument to another, our technique is able to perform a more global analysis, thereby overcoming several limitations of current approaches. We show different results on the correctness and the expressivity of the proposed technique.

Keywords: Datalog with function symbols, bottom-up evaluation, program evaluation termination

1 Introduction

Enriching datalog programs with function symbols has recently seen a surge in interest. Function symbols make modeling easier, increase the expressive power, and allow us to deal with infinite domains. At the same time, this comes at a cost: common inference tasks (e.g., cautious and brave reasoning) become undecidable.

Recent research has focused on identifying classes of logic programs that impose some limitations on the use of function symbols but guarantee decidability of common inference tasks. Efforts in this direction are the class of finitely-ground programs [9] and the more general class of bounded term-size programs [24]. The minimal model of a finitely-ground datalog program is finite, and the well-founded model of a bounded term-size datalog program is finite. Unfortunately, checking if a program is bounded term-size or even finitely-ground is semi-decidable.
Considering the minimal model semantics, decidable subclasses of finitely-ground programs have been proposed (for more details see the last section of this paper). However, there are still many simple practical programs which are finitely-ground but are not detected by any of the current termination criteria. Below are two examples.

**Example 1.** The following program $P_1$ implements the bubble sort algorithm:

\[
\begin{align*}
  r_0 &: \text{bub}(L, [], []) \leftarrow \text{input}(L). \\
  r_1 &: \text{bub}([Y|Z], [X|\text{Cur}], \text{Sol}) \leftarrow \text{bub}([X|[Y|Z]], \text{Cur}, \text{Sol}), X \leq Y. \\
  r_2 &: \text{bub}([X|Z], [Y|\text{Cur}], \text{Sol}) \leftarrow \text{bub}([X|[Y|Z]], \text{Cur}, \text{Sol}), Y < X. \\
  r_3 &: \text{bub}(\text{Cur}, [], [X|\text{Sol}]) \leftarrow \text{bub}([X|[]], \text{Cur}, \text{Sol}).
\end{align*}
\]

Here \text{input} is a base predicate symbol whose extension is a fact containing the list we would like to sort. The bottom-up evaluation of this program always terminates for any input list. The ordered list \text{Sol} can be obtained from the atom \text{bub}([], [], \text{Sol}) in the program’s minimal model.

\[\square\]

**Example 2.** Program $P_2$ below performs a depth-first traversal of an input tree:

\[
\begin{align*}
  r_0 &: \text{visit}(\text{Tree}, [], []) \leftarrow \text{input}(\text{Tree}). \\
  r_1 &: \text{visit}(\text{Left}, [\text{Root}|\text{Visited}], [\text{Right}|\text{ToVisit}]) \leftarrow \\
  &\quad \text{visit}(\text{tree}(\text{Root}, \text{Left}, \text{Right}), \text{Visited}, \text{ToVisit}). \\
  r_2 &: \text{visit}(\text{Next}, \text{Visited}, \text{ToVisit}) \leftarrow \text{visit}(\text{null}, \text{Visited}, [\text{Next}|\text{ToVisit}]).
\end{align*}
\]

Here \text{input} is a base predicate symbol whose extension contains a tree-like structure represented by means of the ternary function symbol \text{tree}. The program visits the nodes of the tree and puts them in a list following a depth-first search. The list \text{L} of visited elements can be obtained from the atom \text{visit}(\text{null}, \text{L}, []) in the program’s minimal model. For instance, if the input tree is

\[
\text{input}(\text{tree}(a, \text{tree}(c, \text{null}, \text{tree}(d, \text{null}, \text{null})), \text{tree}(b, \text{null}, \text{null}))).
\]

the program produces the list \([b, d, c, a]\) containing the nodes of the tree in opposite order w.r.t. the traversal.

\[\square\]

The peculiarity of the above programs is that the overall size of the terms in the body does not increase during their propagation to the head, as there is only a simple redistribution of terms. One of the novelties of the technique proposed in this paper is the capability of doing this kind of analysis, thereby identifying finitely-ground programs that none of the current techniques include.

**Contribution.** In this paper we present a novel technique for checking if a Datalog program is finitely-ground, initially appeared in [6]. Our technique overcomes several limitations of current approaches being able to perform a more global analysis of how terms are propagated from the body to the head of rules. To this end, we use linear constraints to measure and relate the size of head and body atoms of rules and introduce the class of **rule-bounded** programs. We study the relationship between the proposed class and current termination criteria.

**Organization.** Section [2] reports preliminaries on Datalog programs with function symbols. Section [3] introduces the class of rule-bounded programs. Related work and conclusions are reported in Section [4].
2 Preliminaries

Syntax. We assume to have (pairwise disjoint) infinite sets of logical variables, predicate symbols, and function symbols. Each predicate and function symbol \( g \) is associated with an arity, denoted \( \text{arity}(g) \), which is a non-negative integer. Function symbols of arity 0 are called constants. Variables appearing in logic programs are called “logical variables” and will be denoted by upper-case letters in order to distinguish them from variables appearing in linear constraints, which are called “integer variables” and will be denoted by lower-case letters. A term is either a logical variable, or an expression of the form \( f(t_1, ..., t_m) \), where \( f \) is a function symbol of arity \( m \geq 0 \) and \( t_1, ..., t_m \) are terms.

An atom is of the form \( p(t_1, ..., t_n) \), where \( p \) is a predicate symbol of arity \( n \geq 0 \) and \( t_1, ..., t_n \) are terms. A rule \( r \) is of the form \( H \leftarrow B_1, ..., B_k \), where \( k \geq 0 \), and \( H, B_1, ..., B_k \) are atoms. The atom \( H \) is called the head of \( r \) and is denoted by \( \text{head}(r) \). The conjunction \( B_1, ..., B_k \) is called the body of \( r \) and is denoted by \( \text{body}(r) \).

A substitution \( \theta \) is of the form \( \{X_1/t_1, ..., X_n/t_n\} \), where \( X_1, ..., X_n \) are distinct logical variables and \( t_1, ..., t_n \) are terms. The result of applying \( \theta \) to an atom (or term) \( A \), denoted \( A\theta \), is the atom (or term) obtained from \( A \) by simultaneously replacing each occurrence of a logical variable \( X_i \) in \( A \) with \( t_i \) if \( X_i/t_i \) belongs to \( \theta \). Two atoms \( A_1 \) and \( A_2 \) unify if there exists a substitution \( \theta \), called a unifier of \( A_1 \) and \( A_2 \), such that \( A_1\theta = A_2\theta \).

Semantics. Consider a program \( \mathcal{P} \). The Herbrand universe \( H_{\mathcal{P}} \) of \( \mathcal{P} \) is the infinitely infinite set of ground terms which can be built using constants and function symbols appearing in \( \mathcal{P} \). The Herbrand base \( B_{\mathcal{P}} \) of \( \mathcal{P} \) is the set of ground atoms which can be built using predicate symbols appearing in \( \mathcal{P} \) and ground terms of \( H_{\mathcal{P}} \). A rule \( r' \) is a ground instance of a rule \( r \) in \( \mathcal{P} \) if \( r' \) can be obtained from \( r \) by substituting every logical variable in \( r \) with some ground term in \( H_{\mathcal{P}} \). We use \( \text{ground}(r) \) to denote the set of all ground instances of \( r \) and \( \text{ground}(\mathcal{P}) \) to denote the set of all ground instances of the rules in \( \mathcal{P} \), i.e., \( \text{ground}(\mathcal{P}) = \cup_{r \in \mathcal{P}} \text{ground}(r) \). An interpretation of \( \mathcal{P} \) is any subset \( I \) of \( B_{\mathcal{P}} \). The truth value of a ground atom \( A \) w.r.t. \( I \), denoted \( \text{value}_I(A) \), is true if \( A \in I \), false otherwise. A ground rule \( r \) is satisfied by \( I \), denoted \( I \models r \), if there is a ground atom \( A \in \text{body}(r) \) s.t. \( \text{value}_I(A) = \text{false} \) or there \( \text{value}_I(\text{head}(r)) = \text{true} \). Thus, if the body of \( r \) is empty, \( r \) is satisfied by \( I \) if \( \text{value}_I(\text{head}(r)) = \text{true} \). An interpretation of \( \mathcal{P} \) is a model of \( \mathcal{P} \) if it satisfies every ground rule in \( \text{ground}(\mathcal{P}) \). A model \( M \) of \( \mathcal{P} \) is minimal if no proper subset of \( M \) is a model of \( \mathcal{P} \). The minimal model of \( \mathcal{P} \) is unique and is denoted by \( \text{MM}(\mathcal{P}) \).
3 Rule-bounded Programs

In this section, we present rule-bounded programs, a class of programs whose evaluation always terminates and for which checking membership in the class is decidable. Their definition relies on a novel technique which uses linear inequalities to measure terms and atoms’ sizes and checks if the size of the head of a rule is always bounded by the size of some body atom.

We use \( \mathbb{N} \) to denote the set of natural numbers \( \{1, 2, 3, \ldots \} \) and \( \mathbb{N}_0 \) to denote the set of natural numbers including zero. Moreover, \( \mathbb{N}^k = \{(v_1, \ldots, v_k) \mid v_i \in \mathbb{N} \text{ for } 1 \leq i \leq k \} \) and \( \mathbb{N}_0^k = \{(v_1, \ldots, v_k) \mid v_i \in \mathbb{N}_0 \text{ for } 1 \leq i \leq k \} \). Given a \( k \)-vector \( \overline{v} = (v_1, \ldots, v_k) \) in \( \mathbb{N}_0^k \), we use \( \overline{v}[i] \) to refer to \( v_i \), for \( 1 \leq i \leq k \). Given two \( k \)-vectors \( \overline{v} = (v_1, \ldots, v_k) \) and \( \overline{w} = (w_1, \ldots, w_k) \) in \( \mathbb{N}_0^k \), we use \( \overline{v} \cdot \overline{w} \) to denote the classical scalar product, i.e., \( \overline{v} \cdot \overline{w} = \sum_{i=1}^{k} v_i \cdot w_i \).

The following definition introduces the notions of term and atom size.

**Definition 1.** Let \( t \) be a term. The size of \( t \) is recursively defined as follows:

\[
\text{size}(t) = \begin{cases} 
  x & \text{if } t \text{ is a logical variable } X; \\
  m + \sum_{i=1}^{m} \text{size}(t_i) & \text{if } t = f(t_1, \ldots, t_m).
\end{cases}
\]

where \( x \) is an integer variable. The size of an atom \( A = p(t_1, \ldots, t_n) \), denoted \( \text{size}(A) \), is the \( n \)-vector \( (\text{size}(t_1), \ldots, \text{size}(t_n)) \).

In the definition above, an integer variable \( x \) intuitively represents the possible sizes that the logical variable \( X \) can have during the bottom-up evaluation. The size of a term of the form \( f(t_1, \ldots, t_m) \) is defined by summing up the size of its terms \( t_i \)'s plus the arity \( m \) of \( f \). Note that from the definition above, the size of every constant is 0.

**Example 3.** Consider rule \( r_1 \) of program \( \mathcal{P} \) (see Example 1). Using \( \text{lc} \) to denote the list constructor operator “\( \text{lc} \)”, the rule can be rewritten as follows:

\[
\text{bub} \text{(lc}(Y,Z),\text{lc}(X,\text{Cur},\text{Sol})) \leftarrow \text{bub}(\text{lc}(X,\text{lc}(Y,Z)),\text{Cur},\text{Sol}), X \leq Y.
\]

Let \( A \) (resp. \( B \)) be the atom in the head (resp. the first atom in the body). Then,

\[
\text{size}(A) = (2 + y + z, 2 + x + \text{cur, sol}) \quad \text{size}(B) = (2 + [x + (2 + y + z)], \text{cur, sol})
\]

We are now ready to define rule-bounded programs.

**Definition 2 (Rule-bounded programs).** Let \( \mathcal{P} \) be a program and \( \text{pred}(\mathcal{P}) = \{p_1, \ldots, p_h \} \). We say that \( \mathcal{P} \) is rule-bounded if there exist \( k \) vectors \( \overline{\pi}_{p_h} \in \mathbb{N}^{\text{arity}(p_h)} \), \( 1 \leq h \leq k \), such that for every rule \( r \in \mathcal{P} \) with \( A = \text{head}(r) = p_l(t_1, \ldots, t_n) \), there exists an atom \( B = p_j(u_1, \ldots, u_m) \) in \( \text{body}(r) \) s.t. the following inequality is satisfied

\[
\overline{\pi}_{p_l} \cdot \text{size}(B) - \overline{\pi}_{p_j} \cdot \text{size}(A) \geq 0
\]

for every non-negative value of the integer variables in \( \text{size}(B) \) and \( \text{size}(A) \).
Intuitively, for every rule of $\mathcal{P}$, Definition 2 checks if the size of the head atom is bounded by the size some body atom for all possible sizes the terms can assume.

Example 4. Consider again program $\mathcal{P}_1$ of Example 1. To determine if the program is rule-bounded we need to find $\sigma_{input} \in \mathbb{N}$ and $\sigma_{bub} \in \mathbb{N}^3$ such that there is an atom in each rule’s body which satisfy the inequalities derived from rules of $\mathcal{P}_1$ for all non-negative values of the integer variables therein. In particular, selecting the first body atom in each rule we obtain the following linear constraints:

\[
\begin{cases}
\sigma_{input} \cdot l_0 - \sigma_{bub} \cdot (l_0, 0, 0) \geq 0 \\
\sigma_{bub} \cdot (4 + x_1 + y_1 + z_1, cur_1, sol_1) - \sigma_{bub} \cdot (2 + y_1 + z_1, 2 + x_1 + cur_1, sol_1) \geq 0 \\
\sigma_{bub} \cdot (4 + x_2 + y_2 + z_2, cur_2, sol_2) - \sigma_{bub} \cdot (2 + x_2 + z_2, 2 + y_2 + cur_2, sol_2) \geq 0 \\
\sigma_{bub} \cdot (2 + x_3 + 0, cur_3, sol_3) - \sigma_{bub} \cdot (cur_3, 0, 2 + x_3 + sol_3) \geq 0
\end{cases}
\]

where subscripts associated with integer variables are used to refer to the occurrences of logical variables in different rules (e.g., $y_2$ is the integer variable associated to the logical variable $Y$ in rule $r_2$).

By expanding the scalar products and isolating every integer variable we get:

\[
\begin{cases}
(\sigma_{input}[1] - \sigma_{bub}[1]) \cdot l_0 \geq 0 \\
(\sigma_{bub}[1] - \sigma_{bub}[2]) \cdot x_1 + (2\sigma_{bub}[1] - 2\sigma_{bub}[2]) \geq 0 \\
(\sigma_{bub}[1] - \sigma_{bub}[2]) \cdot y_2 + (2\sigma_{bub}[1] - 2\sigma_{bub}[2]) \geq 0 \\
(\sigma_{bub}[1] - \sigma_{bub}[3]) \cdot x_3 + (\sigma_{bub}[2] - \sigma_{bub}[1]) \cdot cur_3 + (2\sigma_{bub}[1] - 2\sigma_{bub}[3]) \geq 0
\end{cases}
\]

The previous inequalities must hold for all $l_0, x_1, y_2, x_3, cur_3 \in \mathbb{N}_0$; it is easy to see that this is the case iff the following system admits a solution:

\[
\begin{cases}
\sigma_{input}[1] - \sigma_{bub}[1] \geq 0 \\
\sigma_{bub}[1] - \sigma_{bub}[2] \geq 0 \\
\sigma_{bub}[1] - \sigma_{bub}[2] \geq 0 \\
\sigma_{bub}[1] - \sigma_{bub}[3] \geq 0 \\
\sigma_{bub}[2] - 2\sigma_{bub}[1] \geq 0 \\
2\sigma_{bub}[1] - 2\sigma_{bub}[3] \geq 0
\end{cases}
\]

A possible solution is $\sigma_{input} = (1)$ and $\sigma_{bub} = (1, 1, 1)$. Consequently, $\mathcal{P}_1$ is rule-bounded.

The method in the previous example to find vectors $\sigma_v$ for all $p \in pred(\mathcal{P})$ can always be applied. That is, we can always isolate the integer variables in the original inequalities and then derive one inequality for each expression that multiplies an integer variable plus the one for the constant term, imposing that all such expressions must be greater than or equal to 0.

Example 5. Applying the method above to the program $\mathcal{P}_2$ of Example 2 we obtain the following constraints:

\[
\begin{cases}
\sigma_{input} \cdot 0 - \sigma_v \cdot (0, 0, 0) \geq 0 \\
(\sigma_v[1] - \sigma_v[2]) \cdot root_1 + (\sigma_v[1] - \sigma_v[3]) \cdot right_1 + (3\sigma_v[1] - 2\sigma_v[2] - 2\sigma_v[3]) \geq 0 \\
(\sigma_v[3] - \sigma_v[1]) \cdot next_2 + 2\sigma_v[3] \geq 0
\end{cases}
\]
where subscript $v$ stands for predicate symbol visit. By setting $\pi_v = (2, 1, 2)$, we get positive integer values of $\pi_v[1], \pi_v[2], \pi_v[3]$ s.t. the inequalities above are satisfied for all $\text{root}_1, \text{right}_1, \text{next}_2 \in \mathbb{N}_0$. Thus, $P_2$ is rule-bounded.

Two key properties of rule-bounded programs are: they are finitely-ground and it is decidable to check whether a given program is rule-bounded.

**Theorem 1.** Every rule-bounded program is finitely-ground.

**Theorem 2.** Checking whether a program is rule-bounded is in NP.

**Theorem 3.** Rule-bounded programs are incomparable with mapping-restricted and bounded programs.

Observe that in the previous theorem we have considered only the most general subclasses of finitely-ground programs proposed so far, which generalize previous classes such as argument-restricted programs [19].

The proposed technique can be further improved by identifying in each rule body atoms that are mutually recursive with the rule head. If the head atom contains only variables occurring in some atom not mutually recursive with its head, the rule is not dangerous and its consideration can be omitted. Moreover, in the remaining rules only body atoms that are mutually recursive with the rule head should be considered to construct the inequality.

### 4 Discussion and Conclusions

A significant body of work has been done on termination of logic programs under top-down evaluation [11, 12, 20, 23, 10, 28, 24, 27, 21, 5, 4, 3] and in the area of term rewriting [35, 30, 2, 12, 13]. Termination properties of query evaluation for normal programs under tabling have been studied in [24, 25, 32].

In this paper, we consider Datalog programs with function symbols under the minimal model semantics, and thus all the excellent works above cannot be straightforwardly applied to our setting—for a discussion on this see, e.g., [9, 1]. In our context, [9] introduced the class of finitely-ground programs, guaranteeing the existence of a finite set of stable models, each of finite size, for programs in the class. Since membership in the class is not decidable, decidable subclasses have been proposed: $\omega$-restricted [31], $\lambda$-restricted [14], finite domain [9], argument-restricted [19], safe and $\Gamma$-acyclic [7], mapping-restricted [8], bounded programs [16]. An adornment-based approach that can be used in conjunction with the techniques above to detect more programs as finitely-ground has been proposed in [17].

Compared with the aforementioned classes, rule-bounded programs allow us to perform a more global analysis and identify many practical programs as finitely-ground, such as those where terms in the body are rearranged in the head, which are not included in any of the classes above. We observe that there are also programs which are not rule-bounded but are recognized as finitely-ground by some of the aforementioned techniques (see Theorems 3).
Similar concepts of “term size” have been considered to check termination of logic programs evaluated in a top-down fashion [29], in the context of partial evaluation to provide conditions for strong termination and quasi-termination [33,18], and in the context of tabled resolution [24,25]. These approaches are geared to work under top-down evaluation, looking at how terms are propagated from the head to the body, while our approach is developed to work under bottom-up evaluation, looking at how terms are propagated from the body to the head. This gives rise to significant differences in how the program analysis is carried out, making one approach not applicable in the setting of the other. As a simple example, the rule $p(X) \leftarrow p(X)$ leads to a non-terminating top-down evaluation, while it is completely harmless under bottom-up evaluation.

We notice that this topic is also related to research done in the database community on termination of the chase procedure, see [15] for a recent survey.

As a direction for future work, we plan to investigate how our techniques can be combined with current termination criteria. Since they look at programs from different standpoints, an interesting issue is to study how they can be integrated so that they can benefit from each other.

References