Termination Analysis for Tabled Logic Programming

Stefaan Decorte, Danny De Schreye, Michael Leuschel,
Bern Martens and Konstantinos Sagonas

Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{stefaan,dannyd,michael,bern,kostis}@cs.kuleuven.ac.be

Abstract. We provide a theoretical basis for studying the termination of tabled logic programs executed under SLG-resolution using a left-to-right computation rule. To this end, we study the classes of ***quasi-terminating*** and ***LG-terminating*** programs (for a set of atomic goals S). These are tabled logic programs where execution of each call from S leads to only a finite number of different (i.e., non-variant) calls, and a finite number of different calls and computed answer substitutions for them, respectively. We then relate these two classes through a program transformation, and present a characterisation of quasi-termination by means of the notion of ***quasi-acceptability*** of tabled programs. The latter provides us with a practical method of proving termination and the method is illustrated on non-trivial examples of tabled logic programs.

1 Introduction

The relative novelty of tabled-based logic programming implementations has given rise to a number of both theoretical and practical questions related to the study of the characteristics of such programs and to improving their performance. This work has been motivated by an initial study on how to adapt advanced program specialisation techniques, originally developed for standard logic programming, to the context of tabled logic programming. In a companion paper [12], we describe how left-propagation of bindings in a program executed under SLG-resolution [4] using a fixed left-to-right computation rule (as SLG is usually implemented in practical tabled-based systems such as XSB [15]) can seriously endanger the termination characteristics of that program. A simple example from [12] is the program

$p(X) \leftarrow p(Y), Y = f(X)$
$X = X \leftarrow$

For the query $\leftarrow p(X)$, the program finitely fails in SLG since its left-to-right evaluation encounters (a variant of) an existing call for which no answers can be produced. By performing one unfolding step with respect to the atom $Y = f(X)$, giving rise to the clause $p(X) \leftarrow p(f(X))$, the same query $\leftarrow p(X)$ will now result in an infinite computation under SLG.

In [12], preconditions for safe (i.e. termination preserving) program specialisation through unfolding are proposed. In the present paper we study the *universal* termination of tabled programs (where we are interested in the termination of the computation after all solutions have been generated), executed under SLG using a fixed left-to-right computation rule, in its own right. We adapt existing results on acceptability of programs with respect to sets of atoms from [6]. We show how this provides a natural necessary and sufficient condition for the termination of tabled logic programs. We also briefly discuss automatic verification of the proposed termination conditions, and present a non-trivial example (Appendix B).

The rest of the paper is structured as follows. In the next section we introduce some preliminaries and present a brief formalisation of SLG-resolution restricted to definite programs. In Section 3 we present the notions of quasi-termination and quasi-acceptability with respect to sets of atoms and prove that they are equivalent. In Section 4 we discuss the stronger notion of LG-termination and, again, characterise it in terms of quasi-termination. We end with a discussion.

2 Preliminaries

Throughout the paper, P will denote a definite tabled logic program. We also assume that in all programs all predicates are tabled (i.e. executed under SLG-resolution). The extended Herbrand Universe, U_P^E, and the extended Herbrand Base, B_P^E, associated with a program P, were introduced in [7]. They are defined as follows. Let $Term_P$ and $Atom_P$ denote the sets of respectively all terms and atoms that can be constructed from the alphabet underlying P. The variant relation, denoted $\approx$, defines an equivalence. U_P^E and B_P^E are respectively the quotient sets $Term_P/\approx$ and $Atom_P/\approx$. For any term t (or atom A), we denote its class in U_P^E (B_P^E) as $\tilde{t}$ (or $\tilde{A}$). However, when no confusion is possible, we omit the tildes. Finally, by $Pred_P$ and Fun_P we denote the sets of predicate and function symbols of P, respectively.

In this paper, we consider termination of SLG-resolution (see [4]), using a fixed left-to-right computation rule, for a *given set of atomic* (top level) *queries* with atoms in $S \subseteq B_P^E$. We will abbreviate SLG-resolution under the left-to-right computation rule by LG-resolution (which for definite programs is similar to OLDT-resolution [16, 11], modulo the fact that OLDT specifies a more fixed control strategy and uses subsumption checking and term-depth abstraction instead of variant checking to determine existence of subgoals and answers in the tables). Below, we formalise these notions. While doing so, we also introduce some example programs that are used throughout the paper to illustrate various aspects of (non)-termination of tabled logic programs.

2.1 SLG-Resolution for Definite Programs

We present a non-constructive definition of SLG-resolution that is sufficient for our purposes, and refer the reader to [2, 4, 16, 17] for more constructive formulations of (variants of) tabled resolution.

Definition 1. (pseudo SLG-tree, pseudo LG-tree) Let P be a definite program, $\mathcal{R}$ be a computation rule, and A an atom. A *pseudo SLG-tree for* $P \cup \{\leftarrow A\}$ *under* $\mathcal{R}$ is a tree τ_A such that:

1. the nodes of τ_A are labeled by goals along with an indication of the selected atom,
2. the arcs are labeled by sets of substitutions (either most general unifiers or computed answer substitutions),
3. the root of τ_A is $\leftarrow A$,
4. the children of the root $\leftarrow A$ are obtained by resolution against all matching (program) clauses in P,
5. the (possibly infinitely many) children of non-root nodes can only be obtained by resolving the selected (using $\mathcal{R}$) atom B of the node with clauses of the form $B\theta \leftarrow$ (not necessarily in P).

If $\mathcal{R}$ is the leftmost literal selection rule, τ_A is called a *pseudo LG-tree* for $P \cup \{\leftarrow A\}$.
We say that a pseudo SLG-tree τ_A for $P \cup \{\leftarrow A\}$ is *smaller* than another pseudo SLG-tree τ'_A for $P \cup \{\leftarrow A\}$ iff τ'_A can be obtained from τ_A by attaching new sub-branches to nodes in τ_A.
A (*computed*) *answer clause* of a pseudo SLG-tree τ_A for $P \cup \{\leftarrow A\}$ is a clause of the form $A\theta \leftarrow$ where θ is the composition of the answer substitutions found on a branch of τ_A whose leaf is labeled by the empty goal.

Intuitively, a pseudo SLG-tree (in an SLG-forest, see Definition 2 below) represents the tabled computation of all answers for a given subgoal labeling the root node of the tree. The trees in the above definition are called *pseudo* SLG-trees because there is no condition yet on which clauses $B\theta \leftarrow$ exactly are to be used for resolution in point 5. These clauses represent the answers found — possibly in another tree of the forest — for the selected atom. This interaction between the trees of an SLG-forest is captured in the following definition.

Definition 2. (SLG-forest, LG-forest) Let P be a program, $\mathcal{R}$ be a computation rule, and S be a (possibly infinite) set of atoms such that no two different atoms in S are variants of each other. $\mathcal{F}$ is an *SLG-forest* for P and S under $\mathcal{R}$ iff $\mathcal{F}$ is a set of minimal pseudo SLG-trees $\{\tau_A \mid A \in S\}$ where

1. τ_A is a pseudo SLG-tree for $P \cup \{\leftarrow A\}$ under $\mathcal{R}$,
2. every selected atom B of each node in some $\tau_A \in \mathcal{F}$ is a variant of an element B' of S, such that every clause resolved with B is a variant of an answer clause of $\tau_{B'}$ and *vice versa*.

Let $Q = \leftarrow A$ be an atomic query. An *SLG-forest* for $P \cup \{Q\}$ is an SLG-forest for a minimal set S with $A \in S$. An *LG-forest* is an SLG-forest containing only pseudo LG-trees.

Point 2 of Definition 2, together with the imposed minimality of trees in a forest, now uniquely determines these trees. So we can drop the designation "pseudo" and refer to (S)LG-trees in an (S)LG-forest. The notion of an (S)LG-forest introduces explicitly the relation between selected atoms and their computed answers. Also note that (S)LG-trees always have a finite depth.

Example 3 (NAT). Let NAT be the program below:

$nat(0) \leftarrow$
$nat(s(X)) \leftarrow nat(X)$

The (unique) (S)LG-forest for NAT and $\{nat(X)\}$, shown in Figure 1, consists of a single (S)LG-tree. Note that this tree has an infinitely branching node.

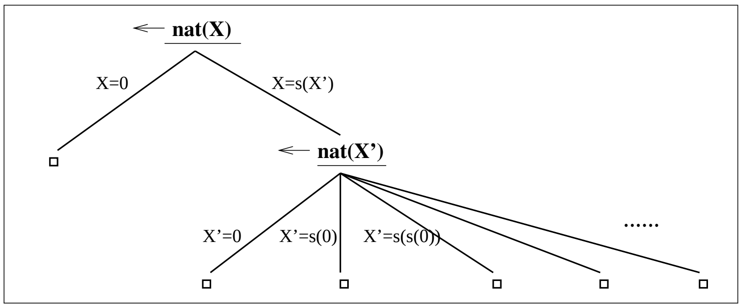

Fig. 1. SLG-forest for $NAT \cup \{\leftarrow nat(X)\}$.

Example 4. (NAT_{sol}) Let NAT_{sol} be the following program:

$nat(0) \leftarrow$
$nat(s(X)) \leftarrow nat(X), sol(nat(X))$
$nat_{sol}(X) \leftarrow nat(X), sol(nat(X))$
$sol(X) \leftarrow$

The LG-forest for NAT_{sol} $\cup\{\leftarrow nat(X)\}$ contains an infinite number of LG-trees; see Figure 2.

Finally, given $S \subseteq B_P^E$, by $Call(P, S)$ we denote the subset of B_P^E such that $\tilde{B} \in Call(P, S)$ whenever an element of $\tilde{B}$ is a selected literal in an LD-derivation for some $P \cup \{\leftarrow A\}$, with $\tilde{A} \in S$. We note that we can use the notions of LD-derivation and LD-computation even in the context of SLG-resolution, as the set of call patterns and the set of computed answer substitutions are not influenced by tabling; see e.g. Theorem 2.1 in [11].

3 Quasi-termination and quasi-acceptability

As the examples in the previous section show, an LG-forest can be infinite. In particular, whenever LD-computation from an initial goal leads to infinitely many different, non-variant calls, the corresponding LG-forest under tabled execution will be infinite. Obviously, such a phenomenon is undesirable, and therefore a

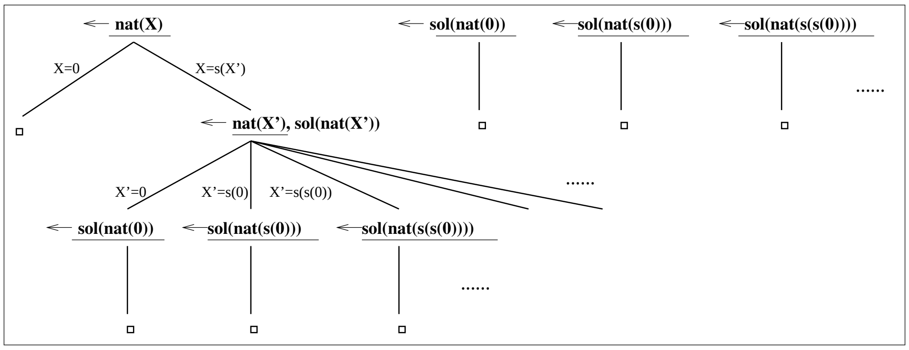

Fig. 2. The (infinite) LG-forest for Example 4.

first basic notion of termination studied in this paper is *quasi-termination* (a term borrowed from [9], defining a similar notion in the context of termination of off-line partial evaluation of functional programs). It is defined as follows.

Definition 5. (quasi-termination) Let P be a program and $S \subseteq B_P^E$. P *quasi-terminates with respect to* S iff for every $A \in S$, $Call(P, \{A\})$ is finite. Also, P *quasi-terminates* iff it quasi-terminates wrt B_P^E.

Lemma 6. *Let P be a program, $A \in B_P^E$, and let $\mathcal{F}$ be the LG-forest for $P \cup \{\leftarrow A\}$. Then P quasi-terminates wrt $\{A\}$ iff $\mathcal{F}$ consists of a finite number of LG-trees.*

This is the termination notion that also forms the heart of the study in [12].

Example 7. Consider the programs NAT and NAT_{sol} of Examples 3 and 4 respectively. With respect to $\{nat(X)\}$, NAT quasi-terminates, but NAT_{sol} does not.

In order to characterise the notion of quasi-termination, we adopt the following concept from [6].

Definition 8. (level mapping) A *level mapping* for a program P is a function $l : B_P^E \to I\!N$.

Definition 9. (finitely partitioning) A level mapping l is *finitely partitioning* iff for all $n \in I\!N : \#l^{-1}(n) < \infty$.

In other words, only finitely many atoms are mapped to any $n \in I\!N$.

We now introduce the notion of *quasi-acceptability*. It is adapted from the acceptability notion defined in [6] and not from the more "standard" definition of acceptability by Apt and Pedreschi in [1]. The reason for this choice is that the quasi-termination property of a tabled program and (set of) goal(s) is *not* invariant under substitution. Consider the following example from [12]:

Example 10. Let $p/2$ be a tabled predicate defined by the following clause.

$p(f(X), Y) \leftarrow p(X, Y)$

Then, the query $\leftarrow p(X, Y)$ quasi-terminates while $\leftarrow p(X, X)$ does not!

The acceptability notion in [1] is expressed in terms of ground instances of clauses and its associated notion of left-termination is expressed in terms of the set of all goals that are *bounded* under the given level mapping. Such sets are closed under substitution. Because quasi-termination lacks invariance under substitution, we need a stronger notion of acceptability, capable of treating *any* set of interest S.

Definition 11. (quasi-acceptability) Let l be a finitely partitioning level mapping on B_P^E. We say that P is *quasi-acceptable with respect to l and S* iff

- for every atom A such that $\tilde{A} \in Call(P, S)$,
- for every clause $H \leftarrow B_1, \ldots, B_n$ in P, such that $mgu(A, H) = \theta$ exists,
- for every initial subsequence $B_1, \ldots, B_{i-1}$ of $B_1, \ldots, B_n$ and every LD-computed answer substitution θ_{i-1} for $\leftarrow (B_1, \ldots, B_{i-1})\theta$:

$$l(A) \geq l(B_i \theta \theta_{i-1})$$

In brief, the main differences with the acceptability notion of [6] are that:

1. level mappings are assumed to be finitely partitioning the extended Herband Base, and
2. decreases of the level mapping are not assumed to be strict.

Intuitively, the reasons for these differences can be understood as follows. Contrary to SLD, computations in which a call has a variant of itself as its (immediate) recursive descendant do not lead to non-termination under tabled evaluation. As we want to allow such computations, requiring a strict decrease of the level mapping is too strong. On the other hand, we do not want to allow that infinitely many different calls within a same level occur (due to the non-strict decrease). To exclude this, we impose the finite partitioning requirement.

More directly, it should be intuitively clear that the combination of the non-strict decrease with finitely partitioning level mappings implies that only a finite number of different calls are allowed to descend from a given call. This corresponds to quasi-termination.

Given these definitions, we can now prove one of the main results of this paper.

Theorem 12. (termination condition) *Let P be a program and $S \subseteq B_P^E$. P is quasi-acceptable wrt some finitely partitioning level mapping l and S iff P is quasi-terminating with respect to S.*

For a proof, we refer to Appendix A.

Quasi-termination captures the property that, under LD-computation, a given atomic goal leads to only finitely many different calls. It is exactly in such cases that tabling aims at achieving actual termination of top-down logic program execution. Hence the importance of quasi-termination as a key property for tabled programs.

Also, in a broader context, techniques for proving quasi-termination can be of great value to ensure termination of off-line specialisation of logic programs (whether tabled or not). Currently, in all off-line partial evaluation methods for logic programs (e.g. [13, 10]) termination has to be ensured manually. In the context of off-line partial evaluation, quasi-termination is actually *identical* to termination of the partial evaluator. Thus, given a technique to establish quasi-termination, one can also establish whether a given binding time annotation will ensure termination or whether further abstraction is called for. This idea has already been successfully applied in the context of functional programming (cf. [8]), using the termination criterion of [9].

4 LG-termination

Even when tabling, quasi-termination as in Definition 5 and Lemma 6 only partially corresponds to our intuitive notion of a 'terminating computation'.

Example 13. Consider the program of Example 3. Given the set $S = \{nat(X)\}$, NAT quasi-terminates with respect to S. This is obvious since $Call(NAT, S) = \{nat(X)\}$, which is finite. The program is also quasi-acceptable with respect to S and *any* finitely partitioning level mapping $B^E_{NAT} \rightarrow \mathbb{N}$.

Nevertheless, this example does not correspond to our intuitive notion of a terminating program. Although the LG-forest is finite, its unique LG-tree is infinitely branching (see Figure 1) and the computation does not terminate. Notice however that the computed answers do not lead to new calls. Therefore, quasi-termination does hold. Also note that the same program is not LD-terminating either: there exists an infinite LD-derivation. To capture this, we define the following stronger termination notion.

Definition 14. (LG-termination) Let P be a program and $S \subseteq B^E_P$. P is *LG-terminating with respect to* S iff for every $A \in S$, the LG-forest for $P \cup \{\leftarrow A\}$ is a finite set of finite LG-trees.
Also, P is *LG-terminating* iff it is LG-terminating wrt B^E_P.

Let us now study how quasi-termination and LG-termination relate to each other. According to Lemma 6, quasi-termination only corresponds to part of the LG-termination notion; it does not capture infinitely branching LG-trees. But a simple program transformation can remedy this.

Definition 15. (sol(ution) transformation) The *sol transformation* is defined as follows:

- For a clause $C = H \leftarrow B_1, \ldots, B_n$ we define
  $C_{sol} = H \leftarrow B_1, sol(B_1), \ldots, B_n, sol(B_n)$

Here, $sol(B_i)$ is the syntactic application of the predicate $sol/1$ on B_i.

- For a program P, we define:

$$
\begin{aligned}
P_{sol} = &\{C_{sol} \mid C \in P\} \cup \\
&\{p_{sol}(X_1,\ldots,X_n) \leftarrow p(X_1,\ldots,X_n), \\
&\qquad\qquad sol(p(X_1,\ldots,X_n)) \mid p/n \in Pred_P\} \cup \\
&\{sol(X) \leftarrow\}
\end{aligned}
$$

where each p_{sol}/n is a new predicate symbol.

- For a set of atoms $S \subseteq B_P^E$, we define
  $S_{sol} = \{\tilde{p}_{sol}(t_1,\ldots,t_n) \mid \tilde{p}(t_1,\ldots,t_n) \in S\}$

The goal of the construction is that P LG-terminates wrt S iff P_{sol} quasi-terminates wrt S_{sol}. Note that P_{sol} is finite provided that P is also finite.

Example 16. The programs NAT of Example 3, and NAT_{sol} of Example 4 are related through the sol transformation.

Example 17. Let P be the following extension of the NAT program.

$t(X) \leftarrow nat(X), fail$
$nat(0) \leftarrow$
$nat(s(X)) \leftarrow nat(X)$

Then $Call(P, \{t(X)\}) = \{t(X), nat(X), fail\}$, so P quasi-terminates wrt $\{t(X)\}$. However, note that the query $\leftarrow t(X)$ does not terminate but fails infinitely in tabled evaluation under a left-to-right computation rule. Using the sol transformation, we have that $\{t(X)\}_{sol} = \{t_{sol}(X)\}$, and P_{sol} is the following program:

$t(X) \leftarrow nat(X), sol(nat(X)), fail, sol(fail)$
$nat(0) \leftarrow$
$nat(s(X)) \leftarrow nat(X), sol(nat(X))$
$t_{sol}(X) \leftarrow t(X), sol(t(X))$
$nat_{sol}(X) \leftarrow nat(X), sol(nat(X))$
$sol(X) \leftarrow$

Now, we have that

$$
\begin{aligned}
&Call(P_{sol}, \{t_{sol}(X)\}) = \\
&\qquad \{t_{sol}(X), t(X), nat(X), fail, sol(nat(0)), sol(nat(s(0))), \ldots\}
\end{aligned}
$$

and as expected P_{sol} does not quasi-terminate wrt $\{t(X)\}_{sol}$.

Theorem 18. *Let P be a program and $S \subseteq B_P^E$. Then P LG-terminates wrt S iff P_{sol} quasi-terminates wrt S_{sol}.*

Proof. (Sketch) As can be seen in Figure 3 for a node in an LG-tree labeled by a compound goal $\leftarrow \underline{B}, \overline{Q}$, where B is the selected atom, every answer clause resolution for B in P (with the answer clause $B\theta$) translates into two answer

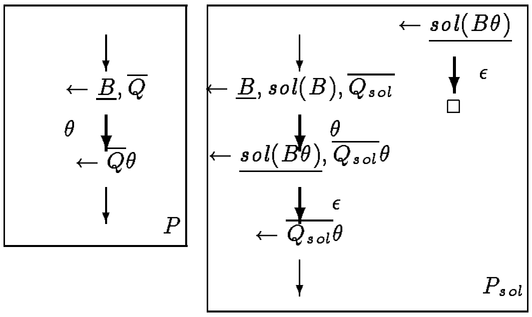

Fig. 3. Relating answer clause resolution in P and P_{sol}.

clause resolutions (for B and $\underline{sol(B\theta)}$) and one new LG-tree (for $\underline{sol(B\theta)}$) in P_{sol}. Note that in the same figure, $\overline{Q_{sol}}$ is the sol-translation of $\overline{Q}$. Let $A \in S$.
($\Rightarrow$) If $\leftarrow A$ LG-terminates in P, then we can build a finite set of finite LG-trees. Thus in P_{sol} only finitely many answer clause resolutions and finitely many LG-trees are added and, by Lemma 6, P_{sol} quasi-terminates wrt $\{A\}_{sol}$.
($\Leftarrow$) If $\leftarrow A$ does not LG-terminate in P, then we either have an infinite set of LG-trees, and thus also an infinite set of LG-trees for P_{sol}. Or we have an infinite branching in some tree, which translates into an infinite set of LG-trees for P_{sol}. So, by Lemma 6, in both cases P_{sol} does not quasi-terminate wrt $\{A\}_{sol}$.

A fully worked out example of a termination proof using the level mappings of Section 3 can be found in Appendix B.

5 Discussion

Tabled logic programming is receiving increasing attention in our community. It avoids many of the shortcomings of SLD(NF) execution and provides a more flexible and often extremely efficient execution mechanism for logic programs. In particular, tabled execution of logic programs terminates more often than execution based on LD-resolution. Nevertheless, termination can still not be guaranteed. Our motivation for the reported work is that we want to port program specialisation techniques for "standard" logic programs to tabled ones. In this attempt, we noticed that simple transformations, which are termination-preserving in the standard logic programming setting, can distort the termination behaviour in the setting of tabled logic programs. This motivated us to start a deeper study of termination of tabled logic programs, in the hope of using the results as tools in the further development and study of optimisation of tabled logic programs through transformations.

There are only relatively few works (in disguise) studying termination under tabled execution. [14], within the context of well-moded programs, gives a sufficient (but not necessary) condition for the bounded term-size property, which implies LG-termination. [9] provides another sufficient (but not necessary) condition for quasi-termination in the context of functional programming.

A specific concern that one might raise with respect to the termination conditions we propose is to what extent existing (automated) termination analyses can be adapted to verify the quasi-acceptability condition. Before addressing this question, we note that all programs that terminate under LD-resolution, are LG-terminating as well. Thus, verification of termination under LD-resolution (i.e., ignoring the designation of predicates as tabled) using an existing automated termination analysis (such as those surveyed in e.g. [5]) is a sufficient proof of the program's quasi-acceptability. Naturally this is not a necessary condition, since there are LG-terminating programs which do not terminate under LD-resolution (from trivial ones as the program in the introduction to more interesting ones as that presented in Appendix B). In such cases, existing termination analyses may require modifications or extensions. The most important issue in this context is to what extent the focus on *finitely partitioning* level mappings becomes a bottleneck in providing level mappings in practice. Or, more specifically, which of the level mappings used in practice in (automated) termination analysis are finitely partitioning?

In most automatic approaches to termination analysis, level mappings are defined in terms of positive linear combinations of *norms*. A norm is a function $n : U_P^E \to I\!N$. Given such a norm n, a level mapping is induced by selecting a number of argument positions, I_p, for each predicate symbol p/k, and by defining: $l(p(t_1, \ldots, t_k)) = \sum_{i \in I_p} p_i n(t_i)$, where all p_i are natural numbers.

Note that if the selected norm n is 'finitely partitioning', meaning that for every $m \in I\!N : \#n^{-1}(m) < \infty$, and $I_p = [1, k]$ as well as $p_i > 0$, then the level mapping l is finitely partitioning as well (assuming that the language underlying P only contains a finite number of predicate symbols). So, the question is: which norms are finitely partitioning ?

Let us first assume that the language underlying P contains only a finite number of constants and functions symbols. In that case, consider any semi-linear norm [3], which are norms that can be defined as:

$$n(f(t_1, \ldots, t_k)) = c_f + \sum_{i \in I_f} n(t_i)$$

where $c_f \in I\!N$, $I_f \subseteq [1, k]$ and depend on f/k only.

Given the restriction on the language underlying P, any semi-linear norm which has $c_f > 0$ and $I_f = [1, k]$, for all f/k in Fun_P, (in other words no part of the term can be ignored) is finitely partitioning. In particular, the well-known *termsize* norm is finitely partitioning (but e.g. the list-length norm is not).

If the underlying language contains an infinite number of constants (e.g. the natural numbers), we can still design very natural finitely partitioning norms by exploiting any existing well-founded ordering on the infinite data set and incorporating that in the norm.

Therefore, automated termination analysis can be quite easily adapted to the case of quasi-termination and in fact requires nothing more than moving from a strict inequality check to verifying the weaker inequality expressed in quasi-acceptability.

However, the requirement that $c_f > 0$ can in some cases complicate the task of finding appropriate norms and level mappings, as the following example illustrates.

Example 19. Let P just consist of the following clause C:

$p([H_1, H_2|T]) \leftarrow p([[H_1, H_1, H_2]|T])$

Let us restrict our attention to calls $p(t)$ where t is a ground list. Then P quasi-terminates (failing finitely) and it even terminates under ordinary, un-tabled evaluation. For the un-tabled case we can simply use the level mapping $l(p(t)) = list\_length(t)$ and we have that the level mapping of the head of a ground instance of C is strictly larger than the level mapping of the corresponding body atom. As previously mentioned, a termination proof under un-tabled execution implies LG-termination. However, such an indirect proof of LG-termination could not be used if the program also contained the clause $p(X) \leftarrow p(X)$. In such a case, the list length norm could not be used to directly prove quasi-acceptability as it is not finitely partitioning (there are infinitely many different terms up to variable renaming with the same list length) and we would have to search for another candidate. Unfortunately, we could not simply use the termsize norm either, because it will lead to the heads of ground instances of C being strictly smaller than the corresponding body atoms. Thus, using the termsize norm we are not able to prove quasi-acceptability and one has to resort to more powerful level mappings. The following, slightly involved, level mapping, where we denote the termsize by $||.||$, will fortunately be sufficient:

$l(p([t_1, \ldots, t_n])) = 2^n(n + \sum_{i\in[1,n]} ||t_i||)$

Now let $p([t_1, t_2|t]) \leftarrow p([[t_1, t_1, t_2]|t])$ be any ground instance of C, where $list\_length([t_1, t_2|t]) = n$, $t = [s_1, \ldots, s_{n-2}]$ and $s = \sum_{i\in[1,n-2]} ||s_i||$. We have that $l(p([t_1, t_2|t])) = 2^n(n + ||t_1|| + ||t_2|| + s)$ which is (strictly) larger than $l(p([[t_1, t_1, t_2]|t])) = 2^{n-1}(n - 1 + 1 + 2||t_1|| + ||t_2|| + s)$.

Acknowledgements

Stefaan Decorte and Michael Leuschel are supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation". Danny De Schreye is a senior research associate of the Belgian National Fund for Scientific Research. Bern Martens is a post-doctoral fellow of the K.U.Leuven Research Council. Konstantinos Sagonas is a post-doctoral fellow of the Fund for Scientific Research — Flanders Belgium (FWO).

References

1. K.R. Apt and D. Pedreschi. Reasoning about Termination of Pure Prolog Programs. *Information and Computation*, 106(1):109–157, 1993.

2. R. Bol and L. Degerstedt. The Underlying Search for Magic Templates and Tabulation. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 793–811, Budapest, Hungary, June 1993. The MIT Press.
3. A. Bossi, N. Cocco, and M. Fabris. Norms on Terms and their use in Proving Universal Termination of a Logic Program. *Theoretical Computer Science*, 124(2):297–328, February 1994.
4. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.
5. D. De Schreye, S. Decorte. Termination of Logic Programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, May/July 1994.
6. D. De Schreye, K. Verschaetse, and M. Bruynooghe. A Framework for Analysing the Termination of Definite Logic Programs with respect to Call Patterns. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'92)*, pages 481–488, ICOT Tokyo, 1992. ICOT.
7. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behaviour of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
8. A. J. Glenstrup and N. D. Jones. BTA algorithms to ensure Termination of off-line Partial Evaluation. In *Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference*, LNCS, pages 25–28. Springer-Verlag, June 1996.
9. C. K. Holst. Finiteness Analysis. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA)*, number 523 in LNCS, pages 473–495. Springer-Verlag, August 1991.
10. J. Jørgensen and M. Leuschel. Efficiently Generating Efficient Generating Extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, number 1110 in LNCS, pages 238–262, Schloß Dagstuhl, 1996. Springer-Verlag.
11. T. Kanamori and T. Kawamura. OLDT-based Abstract Interpretation. *Journal of Logic Programming*, 15(1&2):1–30, January 1993.
12. M. Leuschel, B. Martens, and K. Sagonas. Preserving Termination of Tabled Logic Programs While Unfolding. In N. Fuchs, editor, *Proceedings of Logic Program Synthesis and Transformation (LOPSTR'97)*, Leuven, Belgium, July 1997.
13. T. Mogensen and A. Bondorf. Logimix: A self-applicable Partial Evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *Proceedings of Logic Program Synthesis and Transformation (LOPSTR'92)*, pages 214–227. Springer-Verlag, 1992.
14. L. Plümer. *Termination Proofs for Logic Programs*, number 446 in LNCS. Springer-Verlag, 1990.
15. K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM Press.
16. H. Tamaki and T. Sato. OLD Resolution with Tabulation. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, number 225 in LNCS, pages 84–98, London, July 1986. Springer-Verlag.
17. L. Vieille. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, 1989.

A Termination condition

The following concept will be useful for proving our termination condition. To any fixed P and S, we can associate a call graph as follows.

Definition 20. (call graph associated to S) Let P be a program and $S \subseteq B_P^E$. The *call graph* $\mathit{Call\text{-}Gr}(P,S)$ *associated to* P *and* S is a graph such that:

- its set of nodes is $\mathit{Call}(P,S)$,
- there exists a directed arc from $\tilde{A}$ to $\tilde{B}$ iff
  - there exists a clause $H \leftarrow B_1, \ldots, B_n$ in P, such that $mgu(A,H) = \theta$ exists,[1] and
  - there exists $i \in [1,n]$, such that there is an LD-refutation for

$$\leftarrow (B_1, \ldots, B_{i-1})\theta$$

with computed answer substitution θ_{i-1}, and $B \approx B_i\theta\theta_{i-1}$.

The notion of a call graph has a particularly interesting property, which will be useful in the study of termination.

Proposition 21. (paths and selected atoms) *Let P be a program, $S \subseteq B_P^E$, $\mathit{Call}(P,S)$ and $\mathit{Call\text{-}Gr}(P,S)$ be defined as above. Let p be any directed path in $\mathit{Call\text{-}Gr}(P,S)$. Then there exists an LG-derivation for some element of $\mathit{Call}(P,S)$, such that all the nodes in p occur as selected atoms in the derivation.*

Proof. By definition of $\mathit{Call\text{-}Gr}(P,S)$, for every arc from $\tilde{A}$ to $\tilde{B}$ in $\mathit{Call\text{-}Gr}(P,S)$, there exists a sequence of consecutive LG-derivation steps, starting from $\leftarrow A$ and having a variant of B as its selected atom at the end. Because (a variant of) B is selected at the end-point, any two such derivation-step sequences, corresponding to consecutive arcs in $\mathit{Call\text{-}Gr}(P,S)$, can be composed to form a new sequence of LG-derivation steps. In this sequence, all 3 nodes of the consecutive arcs remain selected atoms in the new sequence of derivation steps. Transitively exploiting the above argument yields the result.

Note that by definition of $\mathit{Call\text{-}Gr}(P,S)$ and $\mathit{Call}(P,S)$, this also implies that there is a sequence of derivation steps starting from $P \cup \{\leftarrow A\}$, with $\tilde{A} \in S$, such that all nodes in the given path p are selected atoms in the derivation sequence.

Theorem 12 (termination condition) *Let P be a program and $S \subseteq B_P^E$. P is quasi-acceptable with respect to some finitely partitioning level mapping l and S iff P is quasi-terminating with respect to S.*

Proof. The "only-if" part of the proof is fairly straightforward. We have that P is quasi-acceptable with respect to l and S. Take any atom A with $\tilde{A} \in S$. Due to the acceptability condition, any call in $\mathit{Call}(P,S)$ directly descending from

[1] Throughout the paper we assume that representatives of equivalence classes are systematically provided with fresh variables, to avoid the necessity of renaming apart.

A, say B, is such that $l(A) \geq l(B)$. The same holds recursively for the atoms descending from B. Thus, the level mapping of any call, recursively descending from A, is smaller than or equal to $l(A) \in \mathbb{N}$. Since l is finitely partitioning, we have that: $\cup_{n \leq l(A)} l^{-1}(n)) < \infty$. Hence, $\#Call(P, \{\tilde{A}\}) < \infty$ and the program is quasi-terminating for $\leftarrow A$.

The "if"-part of the proof is slightly more complicated. Given is that P quasi-terminates for all atomic queries $\leftarrow A$, with $\tilde{A} \in S$. We need to design a finitely partitioning level mapping, l, such that the quasi-acceptability condition with respect to l and s holds.

First of all, we will only define l on the elements of $Call(P, S)$. On elements of the complement of $Call(P, S)$ in B_P^E, l can be assigned any value (as long as we don't group infinitely many in one layer), as these elements do not turn up in the inequality condition of quasi-acceptability. In order to define l on $Call(P, S)$, consider the $Call\text{-}Gr(P, S)$-graph.

A first crucial point in this part of the proof is that the strongly connected components of $Call\text{-}Gr(P, S)$ are necessarily all finite. To see this, assume that $Call\text{-}Gr(P, S)$ contains an infinite strongly connected component. Then, there exists an infinitely long path p, starting from an element in S, through elements of this infinite strongly connected component. By Proposition 21, this means that there exists a derivation for that element in S, such that an infinite number of different atoms from $Call(P, S)$ are selected within this derivation. Obviously, this contradicts quasi-termination.

So, all strongly connected components of $Call\text{-}Gr(P, S)$ are finite. Define $Call\text{-}Gr(P, S)/reduced$ as the graph obtained from $Call\text{-}Gr(P, S)$ by replacing any strongly connected component by a single contracting node and replacing any arc from $Call\text{-}Gr(P, S)$ pointing to (resp. from) any node in that strongly connected component by an arc to (resp. from) the contracting node.

$Call\text{-}Gr(P, S)/reduced$ does not have any (non-trivial) strongly connected components. Moreover, any strongly connected component from $Call\text{-}Gr(P, S)$ that was collapsed into a contracting node of $Call\text{-}Gr(P, S)/reduced$ necessarily consists of only a finite number of nodes.

We now define l as follows.

Let layer-0 be the set of leaves in $Call\text{-}Gr(P, S)/reduced$. We will order these nodes and assign to them an odd number within $\mathbb{N}$. For every atom in a strongly connected component represented by one of the nodes in layer-0, we will assign the level mapping of that atom to be equal to the assigned value of the corresponding contracting node.

Then, we move up to the next layer in $Call\text{-}Gr(P, S)/reduced$. This layer, layer-1, consists of all nodes N, such that:

$$max(\{length(p) \mid p \text{ is a path starting from } N \text{ in } Call\text{-}Gr(P, S)/reduced\}) = 1$$

To any element of layer-1, we assign a natural number n, such that n is larger than all the natural numbers assigned to descendants of this node in layer-0 (note that the graph is finitely branching), but such that there is at least 1 natural number larger than the ones assigned to its descendants which remains unassigned.

We continue this process layer by layer. In each step we make sure that an infinite number of natural numbers remain "unassigned" to nodes, but also that numbers assigned to nodes in higher levels of $Call\text{-}Gr(P,S)$ are strictly larger than the numbers assigned to their descendants. The value of the level mapping on elements of $Call(P,S)$ is as defined for layer-0 above: all calls in a same strongly connected component of $Call(P,S)$ receive the number assigned to their representative in $Call\text{-}Gr(P,S)/reduced$.

Due to this construction, l is finitely partitioning on $Call(P,S)$. Also, by construction again, the quasi-acceptability condition of P with respect to l and S holds.

B An extended termination proof

In this appendix, the approach is illustrated on a non-trivial example. Systems like the one in [6], after the slight adaptations discussed in Section 5, are able to automatically derive this proof. Consider the following context-free grammar, defining well-built numerical expressions, built up from integers and the operators $+$ and $*$.

$Expr \Rightarrow Term \mid Expr + Term$
$Term \Rightarrow Primary \mid Term * Primary$
$Primary \Rightarrow \mathbf{integer} \mid (\; Expr \;)$

If we encode expressions as ground lists of tokens, we can use the following program $Parse$ to parse such expressions in all possible ways.

$expr(A,B) \leftarrow term(A,B)$
$expr(A,B) \leftarrow expr(A,C), C = [+|D], term(D,B)$
$term(A,B) \leftarrow primary(A,B)$
$term(A,B) \leftarrow term(A,C), C = [*|D], primary(D,B)$
$primary(A,B) \leftarrow A = [C|B], integer(C)$
$primary(A,B) \leftarrow A = ['('|C], expr(C,D), D = [')'|B]$

It is not difficult to see that the program is *not* LD-terminating wrt the following set of atoms:

$$S = \{expr(l,t) \mid l \text{ is a list of atoms and } t \text{ is any term } \}$$

For example, the query $\leftarrow expr([1],B)$ generates the answer $B = [\,]$ and then gets stuck in an infinite loop.

The program is however LG-terminating. To see this, consider $Parse_{sol}$ which corresponds to the following program.

$expr(A,B) \leftarrow term(A,B), sol(term(A,B))$
$expr(A,B) \leftarrow expr(A,C), sol(expr(A,C)), C = [+|D], sol(C = [+|D]),$
$\qquad term(D,B), sol(term(D,B))$
$term(A,B) \leftarrow primary(A,B), sol(primary(A,B))$
$term(A,B) \leftarrow term(A,C), sol(term(A,C)), C = [*|D], sol(C = [*|D]),$
$\qquad primary(D,B), sol(primary(D,B))$

$$primary(A,B) \leftarrow A=[C|B], sol(A=[C|B]), integer(C), sol(integer(C))$$
$$primary(A,B) \leftarrow A=['('|C], sol(A=['('|C]), expr(C,D),$$
$$sol(expr(C,D)), D=[')'|B], sol(D=[')'|B])$$
$$expr_{sol}(A,B) \leftarrow expr(A,B), sol(expr(A,B))$$
$$term_{sol}(A,B) \leftarrow term(A,B), sol(term(A,B))$$
$$primary_{sol}(A,B) \leftarrow primary(A,B), sol(primary(A,B))$$
$$sol(X) \leftarrow$$

The following level mapping can then be used to prove quasi-acceptability of $Parse_{sol}$ wrt S_{sol}, which according to Theorem 18 establishes LG-termination of $Parse$ with respect to S.

$$\begin{aligned}
&||term(t_1,t_2)|| \equiv ||expr(t_1,t_2)|| \equiv ||primary(t_1,t_2)|| \\
&\qquad\qquad = 2 \times termsize(t_1) + termsize(t_2) + 1 \\
&||t_1 = t_2|| = termsize(t_1) + termsize(t_2) \\
&||integer(n)|| = abs(n), \text{ if } n \text{ is an integer} \\
&||integer(t)|| = termsize(t), \text{ if } t \text{ is not an integer} \\
&||sol(t)|| = \begin{cases} termsize(t_1) + termsize(t_2) & \\ & \text{if } t \text{ is of the form } p(t_1,t_2) \text{ and} \\ & p/2 \in \{expr/2, term/2, primary/2, =/2\} \\ termsize(t) & \text{otherwise} \end{cases} \\
&||p_{sol}(\bar{t})|| = ||p(\bar{t})||, \forall p/n \in Pred_P
\end{aligned}$$

The following set is a superset of $Call(Parse_{sol}, S_{sol})$:

$$\begin{aligned}
\{&expr(l,t), term(l,t), primary(l,t), \\
&\quad integer(g), sol(t), expr_{sol}(l,t), \\
&\quad\quad term_{sol}(l,t), primary_{sol}(l,t) \mid l \text{ is a ground list of atoms,} \\
&\qquad\qquad\qquad t \text{ is any term and } g \text{ is a ground term}\}
\end{aligned}$$

As $Fun_{Parse_{sol}}$ is a finite set and $||A||$ is defined in terms of the termsize of all arguments of A, $||.||$ is finitely partitioning. We can then prove quasi-acceptability of $Parse_{sol}$ wrt S_{sol}. Consider the first clause defining $expr/2$ and take any atom $expr(l,t) \in Call(Parse_{sol}, S_{sol})$ such that $mgu(expr(l,t), expr(A,B)) = \theta$ exists. Obviously, $||expr(l,t)|| \geq ||term(A,B)\theta||$ as $expr(l,t) = expr(A,B)\theta$.

To prove that $||expr(l,t)|| \geq ||sol(term(A,B))\theta\theta_2||$ for any θ_2 such that θ_2 is a computed answer substitution for $\leftarrow term(A,B)\theta$, we need the following observation. In the following, let $t_1, t_2 \in Term_{Parse_{sol}}$. The set:

$$\begin{aligned}
&\{expr(t_1,t_2), term(t_1,t_2), primary(t_1,t_2) \mid termsize(t_1) \geq termsize(t_2)\} \cup \\
&\{expr_{sol}(t_1,t_2), term_{sol}(t_1,t_2), primary_{sol}(t_1,t_2) \mid termsize(t_1) \geq termsize(t_2)\} \\
&\cup \{t_1 = t_2 \mid termsize(t_1) = termsize(t_2)\} \ \cup \ \{integer(n) \mid n \text{ is an integer}\}
\end{aligned}$$

is a model of $Parse_{sol}$.

Now, whenever θ_2 is a computed answer substitution for $\leftarrow term(A,B)\theta$, $\theta\theta_2$ is a grounding substitution. Consequently, for $term(A,B)\theta\theta_2$, $termsize(A\theta\theta_2) \geq termsize(B\theta\theta_2)$. Therefore and because $A\theta$ is ground, we have that

$$||expr(l,t)|| = ||expr(A,B)\theta|| = 2 \times termsize(A\theta) + termsize(B\theta) + 1$$

$$
\begin{aligned}
&\geq 2 \times termsize(A\theta\theta_2)\\
&\geq termsize(A\theta\theta_2) + termsize(B\theta\theta_2)\\
&= ||sol(term(A, B))\theta\theta_2||
\end{aligned}
$$

Consider now the second clause of $expr/2$. Again, take any atom $expr(l, t) \in Call(Parse_{sol}, S_{sol})$ such that $mgu(expr(l, t), expr(A, B)) = \theta$ exists. We can apply the same arguments as above to infer that $||expr(l, t)|| \geq ||expr(A, C)\theta||$ and that $||expr(l, t)|| \geq ||sol(expr(A, C))\theta\theta_2||$, θ_2 being as above. To prove that $||expr(l, t)|| \geq ||C = [+|D]\theta\theta_3||$ for any θ_3 such that θ_3 is a computed answer substitution for $\leftarrow (expr(A, C), sol(expr(A, C)))\theta$, we need the observation that $expr(A, C)\theta\theta_3$ is ground and thus $termsize(A\theta\theta_3) \geq termsize(C\theta\theta_3)$. Therefore and because $A\theta$ is ground, we have that:

$$
\begin{aligned}
||expr(l, t)|| &= 2 \times termsize(A\theta\theta_3) + termsize(B\theta) + 1\\
&\geq termsize(C\theta\theta_3) + 1 \geq ||C = [+|D]\theta\theta_3||
\end{aligned}
$$

We can use similar arguments on all remaining atoms and clauses.