Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation*

Germán Vidal

DSIC, Technical University of Valencia, Spain
gvidal@dsic.upv.es

Abstract. A logic program strongly quasi-terminates when only a finite number of distinct atoms (modulo variable renaming) are derivable from any given query and computation rule. This notion of quasi-termination, though stronger than related notions that only consider Prolog's computation rule, is essential for ensuring the termination of partial evaluation, where liberal selection policies are often mandatory to achieve a good specialization.
In this paper, we introduce sufficient conditions for the strong termination and quasi-termination of logic programs which are based on the construction of size-change graphs. The class of strongly quasi-terminating logic programs, however, is too restricted. Therefore, we also introduce an annotation procedure that annotates those predicate arguments which are responsible of the non-quasi-termination. As a consequence, the annotated program behaves like a quasi-terminating program if annotated arguments are generalized (i.e., replaced by a fresh variable) when they occur in a computation. We illustrate the usefulness of our approach by designing a simple partial evaluator in which global termination is always ensured *offline* (i.e., statically). A prototype implementation demonstrates its viability.

1 Introduction

Partial evaluation [19] is a well-known technique for program specialization. Basically, given a program, pgm, and a partition of its input data into the so called *static* (i.e., known) and *dynamic* (i.e., unknown) data, a partial evaluator returns a *residual* program $\mathtt{pgm_s}$ which is a specialized version of pgm for the static data s such that $\mathtt{pgm}(\mathtt{s},\mathtt{d}) = \mathtt{pgm_s}(\mathtt{d})$ for all values of the missing dynamic data d. In the partial evaluation of logic programs (also known as partial deduction [32]), both the static and the dynamic data are provided in the form of a query, which is usually less instantiated than ordinary run time queries.

Roughly speaking, given a logic program P and a query Q, a partial evaluation system should construct a finite—possibly incomplete—SLD tree for Q with P such that each atom in the leaves of the tree is an instance of a previously selected atom. The finiteness of this SLD tree can be ensured either *online* or *offline*. Online partial evaluation techniques (e.g., [25, 27]) use rather expensive tests—like the *homeomorphic embedding* [24]—for avoiding infinite derivations. In contrast, offline partial

* This work has been partially supported by the EU (FEDER) and the Spanish MEC under grants TIN2004-00231 and TIN2005-09207-C03-02, and by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054.

evaluation (e.g., [26]) proceeds in two stages: first, the program is analyzed—using a so called binding-time analysis, BTA—so that an *annotated* program is returned; program annotations are used to point out which atoms can be unfolded, which arguments should be generalized (i.e., replaced by fresh variables), etc, in order to guarantee the termination of the specialization process. Then, the second stage is an extension of an SLD interpreter that simply obeys the annotations. Unfortunately, there is no fully automatic BTA for ensuring the termination of offline partial evaluation yet. Recent progress include [9], which presents a fully automatic BTA but only provides partial termination guarantees.

Holst [18] was the first to relate (within the functional programming paradigm) the termination of partial evaluation and the *quasi-termination* of the computations. The notion of quasi-termination can be traced back to term rewriting, where a system is called quasi-terminating if all its derivations contain only a finite number of distinct terms [12]. In logic programming, quasi-termination has been studied, e.g., in [10, 28, 34, 37, 38], mainly in the context of tabled evaluation; in this context, a logic program quasi-terminates when only finitely many different atoms—up to variable renaming—are derivable from any given query.

In this work, we introduce a sufficient condition for the quasi-termination of logic programs that can be used for ensuring the termination of offline partial evaluation. In contrast to many termination analysis for logic programs—which consider Prolog's leftmost computation rule—we need a stronger notion of termination. In particular, one should consider the termination of SLD resolution w.r.t. all possible computation rules. This is essential because liberal selection policies are often mandatory to achieve a good specialization (see, e.g., [1, 25]). Therefore, we consider *strong termination* [2]: a program P and query Q strongly terminate if they universally terminate (i.e., the computation of all solutions terminate) w.r.t. all computation rules. This is the strongest notion of termination and, as noted in [33], strongly terminating programs and queries are generally trivial programs.

Fortunately, the fact that the class of strongly terminating programs is so limited is not a problem in our context. On the one hand, we consider a larger class of programs, namely strongly *quasi-terminating* programs, since quasi-termination suffices to ensure the termination of partial evaluation. On the other hand, even if a program is not strongly quasi-terminating, we can still use the information of the analysis to annotate the problematic arguments of the program atoms. In this way, we can easily design an offline partial evaluator that takes an annotated program and uses an extension of SLD resolution where annotated arguments are generalized so that every computation is still strongly quasi-terminating.

We formalize our quasi-termination analysis by means of *size-change* graphs [22], which (together with monotonicity constraints [4]) are currently used by an increasing number of termination analyzers for different programming languages. Basically, one should first approximate the transition relation of the program by means of size-change graphs; then, we compute the closure of this set under an appropriate composition operator; finally, the program terminates if every idempotent graph in this closure includes a strictly decreasing parameter. Another popular approach to prove termination of logic programs is based on the computation of (an approximation of) the *binary unfoldings* of a program. This approach, however, has only been formalized

for a leftmost computation rule [8] or for a local[1] computation rule [14]. Adapting it for considering strong termination implies redoing almost everything from scratch (and would likely produce a technique closely related to our developments based on size-change graphs).

Our main contributions can be summarized as follows: i) first, we adapt the notion of size-change graph to logic programs; ii) then, we present sufficient conditions for both strong termination and quasi-termination; iii) we also define a program annotation procedure that allows us to ensure the termination of partial evaluation for programs which are not quasi-terminating. iv) finally, we describe the design of an offline partial evaluator based on the program annotation procedure.

The paper is organized as follows. After introducing some preliminaries in the next section, we adapt the notion of size-change graph in Sect. 3, where we also introduce a sufficient condition for termination. Then, Sect. 4 focuses on quasi-termination and presents two alternative characterizations. Sect. 5 presents a simple offline partial evaluator for logic programs in which termination is ensured by means of our quasi-termination analysis. In particular, a program annotation procedure, based on the results of Sect. 4 is defined. Some results with a prototype implementation of the partial evaluator are also presented. Finally, Sect. 6 discusses some related work and Sect. 7 concludes and presents some possibilities for future work.

2 Preliminaries

We assume some familiarity with the standard definitions and notations for logic programs [31]. In this work, we consider a first-order language with a fixed vocabulary of predicate symbols, function symbols, and variables denoted by Π, Σ and $\mathcal{V}$, respectively. We let $\mathcal{T}(\Sigma, \mathcal{V})$ denote the set of *terms* constructed using symbols from Σ and variables from $\mathcal{V}$. An *atom* has the form $p(t_1, \ldots, t_n)$ with $p/n \in \Pi$ and $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$ for $i = 1, \ldots, n$. A *query* is a finite sequence of atoms $\langle A_1, \ldots, A_n \rangle$, where the *empty query* is denoted by *true*. A *clause* has the form $H \leftarrow B_1, \ldots, B_n$ where $H, B_1, \ldots, B_n$, $n \geqslant 0$, are atoms (thus we only consider *definite* programs). A logic *program* is a finite sequence of clauses. Given a program P, the associated extended (non-ground) Herbrand Universe and Base [13] are denoted by U_P^E and B_P^E, respectively. $\mathcal{V}ar(s)$ denotes the set of variables in the syntactic object s (i.e., s can be either a term, an atom, a query, or a clause). A syntactic object s is *ground* if $\mathcal{V}ar(s) = \varnothing$.

Substitutions and their operations are defined as usual. In particular, the set $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is called the *domain* of a substitution σ. The *most general unifier* of two syntactic objects, s_1 and s_2, is denoted by $mgu(s_1, s_2)$. A syntactic object s_1 is *more general* than a syntactic object s_2, denoted $s_1 \leqslant s_2$, if there exists a substitution θ such that $s_2 = s_1\theta$. A *variable renaming* is a substitution that is a bijection on $\mathcal{V}$. Two syntactic objects t_1 and t_2 are *variants* (or equal up to variable renaming), denoted $t_1 \approx t_2$, if $t_1 = t_2\rho$ for some variable renaming ρ.

The notion of *computation rule* $\mathcal{R}$ is used to select an atom within a query for its evaluation. Given a program P, a query $Q = \langle A_1, \ldots, A_n \rangle$, and a computation

[1] Roughly speaking a computation rule is *local* [39] when it always select in a query one of the "most recently" introduced atoms in the derivation (e.g., the atoms that come from the body of the last clause used in the derivation).

rule $\mathcal{R}$, we say that $Q \leadsto_{P,\mathcal{R},\sigma} Q'$ is an *SLD resolution step* for Q with P and $\mathcal{R}$ if $\mathcal{R}(Q) = A_i$, $1 \leqslant i \leqslant n$, is the selected atom,[2] $H \leftarrow B_1, \ldots, B_m$ is a renamed apart clause of P (in symbols $H \leftarrow B_1, \ldots, B_m \ll P$), $\sigma = mgu(A, H)$, and $Q' = (\langle A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n\rangle)\sigma$; we often omit P, $\mathcal{R}$ and/or σ in the notation of an SLD resolution step when they are clear from the context. An *SLD derivation* is a (finite or infinite) sequence of SLD resolution steps. We often use $Q_0 \leadsto_\theta^* Q_n$ as a shorthand for $Q_0 \leadsto_{\theta_1} Q_1 \leadsto_{\theta_2} \ldots \leadsto_{\theta_n} Q_n$ with $\theta = \theta_1 \circ \cdots \circ \theta_n$ (where $\theta = \{\}$ if $n = 0$). An SLD derivation $Q \leadsto_\theta^* Q'$ is *successful* when $Q' = \mathit{true}$; in this case, we say that θ is the *computed answer substitution*. SLD derivations are represented by a (possibly infinite) finitely branching tree.

As it is common practice in *partial evaluation* [32], we adopt the convention that SLD derivations can be either infinite, successful, failed (e.g., $Q \leadsto_\theta^* Q'$ such that $Q' \neq \mathit{true}$ and no SLD resolution step can be applied to Q'), or *incomplete*, in the sense that at any point we are allowed to simply not select any query atom and terminate the derivation.

3 A Sufficient Condition for Strong Termination

In this section, present a sufficient condition for the strong termination of a logic program. For this purpose, we adapt the notions of *size-change graph* and *size-change termination* [22]. Then, we show that size-change termination does not generally imply the (strong) termination of SLD computations. Therefore, we introduce a sufficient condition for termination which includes some additional requirements. We postpone the study of quasi-termination to the next section.

Definition 1 (strong termination [2]**).** *A query Q is strongly terminating w.r.t. a program P if every SLD derivation for Q with P is finite. A program P is strongly terminating w.r.t. a set of atomic queries*[3] *$\mathcal{Q}$ if every $Q \in \mathcal{Q}$ is strongly terminating w.r.t. P. A program P is strongly terminating if it is strongly terminating w.r.t. B_P^E.*

We note that [2] says that a program P is strongly terminating when it is strongly terminating w.r.t. the set of all *ground* queries B_P. Definition 1 above slightly generalizes this notion to arbitrary atoms in order to be useful in the context of partial evaluation.

For conciseness, in the remainder of this paper we write termination to refer to strong termination.

The following auxiliary definitions introduce the notion of calls and the calls-to relation (slightly extended from [8] to consider an arbitrary computation rule).

Definition 2 (calls). *Let P be a program, $\mathcal{R}$ a computation rule, and Q_0 a query. We say that A is a call in a derivation of Q_0 with P and $\mathcal{R}$ iff $Q_0 \leadsto^* Q$ and $\mathcal{R}(Q) = A$. We denote by $\mathit{calls}_P^{\mathcal{R}}(Q_0)$ the set of calls in the computations of Q_0 with P and $\mathcal{R}$.*

[2] Note that $\mathcal{R}$ may also take into account the computation history, which is usual in partial evaluation.

[3] We only consider atomic initial queries for simplicity. Nevertheless, any arbitrary query could easily be encoded by adding a new clause definition to the program.

Definition 3 (calls-to relation $\hookrightarrow$). *We say that there is a call from A to B in a computation of the query Q_0 with the program P and the computation rule $\mathcal{R}$, in symbols $A \overset{Q_0}{\hookrightarrow}_{P,\mathcal{R}} B$, if $A \in calls_P^{\mathcal{R}}(Q_0)$ and $B \in calls_P^{\mathcal{R}}(\langle A \rangle)$. When it is clear from the context, we write $A \hookrightarrow B$ or $A \hookrightarrow_\sigma B$ to emphasize that σ is the substitution associated with a corresponding derivation from $\langle A \rangle$ to $\langle \ldots, B, \ldots \rangle$.*

Trivially, if a program P is terminating w.r.t. S, then the set $calls_P^{\mathcal{R}}(A)$ is finite for every atom $A \in S$ and computation rule $\mathcal{R}$. The inverse claim, however, does not hold: given the program $P = \{p \leftarrow p.\}$, the set $calls_P^{\mathcal{R}}(\langle p \rangle) = \{p\}$ is finite for any computation rule $\mathcal{R}$ while P is clearly not terminating: $\langle p \rangle \rightsquigarrow \langle p \rangle \rightsquigarrow \ldots$

The size-change principle [22] is a recent technique originally aimed at analyzing the termination of functional programs. Intuitively speaking, it consists in tracing size changes of function arguments when going from one function call to another by means of so-called *size-change graphs*. Then, assuming that the measure of size gives rise to a well-founded order, the following principle applies [22]: *If every infinite computation would give rise to an infinitely decreasing value sequence (according to the size-change graphs), then no infinite computation is possible.*

Now, we adapt the definitions of *size-change graph* and *maximal multigraph* to logic programs; we mainly follow the definitions in [36] for rewrite systems, which in turn originates from the original definitions in [22] for first-order functional programs. In the following, a *strict order* $\succ$ is an irreflexive and transitive binary relation on terms. An order $\succ$ is *well-founded* if there are no infinite sequences of the form $t_1 \succ t_2 \succ \ldots$ An order $\succsim$ is a *quasi-order* if it is reflexive and transitive. We say that an order $\succ$ is *closed under substitutions* (or *stable*) if $s \succ t$ implies $s\sigma \succ t\sigma$ for all $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$ and every substitution σ.

Definition 4 (reduction pair [20]). *We say that $(\succsim, \succ)$ is a reduction pair on $\mathcal{T}(\Sigma, \mathcal{V})$ if $\succsim$ is a quasi-order and $\succ$ is a well-founded order on terms where both $\succsim$ and $\succ$ are closed under substitutions and compatible (i.e., $\succsim \circ \succ \subseteq \succ$ or $\succ \circ \succsim \subseteq \succ$ but $\succsim \subseteq \succ$ is not necessary).*

In this work, we consider *symbolic norms* [29] as a basis for defining appropriate reduction pairs:

Definition 5 (symbolic norm [29]). *A symbolic norm is a function $||\cdot|| : \mathcal{T}(\Sigma, \mathcal{V}) \rightarrow \mathcal{T}(I\!N \cup \{+\}, \mathcal{V})$ such that*

$$||t|| = \begin{cases} m + \sum_{i=0}^{n} k_i ||t_i|| & \textit{if } t = f(t_1, \ldots, t_n),\ n \geqslant 0 \\ t & \textit{if } t \textit{ is a variable} \end{cases}$$

where m and $k_1, \ldots, k_n$ are non-negative integer constants depending only on f/n. Note that we associate a variable over integers to each logical variable (we use the same name for both since the meaning of the variable is clear from the context).

Two popular instances of the above definition are the symbolic *term-size norm*

$$||t||_{ts} = \begin{cases} n + \sum_{i=0}^{n} ||t_i||_{ts} & \text{if } t = f(t_1, \ldots, t_n),\ n \geqslant 0 \\ t & \text{if } t \text{ is a variable} \end{cases}$$

and the symbolic *list-length norm*

$$||t||_{ll} = \begin{cases} 1 + ||Xs|| & \text{if } t = [X|Xs] \\ t & \text{if } t \text{ is a variable} \\ 0 & \text{otherwise} \end{cases}$$

For instance, we have $||f(X,Y)||_{ts} = 2 + X + Y$, $||f(a,b)||_{ts} = 2$, $||[a|Y]||_{ll} = 1 + Y$, and $||[a,X]||_{ll} = 2$. The introduction of variables in the range of the norm provides a simple mechanism to express dependencies between the sizes of terms.

In general, given a term t, we have that $||t|| = n_0 + n_1X_1 + \ldots + n_kX_k$ for non-negative integers $n_0, n_1, \ldots, n_k$ and variables $X_1, \ldots, X_k \in \mathcal{V}ar(t)$. By abuse of notation, we say that $||s|| > ||t||$ if $||s|| = n_0 + n_1X_1 + \ldots + n_kX_k$, $||t|| = m_0 + m_1X_1 + \ldots + m_jX_j$, $k \geqslant j \geqslant 0$, and $n_i > m_i$ for some $i \in \{0, \ldots, j\}$ and $n_l \geqslant m_l$ for all other $l \in (\{0, \ldots, j\} \backslash \{i\})$. The definition of $||s|| \geqslant ||t||$ is perfectly analogous.

Now, a reduction pair can easily be defined from a given symbolic norm as follows:

Definition 6 (induced orders). *The pair of orders $(\succsim, \succ)$ induced by a symbolic norm $||\cdot||$ are defined by $s \succ t \Leftrightarrow ||s|| > ||t||$ and $s \succsim t \Leftrightarrow ||s|| \geqslant ||t||$ for all terms s and t.*

The following result proves that the pair of orders induced by a symbolic norm is actually a reduction pair:

Lemma 1. *Let $||\cdot||$ be a symbolic norm on $\mathcal{T}(\Sigma, \mathcal{V})$. Then, the pair of orders $(\succsim, \succ)$ induced by $||\cdot||$ is a reduction pair.*

Proof. The fact that the pair $(\succsim, \succ)$ induced by the symbolic norm $||\cdot||$ is compatible is straightforward since $||s|| > ||t|| \Rightarrow ||s|| \geqslant ||t||$ for all terms s and t. Now we prove that $(\succsim, \succ)$ is also closed under substitutions. Consider two terms s and t such that $s \succ t$. By definition, we have $||s|| > ||t||$ and, thus, $||s|| = n_0 + n_1X_1 + \ldots + n_kX_k$, $||t|| = m_0 + m_1X_1 + \ldots + m_jX_j$, $k \geqslant j \geqslant 0$, $n_i > m_i$ for some $i \in \{0, \ldots, j\}$ and $n_l \geqslant m_l$ for all other $l \in (\{0, \ldots, j\} \backslash \{i\})$. Given a substitution σ, we have $||s\sigma|| = n_0 + n_1||X_1\sigma|| + \ldots + n_k||X_k\sigma||$ and $||t\sigma|| = m_0 + m_1||X_1\sigma|| + \ldots + m_j||X_j\sigma||$. Therefore, $||s\sigma|| > ||t\sigma||$. A similar reasoning can be applied to $\succsim$ and, hence, $(\succsim, \succ)$ is a reduction pair. □

The size-change graphs associated to a logic program, which are parametric w.r.t. a reduction pair, are defined as follows:

Definition 7 (size-change graphs). *Let P be a program and $(\succsim, \succ)$ a reduction pair. We define a size-change graph for every clause $p(s_1, \ldots, s_n) \leftarrow Q$ of P and every atom $q(t_1, \ldots, t_m)$ in Q (if any).*

The graph has n output nodes marked with $\{1_p, \ldots, n_p\}$ and m input nodes marked with $\{1_q, \ldots, m_q\}$. If $s_i \succ t_j$ holds, then we have a directed edge from output node i_p to input node j_q marked with $\succ$.[4] *Otherwise, if $s_i \succsim t_j$ holds, then we have an edge from output node i_p to input node j_q marked with $\succsim$.*

A size-change graph is thus a bipartite graph $\mathcal{G} = (V, W, E)$ where $V = \{1_p, \ldots, n_p\}$ and $W = \{1_q, \ldots, m_q\}$ are the labels of the output and input nodes, respectively, and $E \subseteq V \times W \times \{\succsim, \succ\}$ are the edges.

[4] We sometimes omit the subscripts p and q when they are clear from the context.

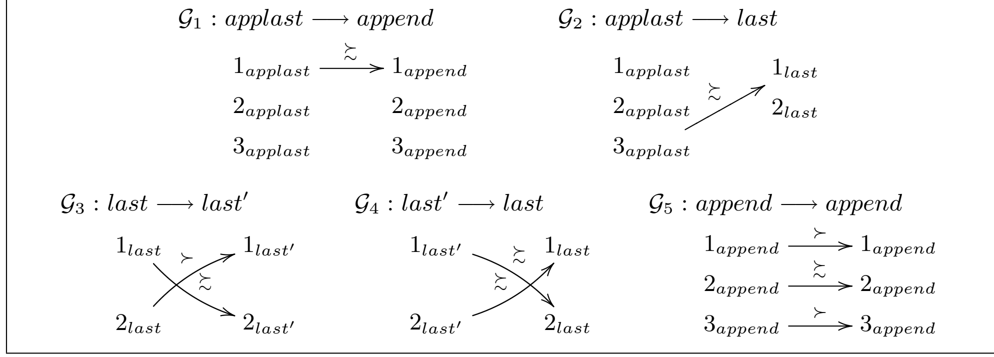

Fig. 1. Size-change graphs for `applast`

Note that our notion of size-change graph is independent of any computation rule, which makes it appropriate for analyzing strong (quasi-)termination.

Example 1. Consider the following deforestation example, which is a slight modification of the `applast` benchmark in the DPPD (Dozens of Problems of Partial Deduction [23]) library to better illustrate the notion of size-change graph:

$(c_1)\quad applast(L, X, Last) \leftarrow append(L, [X], LX),\ last(Last, LX).$

$(c_2)\quad last(X, [X]).$
$(c_3)\quad last(X, [H|T]) \leftarrow last'(T, X).$

$(c_4)\quad last'(T, X) \leftarrow last(X, T).$

$(c_5)\quad append([\,], L, L).$
$(c_6)\quad append([H|L1], L2, [H|L3]) \leftarrow append(L1, L2, L3).$

Let $(\succsim, \succ)$ be the reduction pair induced by the symbolic term-size norm $||\cdot||_{ts}$. Then, we have five size-change graphs, depicted in Fig. 1, which are associated to clauses c_1 (graphs $\mathcal{G}_1$ and $\mathcal{G}_2$), c_3 (graph $\mathcal{G}_3$), c_4 (graph $\mathcal{G}_4$) and c_6 (graph $\mathcal{G}_5$).

In order to focus on program loops, the following definition introduces the notion of *maximal multigraphs*:

Definition 8 (maximal multigraphs). *Given a logic program P, a multigraph of P is either a size-change graph of P or the concatenation (see below) of two multigraphs of P. Given two multigraphs:*

$$\mathcal{G} = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1) \quad \textit{and} \quad \mathcal{H} = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, l_r\}, E_2)$$

w.r.t. the same reduction pair $(\succsim, \succ)$, then the concatenation

$$\mathcal{G} \bullet \mathcal{H} = (\{1_p, \ldots, n_p\}, \{1_r, \ldots, l_r\}, E)$$

is also a multigraph, where E contains an edge from i_p to k_r iff E_1 contains an edge from i_p to some j_q and E_2 contains an edge from j_q to k_r. Furthermore, if some of the edges are labeled with $\succ$, then so is the edge in E; otherwise, it is labeled with $\succsim$.

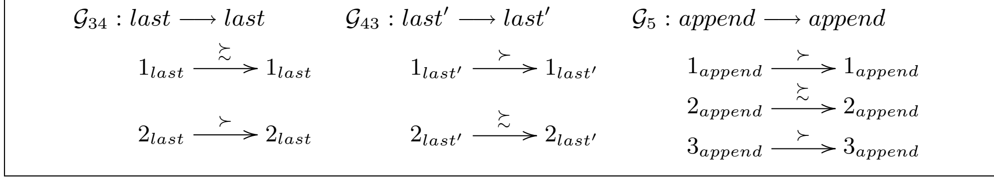

Fig. 2. Maximal multigraphs for `applast`

A multigraph $\mathcal{G}$ of P is called maximal if its input and output nodes are both labeled with $\{1_p, \ldots, n_p\}$ for some predicate p/n and if it is idempotent,[5] *i.e., $\mathcal{G} = \mathcal{G} \bullet \mathcal{G}$.*

Roughly speaking, given the set of size-change graphs of a program, we first compute its transitive closure under the concatenation operator, thus producing a finite set of multigraphs. Then, we only need to focus on the maximal multigraphs of this set because they represent the program loops.

Example 2. Given the size-change graphs of Example 1, we have only three maximal multigraphs: $\mathcal{G}_{34} = \mathcal{G}_3 \bullet \mathcal{G}_4$, $\mathcal{G}_{43} = \mathcal{G}_4 \bullet \mathcal{G}_3$, and $\mathcal{G}_5$, which are shown in Figure 2.

Following the spirit of [22], we say that a program is size-change terminating if every maximal multigraph contains at least one strictly decreasing parameter:

Definition 9 (size-change termination). *A program P is size-change terminating w.r.t. a reduction pair $(\succsim, \succ)$ on $\mathcal{T}(\Sigma, \mathcal{V})$ iff every maximal multigraph contains an edge of the form $i_p \xrightarrow{\succ} i_p$.*

Example 3. Consider the program of Example 1. This program is size-change terminating since every maximal multigraph (shown in Example 2) contains at least one edge labeled with "$\succ$".

Observe that, given a reduction pair with *decidable* orders (e.g., the reduction pair induced by the symbolic term-size norm), size-change termination is decidable as well since there exists a finite number of possible multigraphs. As illustrated in [22], there is a worst case exponential growth factor associated to the computation of multigraphs. Efficiency issues, however, are out of the scope of this paper (see, e.g., the polynomial-time approximation of [21] or the constraint-based approach of [6]).

Clearly, if a program is size-change terminating, then it is terminating w.r.t. any ground query. Size-change termination, however, does not generally imply the termination of a program w.r.t. arbitrary non-ground queries.

Example 4. Consider again the program of Example 1. Although this program is size-change terminating, infinite SLD derivations exist, e.g.:

$$last(X, Y) \leadsto_{\{Y/[W|W_S]\}} last'(W_S, X) \leadsto last(X, W_S) \leadsto_{\{W_S/[R|R_S]\}} \cdots$$

Therefore, some additional requirements are necessary to ensure termination. Let us first recall the notion of *instantiated enough* w.r.t. a symbolic norm.[6]

[5] See [7] for a generalization of this condition where idempotence is not required.

[6] A closely related notion is that of *rigidity* [3], where a term t is rigid w.r.t. a norm $||\cdot||$ if, for any substitution σ, $||t\sigma|| = ||t||$.

Definition 10 (instantiated enough [29]**).** *A term t is instantiated enough w.r.t. a symbolic norm $||\cdot||$ if $||t||$ is an integer (i.e., it does not contain variables).*

The following properties will become useful later:

Lemma 2. *Let $||\cdot||$ be a symbolic norm and t a term which is instantiated enough w.r.t. $||\cdot||$. Then, for every substitution σ, the term $t\sigma$ is also instantiated enough w.r.t. $||\cdot||$ and moreover $||t|| = ||t\sigma||$.*

Proof. Trivial by definition of symbolic norm. □

Lemma 3. *Let $||\cdot||$ be a symbolic norm and s, t be terms such that $||s|| > ||t||$ (resp. $||s|| \geqslant ||t||$). If $s\theta$ is instantiated enough w.r.t $||\cdot||$ for some substitution θ, then $t\theta$ is also instantiated enough w.r.t. $||\cdot||$ and moreover $||s\theta|| > ||t\theta||$ (resp. $||s\theta|| \geqslant ||t\theta||$).*

Proof. The fact that $||s|| > ||t||$ (resp. $||s|| \geqslant ||t||$) implies $||s\theta|| > ||t\theta||$ (resp. $||s\theta|| \geqslant ||t\theta||$) is an immediate consequence of the stability of $>$ (resp. $\geqslant$), as shown in the proof of Lemma 1. Since $s\theta$ is instantiated enough w.r.t. $||\cdot||$, i.e., $||s\theta|| = k$ for some integer k, and $||s\theta|| \geqslant ||t\theta||$, then $t\theta$ must be also instantiated enough w.r.t. $||\cdot||$. □

Now, we present a sufficient condition for termination which is based on the notion of size-change termination. Basically, we require the decreasing parameters of (potentially) looping predicates to be instantiated enough w.r.t. a given symbolic norm in the considered computations.

Theorem 1 (termination). *Let P be a size-change terminating program w.r.t. a reduction pair $(\succsim, \succ)$ induced by a symbolic norm $||\cdot||$. Let S be a set of atoms. If every maximal multigraph of P contains at least one edge $i_p \xrightarrow{\succ} i_p$ such that, for every atom $A \in S$, computation rule $\mathcal{R}$, and atom $p(t_1, \ldots, t_n) \in calls_P^{\mathcal{R}}(A)$, t_i is instantiated enough w.r.t. $||\cdot||$, then P is terminating w.r.t. S.*

Note that t_i should be instantiated enough in every possible derivation w.r.t. any computation rule. As mentioned before, this strong condition is necessary in order to have a useful result for partial evaluation. Obviously, this is an undecidable condition because the set $calls_P^{\mathcal{R}}(A)$ is generally infinite. Nevertheless, this information can be approximated by using a standard groundness or rigidness analysis. Furthermore, in the context of partial evaluation, this information will be available for free from the output of a binding-time analysis (see Sect. 5).

Observe that Theorem 1 allows us to use *local* conditions for proving termination (in the sense of [5]), i.e., rather than searching for a global measure that decreases over all of the loops in the program, termination is guaranteed if for every program loop there exists a measure which decreases as execution follows through that loop. The correctness of this approach (and thus the proof of Theorem 1) is based on Ramsey's Theorem [35].

Proof. We prove the claim by contradiction. Assume that P is not terminating. Then, by [8, Lemma 3.5],[7] we have an infinite chain in the associated calls-to relation for

[7] Although Lemma 3.5 in [8] considers LD derivations (i.e., SLD derivations with a fixed left-to-right computation rule), the proof would be perfectly analogous.

some atom $A \in S$ and computation rule $\mathcal{R}$ of the form $A = A_0 \hookrightarrow_{P,\mathcal{R}} A_1 \hookrightarrow_{P,\mathcal{R}} A_2 \hookrightarrow_{P,\mathcal{R}} \ldots$ We assume that there are no missing calls in this chain, i.e., that for all $A_j \hookrightarrow A_{j+1}$ there is no atom B with $A_j \hookrightarrow B \hookrightarrow A_{j+1}$. Informally speaking, we will prove that there is an infinite sequence of terms $s_0, s_1, \ldots$ where each s_i is an argument of the atom A_i and either $s_i \succ s_{i+1}$ or $s_i \succsim s_{i+1}$ (though there are infinitely many $\succ$), which contradicts the well-foundedness of $\succ$.

Similarly to [36], given two multigraphs $\mathcal{G}$ and $\mathcal{H}$ where $\mathcal{G}$'s input nodes have the same labels as $\mathcal{H}$'s output nodes, we let $\mathcal{G} \circ \mathcal{H}$ be the graph resulting from identifying $\mathcal{G}$'s input and $\mathcal{H}$'s output nodes. Thus $\mathcal{G} \circ \mathcal{H}$ differs from $\mathcal{G} \bullet \mathcal{H}$ in that these intermediate nodes are not dropped.

For each step $A_j \hookrightarrow_{\sigma_j} A_{j+1}$ in the calls-to relation, there is a (renamed apart) clause $H_j \leftarrow B_{j1}, \ldots, B_{jm_j} \ll P$ which is used to perform an SLD resolution step with A_j, where $\theta_j = mgu(A_j, H_j)$, $A_{j+1} = B_{jk_j}\sigma_j$, $1 \leqslant k_j \leqslant m_j$, and $\theta_j \leqslant \sigma_j$ (θ_j is more general than σ_j because some other atoms in the body of the clause could have been solved before selecting B_{jk_j}). Let us denote by $\mathcal{G}_j$ the size-change graph associated to the clause $H_j \leftarrow B_{j1}, \ldots, B_{jm_j}$ and atom B_{jk_j}.

Since P is size-change terminating, every maximal multigraph contains at least one edge $i_p \xrightarrow{\succ} i_p$ for some predicate p. Equivalently, by [36, Lemma 6], a consequence of Ramsey's Theorem, the graph $\mathcal{G}_1 \circ \mathcal{G}_2 \circ \cdots$ contains an infinite path where infinitely many edges are labeled with "$\succ$". W.l.o.g., we assume that this path already starts in $\mathcal{G}_1$ and that $A_0 = p(\ldots)$. For every j, let a_j be the output node in $\mathcal{G}_j$ which is on this path. In the following, we denote by $B|_l$ the l-th argument of atom B. Then, we have $H_j|_{a_j} \succ B_{jk_j}|_{a_{j+1}}$ for all j from an infinite set $J \subseteq \mathbb{N}$ and $H_j|_{a_j} \succsim B_{jk_j}|_{a_{j+1}}$ for $j \in \mathbb{N} \backslash J$. Now, we should prove that $H_j|_{a_j} \succ B_{jk_j}|_{a_{j+1}}$ implies $A_j|_{a_j} \succ A_{j+1}|_{a_{j+1}}$ and that $H_j|_{a_j} \succsim B_{jk_j}|_{a_{j+1}}$ implies $A_j|_{a_j} \succsim A_{j+1}|_{a_{j+1}}$.

For $j = 0$, we have $A_0 \hookrightarrow_{\sigma_0} A_1$, where $H_0 \leftarrow B_{01}, \ldots, B_{0m_0} \ll P$ is a (renamed apart) clause, $\theta_0 = mgu(A_0, H_0)$, $A_1 = B_{0k_0}\sigma_0$, $1 \leqslant k_0 \leqslant m_0$, and $\theta_0 \leqslant \sigma_0$. Assume that $H_0|_{a_0} \succ B_{0k_0}|_{a_1}$ holds (the case $H_0|_{a_0} \succsim B_{0k_0}|_{a_1}$ is perfectly analogous) in the size-change graph associated to the clause $H_0 \leftarrow B_{01}, \ldots, B_{0m_0}$ and atom B_{0k_0}. By the stability of $\succ$, we have that $H_0|_{a_0}\theta_0 \succ B_{0k_0}|_{a_1}\theta_0$ holds; also, since $A_0|_{a_0}\theta_0 = H_0|_{a_0}\theta_0$ (because $\theta_0 = mgu(A_0, H_0)$), we have that $A_0|_{a_0}\theta_0 \succ B_{0k_0}|_{a_1}\theta_0$ also holds. Again by the stability of $\succ$, we have that $A_0|_{a_0}\sigma_0 \succ B_{0k_0}|_{a_1}\sigma_0$ holds and, thus, $A_0|_{a_0}\sigma_0 \succ A_1|_{a_1}$ (since $A_1 = B_{0k_0}\sigma_0$). Now, since $A_0|_{a_0}$ is instantiated enough w.r.t. $||\cdot||$, by Lemma 2, we have that $A_0|_{a_0}\sigma_0$ is also instantiated enough and $||A_0|_{a_0}|| = ||A_0|_{a_0}\sigma_0||$. Therefore, $A_0|_{a_0} \succ A_1|_{a_1}$ also holds. Finally, by Lemma 3, since $A_0|_{a_0}$ is instantiated enough w.r.t. $||\cdot||$, so is $A_1|_{a_1}$.

By applying the same reasoning repeatedly, we have that $A_j|_{a_j} \succ A_{j+1}|_{a_{j+1}}$ holds for all $j \in J$ and $A_j|_{a_j} \succsim A_{j+1}|_{a_{j+1}}$ for all $j \in \mathbb{N} \backslash J$. This is a contradiction to the well-foundedness of "$\succ$". □

Example 5. Consider again the program of Example 1 and the maximal multigraphs of Example 2. Here, Theorem 1 guarantees the termination of SLD resolution for those computations in which the following terms are instantiated enough w.r.t. the symbolic term-size norm $||\cdot||_{ts}$:

- the second argument of every call to predicate *last*,
- the first argument of every call to predicate *last'*, and
- either the first or the third argument of every call to predicate *append*.

4 From Termination to Quasi-Termination

In this section, we turn our attention to (strong) quasi-termination. This is a weaker requirement than termination. Consider, e.g., the program $P = \{ \ p \leftarrow q. \ \ q \leftarrow p. \ \}$. Although this program is clearly not terminating: $\langle p \rangle \leadsto \langle q \rangle \leadsto \langle p \rangle \leadsto \ldots$, it is quasi-terminating since only a finite number of distinct atoms are computed.

Let us introduce the formal definition of strong quasi-termination:

Definition 11 (strong quasi-termination). *A query Q strongly quasi-terminates w.r.t. a program P if, for every computation rule $\mathcal{R}$, the set $call_P^{\mathcal{R}}(Q)$ contains finitely many different nonvariant atoms. A program P is strongly quasi-terminating w.r.t. a set of atomic queries $\mathcal{Q}$ if every $Q \in \mathcal{Q}$ is strongly quasi-terminating w.r.t. P. A program P is strongly quasi-terminating if it is strongly quasi-terminating w.r.t. B_P^E.*

Trivially, (strong) termination implies (strong) quasi-termination. Quasi-termination is relevant for tabled evaluation—see, e.g., [38], where quasi-termination w.r.t. Prolog's leftmost computation rule is considered—and for partial evaluation—see, e.g., [18], where the relation between quasi-termination and partial evaluation is first established, or [17], where the notion of *bounded variation* is introduced in order to ensure the termination of partial evaluation of first-order functional programs.

As mentioned in the introduction, analyzing the *strong* quasi-termination of programs is essential because liberal selection policies are often mandatory to achieve a good specialization during partial evaluation (see, e.g., [1, 25]).

For conciseness, in the remainder of this paper we write quasi-termination to refer to strong quasi-termination.

In order to be able to use size-change graphs to analyze quasi-termination, we need an additional requirement on the reduction pair, as the following example illustrates:

Example 6. Consider the program $P = \{ \ p(X) \leftarrow p(f(X)). \ \}$ and a reduction pair $(\succsim, \succ)$ where all terms built from $f/1$ and the same constant belong to the same equivalence class under $\succsim$, e.g., all terms in $\{a, f(a), f(f(a)), \ldots\}$ are considered "equal". Then, despite the fact that the only maximal multigraph contains an edge $1_p \xrightarrow{\succsim} 1_p$ for the argument of p, infinite non-quasi-terminating SLD derivations exist, e.g., $p(a) \leadsto p(f(a)) \leadsto p(f(f(a))) \leadsto \ldots$, where $a \succsim f(a) \succsim f(f(a)) \succsim \ldots$

In order to overcome this problem, we require the quasi-order to be *well-founded* and *finitely partitioning*. We say that a quasi-order is well-founded whenever its strict part (i.e., when $s \succsim t$ holds but $t \succsim s$ does not) is well-founded. An interesting property of well-founded quasi-orders is that in any infinite quasi-descending sequence $t_0 \succsim t_1 \succsim t_2 \succsim \ldots$, from some point on, all elements are equivalent under the equivalence relation induced by $\succsim$ [12]. The second requirement is introduced in the next definition. Here, we say that an equivalence relation that admits only finite equivalence classes is called *thin* [12].

Definition 12 (finitely partitioning quasi-order). *Let $\succsim$ be a quasi-order on $\mathcal{T}(\Sigma, \mathcal{V})$ and $\sim$ be the associated equivalence relation (i.e., $t_1 \sim t_2$ iff $t_1 \succsim t_2$ and*

$t_2 \succsim t_1$). *We say that* $\succsim$ *is finitely partitioning*[8] *iff the equivalence relation* $\sim$ *on ground terms is thin.*

Then, a quasi-order $\succsim$ is finitely partitioning if there are not infinitely many "equal" ground terms under $\succsim$. Finiteness of equivalence classes is only checked on ground terms since quasi-termination only requires that a finite number of *nonvariant* atoms are computed (which, roughly speaking, amounts to say that all variables are seen as a single fresh constant).

Observe that, if the reduction pair includes a finitely partitioning well-founded quasi-order, the situation of Example 6 is no longer possible. This condition, however, excludes the use of reduction pairs induced by some norms. For instance, the quasi-order of the reduction pair induced by the list-length norm is not finitely partitioning since we can construct an infinite number of lists with the same length. In contrast, it can easily be shown that the quasi-order of the reduction pair induced by the term-size norm is well-founded and finitely partitioning.

Now, we introduce the counterpart of size-change termination for analyzing the quasi-termination of logic programs:

Definition 13 (size-change quasi-termination). *A program P is size-change quasi-terminating w.r.t. a reduction pair $(\succsim, \succ)$ on $\mathcal{T}(\Sigma, \mathcal{V})$ iff every maximal multi-graph associated to a predicate p/n contains either*

(i) at least one edge $i_p \xrightarrow{\succ} i_p$ for some $i \in \{1, \ldots, n\}$, or

(ii) an edge $i_p \xrightarrow{R} i_p$, $R \in \{\succsim, \succ\}$, for all $i = 1, \ldots, n$, where "$\succsim$" is a well-founded finitely partitioning quasi-order.

Intuitively, a program is size-change quasi-terminating if, for every (potentially) looping predicate, at least one argument strictly decreases from one call to another, or *all* arguments (possibly non strictly) decrease from one call to another and the associated quasi-order is well-founded and finitely partitioning.

Similarly to the case of size-change termination, our characterization of size-change quasi-termination does not generally imply the quasi-termination of logic programs. Therefore, we now state a sufficient condition for quasi-termination:

Theorem 2 (quasi-termination). *Let P be a size-change quasi-terminating program w.r.t. a reduction pair $(\succsim, \succ)$ induced by a symbolic norm $||\cdot||$. Let S be a set of atoms. If every maximal multigraph of P contains either*

(i) at least one edge $i_p \xrightarrow{\succ} i_p$ for some $i \in \{1, \ldots, n\}$ such that, for every atom $A \in S$, computation rule $\mathcal{R}$ and $p(t_1, \ldots, t_n) \in calls_P^{\mathcal{R}}(A)$, t_i is instantiated enough w.r.t. $||\cdot||$, or

(ii) an edge $i_p \xrightarrow{\succsim} i_p$ for all $i = 1, \ldots, n$, such that, for every atom $A \in S$, computation rule $\mathcal{R}$ and $p(t_1, \ldots, t_n) \in calls_{\mathcal{R}}^{P}(A)$, $t_1, \ldots, t_n$ are instantiated enough w.r.t. $||\cdot||$,

then P is quasi-terminating w.r.t. S.

[8] Here, we follow the terminology of [10, 38]. In contrast to our definition, they apply this notion to *level mappings* on atoms so that a finitely partitioning level mapping does not map an infinite set of atoms to the same natural number.

Proof. Analogously to the proof of Theorem 1, we prove the claim by contradiction. First, we assume that P is not quasi-terminating. Therefore, the set $calls^P_{\mathcal{R}}(A)$ contains an infinite number of nonvariant atoms for some atom $A \in S$ and computation rule $\mathcal{R}$. Trivially, this means that there exists an infinite chain in the associated calls-to relation of the form $A = A_0 \hookrightarrow_{P,\mathcal{R}} A_1 \hookrightarrow_{P,\mathcal{R}} A_2 \hookrightarrow_{P,\mathcal{R}} \ldots$ with infinitely many nonvariant atoms. We assume that there are no missing calls in this chain, i.e., that for all $A_j \hookrightarrow A_{j+1}$ there is no atom B with $A_j \hookrightarrow B \hookrightarrow A_{j+1}$. Now, we should prove that there is an infinite sequence of terms $s_0, s_1, \ldots$ where each s_i is an argument of the atom A_i and one of the following conditions hold:

- Either $s_i \succ s_{i+1}$ or $s_i \succsim s_{i+1}$ but there are infinitely many $\succ$, which contradicts the well-foundedness of $\succ$. This case is perfectly analogous to the proof of Theorem 1.
- Either $s_i \succ s_{i+1}$ or $s_i \succsim s_{i+1}$ but there are only finitely many $\succ$ and infinitely many $\succsim$. This case is an easy extension of the proof of Theorem 1 by considering that i) since $\succsim$ is well-founded, from some point on, all elements must be equivalent under the equivalence relation $\succsim$ [12], and ii) every equivalence class contains a finite number of distinct term. By applying the same reasoning to every argument of p in A_0, we get a contradiction to the fact that the calls-to chain contains infinitely many nonvariant atoms. □

In the context of partial evaluation, however, we often deal with atoms which contain arguments that are not instantiated enough w.r.t. any norm. There are many common programs, e.g., $\{isNat(0).,\ isNat(s(X)) \leftarrow isNat(X).\ \}$, which are quasi-terminating for any query. For example, given the query $\langle isNat(X_0)\rangle$, it only has the following SLD derivation: $isNat(X_0) \leadsto_{\{X_0/succ(X_1)\}} isNat(X_1) \leadsto_{\{X_1/succ(X_2)\}} \ldots$, and thus the program is quasi-terminating since all queries are variants.

In order to cope with these situations, we now consider an alternative condition for proving quasi-termination. For this purpose, we introduce the following notion:[9]

Definition 14 (bounded norm). *We say that a symbolic norm $||\cdot||$ is bounded if the set $\{s \mid ||t|| \geqslant ||s||\}$ contains a finite number of nonvariant terms for any term t.*

Roughly, a symbolic norm is bounded if, given a term t, there exist only finitely many nonvariant terms which are lesser than or equal to t w.r.t. the symbolic norm $||\cdot||$. Clearly, if a norm is bounded then it is also finitely partitioning.

Note that this condition excludes the use of some norms. For instance, the list-length norm is not bounded (indeed, it is not finitely partitioning and, thus, it cannot be bounded). In contrast, one can easily prove that the term-size norm is bounded.

An atom is called *linear* if there are no multiple occurrences of the same variable. In the following, we consider that the heads of all program clauses are linear. This is not a restriction since every clause with a nonlinear head can be transformed into an equivalent clause with a linear head by adding some equalities to the body.

The following auxiliary lemma states a useful property for linear atoms.

[9] Although the concepts are different, the name is inspired in the notion of bounded atom [2], where an atom A is bounded w.r.t. a level mapping $|\cdot|$ if $|\cdot|$ is bounded on the set of ground instances of A.

Lemma 4. *Let $||\cdot||$ be a symbolic norm. Let $p(t_1,\ldots,t_n) \leftarrow \mathcal{Q}, q(s_1,\ldots,s_m), \mathcal{Q}'$ be a (renamed apart) clause such that $\theta = mgu(p(u_1,\ldots,u_n), p(t_1,\ldots,t_n))$ for some atom $p(u_1,\ldots,u_n)$. If $p(u_1,\ldots,u_n)$ is linear, we have that $||t_i|| \geqslant ||s_j||$ implies $||s_j\theta|| \leqslant max(||u_i||, ||s_j||)$, with $i \in \{1,\ldots,n\}$ and $j \in \{1,\ldots,m\}$.*

Proof. Since both atoms $p(t_1,\ldots,t_n)$ and $p(u_1,\ldots,u_n)$ are linear, in order to simplify the proof, we consider that they have the form $p(x_1,\ldots,x_k,t_{k+1},\ldots,t_n)$ and $p(u_1,\ldots,u_k,y_{k+1},\ldots,y_n)$, where variables $x_1,\ldots,x_k$ do not appear in $t_{k+1},\ldots,t_n$ and variables $y_{k+1},\ldots,y_n$ do not appear in $u_1,\ldots,u_k$ (besides, there are no common variables between both atoms because the clause is renamed apart). Since $||s_j|| \leqslant ||t_i||$, by the stability of $||\cdot||$, we have $||s_j\theta|| \leqslant ||t_i\theta||$. Since $u_i\theta = u_i$ for $i = 1,\ldots,k$, we have $||s_j\theta|| \leqslant ||t_i\theta|| = ||u_i\theta|| = ||u_i||$ for $i = 1,\ldots,k$. Also, since $t_i\theta = t_i$ for $i = k+1,\ldots,n$, we have $||s_j\theta|| = ||s_j||$ for $i = k+1,\ldots,n$, and the claim follows.

Note that the linearity condition cannot be dropped. Consider, for instance, the non-linear query $\langle p(A,A)\rangle$ and the following simple program: $\{\ p(f(X),Y) \leftarrow p(X,Y).\ \}$. Although we have $||f(X)||_{ts} \geqslant ||X||_{ts}$ and $||Y||_{ts} \geqslant ||Y||_{ts}$, non-quasi-terminating SLD derivations exist: $\langle p(A,A)\rangle \leadsto_{\{A/f(X),Y/f(X)\}} \langle p(X,f(X))\rangle \leadsto_{\{X/f(X'),Y'/f(f(X'))\}} \langle p(X',f(f(X')))\rangle \leadsto \ldots$

The following result extends Theorem 2 to deal with arbitrary queries.

Theorem 3 (quasi-termination). *Let P be a size-change quasi-terminating program w.r.t. a reduction pair $(\succsim,\succ)$ induced by a symbolic norm $||\cdot||$. Let S be a set of atoms. If every maximal multigraph contains either*

(i) at least one edge $i_p \stackrel{\succ}{\longrightarrow} i_p$, $i \in \{1,\ldots,n\}$, such that, for every atom $A \in S$, computation rule $\mathcal{R}$ and $p(t_1,\ldots,t_n) \in calls^{\mathcal{R}}_{P}(A)$, t_i is instantiated enough w.r.t. $||\cdot||$,

(ii) an edge $i_p \stackrel{\succsim}{\longrightarrow} i_p$ for all $i = 1,\ldots,n$, such that, for every atom $A \in S$, computation rule $\mathcal{R}$ and $p(t_1,\ldots,t_n) \in calls^{P}_{\mathcal{R}}(A)$, $t_1,\ldots,t_n$ are instantiated enough w.r.t. $||\cdot||$, or

(ii) an edge $i_p \stackrel{R}{\longrightarrow} i_p$, $R \in \{\succsim,\succ\}$, for all $i = 1,\ldots,n$, such that $||\cdot||$ is bounded and for every atom $A \in S$, computation rule $\mathcal{R}$ and $p(t_1,\ldots,t_n) \in calls^{\mathcal{R}}_{P}(A)$, $p(t_1,\ldots,t_n)$ is linear,

then P is quasi-terminating w.r.t. S.

Proof. Analogously to the proof of Theorem 1, we prove the claim by contradiction. First, we assume that P is not quasi-terminating. Therefore, the set $calls^{P}_{\mathcal{R}}(A)$ contains an infinite number of nonvariant atoms for some atom $A \in S$ and computation rule $\mathcal{R}$. Trivially, this means that there exists an infinite chain in the associated calls-to relation of the form $A = A_0 \hookrightarrow_{P,\mathcal{R}} A_1 \hookrightarrow_{P,\mathcal{R}} A_2 \hookrightarrow_{P,\mathcal{R}} \ldots$ with infinitely many nonvariant atoms. We assume that there are no missing calls in this chain, i.e., that for all $A_j \hookrightarrow A_{j+1}$ there is no atom B with $A_j \hookrightarrow B \hookrightarrow A_{j+1}$. Now, we focus on the third case, since the first two cases are identical to those of Theorem 2.

Here, in order to simplify the proof, we assume that predicate p is directly recursive. Therefore, we have an calls-to chain with infinite nonvariant atoms of the form $A_0 \hookrightarrow_{\sigma_0} A_1 \hookrightarrow_{\sigma_1} A_2 \hookrightarrow_{\sigma_2} \ldots$, where $A_i = p(t_{i1},\ldots,t_{in})$, $H_i \leftarrow B_{i1},\ldots,B_{im_i}$,

$\theta_i = mgu(A_i, H_i)$, $\theta_i \leqslant \sigma_i$, and $A_{i+1} = B_{ik_i}\sigma_i$ (θ_i is more general than σ_i because some other atoms in the body of the clause could have been solved before selecting B_{ik_i}) with $1 \leqslant k_i \leqslant m_i$.

W.l.o.g., we assume that $t_{ij} \succsim t_{i+1j}$ for all $i \geqslant 0$ and $j \in \{1, \ldots, n\}$ (this is safe since $t_{ij} \succ t_{i+1j}$ implies $t_{ij} \succsim t_{i+1j}$). Let us consider the first step in the calls-to chain $A_0 \hookrightarrow_{\sigma_0} A_1$. Recall that we denote by $B|_l$ the l-th argument of atom B. If $\sigma_0 = \theta_0$ (either because B_{0k_0} is the first selected atom or because the previously selected atoms share no variables with B_{0k_0}) then, by Lemma 4, we have $||A_1|_l|| = ||B_{0k_0}\theta_0|_l|| \leqslant max(||A_0|_l||, ||B_{0k_0}|_l||)$ for all $l = 1, \ldots, n$.

Otherwise, we consider that σ_0 is the substitution computed by SLD resolution with $\mathcal{Q}_0\theta_0$, where $\mathcal{Q}_0$ is a subset of the sequence of atoms $B_{01}, \ldots, B_{0m_0}$ excluding B_{0k_0}. Let w_0 be the maximum "size" introduced by σ_0 w.r.t. $||\cdot||$ for the variables of B_{0k_0}, i.e., we have $||A_1|_l|| = ||B_{0k_0}\sigma_0|_l|| \leqslant max(||A_0|_l||, ||B_{0k_0}|_l||) + w_0$ for all $l = 1, \ldots, n$.

Let us now consider the second step $A_1 \hookrightarrow_{\sigma_1} A_2$ in the calls-to chain. Following the same reasoning, if $\sigma_1 = \theta_1$ then, by Lemma 4, we have $||A_2|_l|| = ||B_{1k_1}\theta_1|_l|| \leqslant max(||A_1|_l||, ||B_{1k_1}|_l||)$ for all $l = 1, \ldots, n$. Moreover, we assume that the computation rule always selects the same atoms of a given clause in the same order and, thus, $B_{1k_1} = B_{0k_0}$. Hence, $||A_2|_l|| \leqslant max(||A_1|_l||, ||B_{1k_1}|_l||) \leqslant max(max(||A_0|_l||, ||B_{0k_0}|_l||) + w_0, ||B_{1k_1}|_l||) = max(||A_0|_l||, ||B_{0k_0}|_l||) + w_0$ for all $l = 1, \ldots, n$.

Otherwise, we consider that σ_1 is the substitution computed by SLD resolution with $\mathcal{Q}_1\theta_1$. Since we assume that the computation rule always selects the same atoms of a given clause in the same order, we have $\mathcal{Q}_1\theta_1 = \mathcal{Q}_0\theta_1$. Consider the set $\mathcal{X}$ of variables that are shared between $\mathcal{Q}_0$ and B_{0k_0}. Now, we have that $x\sigma_0 \leqslant x\theta_1$, $x \in \mathcal{X}$, and then $x\sigma_1 = x\theta_1$; therefore, we have $||A_2|_l|| \leqslant max(||A_0|_l||, ||B_{0k_0}|_l||) + w_0$ for all $l = 1, \ldots, n$, as in the previous case.

Finally, by applying the same reasoning repeatedly, we conclude that $||A_i|_l|| \leqslant max(||A_0|_l||, ||B_{0k_0}|_l||) + w_0$ for all $l = 1, \ldots, n$ and for all $i > 0$. Therefore, since $||\cdot||$ is bounded, we can only have a finite number of distinct nonvariant atoms whose arguments l are lesser than or equal to $max(||A_0|_l||, ||B_{0k_0}|_l||) + w_0$, which contradicts the fact that the calls-to chain contains infinitely many nonvariant atoms.

To the best of our knowledge, Theorems 2 and 3 present the first local conditions for the strong quasi-termination of a logic program.

On the other hand, if the sufficient conditions of Theorem 3 does not hold, we can still use this result to guide an annotation process, as shown in the next section.

5 Quasi-Termination and Offline Partial Evaluation

In this section, we introduce a program annotation procedure that guarantees the quasi-termination of a program and, thus, the termination of a partial evaluator.

Partial evaluators often follow a simple iterative process [15]. Essentially, in order to specialize a program P w.r.t. the set of atoms $\mathcal{S}$, we construct a finite (generally incomplete) SLD tree for each atom $S \in \mathcal{S}$ and, then, we add to $\mathcal{S}$ all unselected atoms in the leaves of these trees which are not an instance of an atom in $\mathcal{S}$; the process is then restarted with the updated set $\mathcal{S}$. This iterative process terminates when all unselected atoms in the leaves of the SLD trees are instances of atoms in $\mathcal{S}$. Then,

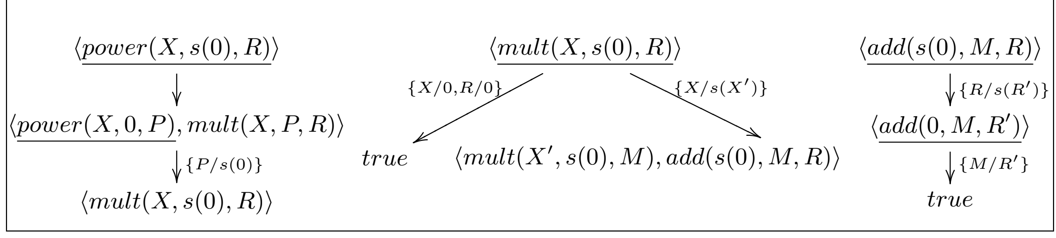

Fig. 3. SLD trees for the partial evaluation of $power(X, s(0), R)$

the specialized program is obtained by producing a so called *resultant* associated to each root-to-leaf branch of the SLD trees (resultants are usually *renamed*, but we ignore this renaming phase here since it is orthogonal to the topic of this paper). In general, the resultant of an SLD derivation $\langle S \rangle \leadsto^*_\sigma \langle S_1, \ldots, S_n \rangle$ is the clause $\sigma(S) \leftarrow S_1, \ldots, S_n$.

Example 7. Consider the following program P to compute the powers of a number:

$$
\begin{array}{ll}
power(X, 0, s(0)). & \\
power(X, s(N), R) & \leftarrow power(X, N, P),\ mult(X, P, R). \\
mult(0, Y, 0). & \\
mul(s(X), Y, R) & \leftarrow mult(X, Y, M),\ add(Y, M, R). \\
add(0, Y, Y). & \\
add(s(X), Y, s(Z)) & \leftarrow add(X, Y, Z).
\end{array}
$$

The partial evaluation of this program w.r.t. the set $\mathcal{S} = \{power(X, s(0), R)\}$ builds the SLD trees depicted in Fig. 3 (selected atoms are underlined). Therefore, the specialized program contains the following resultants:

$$
\begin{array}{ll}
power(X, s(0), R) & \leftarrow mult(X, s(0), R). \\
mult(0, s(0), 0). & \\
mul(s(X), s(0), R) & \leftarrow mult(X, s(0), M),\ add(s(0), M, R). \\
add(s(0), M, s(M)). &
\end{array}
$$

A basic question in partial evaluation is: *when* do we control the termination of the partial evaluation process? *Online* partial evaluators include some (rather expensive) termination tests (like, e.g., the *homeomorphic embedding* [24]) for controlling both the finiteness of the SLD trees—called *local* termination—and the number of atoms to be partially evaluated—called *global* termination. In contrast, *offline* partial evaluators separate the specialization process into two stages:

- First, a so called *binding-time* analysis takes a program and an approximation of the initial set of atoms to be partially evaluated, and returns an annotated program that can be used to control the specialization process (i.e., annotations are added to point out when an atom should be unfolded, when should remain unselected, when a predicate argument should be generalized, i.e., replaced by a fresh variable, etc).
- Then, a rather simple specialization stage—and considerably faster than online partial evaluators—is performed which basically follows the program annotations.

The role of the binding-time analysis is twofold. On the one hand, given an approximation of the initial set of atoms, it should compute an approximation of all possible calls within the program. On the other hand, it should also incorporate a termination analysis in order to add annotations that guarantee both the local and the global termination of the partial evaluation process. We consider that the first task follows the approach of [9], where "regular" binding-types are used. Regular binding-types improve on the classical static (totally known, i.e., ground) and dynamic (possibly unknown) binding-types by introducing regular types [16]. For instance, a binding-type of the form $list = [\,]; [dynamic|list]$ describes the set of all lists whose elements are unknown.

In the following, we use the binding-time analysis of [9] to determine whether a predicate argument is instantiated enough w.r.t. the considered symbolic norm.

5.1 Program Annotation

In some cases, either the considered program is not quasi-terminating or we are not able to prove that it is indeed quasi-terminating. In these cases, the sufficient condition of Theorem 3 can still be used to identify the parameters of the program predicates that are responsible of producing non-quasi-terminating computations.

Algorithm 4. Let P be a program and S be an atomic query. The annotation of program P for S proceeds as follows:

1. We choose a finitely partitioning well-founded quasi-order induced by a symbolic norm $||\cdot||$, e.g., the term-size norm (which is also bound).
2. We construct the maximal multigraphs of P w.r.t. the reduction pair induced by $||\cdot||$.
3. For each maximal multigraph, we check whether the sufficient condition of Theorem 3 holds (in this case, no annotation is added). Otherwise, we proceed as follows: Let $J \subseteq \{1, \ldots, n\}$ be the set of arguments of p for which there is no edge in the maximal multigraph. These arguments are annotated in every call to p in P.
4. Then, for every atom in the body of a clause of P, we annotate all occurrences of the same variable which are not yet annotatated but one (e.g., the leftmost one).

Roughly speaking, steps (3) and (4) above are used to enforce condition (iii) in Theorem 3 when conditions (i) and (ii) of the same theorem do not hold.

These annotations can be used to enure both local and global termination in offline partial evaluation. Regarding global termination, it suffices to generalize (i.e., replace by a fresh variable) every annotated argument of the unselected atoms in the leaves of SLD trees before adding them to the set of (to be) partially evaluated atoms.

As for local termination, we introduce a simple extension of SLD resolution in order to deal with annotated programs and queries.

Definition 15 (generalizing SLD resolution). *Let P be an annotated program according to Algorithm 4. Given a (possibly annotated) query $Q = \langle A_1, \ldots, A_n \rangle$, and a computation rule $\mathcal{R}$, where $\mathcal{R}(Q) = A_i$, $1 \leqslant i \leqslant n$, is the selected atom, we say that $Q \rightsquigarrow Q'$ is a generalizing SLD resolution step for Q with P and $\mathcal{R}$ if either*

– *A_i contains some annotations and, in this case, the derived query Q' has the form $\langle A_1, \ldots, A'_i, \ldots, A_n \rangle$, where A'_i is the result of replacing every annotated term in A_i by a fresh variable, or*
– *A_i does not contain annotations, and then we proceed as in standard SLD resolution.*

It can be shown, as a corollary of Theorem 3, that computations with annotated programs and generalizing SLD resolution are always quasi-terminating.

Example 8. Consider, for instance, the well-known predicate *reverse* with an accumulating parameter:[10]

$$\begin{array}{ll}
(c_1) & \mathit{reverse}(L, \mathit{RL}) \leftarrow \mathit{rev}(L, [\,], \mathit{RL}). \\
(c_2) & \mathit{rev}([\,], A, A). \\
(c_3) & \mathit{rev}([H|T], A, \mathit{RL}) \leftarrow \mathit{rev}(T, [H|A], \mathit{RL}).
\end{array}$$

This program is not quasi-terminating w.r.t. the set $S = \{\mathit{reverse}(L, \mathit{RL})\}$, as the following SLD derivation illustrates:

$$\begin{array}{lll}
\langle \mathit{reverse}(L, \mathit{RL}) \rangle & \leadsto_{\{\,\}} & \langle \mathit{rev}(L, [\,], \mathit{RL}) \rangle \\
& \leadsto_{\{L/[H|T],\ A/[\,]\}} & \langle \mathit{rev}(T, [H], \mathit{RL}) \rangle \\
& \leadsto_{\{T/[H'|T'],\ A/[H]\}} & \langle \mathit{rev}(T', [H', H], \mathit{RL}) \rangle \\
& \leadsto & \ldots
\end{array}$$

In fact, by considering the reduction pair induced by the symbolic term-size norm, we obtain the following maximal multigraph:

$$\begin{array}{llcl}
\mathcal{G}_1: & \mathit{rev} & \longrightarrow & \mathit{rev} \\
& 1_{\mathit{rev}} & \xrightarrow{\succ} & 1_{\mathit{rev}} \\
& 2_{\mathit{rev}} & & 2_{\mathit{rev}} \\
& 3_{\mathit{rev}} & \xrightarrow{\succsim} & 3_{\mathit{rev}}
\end{array}$$

Therefore, the program does not fulfill the conditions of Theorem 3 when the first argument of *rev* is not instantiated enough w.r.t. the term-size norm (as in the derivation above).

Now, by applying Algorithm 4, the second argument of the call to predicate *rev* in clause c_3 is annotated:

$$\begin{array}{ll}
(c_1) & \mathit{reverse}(L, \mathit{RL}) \leftarrow \mathit{rev}(L, [\,], \mathit{RL}). \\
(c_2) & \mathit{rev}([\,], A, A). \\
(c_3) & \mathit{rev}([H|T], A, \mathit{RL}) \leftarrow \mathit{rev}(T, \underline{\underline{[H|A]}}, \mathit{RL}).
\end{array}$$

---

[10] To be precise, the third clause should be written as follows:

$$\mathit{rev}([\,], A, B) \leftarrow A = B.$$

in order to have a linear atom in the head of the clause. However, since there are no predicate calls in the body of this clause, this change is not really relevant for the example.

Therefore, generalizing SLD resolution quasi-terminates:

$$\begin{array}{lll}
\langle reverse(L, RL)\rangle & \rightsquigarrow_{\{\,\}} & \langle rev(L, [\,], RL)\rangle \\
& \rightsquigarrow_{\{L/[H|T],\ A/[\,]\}} & \langle rev(T, \underline{\underline{[H]}}, RL)\rangle \\
& \rightsquigarrow_{generalization} & \langle rev(T, W, RL)\rangle \\
& \rightsquigarrow_{\{T/[H'|T'],\ A/W\}} & \langle rev(T', \underline{\underline{[H'|W]}}, RL)\rangle \\
& \rightsquigarrow_{generalization} & \langle rev(T', W', RL)\rangle \\
& \rightsquigarrow_{\{T'/[H''|T''],\ A/W'\}} & \langle rev(T'', [H''|W'], RL)\rangle
\end{array}$$

because the atom in the last query is a variant of the atom in the fifth query.

5.2 A Prototype Implementation

In order to test the viability of our approach, we have developed a prototype implementation of a rather simple offline partial evaluator for logic programs. The main features of this partial evaluator are as follows:

- Local termination is ensured with a simple *depth*-1 strategy, i.e., a one-step unfolding rule is considered.
- For simplicity, no BTA has been implemented. Therefore, our pre-processing stage adds program annotations according to Algorithm 4 by ignoring the first two cases of Theorem 3 (where rigidness information is necessary).
- As mentioned before, global termination is ensured by generalizing every annotated term of the unselected atoms in the leaves of SLD trees before adding them to the set of (to be) partially evaluated atoms.

The implemented system (500 lines of Prolog code), proff (for simple Prolog offline partial evaluator), can be found at

`http://www.dsic.upv.es/users/elp/german/proff/`

together with some selected examples from the DPPD (Dozens of Problems for Partial Deduction) library [23].

Our preliminary experimental results show that the approach presented so far is viable in practice, although much room for improvement still exists (which was expected given the simplicity of the considered partial evaluator).

6 Related Work

Regarding termination analysis, the closest approaches to our work are the following. First, [2] introduces the notion of strong termination by defining a sound and complete characterization (the so called recurrent programs). We extend Bezem's results by introducing a sufficient condition for strong termination. On the other hand, size-change graphs (originally introduced in [22] in the context of functional programming) are closely related to the *weighted rule graphs* of [30]. However, the weighted rule graphs are built for a specific (leftmost) computation rule and, thus, strong termination cannot be analyzed. Furthermore, the weighted rule graphs are used as an intermediate step to build the query-mapping pairs. In contrast, from the

size-change graphs, we proceed analogously to the binary unfoldings approach [8]: we compute the transitive closure of the size-change graphs in order to identify the program loops. Indeed, the binary unfoldings approach is likely the closest approach to our work. The main difference, as mentioned in the introduction, is that the binary unfoldings approach is defined for the leftmost computation rule [8] or for a local computation rule [14]. Adapting it for considering strong termination would imply redoing almost everything from scratch (and would likely produce a technique almost identical to our developments based on size-change graphs). Finally, there are some approaches based on so called *monotonicity constraints* [4], which are slightly more general than size-change graphs (see [7]).

As for quasi-termination, we find relatively few works devoted to quasi-termination analysis of logic programs. One of the first approaches is [10], where the authors introduce the notion of *quasi-acceptability*, a sufficient and necessary condition for quasi-termination. This work has been extended in several ways. On the one hand, [37] present a sufficient condition which is amenable to automation and extend a constraint-based approach for Prolog termination analysis [11] to the case of Prolog with tabling. On the other hand, [38] considers that not all predicates are tabled (in contrast to [10]) and also introduces modular termination conditions. Unfortunately, all these works consider a fixed leftmost computation rule and, thus, their results are not useful for ensuring the termination of partial evaluation, where *strong* quasi-termination is required.

Finally, regarding the use of quasi-termination analysis for ensuring the termination of offline partial evaluation, there are several related approaches. In particular, we share many similarities with [17], where a quasi-termination analysis based on size-change graphs is used to ensure the termination of an offline partial evaluation scheme for first-order functional programs. However, transferring Glenstrup and Jones' scheme to logic programming is far from trivial (and, indeed, many requirements like having instantiated enough arguments, finitely partitioning or bounded norms, etc, have no counterpart in [17]). We also share the basic scheme with [9], which combines a BTA based on regular types with a termination analysis that follows the binary unfoldings approach (which require a *strict* reduction in the size of some instantiated enough argument). Here, our scheme is more flexible since our termination analysis is valid for any computation rule and it does not require a strict reduction (i.e., a non-strict decrease in all arguments is also allowed). Furthermore, [9] only ensures the local termination of the specialization process (i.e., the finiteness of SLD trees).

To the best of our knowledge, our work presents the first strong quasi-termination analysis for logic programs which is based on local conditions (in the sense of [5]).

7 Discussion

In this paper, we have introduced new sufficient conditions for the strong termination and quasi-termination of logic programs. Our developments are based on the construction of the size-change graphs of the program, which allow us to trace size changes of predicate arguments when going from one call to another. Moreover, by computing the transitive closure of size-change graphs, we can focus on the program

loops and define local conditions for each program loop. Since our results are independent of a particular computation rule, they are particularly useful in the context of partial evaluation, where strong quasi-termination implies the termination of the specialization process. Although the class of strong quasi-terminating programs is rather general, we have also defined an annotation procedure for programs that do not fulfill our sufficient condition. In this case, a slightly extended SLD resolution can still be used to ensure the quasi-termination of computations with the annotated program. The annotation procedure can be incorporated into an offline partial evaluator in order to guarantee its termination.

As for future work, we consider several possibilities. First, we plan to enhance the implemented partial evaluator with a traditional binding-time analysis as well as with more elaborated unfolding rules. Another promising direction for future work is the definition of a mixed online/offline partial evaluation system. For instance, we could first apply the quasi-termination analysis presented in this work so that atoms with no annotated argument can be safely unfolded, while atoms with annotated arguments should be dynamically checked to avoid infinite loops.

References

1. E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Deduction of Logic Programs with Impure Predicates. In *Proc. of LOPSTR'05*, pages 115–132. Springer LNCS 3901, 2006.
2. M. Bezem. Strong Termination of Logic Programs. *Journal of Logic Programming*, 15(1&2):79–97, 1993.
3. A. Bossi, N. Cocco, and M. Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT'91*, pages 153–180. Springer LNCS 494, 1991.
4. A. Brodsky and Y. Sagiv. Inference of Monotonicity Constraints in Datalog Programs. In *Proc. of the 8th ACM Symp. on Princples of Database Systems*, pages 190–199. ACM Press, 1989.
5. M. Codish, S. Genaim, M. Bruynooghe, J. Gallagher, and W. Vanhoof. One Loop at a Time. In *Proc. of the 6th Int'l Workshop on Termination (WST 2003)*, 2003.
6. M. Codish, V. Lagoon, P. Schachte, and P.J. Stuckey. Size-Change Termination Analysis in k-Bits. In *Proc. of the 15th European Symposium on Programming (ESOP 2006)*, pages 230–245. Springer LNCS 3924, 2006.
7. M. Codish, V. Lagoon, and P. Stuckey. Testing for Termination with Monotonicity Constraints. In *Proc. of the 21st Int'l Conf. on Logic Programming (ICLP'05)*, pages 326–340. Springer LNCS 3668, 2005.
8. M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
9. S.-J. Craig, J. Gallagher, M. Leuschel, and K.S. Henriksen. Fully Automatic Binding Time Analysis for Prolog. In *Proc. of the Int'l Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pages 53–68. Springer LNCS 3573, 2005.
10. S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K.F. Sagonas. Termination Analysis for Tabled Logic Programming. In *Proc. of LOPSTR'97*, pages 111–127. Springer LNCS 1463, 1998.
11. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-Based Termination Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 21(6):1137–1195, 1999.

12. N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115, 1987.
13. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
14. M. Gabbrielli and R. Giacobazzi. Goal Independency and Call Patterns in the Analysis of Logic Programs. In *Proc. of the 1994 ACM Symposium on Applied Computing*, pages 394–399. ACM Press, 1994.
15. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of the ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 88–98. ACM, New York, 1993.
16. J.P. Gallagher and K.S. Henriksen. Abstract Domains Based on Regular Types. In *Proc. of Int'l Conf. on Logic Programming (ICLP'04)*, pages 27–42. Springer LNCS 3132, 2004.
17. A.J. Glenstrup and N.D. Jones. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM TOPLAS*, 27(6):1147–1215, 2005.
18. C.K. Holst. Finiteness Analysis. In *Proc. of Functional Programming Languages and Computer Architecture*, pages 473–495. Springer LNCS 523, 1991.
19. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
20. K. Kusakari, M. Nakamura, and Y. Toyama. Argument Filtering Transformation. In *Proc. of the 1st Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'99)*, pages 48–62. Springer LNCS 1702, 1999.
21. C.S. Lee. Finiteness Analysis in Polynomial Time. In *Proc. of the 9th Int'l Symposium on Static Analysis (SAS'02)*, pages 493–508. Springer LNCS 2477, 2002.
22. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. *SIGPLAN Notices (Proc. of POPL'01)*, 28:81–92, 2001.
23. M. Leuschel. The DPPD (Dozens of Problems for Partial Deduction) Library of Benchmarks. URL: `http://www.ecs.soton.ac.uk/~mal/systems/dppd.html`.
24. M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, pages 379–403. Springer LNCS 2566, 2002.
25. M. Leuschel and M. Bruynooghe. Logic Program Specialisation through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming*, 2(4-5):461–515, 2002.
26. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline Specialisation in Prolog using a Hand-Written Compiler Generator. *Theory and Practice of Logic Programming*, 4(1-2):139–191, 2004.
27. M. Leuschel, B. Martens, and D. De Schreye. Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.
28. M. Leuschel, B. Martens, and K.F. Sagonas. Preserving Termination of Tabled Logic Programs while Unfolding. In *Proc. of the 7th Int'l Workshop on Logic Programming Synthesis and Transformation (LOPSTR'97)*, pages 189–205. Springer LNCS 1463, 1998.
29. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In *Proc. of Int'l Conf. on Logic Programming (ICLP'97)*, pages 63–77. The MIT Press, 1997.
30. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Proving Termination for Logic Programs by the Query-Mapping Pairs Approach. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, pages 453–498. Springer LNCS 3049, 2004.

31. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.
32. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
33. D. Pedreschi, S. Ruggieri, and J.-G. Smaus. Characterisations of Termination in Logic Programming. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, pages 376–431. Springer LNCS 3049, 2004.
34. L. Plumer. *Termination Proofs for Logic Programs*. PhD thesis, Universitat Dortmund, 1990.
35. F. Ramsey. On a problem of formal logic. In *Proc. of the London Mathematical Society*, volume 30, pages 264–286, 1930.
36. R. Thiemann and J. Giesl. The Size-Change Principle and Dependency Pairs for Termination of Term Rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229–270, 2005.
37. S. Verbaeten and D. De Schreye. Termination of Simply-Moded Well-Typed Logic Programs under a Tabled Execution Mechanism. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):157–196, 2001.
38. S. Verbaeten, K. Sagonas, and D. De Schreye. Termination Proofs for Logic Programs with Tabling. *ACM Transactions on Computational Logic*, 2(1):57–92, 2001.
39. L. Vieille. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, 1989.