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

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

Abstract

One of the most important challenges in partial evaluation is the design of automatic methods for ensuring the termination of specialisation. It is well known that the termination of partial evaluation can be ensured when the considered computations are *quasi-terminating*, i.e., when only finitely many different calls occur.

In this work, we adapt the use of the so called size-change graphs to logic programming and introduce new sufficient conditions for strong (i.e., w.r.t. any computation rule) termination and quasi-termination. To the best of our knowledge, this is the first sufficient condition for the strong quasi-termination of logic programs. The class of strongly quasi-terminating logic programs, however, is too restrictive. Therefore, we also introduce an annotation procedure that combines the information from size-change graphs and the output of a traditional binding-time analysis. Annotated programs can then be used to guarantee termination of partial evaluation. We finally illustrate the usefulness of our approach by designing a simple partial evaluator in which termination is always ensured *offline* (i.e., statically).

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming; F.3.2 [*Semantics of Programming Languages*]: Partial evaluation; Program analysis

Keywords Quasi-Termination, Partial Deduction

1. Introduction

Partial evaluation [21] is a well-known technique for program specialisation. Essentially, 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 $\texttt{pgm}_\texttt{s}$ which is a specialised version of `pgm` for the static data `s` such that $\texttt{pgm}(\texttt{s},\texttt{d}) = \texttt{pgm}_\texttt{s}(\texttt{d})$ for all values of the dynamic data `d`.

A crucial issue in the design of a partial evaluator is the way in which termination of the specialisation process is ensured. Termination of partial evaluation can in principle be guaranteed when the computations performed at specialisation time only contain finitely many *nonvariant* calls,[1] i.e., when these computations are *quasi-terminating* [13]. In other words, a quasi-terminating program may diverge but every (possibly infinite) computation only contains finitely many different states. Partial evaluators usually include some kind of *memoization*, which means that, if the same function call—or a variant—has already been evaluated, then this function call is not unfolded again. Therefore, thanks to the use of memoization, quasi-termination suffices to ensure the termination of partial evaluation. This relation between quasi-termination and partial evaluation can be traced back to Holst [20], where a finiteness analysis for first-order functional languages is introduced in order to guarantee termination of partial evaluation.

Glenstrup and Jones [18] have recently introduced a binding-time analysis that ensures termination of partial evaluation in the context of functional programming. The technique is based on a quasi-termination analysis that relies on the construction of *size-change* graphs, which were originally introduced by Lee et al. [23] to analyse the termination of first-order functional programs.

In this work, we adapt and extend the approach of Glenstrup and Jones [18] to the partial evaluation of logic programs (also known as *partial deduction* [32]). In this setting, there is no need to introduce a new symbolic execution mechanism for performing partial computations, since SLD resolution—the standard operational semantics of logic programs [31]—can naturally deal with missing information represented by means of logic variables. Furthermore, 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 (and, thus, terminates less often).

Roughly speaking, given a logic program P and a query Q, a partial evaluation system should construct all—possibly incomplete—SLD derivations for Q with P such that each call in the final state of these derivations is a variant of a previously unfolded call. The finiteness of this process can be ensured either *online* or *offline*. Online partial evaluation techniques (see, e.g., [26]) use rather expensive tests—like the *homeomorphic embedding* [25]—for avoiding infinite derivations. In contrast, offline partial evaluation (e.g., [28]) proceeds in two stages: first, the program is analysed—using a so called binding-time analysis, BTA—so that an *annotated* program is returned; program annotations are then used to identify which calls can be unfolded, which arguments should be generalised (i.e., replaced by fresh variables), etc, in order to guarantee termination of the specialisation process. Then, the second stage is a simple extension of an SLD interpreter that mainly obeys the annotations.

Unfortunately, there is no fully automatic BTA for ensuring termination of offline partial evaluation in logic programming yet. Recent progress includes [11], which presents a fully automatic BTA but only provides partial termination guarantees, and [27], where

* This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02.

[1] Two calls are variants if they are equal up to variable renaming.

termination of the partial evaluation process is ensured by supervising the offline process with (more expensive) online techniques.

In order to fill this gap, we first adapt size-change graphs to logic programming [31] and present a sufficient condition for termination. Then, we consider quasi-termination of logic programs. In logic programming, quasi-termination has been studied, e.g., in [12, 29, 33, 35, 36], mainly in the context of tabled evaluation. 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 present two sufficient conditions for quasi-termination. The first condition is similar to that of Glenstrup and Jones [18] but the second one is stronger since it does not necessarily rely on the static data. Actually, our second condition could also be adapted to the setting of functional programming, since the key idea is rather general (see Section 4).

We note that, in contrast to many termination analyses for logic programs (which consider Prolog's leftmost computation rule) we need a stronger notion of termination. In particular, one should consider 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 specialisation (see, e.g., [1, 26]). Therefore, we consider *strong termination* [5]: 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*.

Fortunately, the fact that the class of strongly terminating programs is so limited is not a problem in our context. On one hand, we consider a larger class of programs, namely strongly *quasi-terminating* programs, since quasi-termination suffices to ensure termination of (either online or offline) partial evaluation. On the other hand, if a program is not strongly quasi-terminating (or we are not able to prove that it is indeed quasi-terminating), we can still use the information from the quasi-termination analysis to annotate[2] the problematic arguments of the program's atoms. In this way, we can easily design an offline partial evaluator that takes an annotated program and guarantees termination of the specialisation process automatically.

We formalise our quasi-termination analysis by means of *size-change* graphs [23], which (together with dependency pairs [4] and monotonicity constraints [7]) are currently used by an increasing number of termination analysers for different programming languages. Basically, one should first approximate the transition relation of the program by means of size-change graphs; then, the closure of this set—under an appropriate composition operator—is computed; finally, the program terminates if every idempotent graph in this closure (i.e., every potential program loop) 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 formalised for leftmost [10] or local[3] computation rules [15]. Adapting it for considering strong termination would imply redoing almost everything from scratch (and would likely produce a technique basically identical to our development based on size-change graphs).

Our main contributions can be summarised as follows:

- We adapt the notion of size-change graph to logic programs.
- We present sufficient conditions for both strong termination and quasi-termination of logic programs; to the best of our knowledge, we present the first sufficient condition for strong quasi-termination in the literature.
- We define a fully automatic program annotation procedure that allows us to ensure both the so called local and global termination of partial evaluation for arbitrary (not necessarily quasi-terminating) programs.
- A prototype implementation of an offline partial evaluator that follows the ideas presented in this paper has been undertaken. Preliminary results are encouraging and point out the usefulness and viability of our approach.

The paper is organised as follows. After introducing some preliminaries in the next section, we adapt the notion of size-change graph in Section 3, where we also introduce a sufficient condition for termination. Section 4 focuses on quasi-termination and presents two alternative characterisations. Section 5 presents a simple offline partial evaluator for logic programs in which termination is ensured by means of a program annotation procedure. Finally, Section 6 discusses some related work and Section 7 concludes.

More details and proofs of technical results can be found in [37].

2. Preliminaries

We assume some familiarity with the standard definitions and notations for logic programs [31]. Nevertheless, in order to make the paper as self-contained as possible, we present in this section the main concepts which are needed to understand our development.

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 [14] are denoted by U_P^E and B_P^E, respectively (i.e., $U_P^E = \mathcal{T}(\Sigma, \mathcal{V})$ and $B_P^E = \{p(t_1, \ldots, t_n) \mid p/n \in \Pi \wedge t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})\}$). $\mathcal{V}ar(s)$ denotes the set of variables in the syntactic object s (i.e., s can be 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 application of a substitution θ to a syntactic object s is usually denoted by juxtaposition, i.e., we write $s\theta$ rather than $\theta(s)$. 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. A substitution θ is a unifier of two syntactic objects t_1 and t_2 iff $t_1\theta = t_2\theta$; furthermore, θ is the *most general unifier* of t_1 and t_2, denoted by $mgu(t_1, t_2)$ if, for every other unifier σ of t_1 and t_2, we have that $\theta \leqslant \sigma$.

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 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,[4]

[2] We note that our annotations are different from the usual binding-time annotations in the literature. In our case, annotations are used for identifying those terms that should be generalised in order to guarantee termination.

[3] Roughly speaking, a computation rule is *local* if it always selects one of the atoms that come from the body of the last clause used in the derivation.

[4] Note that $\mathcal{R}$ may also take into account the computation history, which is common in partial evaluation. We ignore this possibility since it is not necessary in this work.

- $H \leftarrow B_1, \ldots, B_m$ 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 $\theta = \theta_n \circ \cdots \circ \theta_1$ (where $\theta = \{\}$ if $n = 0$).[5] An SLD derivation $Q \leadsto_\theta^* Q'$ is *successful* when $Q' = true$; 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 $Q' \neq true$ 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, we first adapt the notions of *size-change graph* and *size-change termination* [23] to logic programming. 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.

We first recall the notion of strong termination [5]:

DEFINITION 3.1 (strong termination). *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 queries $\mathcal{Q}$ 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, according to [5], a program P is strongly terminating when it is strongly terminating w.r.t. the set of all *ground* queries B_P. Definition 3.1 above generalises this notion to arbitrary atoms in order to be useful in the context of partial evaluation where missing information is represented by means of variables.

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

The following auxiliary definitions introduce the notion of calls and the calls-to relation (slightly extended from [10] to consider an arbitrary computation rule).

DEFINITION 3.2 (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 $calls_P^{\mathcal{R}}(Q_0)$ the set of calls in the computations of Q_0 with P and $\mathcal{R}$.*

DEFINITION 3.3 (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 emphasise 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. a set of atoms $\mathcal{A}$, then the set $calls_P^{\mathcal{R}}(A)$ is finite for every atom $A \in \mathcal{A}$ 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: $\langle p \rangle \leadsto \langle p \rangle \leadsto \ldots$

[5] We use $A\theta_1\theta_2\cdots\theta_n$ and $\theta_n \circ \cdots \circ \theta_2 \circ \theta_1(A)$ interchangeably.

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

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

Figure 1. Term-size and list-length symbolic norms

The size-change principle [23] is a recent technique originally aimed at analysing 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 [23]: *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 presentation in [34] for rewrite systems, which in turn originates from the original work of Lee et al. [23] 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.

As in [34], our size-change graphs are parameterized by a reduction pair:

DEFINITION 3.4 (reduction pair). *We say that $(\succsim, \succ)$ is a reduction pair if $\succsim$ is a quasi-order and $\succ$ is a well-founded order where both $\succsim$ and $\succ$ are closed under substitutions and compatible (i.e., $\succsim \circ \succ \subseteq \succ$ and $\succ \circ \succsim \subseteq \succ$ but $\succsim \subseteq \succ$ is not necessary).*

We consider *symbolic norms* [30] as a basis for defining appropriate reduction pairs:

DEFINITION 3.5 (symbolic norm). *A symbolic norm is a function $||\cdot|| : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{T}(\mathbb{N} \cup \{+\}, \mathcal{V})$ such that*

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

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 $||\cdot||_{ts}$ and the symbolic *list-length norm* $||\cdot||_{ll}$, which are shown in Fig. 1.[6] For instance, we have

$$||f(X,Y)||_{ts} = 2 + X + Y \qquad ||f(a,b)||_{ts} = 2$$
$$||[a|Y]||_{ll} = 1 + Y \qquad ||[a,X]||_{ll} = 2$$

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

[6] We use Prolog-like notation for lists, i.e., [ ] denotes the empty list and $[X|Xs]$ denotes a list with head X and tail Xs; furthermore, variables start with an uppercase letter.

$$||t|| = n_0 + n_1X_1 + \ldots + n_kX_k$$

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

- $||s|| = n_0 + n_1X_1 + \ldots + n_jX_j + \ldots + n_kX_k$,
- $||t|| = m_0 + m_1X_1 + \ldots + m_jX_j$, $k \geqslant j \geqslant 0$, and
- $n_i > m_i$ for some $i \in \{0, \ldots, j\}$ and $n_l \geqslant m_l$ for all other $l \in (\{0, \ldots, j\} \backslash \{i\})$.

For simplicity, we consider above that variables in the polynomials are properly ordered to facilitate the comparison. 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 3.6 (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.*

We prove in the extended version of this paper that the pair of orders induced by any symbolic norm is actually a reduction pair (see [37, Lemma 1]).

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

DEFINITION 3.7 (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 $\succ$. 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 labelled 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.

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

EXAMPLE 3.8. *Consider the following deforestation example, which is a slight modification of the* `applast` *benchmark in the DPPD (Dozens of Problems of Partial Deduction [24]) library to better illustrate the notion of size-change graph:*

(c_1) $applast(L, X, Last) \leftarrow append(L, [X], LX),$
$\qquad\qquad\qquad\qquad\qquad\quad last(Last, LX).$
(c_2) $last(X, [X]).$
(c_3) $last(X, [A, B|T]) \leftarrow last'([B|T], X).$

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

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

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. 2, 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 multigraph*:

DEFINITION 3.9 (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:*

$$\mathcal{G} = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1)$$

and

$$\mathcal{H} = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, l_r\}, E_2)$$

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 labelled with $\succ$, then so is the edge in E (thanks to the compatibility of the reduction pair); otherwise, it is labelled with $\succsim$.

A multigraph $\mathcal{G}$ of P is called maximal if its input and output nodes are both labelled with $\{1_p, \ldots, n_p\}$ for some predicate p/n and if it is idempotent,[7] 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 3.10. *Given the size-change graphs of Example 3.8, 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 Fig. 3.*

Following [23], we say that a program is *size-change terminating* if every maximal multigraph contains at least one strictly decreasing parameter:

DEFINITION 3.11 (size-change termination). *A program P is size-change terminating w.r.t. a reduction pair $(\succsim, \succ)$ iff every maximal multigraph of P contains an edge of the form $i_p \xrightarrow{\succ} i_p$.*

EXAMPLE 3.12. *Consider the program of Example 3.8. It is size-change terminating since every maximal multigraph (shown in Fig. 3) contains at least one edge labelled 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 [23], 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 [22] or the constraint-based approach of [8]).

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 (possibly non-ground) queries.

EXAMPLE 3.13. *Consider again the program of Example 3.8. Although this program is size-change terminating, infinite SLD derivations exist, e.g.,*

$$\begin{array}{lll} \langle last(X, Y)\rangle & \rightsquigarrow_{\{Y/[A,B|T]\}} & \langle last'([B|T], X)\rangle \\ & \rightsquigarrow & \langle last(X, [B|T])\rangle \\ & \rightsquigarrow_{\{T/[C|R]\}} & \ldots \end{array}$$

Therefore, some additional requirements are necessary to ensure termination. Let us first recall the notion of *instantiated enough* w.r.t. a symbolic norm.[8]

[7] A generalisation of this condition, where idempotence is not required, can be found in [9].

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

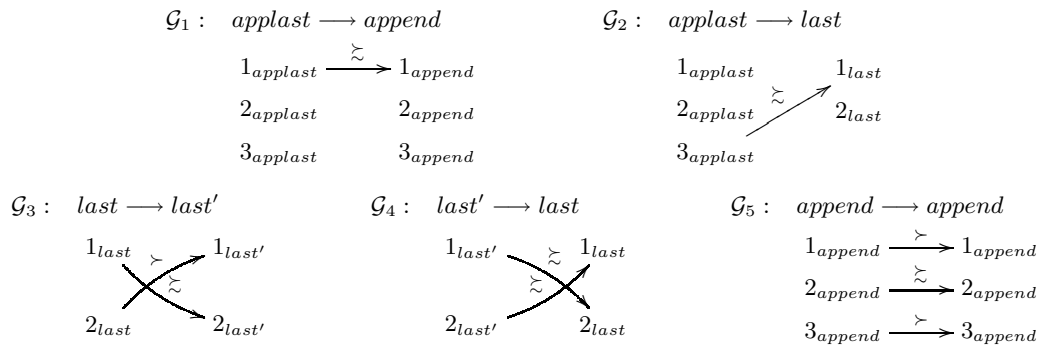

Figure 2. Size-change graphs for `applast`

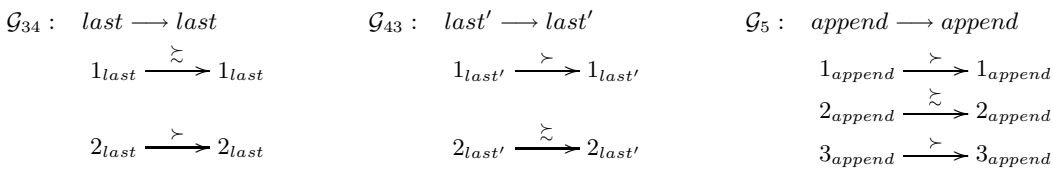

Figure 3. Maximal multigraphs for `applast`

DEFINITION 3.14 (instantiated enough [30]). *A term t is instantiated enough w.r.t. a symbolic norm $||\cdot||$ if $||t||$ is an integer (i.e., the value of the symbolic norm does not contain variables).*

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 3.15 (termination). *Let P be a size-change terminating program w.r.t. a reduction pair $(\succsim, \succ)$ induced by a symbolic norm $||\cdot||$. Let $\mathcal{A}$ 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 $A \in \mathcal{A}$, 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. $\mathcal{A}$.*

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. Unfortunately, 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 rigidity analysis. Furthermore, in the context of partial evaluation, this information will be available for free from the output of a standard binding-time analysis (see Sect. 5).

EXAMPLE 3.16. *Consider again the program of Example 3.8 and the maximal multigraphs of Fig. 3. Here, Theorem 3.15 guarantees 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: $\langle p \rangle \rightsquigarrow \langle q \rangle \rightsquigarrow \langle p \rangle \rightsquigarrow \ldots$, it is quasi-terminating since only a finite number of distinct atoms is computed.

DEFINITION 4.1 (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 nonvariant atoms. A program P is strongly quasi-terminating w.r.t. a set of 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 in logic programming for tabled evaluation—see, e.g., [36], where quasi-termination w.r.t. Prolog's leftmost computation rule is considered.

As mentioned in the introduction, analysing the *strong* quasi-termination of programs is essential for partial evaluation because liberal selection policies are often mandatory to achieve a good specialisation during partial evaluation (see, e.g., [1, 26]).

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 analyse quasi-termination, we need an additional requirement on the reduction pair, as the following example illustrates:

EXAMPLE 4.2. *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.,

$$\langle p(a)\rangle \rightsquigarrow \langle p(f(a))\rangle \rightsquigarrow \langle p(f(f(a)))\rangle \rightsquigarrow \dots$$

where $a \succsim f(a) \succsim f(f(a)) \succsim \dots$

To overcome this problem, we require the quasi-order to be *well-founded* and *finitely partitioning*. 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 $t_0 \succsim t_1 \succsim t_2 \succsim \dots$, from some point on, all elements are equivalent under the equivalence relation induced by $\succsim$ [13]. The second requirement is introduced in the next definition, where we call an equivalence relation that admits only finite equivalence classes *thin* [13].

DEFINITION 4.3 (finitely partitioning quasi-order). *Let $\succsim$ be a quasi-order 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*[9] *iff the equivalence relation $\sim$ on ground terms is thin.*

A quasi-order $\succsim$ is thus finitely partitioning if there are not infinitely many "equal" ground terms under $\succsim$. Finiteness of equivalence classes is only checked on ground terms since a program is quasi-terminating when only 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 4.2 is no longer possible. This condition, however, excludes the use of reduction pairs induced by some symbolic norms. For instance, the quasi-order of a 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 analysing the quasi-termination of logic programs:

DEFINITION 4.4 (size-change quasi-termination). *A program P is size-change quasi-terminating w.r.t. a reduction pair $(\succsim, \succ)$ iff every maximal multigraph of P associated to a predicate p/n fulfils one of the following conditions:*

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

(ii) for all $i = 1,\dots,n$, there exists an edge $i_p \xrightarrow{\succsim} i_p$, where "$\succsim$" is a finitely partitioning well-founded 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 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 characterisation 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 4.5 (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 $\mathcal{A}$ be a set of atoms. If every maximal multigraph of P associated to a predicate p/n fulfils one of the following conditions:*

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

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

then P is quasi-terminating w.r.t. $\mathcal{A}$.

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. In fact, there are many common programs, e.g.,

$$isNat(0).$$
$$isNat(s(X)) \leftarrow isNat(X).$$

which are quasi-terminating for any query. For example, the query $\langle isNat(X_0)\rangle$ only has the following SLD derivation:

$$\langle isNat(X_0)\rangle \rightsquigarrow_{\{X_0/s(X_1)\}} \langle isNat(X_1)\rangle \rightsquigarrow_{\{X_1/s(X_2)\}} \dots$$

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:

DEFINITION 4.6 (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 speaking, a symbolic norm is bounded if, for every term t, there exist only finitely many nonvariant terms which are lesser than or equal to t w.r.t. the symbolic norm $||\cdot||$. Trivially, 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. The following result extends Theorem 4.5:

THEOREM 4.7 (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 $\mathcal{A}$ be a set of atoms. If every maximal multigraph of P associated to a predicate p/n fulfils one of the following conditions:*

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

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

(iii) for all $i = 1,\dots,n$, there exists an edge $i_p \xrightarrow{R} i_p$, with $R \in \{\succsim, \succ\}$, such that $||\cdot||$ is bounded and, for every atom $A \in \mathcal{A}$, computation rule $\mathcal{R}$ and $p(t_1,\dots,t_n) \in calls_P^{\mathcal{R}}(A)$, we have that $p(t_1,\dots,t_n)$ is linear,

then P is quasi-terminating w.r.t. $\mathcal{A}$.

To the best of our knowledge, Theorems 4.5 and 4.7 present the first sufficient conditions for the strong quasi-termination of logic programs.

Note that the linearity condition in case (iii) cannot be dropped from Theorem 4.7. Consider, for instance, the nonlinear query $\langle p(A,A)\rangle$ and the following simple program:

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

[9] We follow the terminology of [12, 36]. 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 number.

Although we have $||f(X)||_{ts} \geqslant ||X||_{ts}$ and $||Y||_{ts} \geqslant ||Y||_{ts}$, non-quasi-terminating SLD derivations exist, e.g.,

$$\begin{array}{lll} \langle p(A,A)\rangle & \leadsto_{\{A/f(X),Y/f(X)\}} & \langle p(X,f(X))\rangle \\ & \leadsto_{\{X/f(X'),Y'/f(f(X'))\}} & \langle p(X',f(f(X')))\rangle \\ & \leadsto & \ldots \end{array}$$

Intuitively speaking, the problem comes from the propagation of bindings between predicate arguments due to unification (see [37, Lemma 4] for more details).

It is worthwhile to note that a similar result could also be applied to (first-order) functional programs. Consider, for instance, the following simple program:

$$\begin{array}{lcl} length([\,]) & \rightarrow & 0 \\ length(x:xs) & \rightarrow & s(length(xs)) \end{array}$$

This program does not fulfil the quasi-termination conditions of [18] if the argument of $length$ is dynamic. However, only finitely many nonvariant calls to function $length$ can be obtained by symbolically evaluating a function call of the form $length(x)$. Therefore, it could be proved quasi-terminating with a functional version of Theorem 4.7. A preliminary result in this direction (although for functional *logic* programs [19] and narrowing-driven partial evaluation [2]) can be found in [3].

If the sufficient condition of Theorem 4.7 does not hold, we can still use this result to ensure the termination of partial evaluation by means of program annotations.

5. Quasi-Termination and Offline Partial Evaluation

In this section, we introduce a program annotation procedure that can be used to guarante quasi-termination of a program when annotated terms are generalised away if they appear in a computation. Thus, it forms an appropriate basis for ensuring termination of an offline partial evaluator automatically.

Partial evaluators often follow an iterative process [16]. Essentially, in order to specialise a program P w.r.t. a set of atoms $\mathcal{S}$,

1. we construct a finite (generally incomplete) SLD tree for each atom $S \in \mathcal{S}$ and
2. 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 the unselected atoms in the leaves of the SLD trees are all either variants or instances of atoms in $\mathcal{S}$. Then, the specialised 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 thus the clause $\sigma(S) \leftarrow S_1,\ldots,S_n$.

EXAMPLE 5.1. *Consider the following program P to compute the powers of a number:*

$power(X,0,s(0)).$
$power(X,s(N),R) \leftarrow power(X,N,P),\ mult(X,P,R).$

$mult(0,Y,0).$
$mul(s(X),Y,R) \leftarrow mult(X,Y,M),\ add(Y,M,R).$

$add(0,Y,Y).$
$add(s(X),Y,s(Z)) \leftarrow add(X,Y,Z).$

A partial evaluation of this program w.r.t. $\{power(X,s(0),R)\}$ that builds the SLD trees depicted in Fig. 4 (selected atoms are underlined) produces a specialised program which contains the following resultants:

$power(X,s(0),R) \leftarrow mult(X,s(0),R).$
$mult(0,s(0),0).$
$mul(s(X),s(0),R) \leftarrow mult(X,s(0),M),\ add(s(0),M,R).$
$add(s(0),M,s(M)).$

A basic question in partial evaluation is: *when* do we control termination of the partial evaluation process? *Online* partial evaluators include some (rather expensive) termination tests (like, e.g., the *homeomorphic embedding* [25]) 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 specialisation 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 specialisation process, e.g., annotations are added to identify when an atom should be unfolded, when it should remain unselected, when a predicate argument should be generalised (i.e., replaced by a fresh variable), etc.
- Then, a rather simple (and considerably faster than in the online case) specialisation stage 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. A recent approach to the first task [11] considers *regular binding-types*. Regular binding-types improve on the classical static (totally known, i.e., ground) and dynamic (possibly unknown) binding-types by introducing regular types [17]. 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 assume that the information from a binding-time analysis like that of [11] is available to determine whether a predicate argument is instantiated enough w.r.t. the considered symbolic norm.

5.1 Program Annotation

Sometimes, either the considered program is not quasi-terminating or we are not able to prove that it is indeed quasi-terminating (according to Theorem 4.7). In these cases, the sufficient condition of Theorem 4.7 can still be used to annotate the parameters of the program predicates that are responsible of producing non-quasi-terminating computations.

ALGORITHM 5.2. *Given a logic program P, the annotation procedure 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 4.7 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.*

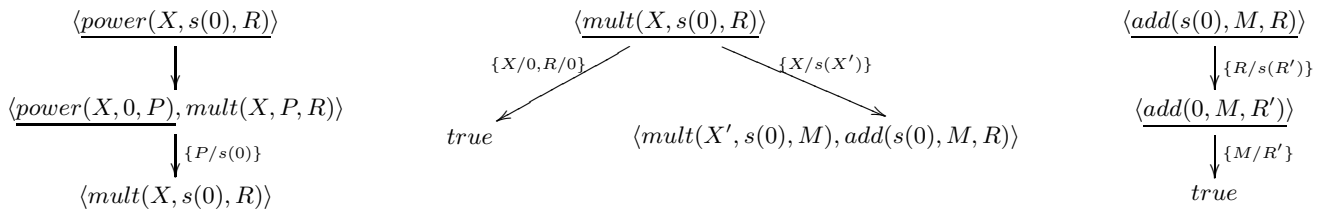

Figure 4. SLD trees for the partial evaluation of $power(X, s(0), R)$

4. Finally, for every atom in the body of a clause of P, we annotate all occurrences of the same dynamic variable which are not yet annotated but one (e.g., the leftmost one).

Roughly speaking, steps (3) and (4) above are used to enforce condition (iii) in Theorem 4.7 when conditions (i) and (ii) of the same theorem do not hold.

We introduce a simple extension of SLD resolution in order to illustrate the usefulness of annotations:

DEFINITION 5.3 (generalizing SLD resolution). *Let P be an annotated program according to Algorithm 5.2. 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 generalising 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 easily be shown, as a corollary of Theorem 4.7, that the computations of an (arbitrary) annotated program obtained by applying Algorithm 5.2 using generalising SLD resolution are always quasi-terminating.

EXAMPLE 5.4. *Consider, for instance, the well-known predicate $reverse$ with an accumulating parameter:*

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

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

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

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

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

Therefore, the program does not fulfil the conditions of Theorem 4.7 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 5.2, the second argument of the call to predicate rev in clause c_3 is annotated:

$$(c_3) \quad rev([H|T], A, RL) \leftarrow rev(T, \underline{\underline{[H|A]}}, RL).$$

Therefore, generalising SLD resolution quasi-terminates:

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

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

Program annotations can be used in different ways in order to ensure both the local and the global termination of a partial evaluator. In the next section, we describe one of such possibilities.

5.2 A Prototype Implementation

We have developed a prototype implementation of an offline partial evaluator for logic programs in order to test the usefulness and viability of our approach. The main features of this partial evaluator are as follows:

- Local termination: given a query, our unfolding strategy selects the leftmost linear atom that, after generalising away annotated arguments, is not a variant of some previously selected atom in the SLD derivation.[10] Annotations are removed before unfolding the selected atom. Note that this implies that generalising SLD is not really used during unfolding (actually, it is only a device to show the relevance of program annotations for ensuring quasi-termination).
- Global termination: once the unfolding strategy returns a set of queries (i.e., the leaves of the SLD tree for a given atom), we collect all atoms in these queries, generalise away annotated arguments, and add the resulting atoms—if they are not variants of the already partially evaluated atoms—to the set of (to be) partially evaluated atoms.
- For simplicity, no BTA has been implemented yet. Therefore, our pre-processing stage adds program annotations according to Algorithm 5.2 by ignoring the first two cases of Theorem 4.7 (where rigidity information is necessary). In other words, the information on which arguments are static and which are dynamic is not taken into account by our partial evaluator (i.e., all arguments are treated as if they were dynamic).

[10] Because of the nonvariant checks, our partial evaluator is not purely offline. Nevertheless, we could also consider a simple one-step unfolding strategy so that the resulting partial evaluator would be purely offline.

Table 1. Benchmark results

Benchmark	run time original	run time specialised	speedup
applast	640 ms.	390 ms.	1.64
depth	280 ms.	120 ms.	2.33
incList	280 ms.	50 ms.	5.60
match	850 ms.	560 ms.	1.52
matchapp	240 ms.	230 ms.	1.04
power	290 ms.	300 ms.	0.97
regexp.r1	130 ms.	90 ms.	1.44
rev_acc_type	510 ms.	490 ms.	1.04
AVERAGE			1.95

The implemented system (around 2000 lines of Prolog code), proff (for Prolog offline partial evaluator), can be found at

`http://www.dsic.upv.es/~gvidal/german/proff/`

together with some selected benchmarks from the DPPD (Dozens of Problems for Partial Deduction) library [24].

Our preliminary experimental results are encouraging. Indeed, despite the simplicity of the implemented partial evaluator (and the fact that the static/dynamic distinction is not taken into account), we measured a speedup factor of 1.95 (i.e., specialised programs run on average 1.95 faster than original ones) on a selection of benchmarks. The results of these experiments are summarised in Table 1, where run times are in milliseconds (running on a Pentium D 2.8GHz, 2GB of RAM, with Fedora Core 5 Linux), and are the average of ten executions with a sufficiently large input query. Times for size-change analysis, program annotation and partial evaluation are not shown because they are always around 10 ms or less. More details can be found in the URL above.

6. Related Work

Regarding termination analysis, the closest approaches to our work are the following. First, Bezem [5] introduced the notion of strong termination by defining a sound and complete characterisation (the so called recurrent programs). We extend Bezem's results by introducing a *sufficient* condition for strong termination (actually, size-change termination, as defined in Definition 3.11 is a sufficient condition for Bezem's strong termination which only considers ground atoms). On the other hand, size-change graphs (originally introduced in [23] 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 analysed. Furthermore, the weighted rule graphs are used as an intermediate step to build the so called query-mapping pairs. In contrast, from the size-change graphs, we proceed analogously to the binary unfoldings approach [10]: 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 [10] or for a local computation rule [15]. 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).

As for quasi-termination, we find relatively few works devoted to quasi-termination analysis of logic programs. One of the first approaches is [12], where the authors introduce the notion of *quasi-acceptability*, a sufficient and necessary condition for quasi-termination. This work has been extended in [36]. Another related approach is [29], where the authors analyse the effects of an unfolding-based program transformation on the termination behaviour of tabled programs. However, these works consider a fixed leftmost computation rule and, thus, their results are not as useful as ours for ensuring termination of partial evaluation, where *strong* quasi-termination is often required.

Finally, regarding the use of quasi-termination analysis for ensuring termination of offline partial evaluation, there are several related approaches. In particular, we share many similarities with [18], where a quasi-termination analysis based on size-change graphs is used to ensure the termination of an offline partial evaluator 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 [18]). Moreover, Theorem 4.7 covers some cases that are not covered by [18] (e.g., when all arguments are dynamic, as illustrated in Section 4). Indeed, this result could also be adapted to functional programs. A preliminary result in this direction (for functional *logic* programs [19] and narrowing-driven partial evaluation [2]) can be found in [3].

We also share the basic scheme with [11], 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 (cf. Theorem 4.7). Furthermore, [11] only ensures the *local* termination of the specialisation 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 and illustrates its usefulness for ensuring termination of partial evaluation.

7. Discussion

In this paper, we have introduced novel 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. By computing the transitive closure of size-change graphs, we can focus only on the (potential) program loops. 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 termination of the specialisation process. Furthermore, since the class of strongly quasi-terminating programs is somehow limited, we have also defined an annotation procedure for programs that do not fulfil our sufficient condition. Then, annotated programs can be used to ensure both local and global termination of partial evaluation.

As for future work, we consider several possibilities. First, we plan to enhance the implemented partial evaluator with more elaborated unfolding rules and a powerful BTA. Also, in our current scheme, the output of a BTA is required for analysing termination. Sometimes, however, the output of the termination analysis can be useful for the BTA. Therefore, the integration of our termination analysis into a BTA seems also interesting. Another 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. In this way, the overload introduced by online partial evaluators would be minimised.

Acknowledgements

We gratefully acknowledge the anonymous referees for many useful comments and suggestions.

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] E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.

[3] G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In *Proc. of the 16th Int'l Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 55–61. Università Ca' Foscari di Venezia, 2006. Extended version to appear in Springer LNCS.

[4] T. Arts and J. Giesl. Termination of Term Rewriting Using Dependency Pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[5] M. Bezem. Strong Termination of Logic Programs. *Journal of Logic Programming*, 15(1&2):79–97, 1993.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.

[11] 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.

[12] 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.

[13] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115, 1987.

[14] 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.

[15] 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.

[16] 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.

[17] J.P. Gallagher and D.A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In *Proc. of 11th Int'l Conf. on Logic Programming (ICLP'94)*, pages 599–613. MIT Press, 1994.

[18] A.J. Glenstrup and N.D. Jones. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM TOPLAS*, 27(6):1147–1215, 2005.

[19] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[20] C.K. Holst. Finiteness Analysis. In *Proc. of Functional Programming Languages and Computer Architecture*, pages 473–495. Springer LNCS 523, 1991.

[21] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[22] 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.

[23] 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.

[24] M. Leuschel. The DPPD (Dozens of Problems for Partial Deduction) Library of Benchmarks. Available at URL: `http://www.ecs.soton.ac.uk/~mal/systems/dppd.html`.

[25] 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.

[26] 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.

[27] M. Leuschel, S.-J. Craig, and D. Elphick. Supervising Offline Partial Evaluation of Logic Programs using Online Techniques. In *Proc. of the Int'l Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*. Springer LNCS, 2006. To appear.

[28] 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.

[29] 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.

[30] 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.

[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] L. Plumer. *Termination Proofs for Logic Programs*. PhD thesis, Universitat Dortmund, 1990.

[34] 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.

[35] 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.

[36] 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.

[37] G. Vidal. Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. Technical report, DSIC, Technical University of Valencia, 2006. Available from URL: `http://www.dsic.upv.es/users/elp/german/papers.html`.