Fast and Accurate Strong Termination Analysis with an Application to Partial Evaluation[⋆]

Michael Leuschel[1], Salvador Tamarit[2], and Germán Vidal[2]

[1] Institut für Informatik, Universität Düsseldorf, D-40225, Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de
[2] DSIC, Technical University of Valencia, E-46022, Valencia, Spain
{stamarit,gvidal}@dsic.upv.es

Abstract. A logic program strongly terminates if it terminates for any selection rule. Clearly, considering a particular selection rule—like Prolog's leftmost selection rule—allows one to prove more goals terminating. In contrast, a strong termination analysis gives valuable information for those applications in which the selection rule cannot be fixed in advance (e.g., partial evaluation, dynamic selection rules, parallel execution). In this paper, we introduce a fast and accurate size-change analysis that can be used to infer conditions for both strong termination and strong quasi-termination of logic programs. We also provide several ways to increase the accuracy of the analysis without sacrificing scalability. In the experimental evaluation, we show that the new algorithm is up to three orders of magnitude faster than the previous implementation, meaning that we can efficiently deal with programs exceeding 25,000 lines of Prolog.

1 Introduction

Analysing the termination of logic programs is a challenging problem that has attracted a lot of interest (see, e.g., [5, 7, 23, 29] and references therein). However, *strong* termination analysis (i.e., termination for any selection rule) has received little attention, a notable exception being the work by Bezem [2], who introduced the notion of strong termination by defining a sound and complete characterisation (the so called recurrent programs). Also, we can find a well established line of research on termination of logic programs with *dynamic* selection rules (e.g., [25, 4, 24, 27, 26]). In these works, however, there are a number of assumptions, like the use of *local* selection rules (a slight extension of the left-to-right selection rule), input-consuming derivations (i.e., derivations where input arguments are not instantiated by SLD resolution steps [3]), etc., which are not useful in our context.

In this work, we consider strong (quasi-)termination[3] so that our results can be applied to any application domain where the selection rule is not known in

[⋆] This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant GVPRE/2008/001, and by the *UPV* (Programs PAID-05-08 and PAID-06-08).

[3] A computation quasi-terminates if it reaches finitely many different states. This is an essential property in many contexts since it allows one to construct a finite representation of the search space, thus allowing for finite analysis and transformation.

advance or should be dynamically defined, e.g., partial evaluation, resolution with dynamic selection rules, parallel execution, etc.

Consider, for instance, the case of partial evaluation [14], a well-known technique for program specialisation. Within the so-called *offline* approach to partial evaluation, there is a first stage called *binding-time analysis* (BTA) that should analyse the termination of the program and also propagate known data following the program's control flow. In this context, one of the main limitation of previous approaches to the offline partial evaluation of logic programs like, e.g., [6], is that the associated BTA is usually rather expensive and does not scale up well to medium-sized programs. Intuitively speaking, this is mainly due to the fact that the termination analysis and the algorithm for propagating known information are interleaved, so that every time a call is annotated as "not unfoldable", the termination analysis has to be re-executed to take into account that some bindings will not be propagated anymore.

In recent work [17, 30], we have shown that this drawback can be overcome by using instead a strong termination analysis based on the size-change principle [15, 28]. In this case, both tasks—termination analysis and propagation of known information—are kept independent, so that the termination analysis is done once and for all before the propagation phase, resulting in major efficiency improvements over the previous approach of [6].

The new BTA scheme of [17], however, still had some shortcomings concerning both efficiency and accuracy. In particular, the size-change analysis involves computing the transitive closure of the so-called *size-change graphs* of the program. This is often an expensive process with a worst case exponential growth factor [15].

In order to overcome this drawback, in this work we introduce an efficient algorithm for the size-change analysis based on the insight that many size change graphs are irrelevant for inferring strong termination and quasi-termination conditions. In particular, we introduce an ordering for size-change graphs, so that only the *weakest* graphs need to be kept without compromising correctness nor accuracy.

Then, we consider the application of the new analysis to the particular domain of offline partial evaluation (cf. Sect. 4) and empirically evaluate the new algorithm. In summary, the empirical results demonstrate the usefulness and scalability of our proposals in practice, meaning that we can efficiently deal with realistic interpreters and systems exceeding 25,000 lines of Prolog.

Finally, in Sect. 5 we develop a further improvement of our new algorithm in the context of partial evaluation. Indeed, the fact that the size-change analysis considers *strong* termination may involve a significant loss of accuracy. For instance, given the clauses

$$\begin{array}{l} p(X) \leftarrow q(X,Y), p(Y). \\ q(s(X),X). \end{array}$$

the size-change analysis infers no relation between the sizes of $p(X)$ and $p(Y)$ in the first clause (while, in contrast, one can easily determine that the argument of

p decreases from one call to the next one by assuming Prolog's leftmost selection rule). Clearly, this makes the size change analysis independent of the selection rule and, particularly, of whether $q(X,Y)$ is unfolded before selecting $p(Y)$ or not. However, in many cases, some partial knowledge is available (e.g., one can safely assume that all *facts* can be unfolded no matter the available information) and could be used to improve the accuracy of the analysis. For this purpose, we develop an extension of the size-change analysis that allows us to propagate some size information from left to right.

2 Fundamentals of Size-Change Analysis

The size-change principle [15] was originally aimed at proving the termination of functional programs. This analysis was adapted to the logic programming setting in [30], where both termination and quasi-termination were analysed. The main difference w.r.t. previous termination analyses for logic programs is that [30] considers *strong* termination, i.e., termination for all computation rules. As mentioned in the introduction, this makes the output of the analysis less accurate but allows the definition of much faster analyses that can be successfully applied in a number of application domains (e.g., for defining a scalable binding-time analysis; see [17] for more details).

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

Size-change analysis is based on constructing graphs that represent the decrease of the arguments of a predicate from one call to another. For this purpose, some ordering on terms is required. Analogously to [28], in [30] reduction pairs $(\succsim, \succ)$ consisting of a quasi-order and a compatible well-founded order (i.e., $\succsim \circ \succ \subseteq \succ$ and $\succ \circ \succsim \subseteq \succ$), both closed under substitutions, were used. The orders $(\succsim, \succ)$ are *induced* from so called *norms*. Here, we only consider the well-known *term-size* norm $||\cdot||_{ts}$ [9] which counts the number of (non-constant) function symbols. The associated induced orders $(\succsim_{ts}, \succ_{ts})$ are defined as follows: $t_1 \succ_{ts} t_2$ (resp. $t_1 \succsim_{ts} t_2$) if $||t_1\sigma||_{ts} > ||t_2\sigma||_{ts}$ (resp. $||t_1\sigma||_{ts} \geqslant ||t_2\sigma||_{ts}$) for all substitutions σ that make $t_1\sigma$ and $t_2\sigma$ ground. For instance, we have $f(s(X),Y) \succ_{ts} f(X,a)$ since $||f(s(X),Y)\sigma||_{ts} > ||f(X,a)\sigma||_{ts}$ for all σ that makes X and Y ground.

We produce a *size-change graph* $\mathcal{G}$ for every pair (H, B_i) of every clause $H \leftarrow B_1, \ldots, B_n$ of the program. Formally,

Definition 1 (size-change graph). *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.

Example 1. Consider the following program *MLIST*:

(c_1) $\quad mlist(L, I, [\,]) \leftarrow empty(L).$
(c_2) $\quad mlist(L, I, LI) \leftarrow nonempty(L), hd(L, X), tl(L, R), ml(X, R, I, LI).$

(c_3) $\quad ml(X, R, I, [XI|RI]) \leftarrow mult(X, I, XI),\ mlist(R, I, RI).$

(c_4) $\quad mult(0, Y, 0).$ $\quad (c_5)$ $\quad mult(s(X), Y, Z) \leftarrow mult(X, Y, Z1),\ add(Z1, Y, Z).$
(c_6) $\quad add(X, 0, X).$ $\quad (c_7)$ $\quad add(X, s(Y), s(Z)) \leftarrow add(X, Y, Z).$

(c_8) $\quad hd([X|\_], X).$ $\quad (c_9)$ $\quad empty([\,]).$

(c_{10}) $\quad tl([\_|R], R).$ $\quad (c_{11})$ $\quad nonempty([\_|\_]).$

which is used to multiply all the elements of a list by a given number. The program is somewhat contrived in order to better illustrate our technique.

Here, the size-change graphs associated to, e.g., clause c_3 are as follows:[4]

$$\begin{tikzcd}
1_{ml} \arrow[r, "\succsim_{ts}"] & 1_{mult} & & 1_{ml} & 1_{mlist} \\
2_{ml} & 2_{mult} & & 2_{ml} \arrow[ur, "\succsim_{ts}"] & 2_{mlist} \\
3_{ml} \arrow[ur, "\succsim_{ts}"] & 3_{mult} & & 3_{ml} \arrow[ur, "\succsim_{ts}"] & 3_{mlist} \\
4_{ml} \arrow[ur, "\succ_{ts}"] & & & 4_{ml} \arrow[ur, "\succ_{ts}"] &
\end{tikzcd}$$

using a reduction pair $(\succsim_{ts}, \succ_{ts})$ induced from the term-size norm.

In order to identify the program *loops*, we should compute roughly a transitive closure of the size-change graphs by composing them in all possible ways.

Definition 2 (graph concatenation, idempotent multigraph). *A multigraph of P is inductively defined to be either a size-change graph of P or the concatenation (see below) of two multigraphs of P. Given two multigraphs:*

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

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. If some of the edges are labelled with $\succ$, then so is the edge in E; otherwise, it is labelled with $\succsim$.

We say that a multigraph $\mathcal{G}$ of P is idempotent when $\mathcal{G} = \mathcal{G} \bullet \mathcal{G}$. Intuitively speaking, an idempotent multigraph represents a chain of multigraphs.

[4] In general, we denote with p/n a predicate symbol of arity n. However, in the examples, we simply write p for predicate p/n when no confusion can arise.

Example 2. For the program *MLIST* of Example 1, we have the following four idempotent multigraphs:

$$\begin{tikzcd}[row sep=tiny]
1_{mlist} & 1_{mlist} \\
2_{mlist} \arrow[r, "\succsim_{ts}"] & 2_{mlist} \\
3_{mlist} \arrow[r, "\succ_{ts}"] & 3_{mlist}
\end{tikzcd}
\qquad
\begin{tikzcd}[row sep=tiny]
1_{ml} & 1_{ml} \\
2_{ml} & 2_{ml} \\
3_{ml} \arrow[r, "\succsim_{ts}"] & 3_{ml} \\
4_{ml} \arrow[r, "\succ_{ts}"] & 4_{ml}
\end{tikzcd}
\qquad
\begin{tikzcd}[row sep=tiny]
1_{mult} \arrow[r, "\succ_{ts}"] & 1_{mult} \\
2_{mult} \arrow[r, "\succsim_{ts}"] & 2_{mult} \\
3_{mult} & 3_{mult}
\end{tikzcd}
\qquad
\begin{tikzcd}[row sep=tiny]
1_{add} \arrow[r, "\succsim_{ts}"] & 1_{add} \\
2_{add} \arrow[r, "\succ_{ts}"] & 2_{add} \\
3_{add} \arrow[r, "\succ_{ts}"] & 3_{add}
\end{tikzcd}$$

that represent how the size of the arguments of the four potentially looping predicates changes from one call to another.

The main termination results from [17, 30] can be summarised as follows:

– A predicate p/n terminates if every idempotent multigraph for p/n contains at least one edge $i_p \xrightarrow{\succ} i_p$, $1 \leq i \leq n$, such that the i-th argument of every call to this predicate is ground.[5]
– A predicate p/n quasi-terminates if every idempotent multigraph for p/n contains edges $j_p^1 \xrightarrow{R_1} 1_p, \ldots, j_p^n \xrightarrow{R_n} n_p$, $R_i \in \{\succ, \succsim\}$, and the arguments $j^1, \ldots, j^n$ are ground in every call to p/n. Additionally, the considered quasi-order $\succsim$ should be well-founded and *finitely partitioning* [8, 29], i.e., there should not be infinitely many "equal" ground terms under $\succsim$.

These conditions, though in principle undecidable, can be approximated in a number of ways. For instance, in the context of partial evaluation, the computed *binding-times*—static for definitely known arguments and dynamic for possibly unknown arguments—can easily be used for this purpose (cf. Sect. 4.1).

3 A Procedure for Size-Change Analysis

In this section, we introduce a fast and accurate procedure for the size-change analysis of logic programs. In principle, a naive procedure for computing the set of idempotent multigraphs of a program may proceed as follows:

1. First, the size-change graphs of the program are built according to Def. 1.
2. Then, after initialising a set $\mathcal{M}$ with the computed size-change graphs, one proceeds iteratively as follows:
   (a) compute the concatenation of every pair of (not necessarily different) multigraphs of $\mathcal{M}$;
   (b) update $\mathcal{M}$ with the new multigraphs.
   This process is repeated until no new multigraphs are added to $\mathcal{M}$.

Unfortunately, such a naive algorithm is unacceptably expensive and does not scale up to even simple programs. Therefore, in the following, we introduce a much more efficient procedure. Intuitively speaking, it improves the naive procedure by taking into account the following observations:

[5] A more relaxed condition based on the notion of *instantiated enough* w.r.t. a norm [22] can be found in [17].

– Firstly, not all size-change graphs need to be constructed, but only those in the path of a (potential) loop. For instance, in Example 1, the size-change graph from *mlist* to *empty* cannot contribute to the construction of any idempotent multigraph.
– Secondly, in many cases, computing the idempotent multigraphs for a single predicate for each loop suffices. For instance, in Example 2, the idempotent multigraphs for both *mlist* and *ml* actually refer to the same loop. This is somehow redundant since either the two multigraphs will point out that both predicates terminate or that both of them may loop.
– Finally, when we have multigraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ for a given predicate p/n such that termination of p/n using $\mathcal{G}_1$ always implies termination of p/n using $\mathcal{G}_2$, then we can safely discard $\mathcal{G}_2$.

These observations allow us to design a faster procedure for size-change analysis. It proceeds in a stepwise manner as follows:

a) Identifying the program loops. In order to identify the (potential) program loops, we first construct the *call graph* of the program, i.e., a directed graph that contains the predicate symbols as vertices and an edge from predicate p/n to predicate q/m for each clause of the form[6] $p(\overline{t_n}) \leftarrow B_1, \ldots, q(\overline{s_m}), \ldots, B_k$, $k \geq 1$, in the program.

For instance, the call graph of program *MLIST* in Example 1 is as follows:

$$\begin{tikzcd}
 & & mlist \arrow[r, bend left] \arrow[lld] \arrow[ld] \arrow[d] \arrow[rd] & ml \arrow[l, bend left] \arrow[r] & mult \arrow[loop below] \arrow[r] & add \arrow[loop below] \\
empty & nonempty & hd & tl & &
\end{tikzcd}$$

Then, we compute the strongly connected components (SCC) of the call graph and delete both trivial SCCs (i.e., SCCs with a single predicate symbol which is not self-recursive) and edges between SCCs. We denote the resulting graph with $scc(P)$ for any program P. E.g., for program *MLIST*, $scc(MLIST)$ is as follows:

$$\begin{tikzcd}
mlist \arrow[r, bend left] & ml \arrow[l, bend left] & mult \arrow[loop below] & add \arrow[loop below]
\end{tikzcd}$$

b) Determining the initial set of size-change graphs. We denote by $sc\_graphs(P)$ a subset of the size-change graphs of program P that fulfils the following condition: there is a size-change graph from atom $p(\overline{t_n})$ to atom $q(\overline{s_m})$ in $sc\_graphs(P)$ iff there is an associated edge from p/n to q/m in $scc(P)$. E.g., for program *MLIST* of Example 1, $sc\_graphs(MLIST)$ contains only four size-change graphs, while the naive approach would have constructed ten size-change graphs.

In principle, only the size-change graphs in $sc\_graphs(P)$ need to be considered in the size-change analysis. This refinement is correct since *idempotent*

[6] We use $\overline{t_n}$ to denote the sequence $t_1, \ldots, t_n$.

multigraphs can only be built from the concatenation of a sequence of size-change graphs that follows the path of a cycle in the call graph (i.e., a path of $scc(P)$).

Furthermore, not all concatenations between these size-change graphs are actually required. As mentioned before, computing a single idempotent multigraph for each (potential) program loop suffices. In the following, we say that S is a *cover set* for $scc(P)$ if S contains *at least* one predicate symbol for each loop in $scc(P)$. We denote by $CS(P)$ the set of cover sets for $scc(P)$.

Definition 3 (initial size-change graphs). *Let P be a program and $S \in CS(P)$ be a cover set for $scc(P)$. We denote by $i\_sc\_graphs(P, S)$ the size-change graphs from $sc\_graphs(P)$ that start from a predicate of S.*

Intuitively, the size-change graphs in $i\_sc\_graphs(P, S)$ will act as the *seeds* of our iterative process for computing idempotent multigraphs. As a consequence, only idempotent multigraphs for the predicates of S are produced. Therefore, the termination result of Sect. 2 should be rephrased as follows:

> A predicate p/n terminates if there exists some (not necessarily different) predicate q/m in the same cycle of $scc(P)$ and every idempotent multigraph of q/m contains at least one edge $i_q \xrightarrow{\succ} i_q$, $1 \leq i \leq m$, such that the i-th argument of every call to this predicate q/m is ground. $(*)$

A similar condition could be given for quasi-termination. Proving the correctness of this refinement is not difficult and relies on the fact that either all predicates in a loop are terminating or none.

Example 3. Given the program $MLIST$ of Ex. 1, both $S_1 = \{mlist/3, mult/3, add/3\}$ and $S_2 = \{ml/4, mult/3, add/3\}$ are cover sets for $scc(MLIST)$. For instance, the set $i\_sc\_graphs(P, S_1)$ contains only the three size-change graphs starting from $mlist/3$, $mult/3$ and $add/3$.

c) Computing the idempotent multigraphs. The core of our improved procedure for size-change analysis is shown in Fig. 1. The algorithm considers the following ordering on multigraphs:

Definition 4 (weaker multigraph). *Given two multigraphs $\mathcal{G}_1 = \langle V_1, W_1, E_1 \rangle$ and $\mathcal{G}_2 = \langle V_2, W_2, E_2 \rangle$, we say that $\mathcal{G}_1$ is weaker than $\mathcal{G}_2$, in symbols $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$, iff the following conditions hold:*

- *the output and input nodes coincide, i.e., $V_1 = V_2$ and $W_1 = W_2$, and*
- *for every edge $i \xrightarrow{R_1} j \in E_1$, $R_1 \in \{\succ, \succsim\}$, there exists an edge $i \xrightarrow{R_2} j \in E_2$, $R_2 \in \{\succ, \succsim\}$, such that $R_1 \sqsubseteq R_2$*

where $\succ \sqsubseteq \succ$, $\succsim \sqsubseteq \succsim$ and $\succsim \sqsubseteq \succ$, but $\succ \not\sqsubseteq \succsim$.

Basically, if a multigraph $\mathcal{G}$ is weaker than another multigraph $\mathcal{H}$, then we have that whenever termination can be proved with $\mathcal{G}$ only, it could also be proved with both $\mathcal{G}$ and $\mathcal{H}$. Indeed, if $\mathcal{G} \sqsubseteq \mathcal{H}$ and $\mathcal{G}' \sqsubseteq \mathcal{H}'$ then $\mathcal{G} \bullet \mathcal{G}' \sqsubseteq \mathcal{H} \bullet \mathcal{H}'$. Thus, by induction, we can prove that for every size change graph derivable from $\mathcal{H}$ there is a corresponding weaker graph derived from $\mathcal{G}$. Therefore, one can safely discard $\mathcal{H}$ from the computed sets of multigraphs. Intuitively speaking, an idempotent multigraph represents a chain of multigraphs, and this chain is only as strong as its weakest segment.

Example 4. Consider the following four clauses extracted from the regular expression matcher from [18]:

$$\begin{array}{l} \mathit{generate}(\mathit{or}(X,\_),H,T) \leftarrow \mathit{generate}(X,H,T). \\ \mathit{generate}(\mathit{or}(\_,Y),H,T) \leftarrow \mathit{generate}(Y,H,T). \\ \mathit{generate}(\mathit{star}(\_),T,T). \\ \mathit{generate}(\mathit{star}(X),H,T) \leftarrow \mathit{generate}(X,H,T1), \mathit{generate}(\mathit{star}(X),T1,T). \end{array}$$

Here, we have the following three size-change graphs:[7]

$$\begin{tikzcd}
1_{gen} \arrow[r, "\succ_{ts}"] & 1_{gen} & 1_{gen} \arrow[r, "\succ_{ts}"] & 1_{gen} & 1_{gen} \arrow[r, "\succsim_{ts}"] & 1_{gen} \\
2_{gen} \arrow[r, "\succsim_{ts}"] & 2_{gen} & 2_{gen} \arrow[r, "\succsim_{ts}"] & 2_{gen} & 2_{gen} & 2_{gen} \\
3_{gen} \arrow[r, "\succsim_{ts}"] & 3_{gen} & 3_{gen} & 3_{gen} & 3_{gen} \arrow[r, "\succsim_{ts}"] & 3_{gen}
\end{tikzcd}$$

using a reduction pair based on the term-size norm, where *generate* is abbreviated to *gen* in the graphs. Here, both the second and third size-change graphs are weaker than the first one, hence the first graph can be safely discarded and also does not have to be concatenated with other graphs.

The algorithm of Fig. 1 follows these principles:

- In every iteration, we only consider concatenations of the form $\mathcal{G}_1 \bullet \mathcal{G}_2$ where $\mathcal{G}_1$ belongs to the current set of multigraphs $\mathcal{M}_i$ and $\mathcal{G}_2$ is one of the original size-change graphs in $\mathit{sc\_graphs}(P)$.
- Also, those graphs that are stronger than some other graphs are removed from the computed multigraphs in every iteration. Here, $\mathcal{M}_{add}$ denotes the weakest multigraphs that should be added to $\mathcal{M}_i$, while $\mathcal{M}_{del}$ keeps track of the already computed graphs (i.e., from $\mathcal{M}_i \cup \mathcal{M}_{add}$) that should be deleted because a weaker multigraph has been produced.

Example 5. Consider again program *MLIST* of Example 1. By using the improved procedure with the cover set $\{\mathit{mlist}/3, \mathit{mult}/3, \mathit{add}/3\}$, only five concatenations are required to get the fixpoint (actually, three of them are only needed to check that a graph is indeed idempotent) and return the final set of idempotent multigraphs (i.e., the first, third and fourth graphs shown in Example 2). With the original algorithm, 48 concatenations were required. This is a simple example, but gives an idea of the speedup factor associated to the new algorithm (more details can be found in Sect. 4).

The following result formally states the correctness of keeping only the weakest multigraphs during the iterative process:

[7] Note that the first two clauses produce the same size-change graph, otherwise we would have four size-change graphs, one for each body atom in the program.

```
1. Input: a program $P$ and a cover set $S \in CS(P)$
2. Initialisation:
   $i := 0$;   $\mathcal{M}_i := i\_sc\_graphs(P,S)$;   $SC := sc\_graphs(P)$
3. repeat
     – $\mathcal{M}_{add} := \emptyset$; $\mathcal{M}_{del} := \emptyset$
     – for all $\mathcal{G}_1 \in \mathcal{M}_i$ and $\mathcal{G}_2 \in SC$ such that $\mathcal{G}_1 \bullet \mathcal{G}_2$ is defined
         (a) $\mathcal{G} := \mathcal{G}_1 \bullet \mathcal{G}_2$
         (b) if $\nexists \mathcal{H} \in (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}$ such that $\mathcal{G} \sqsubseteq \mathcal{H}$ or $\mathcal{H} \sqsubseteq \mathcal{G}$
             then $\mathcal{M}_{add} := \mathcal{M}_{add} \cup \{\mathcal{G}\}$
         (c) if $\exists \mathcal{H} \in (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}$ such that $\mathcal{G} \sqsubseteq \mathcal{H}$ then $\mathcal{M}_{add} := \mathcal{M}_{add} \cup \{\mathcal{G}\}$
             and $\mathcal{M}_{del} := \mathcal{M}_{del} \cup \{\mathcal{H} \in (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}) \mid \mathcal{G} \sqsubseteq \mathcal{H}\}$
     – $\mathcal{M}_{i+1} := (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}$
     – $i := i + 1$
   until $\mathcal{M}_i = \mathcal{M}_{i+1}$
```

Fig. 1. An improved algorithm for size-change analysis

Theorem 1. *Let P be a logic program and $\mathcal{M}$ be the set of idempotent multi-graphs of P computed using the naive algorithm shown at the beginning of this section. Let $\mathcal{M}'$ be the set of idempotent multigraphs computed with the algorithm of Fig. 1 using a cover set S. Then, a predicate $p/n \in S$ is (quasi-)terminating w.r.t. $\mathcal{M}$ iff it is (quasi-)terminating w.r.t. $\mathcal{M}'$.*

As a straightforward corollary, we have that proving termination using the naive algorithm is equivalent to proving termination according to $(*)$ above using the improved algorithm of Fig. 1 for all program predicates (and not only for those predicates in the cover set).

4 Application to Partial Evaluation and Experiments

In this section, we apply our new algorithm to the case of offline partial evaluation of logic programs, both to show the usefulness of the technique in that setting and also to evaluate its scalability in realistic applications.

4.1 Offline Partial Evaluation of Logic Programs

There are two basic approaches to partial evaluation, differing in the way termination issues are addressed [14]. *Online* specializers include a single, monolithic algorithm, while *offline* partial evaluators contain two clearly separated stages: a binding-time analysis (BTA) and the proper partial evaluation. A BTA normally includes both a termination analysis and an algorithm for propagating *static* (i.e., known) information through the program. The output of the BTA is an annotated version of the source program where every call is decorated either with unfold (to be evaluated) or memo (to be residualized, i.e., the call will become part of the residual program); also, every procedure argument is annotated either with static (definitely known at partial evaluation time) or dynamic (possibly unknown at partial evaluation time). Typically, offline partial evaluators are faster but less precise than online partial evaluators.

In the following, *patterns* are expressions of the form $p(b_1, \ldots, b_n)$, with p/n a predicate symbol of arity n and $b_1, \ldots, b_n$ *binding-times*. Here, we consider a simple domain of binding-times with two elements: static and dynamic; more refined domains can be found in, e.g., [6].

An offline partial evaluator takes an annotated program and an initial set of atoms and proceeds iteratively as follows:

- First, the initial atoms are unfolded as much as possible according to the program annotations. This is called the *local* level of partial evaluation.
- Then, every atom in the leaves of the incomplete SLD trees produced in the local level are added—perhaps generalising some of their arguments—to the set of (to be) partially evaluated atoms. This is called the *global* level of partial evaluation.

Similarly, termination issues can be split into local and global termination, i.e., termination of the local and global levels, respectively. Following the (quasi-)termination results sketched at the end of Sect. 2, source programs are annotated as follows:[8]

Local termination. If all idempotent multigraphs for a predicate p/n include an edge $i_p \xrightarrow{\succ} i_p$ and the i-th argument of p/n is static, then all calls to p/n are annotated with unfold; otherwise, they are annotated with memo.

Global termination. If all idempotent multigraphs for a predicate p/n include an edge $j_p \xrightarrow{R} i_p$ such that $R \in \{\succ, \succsim\}$ and its j-th argument is static, then the i-th argument of p/n can be kept as static; otherwise, it should be annotated as dynamic so that it will be generalised at the global level.

4.2 Prolog Implementation and Empirical Evaluation

We have implemented our new algorithm from Fig. 1 (cf. Sect. 3) for size-change analysis in SICStus Prolog. To be able to measure the effectiveness of the restriction to SCCs (i.e., the restriction to $\mathit{sc\_graphs}(P)$) and the restriction to only consider one predicate per loop (i.e., the restriction to $\mathit{i\_sc\_graphs}(P, S)$ for some cover set S), we have provided a way to turn these optimisations off. We also compare to the old implementation from [17], which includes none of the new ideas presented in this paper.

An interesting implementation technique, which all three versions consider (not described in [17]), is the use of hashing[9] to more quickly identify which size-change graphs already exist and which ones can be concatenated with each

[8] The groundness of an argument is now replaced by the argument being static.

[9] We note that, in earlier versions of SICStus, `term_hash` generates surprisingly many collisions; a problem which we reported and which has been fixed in version 4.0.5.

other. All these three algorithms are integrated into the same BTA from [17], which provides a command-line interface. The BTA is by default polyvariant (but can be forced to be monovariant) and uses a domain with the following values: static, list_nv (for lists of non-variable terms), list, nv (for non-variable terms), and dynamic. The user can also provide hints to the BTA (see below). The implemented size-change analysis uses a reduction pair induced from the term-size norm.

Evaluation of Efficiency. Figure 2 contains an overview of our empirical results, where all times are in seconds. A value of 0 means that the timing was below our measuring threshold. The experiments were run on a MacBook Pro with a 2.33 GHz Core2 Duo Processor and 3 GB of RAM. Our BTA was run using SICStus Prolog 4.0.5. The first six benchmarks come from the DPPD [18] library, vanilla, ctl and lambdaint come from [16]. The picemul program is the PIC processor emulator from [11] with 137 clauses and 855 lines of code. javabc and javabc_heap are Java Bytecode interpreters from [10] with roughly 100 clauses. peval are over 2500 lines of Prolog from a partial evaluator for the ground representation from [20]. self_app are the 1925 lines of our size-change analysis and BTA itself. dSL is an interpreter of 444 lines for the dSL specification language [31]. csp is the core interpreter for full CSP-M from [21], consisting of 1771 lines of code. prob is the core interpreter of ProB [19] for B machines, not containing the kernel predicates or the model checker. It consists of 1910 lines of code and deals with B expressions, predicates and substitutions. promela is an interpreter for the full Promela language (see, e.g., [13]), consisting of 1148 lines of code. Finally, goedel is the source code of the Gödel system [12] consisting of 27354 lines of Prolog.[10] The “noentry” annotation in Fig. 2 means that no entry point was provided, hence only the size-change analysis was performed (and no propagation of static information).

The output of the new BTA (without SCC) and the old BTA from [17] are identical as far as local and global annotations are concerned.

In summary, the new size change analysis is always faster and we see improvements of roughly three orders of magnitude on the most complicated examples (up to a factor of 3500 for prob (noentry)). We are able to deal with realistic interpreters and systems exceeding 25K lines of code. For goedel, a small part of the inferred termination conditions are as follows:

```
is_not_terminating(parse_language1, 6, [d,_,_,_,_,_]).
global_binding_times(parse_language1, 6, [s,d,s,s,d,s]).
is_not_terminating(build_delay_condition, 4, [d,d,_,_]).
global_binding_times(build_delay_condition, 4, [s,s,d,d]).
```

In particular, this means that the analysis has inferred that the predicate `parse_language1` can be unfolded if the first argument is static, and that the first, third, fourth and last argument do not need to be generalised to ensure quasi-termination.

[10] Downloaded from `http://www.cs.bris.ac.uk/Research/LanguagesArchitecture/goedel/` and put into a single file, removing module declarations and adapting some of the code for SICStus 4.

Benchmark	Old BTA from [17]	New BTA (without SCC)	New BTA (with SCC)
contains.kmp	0.01	0.00	0.00
imperative-power	2.35	0.03	0.02
liftsolve.app	0.02	0.01	0.01
match-kmp	0.00	0.00	0.00
regexp.r3	0.01	0.00	0.00
ssuply	0.01	0.01	0.01
vanilla	0.01	0.00	0.00
lambdaint	0.17	0.02	0.02
picemul	0.31	0.15	0.15
picemul (noentry)	0.18	0.01	0.01
ctl	0.03	0.02	0.02
javabc	0.03	0.03	0.03
javabc_heap	0.09	0.09	0.09
peval	0.48	0.15	0.06
self_app (noentry)	0.34	0.20	0.05
dSL	0.03	0.01	0.01
csp (noentry)	5.16	0.21	0.09
prob	387.12	1.41	0.61
prob (noentry)	386.63	0.79	0.11
promela (noentry)	330.05	0.35	0.34
goedel (noentry)	1750.90	13.32	2.61

Fig. 2. Empirical results

Compared to the BTA from [6] using binary clauses rather than size-change analysis, the difference is even more striking. This BTA is in turn, e.g., 200 times slower than the old BTA for the picemul example; see [17]. We have also tried the latest version of Terminweb,[11] based upon [5]. However, the online version failed to terminate successfully on, e.g., the picemul example (for which our size-change analysis takes 0.01 s). We have also tried TermiLog,[12] but it timed out after 4 minutes (the maximum time that can be set in the online version).

Evaluation of Precision. Without the use of the SCC optimisations in the algorithm of Fig. 1, the precision remains unchanged w.r.t. [17], and as such the same specialisations can be achieved as described in [17] using hints: e.g., Jones-optimal specialisation for vanilla, reproducing the decompilation from Java byte-code to CLP from [10] or automatically generating the generated code from [11] for picemul.

With the SCC optimisations, we reduce the number of predicates that are memoised. This in turn also reduces the number of hints that a user has to provide to obtain the desired specialisation.

For example, the vanilla example required two hints in [17] and now only one hint is required to obtain a good specialisation. For lambdaint 6 hints were

[11] `http://www.cs.bgu.ac.il/∼mcodish/TerminWeb/`

[12] `http://www.cs.huji.ac.il/∼naomil/termilog.php`

required in [17] to get good performance. Now only two hints are required, expressing the fact that the expression being evaluated and the list of bound variable names are expected to be static and should not be generalised away by the BTA.[13] In the following section we show how the precision of the size-change analysis can be further improved in the setting of partial evaluation, further reducing the need for hints.

5 Propagating Partial Left-To-Right Information

In this section, we extend the size-change analysis in order to right-propagate size information in some cases. Consider, e.g., clause (c_2) in Example 1:

$$(c_2) \quad \mathit{mlist}(L, I, LI) \leftarrow \mathit{nonempty}(L), \mathit{hd}(L, X), \mathit{tl}(L, R), \mathit{ml}(X, R, I, LI).$$

Since our size-change analysis considers *strong* termination, we compare the size of the head of the clause with the size of each atom in the body independently. Therefore, we get no relation between the sizes of list L in the head and its head X and tail R in the call to ml.

In some cases, however, one might assume some additional restrictions. For instance, in many partial evaluators a left-to-right selection rule is used with the only exception that those calls which are annotated with memo are never selected. Therefore, if we know that some calls can be fully unfolded without entering an infinite loop (the case, e.g., of non-recursive predicates), then one can safely propagate the size relationships for the success patterns of these calls to the subsequent atoms in the clause. In principle, these "fully unfoldable" calls can be detected using a standard left-termination analysis (i.e., one that considers a standard left-to-right computation rule), e.g., [5], while size relations of success patterns can be obtained from the computation of the convex hull of [1]. Here, though, we consider that this information is provided by the user by means of hints of the form `'$FULLYUNFOLD'(p,n,size_relations)` where `size_relations` are the interargument size relations for the success patterns of `p/n`. For instance, for the program *MLIST* of Ex. 1, we may have the following hints:

```
'$FULLYUNFOLD'(hd,2,[1>2]).      '$FULLYUNFOLD'(tl,2,[1>2]).
```

which should be read as "when the call to *hd* (resp. *tl*) succeeds, the size of its first argument is strictly greater than the size of its second argument". We note that, in order to be safe, the interargument size relations should be based on the same norm used induce the reduction pair considered in the size-change graphs.

Let us now describe how the size-change analysis can be improved by using this new kind of hints. Consider a clause of the form

$$P \leftarrow Q_1, \ldots, Q_{i-1}, p(t_1, \ldots, t_n), Q_{i+1}, \ldots, Q_m.$$

[13] This does not give exactly the same result; the solution with 6 hints memoises on `eval_if`, which in this case leads to a more efficient version than memoising on `eval`.

together with the hint `'$FULLYUNFOLD'(p,n,I)`. Then, we first replace this clause by the following ones:

$$
\begin{array}{l}
P \leftarrow Q_1, \ldots, Q_{i-1}, p_{entry}(x_1, \ldots, x_k, t_1, \ldots, t_n). \\
p_{entry}(x_1, \ldots, x_k, y_1, \ldots, y_n) \leftarrow p(y_1, \ldots, y_n), p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n). \\
p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n) \leftarrow Q_{i+1}, \ldots, Q_m.
\end{array}
$$

where $\{x_1, \ldots, x_k\} = (\mathcal{V}ar(P, Q_1, \ldots, Q_{i-1}) \cap \mathcal{V}ar(Q_{i+1}, \ldots, Q_m)) \setminus \mathcal{V}ar(p(t_1, \ldots, t_n))$. This transformation is clearly safe w.r.t. SLD resolution since the original clause can be obtained by just unfolding both p_{entry} and p_{exit}.

Now, the size-change graphs of the first and third clauses are computed as usual. For the second clause, however, we assume that the atom $p(y_1, \ldots, y_n)$ could be fully unfolded producing the set of clauses

$$
\begin{array}{l}
p_{entry}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_1 \leftarrow p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_1. \\
\ldots \\
p_{entry}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_j \leftarrow p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_j.
\end{array}
$$

where $\sigma_1, \ldots, \sigma_j$ are the computed answers and the set of interargument size relations I safely approximates the size relations between the arguments of p_{entry} and p_{exit}. Note that we do not need to fully unfold p/n to construct the size-change graphs (it is rather a device to show the correctness of our approach). Formally, for every relation $i > j$ (resp. $i \geqslant j$) in the interargument size relations for p/n, we should add an edge $i_{p_{entry}} \overset{\succ}{\longmapsto} j_{p_{exit}}$ (resp. $i_{p_{entry}} \overset{\succsim}{\longmapsto} j_{p_{exit}}$) to the size-change graph from p_{entry} to p_{exit}. Moreover, we add an edge of the form $i_{p_{entry}} \overset{\succsim}{\longmapsto} i_{p_{exit}}$ since both p_{entry} and p_{exit} are actually the same predicate.

For instance, by considering the previous hints for program *MLIST*, the clause (c_2) is transformed into

$$
\begin{array}{ll}
(c_{21}) & \mathit{mlist}(L, I, LI) \leftarrow \mathit{nonempty}(L), \mathit{hd}_{entry}(L, X, I, LI). \\
(c_{22}) & \mathit{hd}_{entry}(L, X, I, LI) \leftarrow \mathit{hd}(L, X), \mathit{hd}_{exit}(L, X, I, LI). \\
(c_{23}) & \mathit{hd}_{exit}(L, X, I, LI) \leftarrow \mathit{tl}_{entry}(L, R, X, I, LI). \\
(c_{24}) & \mathit{tl}_{entry}(L, R, X, I, LI) \leftarrow \mathit{tl}(L, R), \mathit{tl}_{exit}(L, R, X, I, LI). \\
(c_{25}) & \mathit{tl}_{exit}(L, R, X, I, LI) \leftarrow \mathit{ml}(X, R, I, LI).
\end{array}
$$

Now, by using the interargument size relations for *hd* and *tl*, we construct the following size-change graphs associated to clauses c_{22} and c_{24}:

$$\begin{tikzcd}
1_{hd_{entry}} \arrow[r, "\succsim_{ts}"] \arrow[dr, "\succ_{ts}"] & 1_{hd_{exit}} \\
2_{hd_{entry}} \arrow[r, "\succsim_{ts}"] & 2_{hd_{exit}} \\
3_{hd_{entry}} \arrow[r, "\succsim_{ts}"] & 3_{hd_{exit}} \\
4_{hd_{entry}} \arrow[r, "\succsim_{ts}"] & 4_{hd_{exit}}
\end{tikzcd}
\qquad
\begin{tikzcd}
1_{tl_{entry}} \arrow[r, "\succsim_{ts}"] \arrow[dr, "\succ_{ts}"] & 1_{tl_{exit}} \\
2_{tl_{entry}} \arrow[r, "\succsim_{ts}"] & 2_{tl_{exit}} \\
3_{tl_{entry}} \arrow[r, "\succsim_{ts}"] & 3_{tl_{exit}} \\
4_{tl_{entry}} \arrow[r, "\succsim_{ts}"] & 4_{tl_{exit}} \\
5_{tl_{entry}} \arrow[r, "\succsim_{ts}"] & 5_{tl_{exit}}
\end{tikzcd}$$

Finally, by constructing the size-change graphs for clauses c_{21}, c_{23} and c_{25} as usual, the size-change analysis can now infer the right relation between the sizes of list L in the atom $\mathit{mlist}(L, I, LI)$ and the head X and tail R in the atom $\mathit{ml}(X, R, I, LI)$.

6 Discussion and Conclusion

In this paper, we have presented a new algorithm to perform strong termination and quasi-termination inference using size-change analysis. The experiments have shown that we can analyse the full 25K lines of source code of the Gödel system in under three seconds. The main application of this algorithm is for offline partial evaluation of large programs. In the experimental evaluation we have shown that, with our new algorithm, we can now deal with realistic interpreters, such as the interpreter for the full B specification language from [19]. Together with the selective use of hints [17], we have obtained both a scalable and an effective partial evaluation procedure. The logical next step is to bring this work to practical fruition, by, e.g., optimising the interpreter from [19] for particular specifications, speeding up the animation and model checking process. This challenge has been on our research agenda for quite a while, and we now believe that the goal can be achieved in the near future. One remaining technical hurdle is the treatment of `meta_predicate` annotations (the B interpreter uses meta-predicates to implement delaying versions of negation and findall).

References

1. F. Benoy, A. King, and F. Mesnard. Computing convex hulls with a linear solver. *TPLP*, 5(1-2):259–271, 2005.
2. M. Bezem. Strong Termination of Logic Programs. *Journal of Logic Programming*, 15(1&2):79–97, 1993.
3. Annalisa Bossi, Sandro Etalle, and Sabina Rossi. Properties of input-consuming derivations. *TPLP*, 2(2):125–154, 2002.
4. Annalisa Bossi, Sandro Etalle, Sabina Rossi, and Jan-Georg Smaus. Termination of simply moded logic programs with dynamic scheduling. *ACM Trans. Comput. Log.*, 5(3):470–507, 2004.
5. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
6. S.-J. Craig, J. Gallagher, M. Leuschel, and K.S. Henriksen. Fully Automatic Binding Time Analysis for Prolog. In *Proc. of LOPSTR'04*, pages 53–68. Springer LNCS 3573, 2005.
7. Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.
8. 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.
9. Allen Van Gelder. Deriving constraints among argument sizes in logic programs. *Ann. Math. Artif. Intell.*, 3(2-4):361–392, 1991.
10. Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. Improving the decompilation of Java bytecode to Prolog by partial evaluation. *Electr. Notes Theor. Comput. Sci.*, 190(1):85–101, 2007.
11. K.S. Henriksen and J. Gallagher. Abstract interpretation of pic programs through logic programming. In *SCAM*, pp. 184–196. IEEE Computer Society, 2006.
12. Patricia Hill and John W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

13. Gerard J. Holzmann. The model checker Spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
14. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
15. 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.
16. M. Leuschel, S.-J. Craig, M. Bruynooghe, and W. Vanhoof. Specialising Interpreters Using Offline Partial Deduction. In *Program Development in Computational Logic*, pages 340–375. Springer LNCS 3049, 2004.
17. M. Leuschel and G. Vidal. Fast Offline Partial Evaluation of Large Logic Programs. In *Proc. of LOPSTR'08*, pages 119–134. Springer LNCS 5438, 2009.
18. Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
19. Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
20. Michael Leuschel and Danny De Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *The Journal of Logic Programming*, 36(2):149–193, August 1998.
21. Michael Leuschel and Marc Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Proceedings ICFEM 2008*, LNCS, pages 278–297. Springer-Verlag, 2008.
22. 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.
23. Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Proving Termination for Logic Programs by the Query-Mapping Pairs Approach. In *Program Development in Computational Logic*, pages 453–498, 2004.
24. Elena Marchiori and Frank Teusink. Termination of Logic Programs with Delay Declarations. *J. Log. Program.*, 39(1-3):95–124, 1999.
25. Lee Naish. Coroutining and the construction of terminating logic programs. *Australian Computer Science Communications*, 15(1):181–190, 1993.
26. Jan-Georg Smaus. Termination of Logic Programs Using Various Dynamic Selection Rules. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 43–57. Springer, 2004.
27. J.-G. Smaus, P.M. Hill, and A. King. Verifying termination and error-freedom of logic programs with `block` declarations. *TPLP*, 1(4):447–486, 2001.
28. 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.
29. 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.
30. G. Vidal. Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In *Proc. of PEPM'07*, pages 51–60. ACM Press, 2007.
31. Bram De Wachter, Alexandre Genon, Thierry Massart, and Cédric Meuter. The formal design of distributed controllers with $_{\mathrm{d}}$sl and Spin. *Formal Asp. Comput.*, 17(2):177–200, 2005.