# Fast Offline Partial Evaluation of Logic Programs[☆]

Michael Leuschel[a], Germán Vidal[b,*]

[a]*Institut für Informatik, Universität Düsseldorf, D-40225, Düsseldorf, Germany*
[b]*MiST, DSIC, Universitat Politècnica de València, E-46022, Valencia, Spain*

## Abstract

One of the most important challenges in partial evaluation is the design of automatic methods for ensuring the termination of the process. In this work, we introduce sufficient conditions for the strong (i.e., independent of a computation rule) termination and quasi-termination of logic programs which rely on the construction of size-change graphs. We then present a fast binding-time analysis that takes the output of the termination analysis and annotates logic programs so that partial evaluation terminates. In contrast to previous approaches, the new binding-time analysis is conceptually simpler and considerably faster, scaling to medium-sized or even large examples.

*Keywords:* partial evaluation, termination analysis, logic programming

## 1. Introduction

Partial evaluation [29] is a well-known technique for program specialisation. Essentially, given a program, $\mathtt{pgm}$, and a partition of its input data into the *static* (i.e., known) and *dynamic* (i.e., possibly unknown) data, a partial evaluator returns a *residual* program $\mathtt{pgm_s}$ which is a specialised version of $\mathtt{pgm}$ for the static data $\mathtt{s}$ such that $\mathtt{pgm(s,d)} = \mathtt{pgm_s(d)}$ for all values $\mathtt{d}$ of the dynamic data. An essential component of partial evaluation is a mechanism for *symbolic* execution so that statements are executed when static data suffice to determine the control flow and residualized—to be executed at run time—otherwise.

Partial evaluation techniques have been applied successfully to optimise a large number of software tasks, like pattern recognition, computer graphics, database query answering and integrity constraints checking, scientific computing, etc. [29]. Besides program specialisation, partial evaluation is also a powerful tool to remove the *interpretation overhead* of domain specific languages implemented as libraries of another well established programming language. Moreover, as formalized by the so called Futamura projections [18], partial evaluators constitute a promising approach to automatic compiler generation.

A crucial issue in the design of a partial evaluator is the way in which the 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 calls, i.e., when these computations are *quasi-terminating* [15]. Partial evaluators usually include some kind of *memoization*, which means that, if the same function call has already been evaluated, then this function call is not unfolded again. This is why quasi-termination suffices to ensure the termination of partial evaluation. This relation between quasi-termination and partial evaluation can be traced back to Holst [28], where a finiteness analysis for first-order functional languages was introduced in order to guarantee the termination of partial evaluation.

There are two main approaches to *partial evaluation*, depending on *when* termination issues are addressed. *Online* partial evaluators basically include an augmented interpreter that tries to evaluate the program constructs as much as possible—using the partially known input data—while still ensuring the termination of the process. *Offline* partial evaluators, on the other hand, require a *binding-time analysis* (BTA) to be run before specialisation, which annotates the source code to be specialised. Roughly speaking, the BTA annotates the various calls in the source program as either unfold (executed by the partial evaluator) or memo (executed at run time, i.e., memoized), and annotates the arguments of the calls themselves as static (known at specialisation time) or dynamic (only definitely known at run time).

Online approaches tend to be more accurate but also computationally more expensive, so that they do not scale up well to medium and large programs. In contrast, offline partial evaluators usually deal with abstractions of the static data (rather than its actual values) and, thus, they are often less accurate but much faster.

Glenstrup and Jones [23] have 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. [31] to analyse the termination of first-order functional programs. In this work, we follow a similar scheme in order to define a fast offline partial evaluation scheme for logic programs [44]. In contrast to other approaches, 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—can naturally deal with missing information represented by means of logic variables.

In the context of logic programming, a BTA should ensure that the annotations of the arguments are correct, in the sense that an argument marked as static will be ground in all possible specialisations. It should also ensure that the specialiser will always terminate. This can be broadly classified into local and global termination [33]. The *local* termination of the process implies that no atom is infinitely unfolded. The *global* termination ensures that only a finite number of atoms are unfolded. In previous work, Craig et al [13] have presented a fully automatic BTA for logic programs, whose output can be used for the offline partial evaluator LOGEN [36]. Unfortunately, this BTA still suffers from some serious practical limitations:

- The current implementation does not guarantee global termination.

- The technique and its implementation are quite complicated, consisting of a combination of various other analyses: model-based binding-time inference, binary clause

generation for left-termination analysis,[1] and inter-argument size relation analysis with polyhedra. To make matters more complicated, these analyses also run on different Prolog systems. As a consequence, the current implementation is quite fragile and hard to maintain.

- In addition to the implementation complexity, the technique is also very slow and does not scale to medium-sized examples.

Basically, the main bottleneck of previous approaches based on a left-termination analysis (like, e.g., the analysis based on the *abstract binary unfoldings* [11] as in [13]) is that every time the annotation of an atom changes from unfold to memo during the BTA, the termination analysis should be redone from scratch in order to take into account that this atom will not be unfolded (which implies that some bindings might not be propagated to the next atoms anymore).

In order to overcome these drawbacks, we introduce a (quasi-)termination analysis for logic programs that is *independent* of the selection rule. This approach is clearly less precise than other termination analyses that take into account a particular selection strategy—since no variable bindings are propagated between the body atoms of a clause—but, as a counterpart, is also much faster—since the termination analysis is only run once—and well suited for partial evaluation where flexible selection strategies are often mandatory (see, e.g., [1, 33]). Our BTA for logic programs has the following advantages:

- it is conceptually simpler and considerably faster, scaling to medium-sized or even large examples (e.g., our BTA is able to deal with programs of more than 25,000 lines of code; see Section 7);

- the technique does ensure both local and global termination;

- its accuracy can be improved by making use of user-provided hints or partially taking into account the selection strategy.

We have implemented the new approach, and we will show on experimental results that the new technique is indeed much faster and much more scalable. On some examples, the accuracy is still sub-optimal, and we present various ways to improve this. Still, this is the first BTA for logic programs that can be applied to large programs, and as such is an important step forward.

This work presents our recent advances in the research of offline partial evaluation of logic programs. In particular, it includes and extends previous contributions originally introduced in [53, 42, 40, 41].

The paper is organized as follows. After some preliminaries, Section 3 presents an overview of an offline partial evaluator for logic programs. Then, Section 4 introduces a termination analysis based on the size-change principle. An efficient algorithm for size-change analysis is presented in Section 5, which is then used in the definition of a fully automated binding-time analysis in Section 6. The usefulness and viability of our approach is shown in Section 7, where the results of an experimental evaluation are

---

[1]We use the shorthand *left-termination* analysis to refer to a termination analysis which considers the leftmost selection strategy.

3

discussed. Finally, Section 8 compares our approach to some related work and Section 9 concludes and presents some directions for further research.

## 2. Preliminaries

We assume some familiarity with the standard definitions and notations for logic programs [44]. 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 $\Pi$, $\Sigma$ and $\mathcal{V}$, respectively. We let $\mathcal{T}(\Sigma, \mathcal{V})$ denote the set of *terms* constructed using symbols from $\Sigma$ 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 [16] are denoted by $U_P^E$ and $B_P^E$, respectively (i.e., $U_P^E = \mathcal{T}(\Sigma, \mathcal{V})$ and $B_P^E = \{p(t_1, \ldots, t_n) \mid p/n \in \Pi \land t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})\}$). $\mathcal{V}ar(s)$ 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 $\mathcal{V}ar(s) = \emptyset$.

Substitutions—which are denoted by greek letters (e.g., $\sigma$, $\theta$, $\rho$)—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 $\sigma$. The application of a substitution $\theta$ 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* (equivalently, *less instantiated*) than a syntactic object $s_2$, denoted $s_1 \leqslant s_2$, if there exists a substitution $\theta$ such that $s_1\theta = s_2$. 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 $\rho$. A substitution $\theta$ is a unifier of two syntactic objects $t_1$ and $t_2$ iff $t_1\theta = t_2\theta$; furthermore, $\theta$ is the *most general unifier* of $t_1$ and $t_2$, denoted by $mgu(t_1, t_2)$ if, for every other unifier $\sigma$ 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,[2]

- $H \leftarrow B_1, \ldots, B_m$ is a renamed apart clause of $P$ (in symbols $H \leftarrow B_1, \ldots, B_m \ll P$),

- $\sigma = mgu(A_i, H)$, and

- $Q' = (\langle A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n \rangle)\sigma$.

_____

[2]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 orthogonal to the topics of this work. We also ignore the fact that technically speaking the atom $A_i$ may occur twice or more in $Q$.

We often omit $P$, $\mathcal{R}$ and/or $\sigma$ in the notation of an SLD resolution step when they are clear from the context. An *SLD derivation* is a (possibly 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 $\theta = \theta_n \bullet \cdots \bullet \theta_1$ (where $\theta = \emptyset$ if $n = 0$).[3] A finite SLD derivation $Q \leadsto_\theta^* Q'$ is *successful* when $Q' = true$; in this case, we say that $\theta$ restricted to $\mathcal{V}ar(Q)$ is the *computed answer substitution*. SLD derivations are represented by a (possibly infinite) finitely branching tree.

As it is common practice in *partial evaluation* [45], 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.[4]

## 3. Overview: Offline Partial Evaluation

In the context of logic programs, a partial evaluator typically specialises a given source program $P$ for a particular atom of interest $A$. The partial evaluator then produces a specialised version of $P$, which can be called for instances of $A$. The information provided to the partial evaluator is thus twofold: it knows the entry predicate $p$ that is going to be used (namely the predicate of $A$), as well as (possibly) the values of certain arguments of $p$ for all possible entry calls at runtime. For example, given $A = p(1, X, [Y, Z])$, possible runtime instances are $p(1, a, [1, 3])$ or $p(1, V, [b, V])$, and the partial evaluator knows that the entry predicate is going to be $p$ (with arity 3) and it knows that the first argument to $p$ will always be 1 (at least for the initial, top-level call to $P$). The partial evaluator also has partial information about the third argument, i.e., that it must always be a list of length 2. The first argument to $p$ is called *static*: we know the value of the argument already at specialisation time. The second argument to $p$ is called *dynamic*: we have no information at all about the possible values that may occur at runtime. The third argument is often called *partially static*: we have some information about the possible values at runtime, but not enough to infer the exact value.

The partial evaluator exploits this information by propagating the entry point information using *unfolding* (which at the same time constructs specialised clauses) and maintaining a set $Q$ of potential calls at runtime.

As already mentioned, approaches to partial evaluation are categorized into online and offline ones. Motivations for offline rather than online partial evaluation are manifold, ranging from improved performance to easing self-application and generating compilers from interpreters. In this paper, we focus on an offline approach that follows the scheme shown in Figure 1. Intuitively speaking, the specialisation process is partitioned into two phases as follows:

1. A *binding-time analysis* (BTA) which annotates the source program to be specialised. Basically, the BTA annotates every call of the source program as either unfold (executed by the partial evaluator) or memo (executed at runtime, i.e., memoized). The BTA also annotates the arguments of the calls themselves with the values

---

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

[4]Partial evaluation often require computing incomplete derivations in order to guarantee the termination of the process.
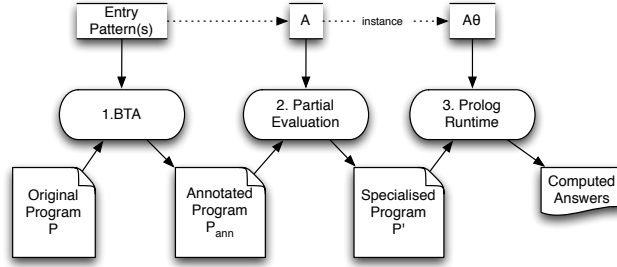
Figure 1: Offline Partial Evaluation Scheme

**Input:** an annotated program $P$ and an atom $A$
**Output:** a partially evaluated program $P_A$
**Initialization:** $i := 0$; $Q := \{A\}$
**Repeat**
    $Q' := Q$;
    $P' := unfold(Q, P)$;
    $Q := abstract(Q, P')$;
**Until** $Q' = Q$ (modulo variable renaming)
**Return:** $P'$

Figure 2: Basic algorithm for partial evaluation

of a binding-time domain. For instance, for the simplest domain $\{\mathsf{static}, \mathsf{dynamic}\}$, the BTA annotates the arguments as either $\mathsf{static}$ (definitely known at speciali- sation time) or $\mathsf{dynamic}$ (only definitely known at runtime). The BTA is given an entry pattern, stating the entry predicate for specialisation as well as which of its arguments will be static. In our example above, the pattern could look like $p(\mathsf{static}, \mathsf{dynamic}, \mathsf{dynamic})$.[5] Observe that the BTA does not yet know the values of the static arguments.

2. A *partial evaluation* phase, which mainly follows the annotations of the BTA, now using and propagating the values of the static arguments. This phase basically proceeds by following the algorithm of Figure 2 (see, e.g., [20]). Here, we take as input an annotated program and an initial atom[6] and proceed iteratively as follows:

- Each atom in the current set $Q$ of atoms to be partially evaluated is unfolded as much as possible following the $\mathsf{unfold}$ and $\mathsf{memo}$ annotations in $P$. This process is usually referred to as the *local control* of partial evaluation

- For each non-failing root-to-leaf derivation $A \rightsquigarrow_\sigma G$, a residual clause—often called resultant [45]—of the form $A\sigma \leftarrow G$ is returned, gradually giving rise to the unfolded program $P'$.

- All atoms in the bodies of $P'$ should be added to $Q$ in order to guarantee the *closedness* [45] of the specialised program (i.e., to ensure that every call

---

[5] Actually, it could be more accurate depending on the binding-time domain, e.g., it could have the form $p(\mathsf{static}, \mathsf{dynamic}, \mathsf{list})$, see Section 6.

[6] We present a simple scheme where only atomic queries are considered. Current partial evaluators deal with conjunctions instead, giving rise to much more accurate results. Since this is orthogonal to the topics of this paper, we keep the basic scheme for simplicity.

that might occur at runtime will be appropriately covered by some specialised procedure). In order to guarantee the finiteness of the process, an abstraction operator is usually applied. It basically generalizes those atom subterms which are annotated as dynamic (i.e., replaces them with fresh variables) and discards those atoms which are variants of the atoms which are already in $Q$. This part of the process is usually called the *global control* of partial evaluation.

Therefore, a BTA should ensure termination at both the local and global control levels. In other words, it should ensure local and global termination, which are defined as follows:

**Local termination:** by unfolding calls annotated as unfold and never selecting calls annotated as memo at partial evaluation time, all SLD derivations are finite.

**Global termination:** by generalizing the dynamic parts of collected atoms and discarding variants, the set of specialised atoms cannot grow infinitely.

As it is common practice, an essential component of our BTA is the termination analysis. In particular, we analyze both termination and quasi-termination. While termination results are useful to infer unfold/memo annotations, the quasi-termination analysis is essential to determine which predicate arguments should definitely be annotated as dynamic in order to ensure global termination.

## 4. Size-Change Termination Analysis

In this section, we present strong termination and quasi-termination analyses for logic programs. For this purpose, we adapt the size-change termination analysis of Lee et al [31] to the logic programming setting.

As mentioned before, we focus in *strong* termination in order to keep the analysis independent of a particular selection strategy. Our notion of strong termination, which is a slight generalization of the standard notion of Bezem [6], is defined as follows:

**Definition 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. $P$.

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

The following auxiliary definitions introduce the notions of *calls* and *calls-to relation*; they are slight extensions of the same notions in [11] in order to consider an arbitrary computation rule.

**Definition 2 (calls).** Let $P$ be a program, $\mathcal{R}$ a computation rule, and $Q_0$ a query. We say that $A$ is a call in a derivation of $Q_0$ with $P$ and $\mathcal{R}$ iff $Q_0 \rightsquigarrow^* Q$ 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 (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 $A \overset{Q_0}{\hookrightarrow}_{P,\mathcal{R}} B\theta$, if $A \in calls_P^{\mathcal{R}}(Q_0)$, $\langle A \rangle \rightsquigarrow_{\sigma} \langle \ldots, B, \ldots \rangle \rightsquigarrow_{\theta}^* \langle \ldots, B\theta, \ldots \rangle$ and $\mathcal{R}(\langle \ldots, B\theta, \ldots \rangle) = B\theta$. When it is clear from the context, we write $A \hookrightarrow B\theta$ or $A \hookrightarrow_{\delta} B\theta$ to emphasise that $\delta$ is the substitution associated with a corresponding derivation from $\langle A \rangle$ to $\langle \ldots, B\theta, \ldots \rangle$ (i.e., $\delta = \sigma\theta$ above).

We note that, in contrast to [11], our calls-to relation only considers *direct* calls, i.e., $A \hookrightarrow B$ implies that $B$ is an atom in the body of the clause used to unfold $A$ (possibly instantiated when other siblings are first selected and unfolded). The relation of [11] can be seen as the transitive closure of our definition.

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: $\langle p \rangle \rightsquigarrow \langle p \rangle \rightsquigarrow \ldots$

The size-change principle [31] is a 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 *size-change graphs*. Then, assuming that the measure of size gives rise to a well-founded order, the following principle applies [31]: *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 *idempotent multigraph* to logic programs; we mainly follow the presentation in [47] for rewrite systems, which in turn originates from the original work of [31] for first-order functional programs. We do not make any assumption on the orders to be partial or total in this work (which mainly affects to the accuracy of the analysis but not its correctness).

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 $\sigma$. By abuse of notation, we say that a quasi-order is well-founded whenever its strict part (i.e., $>$ defined by $s > t$ iff $s \succsim t \wedge t \not\succsim s$) is well-founded. An interesting property of well-founded quasi-orders is that, in any infinite quasi-descending sequence $t_0 \succsim t_1 \succsim t_2 \succsim \ldots$, from some point on, all elements are equivalent under the equivalence relation induced by $\succsim$ [15].

Our size-change graphs are parameterized by a reduction pair:

**Definition 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., $\succsim \bullet \succ \subseteq \succ$ and $\succ \bullet \succsim \subseteq \succ$ but $\succ \subseteq \succsim$ is not necessary).

Observe that, as $\succsim$ is reflexive, we always have $\succsim \bullet \succ \supseteq \succ$ and $\succ \bullet \succsim \supseteq \succ$. An example of reduction pair can be found below.

In logic programming, however, termination analyses usually rely on the use of *norms* which measure the size of terms. Now, we show how a reduction pair can easily be induced by a given norm in some cases. For this purpose, we focus on this paper on *symbolic norms* [43]:

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

$$||t|| = \begin{cases} m + \sum_{i=1}^n k_i ||t_i|| & \text{if } t = f(t_1, \ldots, t_n), \ n \geqslant 0 \\ t & \text{if } t \text{ is a variable} \end{cases}$$

$$||t||_{ts} = \begin{cases} n + \sum_{i=1}^n ||t_i||_{ts} & \text{if } t = f(t_1, \ldots, t_n), \ n \geqslant 0 \\ t & \text{if } t \text{ is a variable} \end{cases}$$

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

Figure 3: Term-size and list-length symbolic norms

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. 3.[7] For instance, we have

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

The introduction of variables in the range of the norm provides a simple mechanism to express dependencies between the sizes of terms.

**Definition 6.** By abuse of notation, given two terms $s$ and $t$, we say that $||s|| > ||t||$ (respec. $||s|| \geqslant ||t||$) if $||s\sigma|| > ||t\sigma||$ (respec. $||s\sigma|| \geqslant ||t\sigma||$) holds for all substitutions $\sigma$ that make $||s\sigma||$ and $||t\sigma||$ integer constants.

By definition, the relations $>$ and $\geqslant$ are stable (closed under substitutions):

**Lemma 1.** *Given two terms $s$ and $t$, we have that $||s|| > ||t||$ (respec. $||s|| \geqslant ||t||$) implies $||s\sigma|| > ||t\sigma||$ (respec. $||s\sigma|| \geqslant ||t\sigma||$) for all substitutions $\sigma$.*

The next lemma shows a basic property of symbolic norms that will become useful to prove the correctness of our termination analysis:

**Lemma 2.** *Let $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$ be terms. Then both $||s|| > ||t||$ and $||s|| \geqslant ||t||$ imply $\mathcal{V}ar(||s||) \supseteq \mathcal{V}ar(||t||)$.*

PROOF. By contradiction. Let us assume that $||s|| > ||t||$ (the proof for $||s|| \geqslant ||t||$ would be perfectly analogous) but there exists a variable $x \in \mathcal{V}ar(||t||)$ such that $x \notin \mathcal{V}ar(||s||)$. Therefore, we can construct a ground substitution $\sigma$ that maps all variables but $x$ to constants and $x$ to a sufficiently large term so that $||s\sigma|| > ||t\sigma||$ does not hold, which contradicts our initial assumption. $\square$

For example, $||[a,b|T]||_{ll} > ||[H|T]||_{ll}$ and $var(||[a,b|T]||_{ll}) = var(||[H|T]||_{ll}) = T$.

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

---

[7]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.

**Definition 7 (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 $t$.

For example, the reduction pair $(\succsim_{ts}, \succ_{ts})$ induced by the term-size norm $||\ ||_{ts}$ is defined as follows:

- $s \succ_{ts} t$ iff $||s||_{ts} > ||t||_{ts}$ and

- $s \succsim_{ts} t$ iff $||s||_{ts} \geqslant ||t||_{ts}$.

Therefore, we have $s(X) \succ_{ts} X$, $f(X) \succsim_{ts} f(X)$, $f(X,Y) \succsim_{ts} f(X,a)$, etc.

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

**Lemma 3.** *Let* $||\cdot||$ *be a symbolic norm on* $\mathcal{T}(\Sigma, \mathcal{V})$. *Then, the pair of orders* $(\succsim, \succ)$ *induced by* $||\cdot||$ *is a reduction pair.*

PROOF. The fact that the pair $(\succsim, \succ)$ induced by the symbolic norm $||\cdot||$ is compatible is straightforward since $>$ and $\geqslant$ over natural numbers are compatible.

Now we prove that $(\succsim, \succ)$ is also closed under substitutions. Consider two terms $s$ and $t$ such that $s \succ t$. We want to prove that $s\theta \succ t\theta$ for all substitution $\theta$. We proceed by contradiction. Assume that there exists a substitution $\delta$ such that $s\delta \not\succ t\delta$. By definition of induced orders, there exists some substitution $\sigma$ such that $||s\delta\sigma||$ and $||t\delta\sigma||$ are integer constants but $||s\delta\sigma|| \not\succ ||t\delta\sigma||$. Since $s \succ t$, we have that $||s\vartheta|| > ||t\vartheta||$ for all substitution $\vartheta$ that makes $||s\vartheta||$ and $||t\vartheta||$ integer constants. Therefore, we have a contradiction when $\vartheta = \delta\sigma$. A similar reasoning can be applied to $\succsim$. Hence, $(\succsim, \succ)$ is a reduction pair. $\square$

Now we use size-change graphs to trace the size relationship between the arguments of the atom in the head of clause and the arguments in the body atoms of the same clause. The graphs are parametric w.r.t. a reduction pair:

**Definition 8 (size-change graphs [31]).** 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 $\succ$. If $s_i \not\succ t_j$ and $s_i \succsim t_j$ 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 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.

The *call graph* of a program is often useful to identify the structure of its size-change graphs. As it is common practice, the call graph of the program is 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 $p(t_1, \ldots, t_n) \leftarrow B_1, \ldots, q(s_1, \ldots, s_m), \ldots, B_k$, $k \geq 1$, in the program.
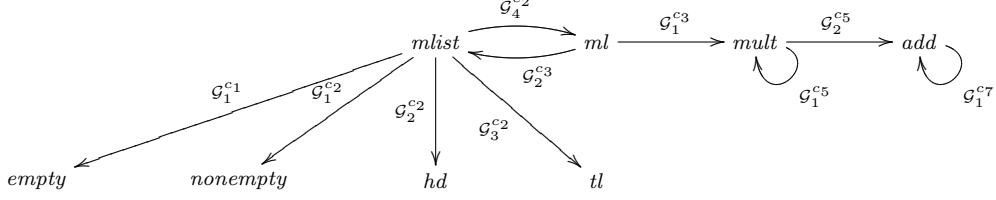
Figure 4: Call graph for program *MLIST*

**Example 1.** Consider the following program *MLIST*:

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

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

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

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

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

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 the technique.

The call graph of program *MLIST* is shown in Figure 4 (the reader can ignore the labels for the moment). The size-change graphs of the program, using a reduction pair $(\succsim_{ts}, \succ_{ts})$ induced by the term-size norm, are shown in Figure 5, where we denote by $\mathcal{G}_n^{c_i}$ the $n$-th size-change graph associated to clause $c_i$.[8] Trivially, there is a size-change graph for every edge of the call graph, as illustrated in Figure 4, where edges are labeled with the names of the associated size-change graphs.

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. For this purpose, the notion of *idempotent multigraph* is introduced:

**Definition 9 (composition, idempotent multigraphs [31, 47]).** Let $P$ a logic program. A multigraph of $P$ is inductively defined to be either a size-change graph of $P$ or the composition (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 composition is defined as follows:

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

---

[8]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.

$$\mathcal{G}_1^{c1}: \quad mlist \quad \longrightarrow \quad empty \qquad \mathcal{G}_1^{c2}: \quad mlist \quad \longrightarrow \quad nonempty \qquad \mathcal{G}_2^{c2}: \quad mlist \quad \longrightarrow \quad hd$$

$$1_{mlist} \xrightarrow{\succsim_{ts}} 1_{empty} \qquad\qquad 1_{mlist} \xrightarrow{\succsim_{ts}} 1_{nonempty} \qquad\qquad 1_{mlist} \xrightarrow{\succsim_{ts}} 1_{hd}$$

$$2_{mlist} \qquad\qquad\qquad 2_{mlist} \qquad\qquad\qquad\qquad 2_{mlist} \qquad 2_{hd}$$

$$3_{mlist} \qquad\qquad\qquad 3_{mlist} \qquad\qquad\qquad\qquad 3_{mlist}$$

$$\mathcal{G}_3^{c2}: \quad mlist \quad \longrightarrow \quad tl \qquad \mathcal{G}_4^{c2}: \quad mlist \quad \longrightarrow \quad ml$$

$$1_{mlist} \xrightarrow{\succsim_{ts}} 1_{tl} \qquad\qquad 1_{mlist} \qquad 1_{ml}$$

$$2_{mlist} \qquad 2_{tl} \qquad\qquad 2_{mlist} \xrightarrow{\succsim_{ts}} 2_{ml}$$

$$3_{mlist} \qquad\qquad\qquad 3_{mlist} \xrightarrow{\succsim_{ts}} 3_{ml}$$

$$\qquad\qquad\qquad\qquad\qquad\qquad\qquad 4_{ml}$$

$$\mathcal{G}_1^{c3}: \quad ml \quad \longrightarrow \quad mult \qquad \mathcal{G}_2^{c3}: \quad ml \quad \longrightarrow \quad mlist \qquad \mathcal{G}_1^{c5}: \quad mult \quad \longrightarrow \quad mult$$

$$1_{ml} \xrightarrow{\succsim_{ts}} 1_{mult} \qquad\qquad 1_{ml} \xrightarrow{\succsim_{ts}} 1_{mlist} \qquad\qquad 1_{mult} \xrightarrow{\succ_{ts}} 1_{mult}$$

$$2_{ml} \xrightarrow{\succsim_{ts}} 2_{mult} \qquad\qquad 2_{ml} \xrightarrow{\succsim_{ts}} 2_{mlist} \qquad\qquad 2_{mult} \xrightarrow{\succsim_{ts}} 2_{mult}$$

$$3_{ml} \xrightarrow{\succ_{ts}} 3_{mult} \qquad\qquad 3_{ml} \xrightarrow{\succ_{ts}} 3_{mlist} \qquad\qquad 3_{mult} \qquad 3_{mult}$$

$$4_{ml} \qquad\qquad\qquad\qquad 4_{ml}$$

$$\mathcal{G}_2^{c5}: \quad mult \quad \longrightarrow \quad add \qquad \mathcal{G}_1^{c7}: \quad add \quad \longrightarrow \quad add$$

$$1_{mult} \xrightarrow{\succ_{ts}} 1_{add} \qquad\qquad 1_{add} \xrightarrow{\succsim_{ts}} 1_{add}$$

$$2_{mult} \xrightarrow{\succsim_{ts}} 2_{add} \qquad\qquad 2_{add} \xrightarrow{\succ_{ts}} 2_{add}$$

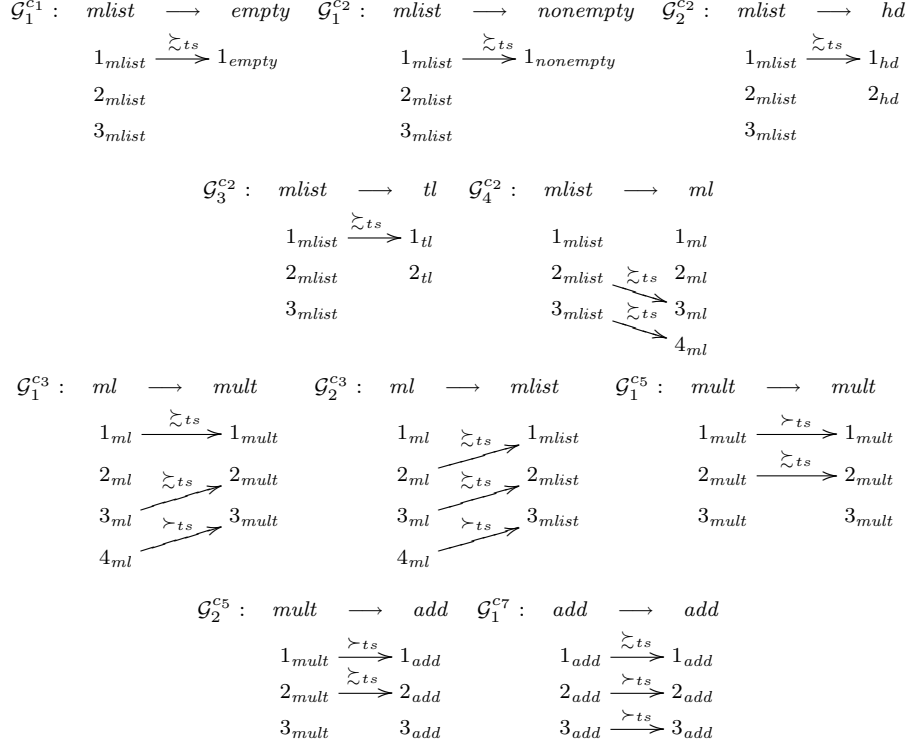$$3_{mult} \qquad 3_{add} \qquad\qquad 3_{add} \xrightarrow{\succ_{ts}} 3_{add}$$

Figure 5: Size-change graphs for *MLIST*

with

$$E = \{(i_p, k_r, L_1 \bullet L_2) \mid \exists j_q.(i_p, j_q, L_1) \in E_1 \wedge (j_q, k_r, L_2) \in E_2\}$$

is also a multigraph, where we define $L_1 \bullet L_2 = \text{``}\succsim\text{''}$ if both $L_1 = L_2 = \text{``}\succsim\text{''}$ and $L_1 \bullet L_2 = \text{``}\succ\text{''}$ in all other cases.[9]

A multigraph $\mathcal{G}$ of $P$ is called *idempotent* if $\mathcal{G} = \mathcal{G} \bullet \mathcal{G}$.

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

**Example 2.** Consider the size-change graphs of program *MLIST* in Example 1 which are shown in Figure 5. Intuitively, only the composition of the size-change graphs associated to the loops of the call graph (Fig. 4) could give rise to idempotent multigraphs (since the result of other compositions would never be idempotent). In particular, we can construct

---

[9]We note that, when there are multiple edges between the same nodes, we keep the edge with the strongest relation in the examples, i.e., given two edges labeled with $\succ$ and $\succsim$, we just show the edge labeled with $\succ$.
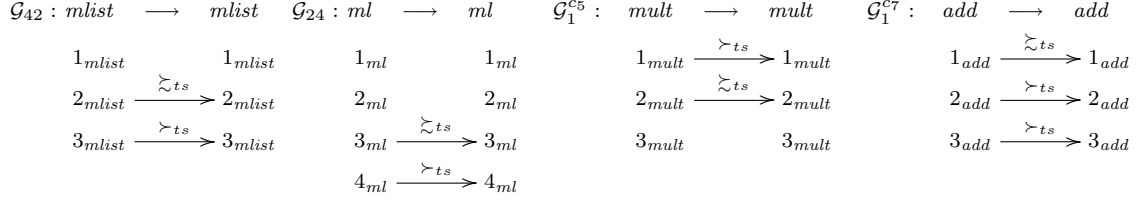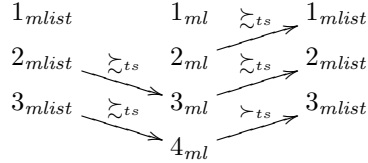
$$\mathcal{G}_{42}: mlist \longrightarrow mlist \quad \mathcal{G}_{24}: ml \longrightarrow ml \quad \mathcal{G}_1^{c_5}: \quad mult \longrightarrow mult \quad \mathcal{G}_1^{c_7}: \quad add \longrightarrow add$$

$$
\begin{array}{cc}
1_{mlist} & 1_{mlist} \\
2_{mlist} \xrightarrow{\succsim_{ts}} 2_{mlist} \\
3_{mlist} \xrightarrow{\succ_{ts}} 3_{mlist}
\end{array}
\qquad
\begin{array}{cc}
1_{ml} & 1_{ml} \\
2_{ml} & 2_{ml} \\
3_{ml} \xrightarrow{\succsim_{ts}} 3_{ml} \\
4_{ml} \xrightarrow{\succ_{ts}} 4_{ml}
\end{array}
\qquad
\begin{array}{cc}
1_{mult} \xrightarrow{\succ_{ts}} 1_{mult} \\
2_{mult} \xrightarrow{\succsim_{ts}} 2_{mult} \\
3_{mult} & 3_{mult}
\end{array}
\qquad
\begin{array}{cc}
1_{add} \xrightarrow{\succsim_{ts}} 1_{add} \\
2_{add} \xrightarrow{\succ_{ts}} 2_{add} \\
3_{add} \xrightarrow{\succ_{ts}} 3_{add}
\end{array}
$$

Figure 6: Idempotent multigraphs for *MLIST*

the following four idempotent multigraphs: $\mathcal{G}_{42} = \mathcal{G}_4^{c_2} \bullet \mathcal{G}_2^{c_3}$, $\mathcal{G}_{24} = \mathcal{G}_2^{c_3} \bullet \mathcal{G}_4^{c_2}$, $\mathcal{G}_1^{c_5}$, and $\mathcal{G}_1^{c_7}$, which are shown in Figure 6. Observe that multigraphs $\mathcal{G}_{42}$ and $\mathcal{G}_{24}$ actually represent the same loop. We will introduce an efficient algorithm for computing the idempotent multigraphs in Section 5.

In the following, similarly to [47], given two multigraphs $\mathcal{G}$ and $\mathcal{H}$ where $\mathcal{G}$'s input nodes have the same labels as $\mathcal{H}$'s output nodes, we let the *concatenation* $\mathcal{G} \circ \mathcal{H}$ be the graph resulting from identifying $\mathcal{G}$'s input and $\mathcal{H}$'s output nodes. Thus $\mathcal{G} \circ \mathcal{H}$ only differs from $\mathcal{G} \bullet \mathcal{H}$ in that these intermediate nodes are not dropped. For example, the graph $\mathcal{G}_4^{c_2} \circ \mathcal{G}_2^{c_3}$ is as follows:

$$
\begin{array}{ccc}
1_{mlist} & 1_{ml} \xrightarrow{\succsim_{ts}} & 1_{mlist} \\
2_{mlist} \xrightarrow{\succsim_{ts}} & 2_{ml} \xrightarrow{\succsim_{ts}} & 2_{mlist} \\
3_{mlist} \xrightarrow{\succsim_{ts}} & 3_{ml} \xrightarrow{\succ_{ts}} & 3_{mlist} \\
& 4_{ml} &
\end{array}
$$

(compare this graph with $\mathcal{G}_{42} = \mathcal{G}_4^{c_2} \bullet \mathcal{G}_2^{c_3}$ shown in Figure 6).

The relevance of idempotent multigraphs is shown in the following result from [47], which in turn follows the same ideas as the proof of Theorem 4 in [31]. Basically, both results are a consequence of Ramsey's theorem [46].

**Lemma 4.** *[31, 47] Let $\mathcal{S}$ be a finite set of size-change graphs. Then,*

- *every graph $\mathcal{G}_1 \circ \mathcal{G}_2 \circ \dots$ with an infinite sequence $\mathcal{G}_1, \mathcal{G}_2, \dots \in \mathcal{S}$ has an infinite path containing infinitely many edges labelled with "$\succ$"*

  *iff*

- *every idempotent multigraph $\mathcal{G}_1 \bullet \mathcal{G}_2 \bullet \dots \bullet \mathcal{G}_m$ with $\mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_m \in \mathcal{S}$ has an edge of the form $i_p \xrightarrow{\succ} i_p$.*

Following [31], we say that a program is *size-change terminating* if every idempotent multigraph contains at least one strictly decreasing parameter:[10]

**Definition 10 (size-change termination).** A program $P$ is size-change terminating w.r.t. a reduction pair $(\succsim, \succ)$ iff every idempotent multigraph of $P$ contains an edge of the form $i_p \xrightarrow{\succ} i_p$.

_____

[10]A reformulation of this condition, where idempotence is not required, can be found in [10].

**Example 3.** Consider the program of Example 1. It is size-change terminating since every idempotent multigraph (shown in Fig. 6) contains at least one edge labelled with "$\succ$".

We 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 [31], there is a worst case exponential growth factor associated to the computation of multigraphs. There exist, though, a number of proposals to improve the efficiency of the size-change analysis which are orthogonal to the topics of this paper (we refer the interested reader to, e.g., the polynomial-time approximation of [30] or the constraint-based approach of [9]).

*4.1. Termination*

In this section, we introduce a sufficient condition for the (strong) *termination* of logic programs that is based on the notion of size-change termination defined above. In the context of functional programming, the notion of size-change termination suffices to ensure that the program is terminating. In the logic programming setting, this is only the case for *ground* queries. In general, though, size-change termination does not generally imply the termination of a program w.r.t. arbitrary (possibly non-ground) queries.

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

$$\langle add(X, Y, Z) \rangle \quad \leadsto_{\{Y/s(Y'), Z/s(Z')\}} \quad \langle add(X, Y', Z') \rangle \quad \leadsto_{\{Y'/s(Y''), Z'/s(Z'')\}} \quad \cdots$$

Therefore, some additional requirements are necessary to ensure termination. Let us first recall the notion of *instantiated enough* w.r.t. a symbolic norm.[11]

**Definition 11 (instantiated enough [43]).** A term $t$ is instantiated enough w.r.t. a symbolic norm $|| \cdot ||$ if $||t||$ is an integer constant.

The next auxiliary lemmas show some basic properties of symbolic norms that will become useful to prove the main result of this section (cf. Theorem 1).

**Lemma 5.** *Let $|| \cdot ||$ be a symbolic norm and $t$ a term which is instantiated enough w.r.t. $|| \cdot ||$. Then, for every substitution $\sigma$, the term $t\sigma$ is also instantiated enough w.r.t. $|| \cdot ||$ and moreover $||t|| = ||t\sigma||$.*

PROOF. Trivial by definition of symbolic norm. □

**Lemma 6.** *Let $|| \cdot ||$ be a symbolic norm and $s, t$ be terms such that $||s|| > ||t||$ (resp. $||s|| \geqslant ||t||$). If $s\theta$ is instantiated enough w.r.t $|| \cdot ||$ for some substitution $\theta$, then $t\theta$ is also instantiated enough w.r.t. $|| \cdot ||$ and moreover $||s\theta|| > ||t\theta||$ (resp. $||s\theta|| \geqslant ||t\theta||$).*

---

[11] A closely related notion is that of *rigidity* [7], where a term $t$ is rigid w.r.t. a norm $|| \cdot ||$ if, for any substitution $\sigma$, $||t\sigma|| = ||t||$.

PROOF. Since $||s|| > ||t||$ (resp. $||s|| \geqslant ||t||$), by Lemma 1 (which states the stability of $>$ and $\geqslant$), we have $||s\theta|| > ||t\theta||$ (resp. $||s\theta|| \geqslant ||t\theta||$). Since $s\theta$ is instantiated enough w.r.t. $||\cdot||$, i.e., $||s\theta|| = k$ for some integer $k$, and $||s\theta|| \geqslant ||t\theta||$ then $||t\theta||$ must also be an integer and, thus, $t\theta$ is instantiated enough w.r.t. $||\cdot||$. □

**Lemma 7.** *Let $P$ be a program and $Q$ be a query. Then, there is an infinite SLD derivation for $Q$ with $P$ if and only if there is an infinite chain in the calls-to relation.*

PROOF. The proof is perfectly analogous to the proof of Lemma 3.5 in [11], since the only significant difference is that [11] considers a leftmost selection rule, while we consider arbitrary SLD derivations. □

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

**Theorem 1 (termination).** *Let $P$ be a program and let $(\succsim, \succ)$ be a reduction pair induced by a symbolic norm $||\cdot||$. Let $\mathcal{A}$ be a set of atoms. If every idempotent multigraph of $P$ contains at least one edge $i_p \overset{\succ}{\longrightarrow} i_p$ 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. $\mathcal{A}$.*

PROOF. We prove the claim by contradiction. Assume that $P$ is not terminating w.r.t. $\mathcal{A}$. Then, by Lemma 7, we have an infinite chain in the associated calls-to relation for some atom $A \in \mathcal{A}$ and computation rule $\mathcal{R}$ of the form $A = A_0 \hookrightarrow_{P,\mathcal{R}} A_1 \hookrightarrow_{P,\mathcal{R}} A_2 \hookrightarrow_{P,\mathcal{R}} \cdots$ For each step $A_j \hookrightarrow_{\sigma_j} A_{j+1}$ in the calls-to relation, there is a (renamed apart) clause $H_j \leftarrow B_1^j, \ldots, B_{m_j}^j \ll P$ which is used to perform an SLD resolution step with $A_j$, where $\theta_j = mgu(A_j, H_j)$, $A_{j+1} = B_{k_j}^j \sigma_j$, $1 \leqslant k_j \leqslant m_j$, and $\theta_j \leqslant \sigma_j$ ($\theta_j$ is more general than $\sigma_j$ because some siblings in the body of the clause could have been selected before $B_{k_j}^j$).

Informally speaking, now we have to prove that there is an infinite sequence of terms $s_0, s_1, \ldots$ where each $s_j$ is an argument of the atom $A_j$ and either $s_j \succ s_{j+1}$ or $s_j \succsim s_{j+1}$ (though there are infinitely many $\succ$), which contradicts the well-foundedness of $\succ$.

First, we know that for each step $A_j \hookrightarrow_{\sigma_j} A_{j+1}$ there is a corresponding size-change graph $\mathcal{G}_j$. Therefore, we have an associated infinite graph[12] of the form $\mathcal{G}_0 \circ \mathcal{G}_1 \circ \ldots$ Since the conditions in the claim of the theorem are stronger, $P$ is trivially size-change terminating. Therefore, by Lemma 4, the graph $\mathcal{G}_0 \circ \mathcal{G}_1 \circ \cdots$ should contain an infinite path where infinitely many edges are labeled with "$\succ$". W.l.o.g., we assume that this path already starts in node $i_p$ of $\mathcal{G}_0$ and that $A_0 = p(\ldots)$ such that there is a maximal multigraph for $p/n$ including a strict edge for $i_p$.[13] For every $j$, let $a_j$ be the output node in $\mathcal{G}_j$ which is on this path. In the following, we denote by $B|_l$ the $l$-th argument of atom $B$. Then, we have $H_j|_{a_j} \succ B_{k_j}^j|_{a_{j+1}}$ for all $j$ from an infinite set $J \subseteq \mathbb{N}$ and

---

[12]Recall that $\mathcal{G} \circ \mathcal{H}$ differs from $\mathcal{G} \bullet \mathcal{H}$ in that intermediate nodes are not dropped.
[13]If this is not the case, one just needs to skip a finite prefix of the path in order to meet the required conditions (see the proof of Lemma 6 in [47]).

$H_j|_{a_j} \succsim B^j_{k_j}|_{a_{j+1}}$ for $j \in I\!\!N \backslash J$. Now, we should prove that $H_j|_{a_j} \succ B^j_{k_j}|_{a_{j+1}}$ implies $A_j|_{a_j} \succ A_{j+1}|_{a_{j+1}}$ and that $H_j|_{a_j} \succsim B^j_{k_j}|_{a_{j+1}}$ implies $A_j|_{a_j} \succsim A_{j+1}|_{a_{j+1}}$.

For $j = 0$, we have $A_0 \hookrightarrow_{\sigma_0} A_1$, where $H_0 \leftarrow B^0_1, \ldots, B^0_{m_0} \lll P$ is a (renamed apart) clause, $\theta_0 = mgu(A_0, H_0)$, $A_1 = B^0_{k_0} \sigma_0$, $1 \leqslant k_0 \leqslant m_0$, and $\theta_0 \leqslant \sigma_0$. Assume that $H_0|_{a_0} \succ B^0_{k_0}|_{a_1}$ holds (the case $H_0|_{a_0} \succsim B^0_{k_0}|_{a_1}$ is perfectly analogous) in the size-change graph associated to the clause $H_0 \leftarrow B^0_1, \ldots, B^0_{m_0}$. By the stability of $\succ$, we have that $H_0|_{a_0}\theta_0 \succ B^0_{k_0}|_{a_1}\theta_0$ holds; also, since $A_0|_{a_0}\theta_0 = H_0|_{a_0}\theta_0$ (because $\theta_0 = mgu(A_0, H_0)$), we have that $A_0|_{a_0}\theta_0 \succ B^0_{k_0}|_{a_1}\theta_0$ also holds. Again by the stability of $\succ$, we have that $A_0|_{a_0}\sigma_0 \succ B^0_{k_0}|_{a_1}\sigma_0$ holds and, thus, $A_0|_{a_0}\sigma_0 \succ A_1|_{a_1}$ (since $A_1 = B^0_{k_0}\sigma_0$). Now, since $A_0|_{a_0}$ is instantiated enough w.r.t. $||\cdot||$ (since there is an edge $a_{0_p} \xrightarrow{\succ} a_{0_p}$ with $a_{0_p} = i_p$ as stated above), by Lemma 5, we have that $A_0|_{a_0}\sigma_0$ is also instantiated enough and $||A_0|_{a_0}|| = ||A_0|_{a_0}\sigma_0||$. Therefore, $A_0|_{a_0} \succ A_1|_{a_1}$ also holds. Finally, by Lemma 6, since $A_0|_{a_0}$ is instantiated enough w.r.t. $||\cdot||$, so is $A_1|_{a_1}$.

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

Note that the strictly decreasing arguments should be instantiated enough in every possible derivation w.r.t. any computation rule. Although this condition is undecidable in general (since the set of derivable calls is infinite in general), it can be approximated by using the binding-times of the computed call patterns (cf. Section 6.2.1).

### 4.2. Quasi-Termination

Now, we focus on *quasi-termination*, a weaker notion than termination. As in the previous section, we first introduce a notion called *size-change quasi-termination* which is not enough for logic programs. Then, we show how (strong) quasi-termination of logic programs can be ensured starting from size-change quasi-termination.

Consider, e.g., the program $P = \{ p \leftarrow q., \quad q \leftarrow p. \}$. Although this program is clearly non-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. As mentioned before, ensuring the termination of partial evaluation on quasi-terminating programs is rather simple (e.g., by using a form of memoisation), which justifies our interest in this property.

**Definition 12 (strong quasi-termination).** A query $Q$ is strongly quasi-terminating w.r.t. a program $P$ if, for every computation rule $\mathcal{R}$, the set $call^{\mathcal{R}}_P(Q)$ 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. $P$.

Trivially, (strong) termination implies (strong) quasi-termination. Quasi-termination is also relevant in logic programming for *tabled* evaluation since the quasi-termination of SLD resolution implies the termination of the tabled mechanism (see, e.g., [52], where quasi-termination w.r.t. Prolog's *leftmost* computation rule is considered).

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 5.** Consider the following program $P$:

$$p([X]) \leftarrow p([f(X)]).$$

and a reduction pair $(\succsim, \succ)$ induced by the symbolic list-length norm (see Fig. 3); note that, according to this symbolic norm, all lists with the same number of elements are considered "equal".

The size-change analysis for this program returns a single idempotent multigraph which contains an edge $1_p \xrightarrow{\succsim} 1_p$ for the argument of $p$. However, infinite non-quasi-terminating SLD derivations exist, e.g.,

$$\langle p([a]) \rangle \rightsquigarrow \langle p([f(a)]) \rangle \rightsquigarrow \langle p([f(f(a))]) \rangle \rightsquigarrow \ldots$$

where $[a] \succsim [f(a)] \succsim [f(f(a))] \succsim \ldots$

To overcome this problem, we require the quasi-order to be *well-founded* (see page 8) and *finitely partitioning*. The second requirement is introduced in the next definition:

**Definition 13 (finitely partitioning quasi-order).** We say that a quasi-order $\succsim$ is finitely partitioning[14] iff for all ground $t$, the set $\{s \mid t \succsim s \wedge \mathcal{V}ar(s) = \emptyset\}$ is finite.

A quasi-order $\succsim$ is thus finitely partitioning if there are not infinitely many "equal" or "smaller" ground terms under $\succsim$. We only consider ground terms since our notion of quasi-termination requires a finite number of *nonvariant* atoms. In other words, all variables are seen as a single fresh constant (i.e., $p(X)$, $p(X')$, $p(X'')$, ..., are the same atom from the point of view of quasi-termination), so we can safely ignore variables in the definition of finitely partitioning quasi-orders.

Nevertheless, we note that, given a a finitely partitioning quasi-order induced by a norm, the terms which are instantiated enough w.r.t. such a norm are necessarily ground:

**Lemma 8.** *Let $(\succsim, \succ)$ be a reduction pair induced by a symbolic norm $\|\cdot\|$. If $\succsim$ is finitely partitioning, then a term $t$ is instantiated enough w.r.t. $\|\cdot\|$ iff $t$ is ground.*

PROOF. We only prove the "only if" part since the "if" claim is trivial. We proceed by contradiction. Let $t$ be a non-ground term which is instantiated enough w.r.t. $\|\cdot\|$. By Lemma 5, $\|t\sigma\| = \|t\|$ for all substitution $\sigma$. Therefore, assuming a non-trivial signature,[15] we have an infinite number of non-variant ground terms (i.e., we only consider those $\sigma$ such that $t\sigma$ is ground) which are equivalent under $\|\cdot\|$ and, thus, under $\succsim$, which contradicts the fact that $\succsim$ is finitely partitioning. □

Thanks to the above result, we will just require ground arguments instead of instantiated enough arguments (as in Theorem 1) when a finitely partitioning quasi-order is considered (see the claim of Theorem 2 below).

---

[14]We follow the terminology of [14, 52]. 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.

[15]In the following, we assume that the considered program signature is always non trivial, i.e., it contains at least one function symbol of arity greater than zero.

Observe that, if the reduction pair includes a finitely partitioning well-founded quasi-order, the situation of Example 5 is no longer possible. Trivially, the quasi-order of the reduction pair induced by the term-size norm is well-founded and finitely partitioning. 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 (see the extension in Section 6.2.2 though).

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

**Definition 14 (size-change quasi-termination).** Let $P$ be a program and let $(\succsim, \succ)$ be a reduction pair, where "$\succsim$" is a finitely partitioning well-founded quasi-order. We say that $P$ is size-change quasi-terminating w.r.t. $(\succsim, \succ)$ iff every idempotent multigraph of $P$ associated to a predicate $p/n$ fulfils at least one of the following two conditions:

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

(ii) for all $i = 1, \ldots, n$, there exists an edge $j_p \xrightarrow{R} i_p$ for some $j \in \{1, \ldots, n\}$, with $R \in \{\succ, \succsim\}$.

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 are bounded by some argument of the previous call 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 would be appropriate in a functional context but does not generally imply the quasi-termination of logic programs. In this case, however, the reason is more subtle and related to non-linearity,[16] as the following example illustrates:

**Example 6.** Consider, for instance, the non-linear query $\langle p(A, A) \rangle$ and the following simple program:
$$p(f(X), Y) \leftarrow p(X, Y).$$
Although we have $||f(X)||_{ts} \geqslant ||X||_{ts}$ and $||Y||_{ts} \geqslant ||Y||_{ts}$, non-quasi-terminating SLD derivations exist, e.g.,

$$\langle p(A, A) \rangle \quad \rightsquigarrow_{\{A/f(X), Y/f(X)\}} \quad \langle p(X, f(X)) \rangle$$
$$\rightsquigarrow_{\{X/f(X'), Y'/f(f(X'))\}} \quad \langle p(X', f(f(X'))) \rangle$$
$$\rightsquigarrow \quad \ldots$$

Intuitively speaking, the problem comes from the propagation of bindings between predicate arguments due to unification.

Before we proceed with the main result of this section, we need some preparatory results. Basically, we need a result relating idempotent multigraphs and the paths in the infinite composition of size-change graphs, analogously to Lemma 4. This is a key result to state the correctness of the characterization of size-change quasi-termination. In the following, we write $j_p \longrightarrow i_q$ as a shorthand for $j_p \xrightarrow{R} i_q$ with $R \in \{\succ, \succsim\}$.

---

[16] An atom is called *linear* if there are no multiple occurrences of the same variable.

**Lemma 9.** *Let $\mathcal{S}$ be a finite set of size-change graphs. Then, the following statements are equivalent:*

1. *For every infinite graph $\mathcal{G}_1 \circ \mathcal{G}_2 \circ \ldots$ with $\mathcal{G}_1, \mathcal{G}_2, \ldots \in \mathcal{S}$, there is a natural number $k > 0$ such that, for every node $v$ in every size-change graph $\mathcal{G}_r$ with $r > k$, there is a path from some node in $\mathcal{G}_k$ to node $v$ in $\mathcal{G}_r$.*

2. *In every idempotent multigraph $\mathcal{G}_1 \bullet \mathcal{G}_2 \bullet \ldots \bullet \mathcal{G}_m$, $m \geqslant 1$, for a predicate symbol $p/n$ with $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_m \in \mathcal{S}$, there exists, for all $i = 1, \ldots, n$, some $j \in \{1, \ldots, n\}$ such that $\mathcal{G}_1 \bullet \mathcal{G}_2 \bullet \ldots \bullet \mathcal{G}_m$ contains an edge $j_p \longrightarrow i_p$.*

PROOF. The proof follows a similar scheme as that of the proof of Lemma 4 (in particular, we follow the proof scheme of Lemma 6 in [47]).

We first prove "(1) $\Rightarrow$ (2)" by contradiction. Assume that there exists an idempotent multigraph $\mathcal{G} = \mathcal{G}_1 \bullet \mathcal{G}_2 \bullet \ldots \bullet \mathcal{G}_m$ for some predicate symbol $p/n$ which has no edge $j_p \longrightarrow i_p$ for some $i \in \{1, \ldots, n\}$, where $\mathcal{G} = (\{1_p, \ldots, n_p\}, \{1_p, \ldots, n_p\}, E)$. By claim (1), we have that the infinite graph $\mathcal{G}_1 \circ \mathcal{G}_2 \circ \ldots \circ \mathcal{G}_m \circ \mathcal{G}_1 \circ \mathcal{G}_2 \circ \ldots \circ \mathcal{G}_m \circ \ldots$ must have a path from some node in $\mathcal{G}_k$ to every node $v$ in every graph $\mathcal{G}_r$ with $k > 0$ and $k < r \leqslant m$. By construction, we can assume that $k = 1$. Therefore, there is a path to every node $i_p$ of $\mathcal{G}_m$ starting from some node $j_p$ of $\mathcal{G}_1$. Hence, the edge $j_p \longrightarrow i_p$ belongs to $\mathcal{G}$ and we get a contradiction.

Now we prove "(2) $\Rightarrow$ (1)" (by contradiction too). Assume that there exists an infinite graph $\mathcal{G}_1 \circ \mathcal{G}_2 \circ \ldots$ with $\mathcal{G}_1, \mathcal{G}_2, \ldots \in \mathcal{S}$ such that there exists some node $v$ in a graph $\mathcal{G}_r$ such that there is no path from any node of graph $\mathcal{G}_k$ to $v$ in $\mathcal{G}_r$, $0 < k < r$. First, for all pair of numbers $(a, b)$ with $a < b$ we let $\mathcal{G}_{a,b}$ be the multigraph resulting from the composition of $\mathcal{G}_a, \mathcal{G}_{a+1}, \ldots, \mathcal{G}_{b-1}$, i.e., $\mathcal{G}_a \bullet \mathcal{G}_{a+1}, \bullet \cdots \bullet \mathcal{G}_{b-1}$. As there are finitely many possible multigraphs, by Ramsey's theorem, there is an infinite $I \subseteq \mathbb{N}$ such that $\mathcal{G}_{a,b}$ is always the same graph for all $a, b \in I$ with $a < b$. We call this graph $\mathcal{G}$. Note that $\mathcal{G}$ is an idempotent multigraph: for $n_1 < n_2 < n_3$ with $n_i \in I$, we have $\mathcal{G}_{n_1,n_3} = \mathcal{G}_{n_1} \bullet \cdots \bullet \mathcal{G}_{n_2-1} \bullet \mathcal{G}_{n_2} \bullet \cdots \bullet \mathcal{G}_{n_3-1} = \mathcal{G}_{n_1,n_2} \bullet \mathcal{G}_{n_2,n_3}$ and, thus, $\mathcal{G} = \mathcal{G} \bullet \mathcal{G}$. Thus, for our original infinite graph, we have

$$\mathcal{G}_1 \circ \mathcal{G}_2 \circ \ldots = \mathcal{G}_1 \circ \cdots \circ \mathcal{G}_{n_1-1} \circ \mathcal{G}_{n_1} \circ \cdots \circ \mathcal{G}_{n_2-1} \circ \mathcal{G}_{n_2} \circ \cdots$$

Now, let us assume that node $v$ belongs to a graph $\mathcal{G}_{n_t-1}$ for some $t > 1$ and let us fix $k = n_1$. By assumption, there is no path to node $v$ from any node of graph $\mathcal{G}_{n_1}$. Then, this must also be true for the graph

$$\mathcal{G}_{n_1} \bullet \cdots \bullet \mathcal{G}_{n_2-1} \circ \mathcal{G}_{n_2} \bullet \cdots \bullet \mathcal{G}_{n_3-1} \circ \cdots = \mathcal{G}_{n_1,n_2} \circ \mathcal{G}_{n_2,n_3} \circ \cdots = \mathcal{G} \circ \mathcal{G} \circ \cdots$$

However, since $\mathcal{G}$ is an idempotent multigraph then for all $i = 1, \ldots, n$, there exists some $j \in \{1, \ldots, n\}$ with $j_p \longrightarrow i_p$, so we get a contradiction. The proof when $v$ belongs to another graph, say $\mathcal{G}_{n_t-2}$, proceeds analogously by considering the graph

$$\mathcal{G}_{n_2-1} \bullet \mathcal{G}_{n_2} \bullet \cdots \bullet \mathcal{G}_{n_3-2} \circ \mathcal{G}_{n_3-1} \bullet \mathcal{G}_{n_3} \bullet \cdots \bullet \mathcal{G}_{n_4-2} \circ \cdots = \mathcal{G}' \circ \mathcal{G}' \circ \cdots$$

with $k = n_2 - 1$ since the graphs $\mathcal{G}_{n_i-1} \bullet \mathcal{G}_{n_i+1} \bullet \cdots \bullet \mathcal{G}_{n_{i+1}-2}$ are also idempotent. $\qquad \square$

Finally, the next result is essential to infer quasi-termination from our notion of size-change quasi-termination. In the following, we say that a (possibly infinite) sequence of ground terms $t_0, t_1, \ldots$ is quasi-terminating if the set $\{t_0, t_1, \ldots\}$ is finite.

**Lemma 10.** *Let $(\succsim, \succ)$ be a reduction pair such that $\succsim$ is well founded and finitely partitioning. Let $s_0 \succsim s_1 \succsim s_2 \succsim \ldots$ be a (possibly infinite) sequence of ground terms and let $t_0, t_1, t_2, \ldots$ be a (possibly infinite) sequence of ground terms such that there exists $k \geqslant 0$ where $s_i \succsim t_{i+1}$ for all $i \geqslant k$. Then, both sequences $s_0, s_1, s_2, \ldots$ and $t_0, t_1, t_2, \ldots$ are quasi-terminating.*

PROOF. Since $\succsim$ is well founded, there exists some $n \geqslant 0$ such that all the terms in the (possibly infinite) set $\{s_n, s_{n+1}, s_{n+2}, \ldots\}$ are equivalent under the equivalence relation induced by $\succsim$ [15], i.e., $s_n \sim s_{n+1} \sim \ldots$ where $\sim$ is the associated equivalence relation (i.e., $t_1 \sim t_2$ iff $t_1 \succsim t_2$ and $t_2 \succsim t_1$). Therefore, the sequence $s_0, s_1, s_2, \ldots$ is quasi-terminating. Moreover, given $m = max(n, k)$, we have $s_n \succsim t_{i+1}$ for all $i \geqslant m$. Hence, since $\succsim$ is finitely partitioning, the set $\{t_{m+1}, t_{m+2}, \ldots\}$ is finite and, thus, the sequence $t_0, t_1, t_2, \ldots$ is quasi-terminating too. $\qquad\square$

We can now state and prove a sufficient condition for quasi-termination. Observe that the assumptions of the next theorem correspond to the conditions for size-change quasi-termination (cf. Definition 14).

**Theorem 2 (quasi-termination).** *Let $P$ be a program and let $(\succsim, \succ)$ be a reduction pair induced by a symbolic norm $|| \cdot ||$, where "$\succsim$" is a finitely partitioning well-founded quasi-order. Let $\mathcal{A}$ be a finite set of atoms. If every idempotent 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 $i \in \{1, \ldots, n\}$ 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. $|| \cdot ||$, or*

*(ii) for every atom $A \in \mathcal{A}$, computation rule $\mathcal{R}$ and atom $p(t_1, \ldots, t_n) \in calls_{\mathcal{R}}^P(A)$, we have that, for all $i = 1, \ldots, n$, there exists an edge $j_p \longrightarrow i_p$ for some $j \in \{1, \ldots, n\}$ such that $t_1, \ldots, t_n$ are ground,*

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

PROOF. We prove the claim by contradiction. Let us assume that $P$ is not quasi-terminating. Therefore, the set $calls_{\mathcal{R}}^P(A)$ contains an infinite number of nonvariant atoms for some atom $A \in \mathcal{A}$ and computation rule $\mathcal{R}$. Hence, there exists an infinite chain in the associated calls-to relation of the form $A = A_0 \hookrightarrow_{P, \mathcal{R}} A_1 \hookrightarrow_{P, \mathcal{R}} A_2 \hookrightarrow_{P, \mathcal{R}} \cdots$ with infinitely many nonvariant atoms. For each step $A_j \hookrightarrow_{\sigma_j} A_{j+1}$ in the calls-to relation, there is a (renamed apart) clause $H_j \leftarrow B_1^j, \ldots, B_{m_j}^j \ll P$ which is used to perform an SLD resolution step with $A_j$, where $\theta_j = mgu(A_j, H_j)$, $A_{j+1} = B_{k_j}^j \sigma_j$, $1 \leqslant k_j \leqslant m_j$, and $\theta_j \leqslant \sigma_j$ ($\theta_j$ is more general than $\sigma_j$ because some siblings in the body of the clause could have been solved before selecting $B_{k_j}^j$).

For each step $A_j \hookrightarrow_{\sigma_j} A_{j+1}$ there is a corresponding size-change graph $\mathcal{G}_j$. Then, we have an associated infinite graph $\mathcal{G}_0 \circ \mathcal{G}_1 \circ \ldots$ If there is an infinite sequence $I = \{i_0, i_1, \ldots\}$ such that $\mathcal{G}_0 \circ \mathcal{G}_1 \circ \cdots = \mathcal{G}_0 \circ \cdots \circ \mathcal{G}_{i_0} \circ \cdots \circ \mathcal{G}_{i_1 - 1} \circ \mathcal{G}_{i_1} \circ \cdots \circ \mathcal{G}_{i_2 - 1} \circ \cdots$ and every composition $\mathcal{G}_{i_s} \bullet \mathcal{G}_{i_s + 1} \cdots \bullet \mathcal{G}_{i_{s+1} - 1}$, $s \in I$, yields the same idempotent multigraph $\mathcal{G}$ and $\mathcal{G}$ satisfies (i), then we have size-change termination and the proof proceeds analogously to that of Theorem 1 by applying Lema 4.

Otherwise, by Lemma 9, there exists some $k > 0$ such that, for all node $v$ of $\mathcal{G}_r$, $r > k$, there exists a path that starts in $\mathcal{G}_k$ and reaches $v$. Since the considered derivation is not quasi-terminating and there is finitely many predicate symbols, we can choose a predicate symbol $p/n$ such that the nodes associated to its arguments occur infinitely often in this graph. Let us denote by $p(t_1^1, \ldots, t_n^1), p(t_1^2, \ldots, t_n^2), p(t_1^3, \ldots, t_n^3), \ldots$ the infinite sequence of calls to $p/n$ associated to the graphs $\mathcal{G}_r$, $r > k$, labelled with $\{1_p, \ldots, n_p\}$. Therefore, there must be at least one infinite path that always traverses the same argument, say the $i$-th argument, so that

$$t_i^1 \succsim t_i^2 \succsim t_i^3 \succsim \ldots$$

By Lemma 10, this sequence is quasi-terminating and, thus, this argument cannot give rise to infinitely many non-variant terms. For the remaining arguments $j \in \{1, \ldots, n\}$, $j \neq i$, either there is also an infinite path that always traverses this argument and we proceed as before, or we have

$$t_i^1 \succsim t_j^2, \ t_i^2 \succsim t_j^3, \ t_i^3 \succsim t_j^4, \ldots$$

and, again by Lemma 10, this sequence cannot give rise to infinitely many non-variant terms. Therefore, only finitely many non-variant calls to $p/n$ exist and we get a contradiction. $\qquad\square$

The relevance of this result for ensuring global termination will be shown in Section 6.2.2.

In the next section, we focus on the efficient implementation of size-change analysis.

## 5. An Efficient Algorithm for Size-Change Analysis

As we have seen in the previous section, the size-change analysis involves computing the composition closure of the size-change graphs of the program. In this section, we introduce an efficient algorithm for computing this closure based on the insight that many size-change graphs are irrelevant for inferring strong termination and quasi-termination conditions in our setting.

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. 8.
2. Then, after initialising a set $\mathcal{M}$ with the computed size-change graphs, one proceeds iteratively as follows:
   (a) compute the composition 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 too expensive and does not scale up to large programs. Therefore, in the following, we introduce a more efficient procedure. Our algorithm does not compute all size-change graphs, but only a subset of them which is sufficient to produce correct annotations for partial evaluation. Intuitively speaking, it improves the naive procedure by taking into account the following observations:

- Firstly, only the size-change graphs in the path of a (potential) loop need to be constructed. 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 to compute correct annotations for partial evaluation. For instance, for program *MLIST*, the idempotent multigraphs for both *mlist* and *ml* actually refer to the same loop. This is somehow redundant since we have that the two multigraphs will point out that both predicates terminate or that both of them may loop. Actually, in the context of partial evaluation, it suffices to annotate one predicate in every loop as memo (i.e., non unfoldable) in order to avoid non-termination at partial evaluation time.

Some of these observations are actually well-known given the ubiquity of closure-type computations in program analysis. Similar opimisations can be found in, e.g., [5, 17, 22] (see also references herein).

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

### 5.1. Identifying the program loops

In order to identify the (potential) program loops, we use the call graph of the program (cf. Section 4). 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 $P$.

For instance, for program *MLIST*, the strongly connected components $scc(MLIST)$ of the call graph shown in Figure 4 is as follows:

$$mlist \overset{\longrightarrow}{\longleftarrow} ml \qquad mult \circlearrowright \qquad add \circlearrowright$$
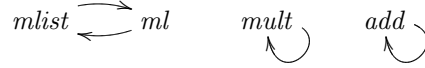
### 5.2. Determining the initial set of size-change graphs

We denote by $sc\_graphs(P)$ the subset of the size-change graphs of a program $P$ that fulfils the following condition: there is a size-change graph $(\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E)$ in $sc\_graphs(P)$ iff there is an associated edge $p/n \longrightarrow q/m$ in $scc(P)$.

**Example 7.** Given the program *MLIST* of Example 1, while the naive approach would have constructed ten size-change graphs (depicted in Figure 5), $sc\_graphs(MLIST)$ contains only four size-change graphs: $\mathcal{G}_4^{c_2}$, $\mathcal{G}_2^{c_3}$, $\mathcal{G}_1^{c_5}$, and $\mathcal{G}_1^{c_7}$.

In principle, only the size-change graphs in $sc\_graphs(P)$ need to be considered in the size-change analysis. This refinement is trivially correct since *idempotent* multigraphs can only be built from the composition of a sequence of size-change graphs that follows the path of a cycle in the call graph (i.e., a path of $scc(P)$).

**Lemma 11.** *Let $P$ be a program and let $\mathcal{S}$ be the associated set of size-change graphs. For all idempotent multigraph $\mathcal{G}_0 \bullet \ldots \bullet \mathcal{G}_k$, with $\mathcal{G}_0, \ldots, \mathcal{G}_k \in \mathcal{S}$, we have that $\mathcal{G}_i \in sc\_graphs(P)$ for all $i = 1, \ldots, k$.*

22

Furthermore, not all compositions 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 all cover sets for $scc(P)$.[17]

**Definition 15 (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 $sc\_graphs(P)$ whose output nodes correspond to the arguments of a predicate in $S$.

Intuitively, the size-change graphs in $i\_sc\_graphs(P, S)$ will act as the *seeds* of our iterative process for computing idempotent multigraphs.

**Example 8.** Given the program *MLIST* of Example 1, we have that both

$$S_1 = \{mlist/3, mult/3, add/3\} \qquad \text{and} \qquad S_2 = \{ml/4, mult/3, add/3\}$$

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 $add/3$ (i.e., the size-change graphs $\mathcal{G}_4^{c_2}$, $\mathcal{G}_1^{c_5}$ and $\mathcal{G}_1^{c_7}$, as depicted in Figure 5). In contrast, the set $i\_sc\_graphs(P, S_2)$ contains the size-change graphs $\mathcal{G}_2^{c_3}$, $\mathcal{G}_1^{c_5}$ and $\mathcal{G}_1^{c_7}$ of Figure 5. Therefore, with $S_1$ we choose $mlist/3$ as the representative node of the loop, and with $S_2$ we choose $ml/4$ instead.

The correctness of our approach is stated in the following lemma:

**Lemma 12.** *Let $P$ be a program and $S \in CS(P)$ be a cover set for $scc(P)$. Let $\mathcal{M}$ be the set of idempotent multigraphs of $P$ computed using the naive algorithm shown at the beginning of this section and let $\mathcal{M}'$ be the set of idempotent multigraphs computed starting only with the size-change graphs in $i\_sc\_graphs(P, S)$. Then, every idempotent multigraph $\mathcal{G}$ of $\mathcal{M}$ fulfills the following conditions:*

- *there is at least one edge $i_p \xrightarrow{\succ} i_p$ or*
- *there are edges $j_p \longrightarrow i_p$ for all nodes $i_p$ of $\mathcal{G}$*

*iff every idempotent multigraph of $\mathcal{M}'$ does.*

PROOF. In the traditional approach, we get an idempotent multigraph for each node of each SCC of $scc(P)$. Trivially, every of this idempotent graphs are obtained from compositions of the form $\mathcal{G}_0 \bullet \ldots \bullet \mathcal{G}_n$ such that the output nodes of $\mathcal{G}_0$ and the input nodes of $\mathcal{G}_n$ refer to the arguments of the same predicate in $scc(P)$. Moreover, given an SCC, all the computed idempotent multigraphs must be clearly *coherent*, i.e., all of them: $\mathcal{G}_0 \bullet \ldots \bullet \mathcal{G}_n$, $\mathcal{G}_1 \bullet \ldots \bullet \mathcal{G}_n \bullet \mathcal{G}_0$, $\mathcal{G}_2 \bullet \ldots \bullet \mathcal{G}_n \bullet \mathcal{G}_0 \bullet \mathcal{G}_1$, etc., contain an edge $i_p \xrightarrow{\succ} i_p$ (not necessarily for the same argument) or none, and similarly for the second condition. Therefore, limiting ourselves to obtaining just one idempotent multigraph per SCC gives the same information and the claim follows. $\qquad \square$

1. **Input:** a program $P$ and a cover set $S \in CS(P)$
2. **Initialisation:**
   $i := 0; \quad \mathcal{M}_i := i\_sc\_graphs(P, S); \quad SC := sc\_graphs(P)$
3. **repeat**

   - $\mathcal{M}_{i+1} := \mathcal{M}_i$
   - 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

     $\mathcal{M}_{i+1} := \mathcal{M}_{i+1} \cup (\mathcal{G}_1 \bullet \mathcal{G}_2)$
   - $i := i + 1$

   **until** $\mathcal{M}_i = \mathcal{M}_{i+1}$
4. **Return** $\{\mathcal{G} \in \mathcal{M}_i \mid \mathcal{G} \text{ is idempotent}\}$

Figure 7: An improved algorithm for size-change analysis (closure computation)

*5.3. Computing the idempotent multigraphs*

   The core of our improved procedure for size-change analysis is shown in Fig. 7. Note that, in every iteration, we only consider compositions 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 $sc\_graphs(P)$. Once a fixpoint is reached, we select the idempotent multigraphs.

**Example 9.** Consider the following four clauses extracted from the regular expression matcher of [32]:

$$generate(or(X, \_), H, T) \leftarrow generate(X, H, T).$$
$$generate(or(\_, Y), H, T) \leftarrow generate(Y, H, T).$$
$$generate(star(\_), T, T).$$
$$generate(star(X), H, T) \leftarrow generate(X, H, T1), generate(star(X), T1, T).$$

Here, we have the following three size-change graphs:[18]

$$
\begin{array}{ccc}
1_{gen} \xrightarrow{\succ_{ts}} 1_{gen} & 1_{gen} \xrightarrow{\succ_{ts}} 1_{gen} & 1_{gen} \xrightarrow{\succsim_{ts}} 1_{gen} \\
2_{gen} \xrightarrow{\succsim_{ts}} 2_{gen} & 2_{gen} \xrightarrow{\succsim_{ts}} 2_{gen} & 2_{gen} \qquad 2_{gen} \\
3_{gen} \xrightarrow{\succsim_{ts}} 3_{gen} & 3_{gen} \qquad 3_{gen} & 3_{gen} \xrightarrow{\succsim_{ts}} 3_{gen}
\end{array}
$$

using a reduction pair based on the term-size norm, where *generate* is abbreviated to *gen* in the graphs. Following the algorithm of Figure 7, we first compute a fixpoint, which

---

[17]Our implementation uses a greedy algorithm to determine a valid cover set for $scc(P)$. Note, however, that it does not necessarily find a minimum cover set.

[18]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.

contains the above three graphs, and also the following two new multigraphs:

$$1_{gen} \xrightarrow{\;\succ_{ts}\;} 1_{gen} \qquad\qquad 1_{gen} \xrightarrow{\;\succ_{ts}\;} 1_{gen}$$

$$2_{gen} \qquad\quad 2_{gen} \qquad\qquad\qquad 2_{gen} \qquad\quad 2_{gen}$$

$$3_{gen} \xrightarrow{\;\succsim_{ts}\;} 3_{gen} \qquad\qquad 3_{gen} \qquad\quad 3_{gen}$$

In this case, all five graphs are idempotent and, thus, all of them are returned by the algorithm.

## 6. A Binding-Time Analysis for Logic Programs

In this section, we present a fully automatic BTA for (definite) logic programs. Our algorithm is parametric w.r.t.

- a domain $\mathcal{D}$ of binding-times and

- the associated function for propagating binding-times through the atoms of clause bodies.

For instance, one can consider a simple domain with the basic binding-times static (definitely known at partial evaluation time) and dynamic (possibly unknown at partial evaluation time). We assume that all binding-time domains contain these two binding times, but they could also include, e.g., the elements such as:

- nonvar: the argument is not a variable at partial evaluation time, i.e., the top-level function symbol is known;

- list: the argument is definitely bound to a finite list of possibly unknown arguments at partial evaluation time.

- list_nonvar: the argument is definitely bound to a finite list, whose elements are not variables;

In the LOGEN system [36], an *offline* partial evaluator for Prolog, the user can also define their own binding-times [13] (called *binding-types* in this context), and one can use the pre-defined list-constructor to define additional types such as list(dynamic) to denote a list of known length with dynamic elements, or list(nonvar) to denote a list of known length with non-variable elements.

A binding-time domain is a set $\mathcal{D}$ of *binding-times* along with a concretisation function $\gamma : \mathcal{D} \to \wp(\mathcal{T}(\Sigma, \mathcal{V}))$. We assume the existence of a bottom element $\perp$ with $\gamma(\perp) = \emptyset$ and a top element $\top$ with $\gamma(\top) = \mathcal{T}(\Sigma, \mathcal{V})$. The top element is usually called dynamic.

The concretisation function for the binding-times mentioned above are defined as follows: $\gamma(\text{static})$ is the set of all ground terms, $\gamma(\text{nonvar})$ is the set of all non-variable terms, $\gamma(\text{list})$ is the set of all terms representing a list of known length, $\gamma(\text{list\_nonvar})$ is the set of all terms representing a list of known length whose elements are non-variable terms.

The concretisation function $\gamma$ induces an order on $\mathcal{D}$: $b_1 \sqsubseteq b_2$ iff $\gamma(b_1) \subseteq \gamma(b_2)$. I.e., $b_1 \sqsubseteq b_2$ denotes that $b_1$ is *less dynamic* than $b_2$. Given this order, we denote the least

upper bound of binding-times $b_1$ and $b_2$ by $b_1 \sqcup b_2$ and the greatest lower bound by $b_1 \sqcap b_2$. We have that for all $x \in \mathcal{D}$:

$$\bot \sqsubseteq x \sqsubseteq \mathsf{dynamic}$$

From this follows that for all $x \in \mathcal{D}$ we have:

$$\bot \sqcup x = x \sqcup \bot = x$$
$$\mathsf{dynamic} \sqcup x = x \sqcup \mathsf{dynamic} = \mathsf{dynamic}$$
$$\bot \sqcap x = x \sqcap \bot = \bot$$
$$\mathsf{dynamic} \sqcap x = x \sqcap \mathsf{dynamic} = x$$

For the particular binding-times mentioned above, we have for example $\bot \sqsubseteq \mathsf{static} \sqsubseteq \mathsf{nonvar} \sqsubseteq \mathsf{dynamic}$ and $\mathsf{static} \sqcup \mathsf{list} = \mathsf{dynamic}$.

Given a set of binding-times $W$, we define $\sqcup W$ as follows ($\sqcap W$ is defined in a similar way):

$$\sqcup W \quad = \quad \begin{cases} \bot & \text{if } W = \emptyset \\ b & \text{if } W = \{b\} \\ b_1 \sqcup (b_2 \sqcup (\ldots \sqcup (b_{n-1} \sqcup b_n))) & \text{if } W = \{b_1, \ldots, b_n\}, \ n > 0 \end{cases}$$

In the following, a *pattern* is defined as an expression of the form $p(b_1, \ldots, b_n)$ where $p/n$ is a predicate symbol and $b_1, \ldots, b_n$ are binding-times. The concretisation function $\gamma$ and the induced order $\sqsubseteq$ and operators $\sqcup$ and $\sqcap$ are extended to patterns in the natural way. Also, given a pattern $p(b_1, \ldots, b_n)$, the function $\bot(p(b_1, \ldots, b_n))$ returns a new pattern $p(\bot, \ldots, \bot)$ where all arguments are set to the bottom element $\bot$.[19]

Given a binding-time domain, we consider an associated domain of *abstract substitutions* that map variables to binding-times. An abstract substitution can be viewed as representing a set of concrete substitutions. More formally, we extend $\gamma$ to abstract substitutions as follows:

$$\gamma(\{v_1/b_1, \ldots, v_k/b_k\}) =_{\mathsf{def}} \{\{v_1/t_1, \ldots, v_k/t_k\} \mid \text{such that } t_i \in \gamma(b_i)\}$$

We now introduce the auxiliary functions *asub* and *pat* to produce abstract substitutions from patterns and vice versa. Their role within the call and success pattern analysis will be clarified below in Section 6.1.

Given an atom $A$ and a pattern $\pi$ with $pred(A) = pred(\pi)$,[20] the partial function $asub(A, \pi)$ returns an abstract substitution $\sigma$ for the variables of $A$ so that $A\sigma$ matches the pattern $\pi$. More formally, $asub(p(t_1, \ldots, t_n), p(b_1, \ldots, b_n)) = \sigma$ means that for all concrete substitutions $\sigma'$ with $p(t_1, \ldots, t_n)\sigma' \in \gamma(p(b_1, \ldots, b_n))$ we have that $\sigma' \in \gamma(\sigma)$.

For instance, for a simple binding-time domain $\mathcal{D} = \{\mathsf{static}, \mathsf{dynamic}\}$, a valid function *asub* can be defined as follows:

$asub(p(t_1, \ldots, t_n), p(b_1, \ldots, b_n))$
$\qquad\qquad =_{\mathsf{def}} \ \{x/b \mid x \in \mathcal{V}ar(p(t_1, \ldots, t_n)) \wedge b = \sqcap\{b_i \mid x \in \mathcal{V}ar(t_i), \ i = 1, \ldots, n\}\}$

---

[19] A technical difficulty arises if the arity of $n$ is 0. The implementation in Section 7 actually uses a special pattern $\bot$ rather than a binding time $\bot$: indeed, either all arguments are $\bot$ or none are $\bot$. Many binding-time analysis actually forego the $\bot$ element and as such do not detect failing predicates.

[20] Auxiliary function *pred* returns the predicate name and arity of an atom or pattern.

---

1. **Input:** a program $P$ and an entry pattern $\pi_{entry}$
2. **Initialisation:** $\mu := \emptyset$; $addentry(\pi_{entry})$
3. **repeat**
   **for** all patterns $\pi_{call} \in dom(\mu)$:
       **for** all clauses $H \leftarrow B_1, \ldots, B_n \in P$, $n \geqslant 0$, with $pred(H) = pred(\pi_{call})$:
       (a) $\sigma_0 := asub(H, \pi_{call})$ // determine binding-times for variables in clause head
       (b) $\sigma_0 := \sigma_0 \cup \{v/\mathsf{dynamic} \mid v \in \mathcal{V}ar(B_1, \ldots, B_n) \setminus \mathcal{V}ar(H)\}$ // mark existential
           variables as dynamic
       (c) **for** i = 1 to n:
           $\sigma_i := get(B_i, \sigma_{i-1})$ // get success substitution of $B_i$ and compose with $\sigma_{i-1}$
       (d) $\mu(\pi_{call}) := \mu(\pi_{call}) \sqcup pat(H, \sigma_n)$ // update success information
   **until** $\mu$ doesn't change

---

Figure 8: Call and success pattern analysis

Roughly speaking, a variable that appears only in one argument $t_i$ will be mapped to the corresponding binding time $b_i$; if the same variable appears in several arguments, then it is mapped to the greatest lower bound of the corresponding binding-times of these arguments. For instance, we have

$$asub(p(X, X), p(\mathsf{static}, \mathsf{dynamic})) = \{X/\mathsf{static}\}$$

Observe that the greatest lower bound is used to compute the less dynamic binding-time of a variable when it is bound to different values.

Given an atom $A = p(t_1, \ldots, t_n)$ and an abstract substitution $\sigma$, function $pat(A, \sigma)$ returns a pattern $p(b_1, \ldots, b_n)$ in which the binding-time of every argument is determined by the abstract substitution. More formally, for all $\sigma' \in \gamma(\sigma)$ we must have that $A\sigma' \in \gamma(pat(A, \sigma))$. For example, when $t_i$ contains no variables $pat$ computes an element $b_i \in \mathcal{D}$ (usually the most precise one) which represents the term (in this case $pat$ corresponds to the abstraction function $\alpha$ of abstract interpretation).

For the binding-time domain $\mathcal{D} = \{\mathsf{static}, \mathsf{dynamic}\}$, this function can be formally defined as follows:

$$pat(p(t_1, \ldots, t_n), \sigma) =_{\mathsf{def}} p(b_1, \ldots, b_n) \text{ where } b_i = \sqcup\{x\sigma \mid x \in \mathcal{V}ar(t_i)\}, \ i = 1, \ldots, n$$

E.g., we have

$$pat(p(X, X), \{X/\mathsf{static}\}) = p(\mathsf{static}, \mathsf{static})$$

and

$$pat(q(f(X, Y)), \{X/\mathsf{static}, Y/\mathsf{dynamic}\}) = q(\mathsf{dynamic})$$

Now, we present our BTA algorithm in a stepwise manner.

### 6.1. Call and Success Pattern Analysis

The first step towards a BTA is basically a simple call and success pattern analysis parameterized by the considered binding-time domain $(\mathcal{D}, \sqsubseteq)$. The algorithm is shown in

Fig. 8. Here, we keep a *memo* table $\mu$ with the call and success patterns already found in the analysis, i.e., if $\mu(\pi_{call}) = \pi_{success}$ then we have a call pattern $\pi_{call}$ with associated success pattern $\pi_{success}$; initially, all success patterns have static arguments. In order to add new entries to the memo table, we use the following function *addentry*:

$$addentry(pattern) \quad =_{\mathsf{def}} \quad \textbf{if } pattern \notin dom(\mu) \textbf{ then } \mu(pattern) := \bot(pattern) \textbf{ fi}$$

where the notation $\mu(pattern) := \bot(pattern)$ is used to denote an update of $\mu$.

Basically, the algorithm takes a logic program $P$ and an entry pattern $\pi_{entry}$ and, after initializing the memo table, enters a loop until the memo table reaches a fixed point. Every iteration of the main loop proceeds as follows:

- for every call pattern $\pi_{call}$ with a matching clause $H \leftarrow B_1, \ldots, B_n$, we first compute the entry abstract substitution $asub(H, \pi_{call})$;

- then, we use function *get* to propagate binding-times through the atoms of the body, thus updating correspondingly the memo table (a global parameter of *get*) with the call and (initial) success patterns for every atom:

$$get(B, \sigma) \quad =_{\mathsf{def}} \quad addentry(pat(B, \sigma)); \textbf{ return } override(asub(B, \mu(pat(B, \sigma))), \sigma)$$

  The auxiliary override function is defined as follows:

$$override(\sigma_1, \sigma_2) \quad =_{\mathsf{def}} \quad \sigma_1 \cup \{x/d \mid x/d \in \sigma_2 \wedge x \notin dom(\sigma_1)\}$$

  i.e., it overrides the bindings of $\sigma_2$ with those of $\sigma_1$.

- finally, we update in the memo table the success pattern associated to the call pattern $\pi_{call}$ using the exit abstract substitution of the clause.

The role of functions *asub* and *pat* when analyzing a clause $H \leftarrow B_1, \ldots, B_n$ can be graphically represented as follows:
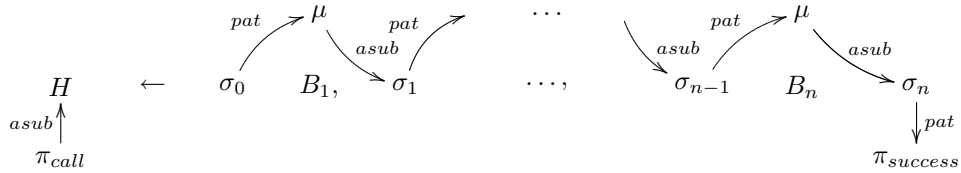


In the next section, we present a BTA that slightly extends this algorithm.

*6.2. A BTA Ensuring Local and Global Termination*

In contrast to the call and success pattern analysis of Fig. 8, a BTA should annotate every call with either unfold or memo so that

- all atoms marked as unfold can be unfolded as much as possible (as indicated by the annotations) while still guaranteeing the local termination, and

- global termination is guaranteed by generalising the dynamic arguments whenever a new atom is added to the set of (to be) partially evaluated atoms; also, all arguments marked as static must indeed be ground.

1. **Input:** a program $P$ and an entry pattern $\pi_{entry}$

2. **Initialisation:** $\mu := \emptyset$; $addentry(\pi_{entry})$; $\boxed{memo := \emptyset}$

3. **repeat**

   **for** all patterns $\pi_{call} \in dom(\mu)$:

       for all clauses $H \leftarrow B_1, \ldots, B_n \in P$, $n \geqslant 0$, with $pred(H) = pred(\pi_{call})$:

       (a) $\sigma_0 := asub(H, \pi_{call})$ // determine binding-times for variables in clause head

       (b) $\sigma_0 := \sigma_0 \cup \{v/\mathsf{dynamic} \mid v \in \mathcal{V}ar(B_1, \ldots, B_n) \setminus \mathcal{V}ar(H)\}$ // mark existential
           variables as dynamic

       (c) **for** i = 1 to n:
   $$\sigma_i := \boxed{cond\_get(B_i, \sigma_{i-1}, ppoint(B_i))}$$

       (d) $\mu(\pi_{call}) := \mu(\pi_{call}) \sqcup pat(H, \sigma_n)$ // update success information

   **until** $\mu$ doesn't change

Figure 9: A BTA ensuring local and global termination

Figure 9 shows a BTA that slightly extends the call and success pattern analysis of Fig. 8 (differences appear in a box). The main changes are as follows:

- We consider that each call $B$ is uniquely identified by a program point $ppoint(B)$. The algorithm keeps track of the considered program points using the set $memo$ (initially empty).

- The function for propagating binding-times, now called $cond\_get$, takes an additional parameter: the program point of the considered atom; function $cond\_get$ is defined in terms of $get$ as follows:

$$cond\_get(B, \sigma, pp) \quad =_{\mathsf{def}} \quad \textbf{if } unfold(pat(B, \sigma)) \wedge pp \notin memo \textbf{ then}$$
$$\textbf{return } get(B, \sigma)$$
$$\textbf{else}$$
$$addentry(gen(pat(B, \sigma))); memo := memo \cup \{pp\};$$
$$\textbf{return } \sigma$$
$$\textbf{fi}$$

If the atom at the program point $pp$ is unfolded then $cond\_get$ calls the $get$ function from Fig. 8) to obtain and apply the success information for the call $B$. In the other case, the call $B$ is memoized and hence $cond\_get$ returns $\sigma$ unchanged. In addition, the call $B$ is generalized using $gen$ described below and registered as a separate entry point for the analysis.

The introduction of this **if-then-else** construct highlights one of the major differences between abstract interpretation and binding-time analysis (see also [8]): the result of the analysis influences the "program" to be analysed.

- Auxiliary functions $gen$ and $unfold$ are defined as follows:

29

– Given a pattern $\pi$, function $unfold(\pi)$ returns true if $\pi$ is safe for unfolding, i.e., if it guarantees *local* termination.

– Given a pattern $\pi$, function $gen(\pi)$ returns a generalization of $\pi$ (i.e., $\pi \sqsubseteq gen(\pi)$) that ensures *global* termination.

Precise definitions for *gen* and *unfold* will be presented in the next sections.

### 6.2.1. Local Control and Termination

First, we extend the notion of "instantiated enough" to binding-times as follows: a binding-time $b$ is instantiated enough w.r.t. a symbolic norm $|| \cdot ||$ if, for all terms $t$ approximated by the binding-time $b$, $t$ is instantiated enough w.r.t. $|| \cdot ||$.

Now, we introduce an appropriate definition of *unfold* that approximates the conditions of Theorem 1:

**Definition 16 (local termination).** Let $P$ be a program and let $(\succsim, \succ)$ be a reduction pair induced by a symbolic norm $|| \cdot ||$. Let $\mathcal{G}$ be the idempotent multigraphs from the size-change analysis and $\pi = p(b_1, \ldots, b_n)$ be a pattern. Function $unfold(\pi)$ returns true if every idempotent multigraph for $p/n$ in $\mathcal{G}$ contains at least one edge $i_p \xrightarrow{\succ} i_p$, $1 \leqslant i \leqslant n$, such that $b_i$ is instantiated enough w.r.t. $|| \cdot ||$.

For instance, given the idempotent size-change graphs of Figure 6 and the binding-time domain $\mathcal{D} = \{\mathsf{static}, \mathsf{dynamic}\}$, we have

$$unfold(mlist(\mathsf{dynamic}, \mathsf{dynamic}, \mathsf{static})) = true$$

since there is an edge $3_{mlist} \xrightarrow{\succ} 3_{mlist}$ and the binding-time $\mathsf{static}$ (representing only ground terms) is clearly instantiated enough w.r.t. any norm. In contrast, we have

$$unfold(mlist(\mathsf{static}, \mathsf{static}, \mathsf{dynamic})) = false$$

since the only edge labeled with "$\succ$" is $3_{mlist} \xrightarrow{\succ} 3_{mlist}$ and the binding-time $\mathsf{dynamic}$ is not instantiated enough w.r.t. any norm.

### 6.2.2. Global Control and Quasi-Termination

In order to ensure the global termination of the specialisation process, we should ensure that only a finite number of non-variant atoms are added to the set of (to be) partially evaluated atoms, i.e., that the sequence of atoms is *quasi-terminating*. This is a weaker requirement than termination.

Now, we introduce an appropriate definition of *gen* that approximates the conditions of Theorem 2:

**Definition 17 (global termination).** Let $P$ be a program and let $(\succsim, \succ)$ be a reduction pair where "$\succsim$" is a finitely partitioning well-founded quasi-order. Let $\mathcal{G}$ be the idempotent multigraphs computed by the size-change analysis. Given a pattern $\pi = p(b_1, \ldots, b_n)$, function $gen(\pi)$ returns $p(b'_1, \ldots, b'_n)$ where $b'_i = b_i$ if every idempotent multigraph for $p/n$ in $\mathcal{G}$ either

1. contains an edge $j_p \xrightarrow{\succ} j_p$ such that $b_j$ is instantiated enough w.r.t. $|| \cdot ||$

30

2. or contains an edge $j_p \xrightarrow{R} i_p$, $R \in \{\succ, \succsim\}$, for some $j \in \{1, \dots, n\}$, such that $b_i$ is static,

and $b'_i = $ dynamic otherwise.

For example, given the idempotent multigraphs of Figure 6, we have

$$gen(mlist(\text{static}, \text{static}, \text{static})) = mlist(\text{dynamic}, \text{static}, \text{static})$$

since there is no edge $j_{mlist} \xrightarrow{R} 1_{mlist}$, which means that the first argument must be generalized at the global level in order to guarantee the (global) termination of the partial evaluation process.

As mentioned before, requiring quasi-orders induced by finitely partitioning norms is often too restrictive (e.g., the list-length norm is not finitely partitioning). In the context of partial evaluation, however, symbolic norms need not be finitely partitioning as long as the *problematic* parts of the terms are generalized at the global level. For instance, we can safely use the symbolic list-length norm as long as the list elements are replaced by fresh variables in the global level. This idea can be formalized by means of the *most general generalization* operator:

**Definition 18 ($mgg$).** Let $|| \cdot ||$ be a symbolic norm. Given a term $t$, we denote by $mgg^{||\cdot||}(t)$ the most general generalization of $t$ such that $||t|| = ||mgg^{||\cdot||}(t)||$. We also let $mgg^{||\cdot||}(p(t_1, \dots, t_n)) = p(mgg^{||\cdot||}(t_1), \dots, mgg^{||\cdot||}(t_n))$.

For instance, given the term $t = [s(N), b]$, we have $mgg^{||\cdot||_{ll}}(t) = [X, Y]$ but $mgg^{||\cdot||_{ts}}(t) = [s(N), b]$.

Therefore, one can ensure the global termination of partial evaluation when using arbitrary symbolic norms in the size-change analysis as long as

- dynamic parts of arguments are replaced by fresh variables in the global level (this is already done by current offline partial evaluators) and

- every atom $A$ is replaced by $mgg^{||\cdot||}(A)$ in the global level, where $||\cdot||$ is the symbolic norm used in the size-change analysis.

## 7. Implementation and Empirical Evaluation

In this section we want to examine the scalability of our approach, and also examine how accurate it is compared to an online partial evaluation system.

Our new binding-time analysis is still being continuously extended and improved. The implementation was done in SICStus Prolog and provides a command-line interface which is available at `http://www.stups.uni-duesseldorf.de/w/Size-Change-BTA`. The BTA is by default polyvariant[21] (but can be forced to be monovariant) and uses a domain with the following values: static, list_nonvar (for lists of non-variable terms), list, nonvar (for non-variable terms), and dynamic. The user can also provide "hints" to the BTA (see below).

---

[21] A BTA is called *polyvariant* when an atom may give rise to multiple specialised version, and *monovariant* when only a single specialised version is produced.

The implemented size-change analysis uses a reduction pair induced by the symbolic term-size norm. The list-length norm can be enabled using a command-line option.[22] The implementation also supports programs with negation: the BTA ensures that no success substitutions are propagated for negated calls; apart from this a negated call is treated like an ordinary call.

We provide some preliminary experimental results below. The experiments were run on a MacBook Air with a 1.8 GHz i7 Processor and 4 GB of RAM. Our BTA was run using SICStus Prolog 4.2.3, and LOGEN and its generated specialisers were run using Ciao Prolog 1.14.2. We compared the results against the online specialiser ECCE [37], which was compiled using SICStus Prolog 3.12.8. This system also ensures local and global termination, using the online approach. Other systems to ensure termination for offline partial evaluation of Prolog were not available to us (or in the case of [13] cannot be run on current hardware; we return to this in Section 7.2.1 below).

### 7.1. Results in Fully Automatic Mode

We first present some experiments using our BTA in fully automatic mode. Figure 10 contains an overview of our empirical results, where all times are in seconds. A value of 0.00 means that the timing was below our measuring threshold. Note that the granularity of the SICStus Prolog run time statistics is 10 ms.

The columns *sca* and *bta* contain the time required for the size-change analysis and the ensuing binding-time analysis respectively. The *logen* column contains the time to specialise the original source program using the annotations provided by the BTA. For comparison, the *ecce* column contains the time to perform an online specialisation using ECCE. The quality of the specialised code is then analyzed in the "Runtime" columns: the runtime of the original code is presented in the *orig* column, while the runtime of the specialised code generated using our BTA together with LOGEN is shown in the logen column. The speedup obtained by LOGEN is computed in the next column. The last two columns contain the runtime and speedup for the specialised program generated by ECCE.

Note that we did not measure the time to start up either SICStus Prolog or Ciao Prolog for any of the columns.

### 7.1.1. DPPD

The first six benchmarks in Fig. 10 come from the original DPPD [32] library. As can be seen, the runtime of both the size-change analysis and the BTA was almost always below the measuring threshold. Good speedups were obtained for ssuply and regexp.r3. For the other benchmarks, the speedup is somewhat disappointing, especially when compared with ECCE. For liftsolve (an interpreter for the ground representation specialised for append as object program) we obtain a reasonable speedup, but the specialised program generated by ECCE is considerably faster. The most disappointing benchmark is probably imperative-power, where we even get a slight slow-down.

In summary, while the BTA is indeed very fast, the speed of the resulting specialised programs is reasonable, but still slightly disappointing. We will show later in Section 7.3 how we can substantially improve this picture by the selective use of hints.

---

[22] Maybe surprisingly, enabling the list-length norm turns out to have very little practical benefit at all (the reasons for which are beyond the scope of this paper).

| Benchmark | Specialisation | | | | Runtime | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | sca | bta | logen | ecce | orig | logen | $\times_{logen}$ | ecce | $\times_{ecce}$ |
| contains.kmp | 0.00 | 0.00 | 0.001 | 0.10 | 0.040 | 0.037 | 1.09 | 0.007 | 6.00 |
| imperative_power | 0.03 | 0.01 | 0.004 | 0.68 | 0.063 | 0.077 | 0.83 | 0.037 | 1.73 |
| liftsolve | 0.01 | 0.00 | 0.004 | 0.08 | 0.083 | 0.050 | 1.67 | 0.003 | 27.77 |
| match | 0.00 | 0.00 | 0.001 | 0.02 | 0.950 | 0.700 | 1.36 | 0.470 | 2.02 |
| regexp.r3 | 0.00 | 0.01 | 0.002 | 0.04 | 0.950 | 0.400 | 2.38 | 0.310 | 3.06 |
| ssuply | 0.00 | 0.01 | 0.000 | 0.02 | 0.057 | 0.003 | 17.00 | 0.003 | 17.00 |
| vanilla | 0.00 | 0.01 | 0.001 | 0.03 | 0.040 | 0.013 | 3.00 | 0.007 | 6.00 |
| ctl | 0.00 | 0.00 | 0.002 | 0.24 | 3.120 | 5.510 | 0.57 | 0.350 | 8.91 |
| lambdaint | 0.01 | 0.01 | 0.003 | 0.22 | 0.450 | 0.420 | 1.07 | 0.020 | 22.50 |
| dbaccess | 0.00 | 0.02 | 0.001 | 0.12 | 0.380 | 0.020 | 19.00 | 0.050 | 7.60 |
| javabc | 0.01 | 0.01 | 0.002 | 0.48 | 1.320 | 0.400 | 3.30 | 0.090 | 14.67 |
| picemul | 0.01 | 0.10 | 2.185 | 426.00 | - | - | - | - | - |
| goedel | 1.55 | 0.09 | - | - | - | - | - | - | - |

Figure 10: Empirical Results in Fully Automatic Mode

### 7.2. Interpreters

Next, we examine the performance of our BTA on a few interpreters (which are also available from [32]):

- vanilla is a variation of the vanilla metainterpreter specialised for double-append as object program from [34],

- lambdaint is an interpreter for a simple functional language from [34],

- ctl is the CTL model checker from [39], specialised for the formula ef(p(unsafe)) and a parametric Petri net (see also [36]),

- dbaccess is an interpreter for role-based access control from [2],

- javabc is a Java Bytecode interpreter from [25] with roughly 100 clauses.

Again, the runtime of the size-change analysis and the BTA is very fast and almost negligible. Apart from ctl and lambdaint we also get reasonable speedups. For javabc, we were able to reproduce the decompilation from Java bytecode to CLP from [25] using our BTA together with LOGEN.

The lambdaint interpreter contains some side-effects and non-declarative features. It can still be run through ECCE, but there is actually no guarantee that ECCE will preserve the side-effects and their order. Our BTA is again very fast, but unfortunately resulting in little speedup over the original. Still, the BTA from [13] could not cope with the program at all and we at least obtain a correct starting point.

For the ctl interpreter, the slow down is caused by non-leftmost unfolding. We could prohibit non-leftmost unfolding in the BTA.

Again, we will show in Section 7.3 how we can substantially improve this picture by the selective use of hints.

*7.2.1. Scalability: PIC Emulator and Gödel System*

To validate the scalability of our BTA we have also tried our new BTA on a larger example, the PIC processor emulator from [26]. It consists of 137 clauses and 855 lines of code. The purpose here was to specialise the PIC emulator for a particular PIC machine program, in order to run various static analyses on it.[23] The old BTA from [13] took 1 m 39 s (on a Linux server which should correspond roughly to 57 seconds on the MacBook Air used here).[24] Furthermore the generated annotation file is erroneous and could not be used for specialisation. With our new BTA a correct annotation is generated in less than half a second; the ensuing specialisation by LOGEN took 2.4 s. The generated code is very similar to the one obtained using a manually constructed annotation in Section 3 of [26] or in [35]. In fact, it is slightly more precise and with a simple hint (see Sect. 7.3), we were able to *reduce* specialisation so as to obtain the exact same code as [35] for the main interpreter loop. With ECCE it took over 7 minutes to construct a (very large) specialised program.

goedel is the source code of the Gödel system [27] consisting of 27,354 lines of Prolog.[25] Unfortunately, the code does not run on the current version of SICStus Prolog, hence we were only able to generate an annotation. We used the pattern `parse_language1(s,d,d,d,d,d)` as entry point for the BTA. 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 been able to infer 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. The size change analysis runs in less than two seconds, and the BTA is again extremely fast.

In summary, these examples clearly show that we have attained our goal of being able to successfully analyse medium-sized and even large examples with our BTA. Our size change analysis is two orders of magnitude faster than our initial implementation reported in [42] for goedel and twice as fast for picemul. Compared to the BTA from [13] using binary clauses rather than size-change analysis, the difference is even more striking: the BTA from [13] is in turn, e.g., 200 times slower than [42] for the picemul example; see [42]. We have also tried to use the latest version of Terminweb,[26] based upon [11]. 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 to use TermiLog,[27] but it timed out after 4 minutes (the maximum time that can be set in the online version).

---

[23] The emulator cannot be run as is using an existing Prolog system, as the built-in arithmetic operations have to be treated like constraints.

[24] We were unable to get [13] working on the MacBook Air and had to resort to using our webserver (with 26 MLIPS compared to the MacBook Air's 45 MLIPS).

[25] 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.

[26] `http://www.cs.bgu.ac.il/~mcodish/TerminWeb/`

[27] `http://www.cs.huji.ac.il/~naomil/termilog.php`

One cannot really draw clear-cut conclusions about the performance of Terminweb and TermiLog; but the experiments at least seem to confirm that the tackled programs are not trivial from the perspective of termination analysis.

### 7.3. Improving the Results with Hints

While the above experiments show that we have basically succeeded in obtaining a fast BTA, the specialisation results are still unsatisfactory for many examples. There are several causes for this:

1. One cause regards the particular binding-time domain used. For example, the lambdaint interpreter contains an environment which is a list of bindings from variables to values, such as `[x/2, y/3]`. During specialisation, the length of the list as well as the variable names are known, and the values are unknown. However, the closest binding-time value is `list_nv`, meaning that the BTA and the specialiser would throw away the variable names (i.e., the specialiser will work with `[A/B,C/D]` rather than with `[x/B,y/D]`). One solution is to improve our BTA to work with more sophisticated binding-time domains, possibly inferring the interesting abstract values using a type inference. Another solution is a so called "binding-time improvement" (bti) [29], whereby we rewrite the interpreter to work with two lists (i.e., `[x,y]` and `[2,3]`) rather than one. The first list (i.e., `[x,y]`) can now be classified as `static`, thereby keeping the desired information and allowing the specialiser to remove the overhead of variable lookups. We have performed this transformation for lambdaint and liftsolve. The results can be found in Fig. 11.

2. Another reason is an inherent limitation of using size-change analysis, namely the fact that the selection rule is ignored. This both gives our BTA its speed and scalability, but it also induces a precision loss. One way to solve this issue is for the user to be able to selectively insert "hints" into the source code, overriding the BTA. For the moment we support hints that force unfolding (resp. memoisation) of certain calls or predicates, as well as ways to prevent generalisation of arguments of memoised predicates.

The main idea of using hints is to have just a few of them, in the original source code in a format that a user can comprehend. Compared to editing the annotation file generated by our BTA, the advantage of hints is that the source file can still be freely edited; there is no need to synchronise annotations with edited code as in earlier work (such as the Pylogen interface [12]). Also, the propagation of binding-times is still fully performed by the BTA (and no binding-time errors can be introduced). Moreover, unfolding and generalisation decisions for predicates without hints are also fully taken care of by our algorithm. There is obviously the potential for a user to override the BTA in such a way that the specialisation process will no longer terminate. Note, however, that one can still use the watchdog mode [35] to pinpoint such errors.

Figure 11 contains a selection of benchmarks from Fig. 10, where we have applied hints and sometimes also binding-time improvements.

*contains.* For the contains example, the following hint (together with enabling the list-length norm) was sufficient to improve the speed more than four-fold:

```
'$MEMOANN'(con,2,[d,s]).
```

| Benchmark | Specialisation | | | | Runtime | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | sca | bta | logen | ecce | orig | logen | $\times_{logen}$ | ecce | $\times_{ecce}$ |
| contains.kmp | 0.00 | 0.00 | 0.001 | 0.10 | 0.040 | 0.037 | 1.09 | 0.007 | 6.00 |
| + hints + ll + scc | 0.00 | 0.01 | 0.001 | ” | ” | 0.007 | 6.00 | ” | ” |
| imperative_power | 0.03 | 0.01 | 0.004 | 0.68 | 0.063 | 0.077 | 0.83 | 0.037 | 1.73 |
| + hints + scc | 0.03 | 0.01 | 0.003 | ” | ” | 0.043 | 1.46 | ” | ” |
| liftsolve | 0.01 | 0.00 | 0.004 | 0.08 | 0.083 | 0.050 | 1.67 | 0.003 | 27.77 |
| + bti + scc | 0.01 | 0.00 | 0.001 | ” | ” | 0.007 | 12.50 | ” | ” |
| ctl | 0.00 | 0.00 | 0.002 | 0.24 | 3.120 | 5.510 | 0.57 | 0.350 | 8.91 |
| + bti | 0.02 | 0.00 | 0.001 | ” | ” | 0.830 | 3.76 | ” | ” |
| + bti + hints + scc | 0.01 | 0.01 | 0.003 | ” | ” | 0.320 | 9.75 | ” | ” |
| lambdaint | 0.01 | 0.01 | 0.003 | 0.22 | 0.450 | 0.420 | 1.07 | 0.020 | 22.50 |
| + hints + scc | 0.00 | 0.01 | 0.003 | ” | ” | 0.220 | 2.04 | ” | ” |
| + bti + hints + scc | 0.01 | 0.01 | 0.003 | ” | ” | 0.080 | 5.62 | ” | ” |

Figure 11: Empirical results with scc (the algorithm of Fig. 7), hints and binding-time improvements

This hint tells the BTA not to generalise the second argument to `con` away during memoisation. Indeed, the contains program includes the following clause:

```
con([H|T],P) :- new(H,P,NP), con(T,NP).
```

Here, the size-change analysis assumes that the second argument of `con` can grow (as it does not know whether `new(H,P,NP)` will be unfolded or not).

Note that the above hint actually ensures that the BTA will exactly use the pattern `con(d,s)`. This means that the BTA will always generalise the first argument to `con` and will throw an error if the second argument is not static. More formally, in the context of `'$MEMOANN'` annotations the following auxiliary function $gen'$ replaces the function $gen$ in Figure 9:

$$gen'(p(b_1,\ldots,b_n)) =_{\mathsf{def}} \quad \begin{aligned} &\textbf{if } \exists \text{ a fact } \texttt{'\$MEMOANN'}(p, n, [a_1, \ldots, a_n]) \textbf{ then} \\ &\quad \textbf{if } p(b_1,\ldots,b_n) \sqsubseteq p(a_1,\ldots,a_n) \textbf{ then} \\ &\quad\quad \textbf{return } p(a_1,\ldots,a_n) \\ &\quad \textbf{else throw error fi} \\ &\textbf{else} \\ &\quad \textbf{return } gen(p(b_1,\ldots,b_n)) \\ &\textbf{fi} \end{aligned}$$

*liftsolve.* For the liftsolve interpreter for ground representation, we have performed a binding-time improvement, after which the BTA obtains a near optimal result fully automatically. Indeed, as mentioned above, we have split a list of bindings containing terms of the form `sub(Var,T)` into two lists (see Figure 12). As you can see in Figure 11, this rewriting was sufficient to improve the performance of the specialised program by one order of magnitude; no hints were required in this case.

*lambdaint.* For the lambdaint interpreter, we again performed a binding-time improvement, separating a list into two lists. As can be seen in Figure 11, this already improves

**Original:**

```
mkng(var(N),X,[],[sub(N,X)]).
mkng(var(N),X,[sub(N,X)|T],[sub(N,X)|T]).
mkng(var(N),X,[sub(M,Y)|T],[sub(M,Y)|T1]) :-
    N \== M,mkng(var(N),X,T,T1).
mkng(term(F,Args),term(F,IArgs),InSub,OutSub) :-
    l_mkng(Args,IArgs,InSub,OutSub).
```

**Binding-Time Improved Version:**

```
mkng(var(N),X,[],[],[N],[X]).
mkng(var(N),X,[N|T1],[X|T2],[N|T1],[X|T2]).
mkng(var(N),X,[M|T1],[Y|T2],[M|TT1],[Y|TT2]) :-
    N \== M,mkng(var(N),X,T1,T2,TT1,TT2).
mkng(term(F,Args),term(F,IArgs),InSub1,InSub2,OutSub1,OutSub2) :-
    l_mkng(Args,IArgs,InSub1,InSub2,OutSub1,OutSub2).
```

Figure 12: Binding-time improvement for lambdaint

the performance somewhat. By further adding two hints, however, we get specialised programs which correspond almost exactly to the results obtained in [34], when hand-crafting the annotations with custom binding-times. The hints for lambdaint are the following ones:

```
'$MEMOANN'(eval,4,[s,s,list,d]).
'$MEMOANN'(l_eval,4,[s,s,list,d]).
```

We tell the BTA, that we consider calls `eval(s,s,list,d)` and `l_eval(s,s,list,d)` to be quasi-terminating. As described earlier, the BTA also checks that at every program point we obtain calls that are at least as precise.

As we can see, by using hints and some binding-time improvements, we are now able to specialise larger interpreters predictably and effectively, obtaining very good speedups.

## 8. Related Work

The first attempts at ensuring (local) termination for offline partial evaluation were developed in [24] in the context of functional programming; the core ideas later evolved into the size-change principle. Regarding offline partial evaluation of logic programs, the closest previous work is that of Craig et al [13], which itself is a further development of Section 6 of [36]. A very similar approach is also described in [50]. [13] develops a fully automatic BTA for logic programs. As in our case, the output of this BTA is an annotated program that can be used as input to the offline partial evaluator LOGEN [36]. The BTA of [13], however, suffered from some drawbacks: it did not ensure global termination and it was computationally very expensive (due to the interleaving between the left-termination analysis and the propagation of binding-times, as witnessed by the experimental results shown in the previous section). Our BTA is also fully automatic but guarantees both local and global termination. Furthermore, it is much faster, so that it scales up well to medium and even large Prolog programs.

37

To the best of our knowledge, the first binding-time analysis for logic programming was [8]. The approach of [8] obtains the required annotations by analysing the behaviour of an *online* specialiser on the subject program. Unfortunately, the approach was overly conservative. Indeed, [8] decides whether or not to unfold a call based on the original program without taking the current annotations into account. This means that a call can either be completely unfolded or not at all. Also, the approach was never fully implemented and integrated into a partial evaluator. The papers [49, 48, 51] describe various BTAs for the logic programming language Mercury, even addressing issues such as modularity and higher-order predicates. An essential part of these approaches is the classification of unifications into tests, assignments, constructions and deconstructions. Hence, these works cannot be easily ported to a Prolog setting, although some ideas can be found in [51].

Regarding the strong termination analysis, the closest approaches to our work are the following. First, Bezem [6] introduced the notion of strong termination by defining a sound and complete characterisation (the *recurrent* programs). We extend Bezem's results by introducing a *sufficient* condition for strong termination (actually, size-change termination, as defined in Definition 10 is a sufficient condition for Bezem's strong termination which only considers ground atoms).

As for the size-change analysis, it was originally introduced in [31] in the context of functional programming and later improved in a number of ways (see a detailed account in [3]). Size-change graphs are also closely related to the *weighted rule graphs* for logic programs of [43]. 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 [11]: 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 [11] or for a local computation rule [19]. 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 [14], where the authors introduce the notion of *quasi-acceptability*, a sufficient and necessary condition for quasi-termination. This work has been extended in [52]. Another related approach is [38], 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 [23], 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 norms, etc, have no counterpart in [23]). Besides

the paradigm shift, there are some other differences between our approach and that of Glenstrup and Jones. In particular, their notion of quasi-termination is based on the concept of "bounded static variation" (extended to so called "bounded anchoring"), i.e., identifying parameters that can only take a finite number of different values and, thus, make the computation stop. In contrast, our approach relies on a natural extension of the notion of size-change termination using a (well-founded and finitely partitioning) quasi-order. Another difference is the underlying partial evaluation algorithm considered. In [23], a simpler one-step unfolding strategy is mostly considered (nevertheless, the paper also shows how this limitation can be overcome) and, hence, only global termination must be ensured. In our approach, we deal both with local and global termination since our local level is not trivial.

Finally, let us briefly compare to our previous work on this topic. As mentioned in the introduction, this paper includes and extends previous contributions originally introduced in [53, 42, 40, 41]. To be more precise, the following contributions are original from this paper:

- We formally prove the correctness of both the termination and quasi-termination analyses based on the size-change principle. [53] did not include correctness results and, moreover, the requirements for quasi-termination for non-ground terms were not feasible in practice.

- We provide correctness results for the algorithm that performs size-change analysis. In particular, we extend the developments in [40, 41] where no technical results were proved.

- The definition of a refined (w.r.t. that presented in [42]) BTA for logic programs that uses the results of the termination and quasi-termination analyses. In particular, the analysis is now independent of the considered binding-time domain. Moreover, an implementation of the BTA is now publicly available through a web interface.

To the best of our knowledge, our work presents the first fully automatic BTA for logic programs that guarantees both local and global termination by using *strong* termination and quasi-termination analyses based on the construction of size-change graphs.


## 9. Discussion and Future Work

We have presented a very fast BTA, which is able to cope with larger programs than previous approaches and, for the first time, ensures both local and global termination. For this purpose, we have introduced an 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 two seconds. In the experimental evaluation we have shown that our BTA can now deal with realistic interpreters. Together with the selective use of hints [42], we have obtained both a scalable and an effective partial evaluation procedure.

In conclusion, our BTA is well suited to be applied to larger programs. The accuracy of the annotations is not yet optimal, but in conjunction with hints we have obtained a fast BTA with very good specialisation results. Nevertheless, there is still room for improvement.

As a future work, we mainly consider different possibilities to make the analysis more accurate. For this purpose, we would like to infer which arguments are of *bounded static variation* [23], i.e., arguments whose size may increase from one call to another, but that can only take a finite number of different values (consider, e.g., the value of a program counter) since these arguments need not be marked as dynamic. Another way to improve the accuracy of the BTA consists in also running a standard left-termination analysis (such as, e.g., the termination analysis based on the abstract binary unfoldings [11]), so that left-terminating atoms are marked with a new annotation call (besides unfold and memo, which keep the same meaning). Basically, while atoms annotated with unfold allow us to perform an unfolding step and then the annotations of the derived goal must be followed, atoms annotated with call could be *fully* unfolded. In principle, this extension may allow us to avoid some of the loss of accuracy due to considering a *strong* termination analysis during the BTA.

As an alternative approach, we would like to explore the definition of a BTA using SAT solving techniques. In particular, one could define a SAT problem involving both the constraints from the left-termination analysis (e.g., as in [4]) with the constraints associated to the propagation of binding times. This approach will be more accurate than the one presented in this paper since a fixed selection rule would be considered in the termination analysis. Moreover, an advantage of such an approach is that one can perform the analysis without fixing *a priori* a symbolic norm but rather leaving the SAT solver to find the right norm for proving termination (as it is done, e.g., in the termination prover AProVE [21]). Finally, we also plan to investigate the extension of this analysis to deal with quasi-termination and, more importantly, whether non-quasi-terminating arguments could be identifed (as we do in Definition 17 by inspecting the idempotent multigraphs produced by the size-change termination analysis), since this is essential to produce static/dynamic annotations that guarantee global termination.

One major challenge is to make our technique in particular, and partial evaluation in general, available to programmers for real-life source code. For this, we need to safely deal with all aspects of Prolog (higher-order predicates such as call or maplist, the cut, assert and retract, input/output), with minimal user-annotations and also enabling the source code to be changed with minimal amount of effort. Our new technique goes a long way towards minimising the number of required user annotations, but further research and development work is required to build a tool that can be used reliably on arbitrary source code with predictable outcome.

## References

[1] E. Albert, G. Puebla, J. Gallagher, Non-Leftmost Unfolding in Partial Deduction of Logic Programs with Impure Predicates, in: Proc. of LOPSTR'05, Springer LNCS 3901, 2006, pp. 115–132.

[2] S. Barker, M. Leuschel, M. Varea, Efficient and flexible access control via Jones-optimal logic program specialisation, Higher-Order and Symbolic Computation 21 (1-2) (2008) 5–35.

[3] A. Ben-Amram, Size-change termination and constraint transition systems (2012).
URL http://www2.mta.ac.il/~amirben/sct.html

[4] A. Ben-Amram, M. Codish, A SAT-Based Approach to Size Change Termination with Global Ranking Functions, in: C. Ramakrishnan, J. Rehof (eds.), Proc. of the 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07), Springer LNCS 5028, 2008, pp. 46–55.

[5] A. M. Ben-Amram, C. S. Lee, Program termination analysis in polynomial time, ACM Trans. Program. Lang. Syst. 29 (1).

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

[7] A. Bossi, N. Cocco, M. Fabris, Proving Termination of Logic Programs by Exploiting Term Properties, in: S. Abramsky, T. Maibaum (eds.), Proc. of TAPSOFT'91, Springer LNCS 494, 1991, pp. 153–180.

[8] M. Bruynooghe, M. Leuschel, K. Sagonas, A polyvariant binding-time analysis for off-line partial deduction, in: C. Hankin (ed.), Proceedings of the European Symposium on Programming (ESOP'98), LNCS 1381, Springer-Verlag, 1998, pp. 27–41.

[9] M. Codish, V. Lagoon, P. Schachte, P. Stuckey, Size-Change Termination Analysis in $k$-Bits, in: Proc. of the 15th European Symposium on Programming (ESOP 2006), Springer LNCS 3924, 2006, pp. 230–245.

[10] M. Codish, V. Lagoon, P. Stuckey, Testing for Termination with Monotonicity Constraints, in: Proc. of the 21st Int'l Conf. on Logic Programming (ICLP'05), Springer LNCS 3668, 2005, pp. 326–340.

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

[12] S. Craig, Practicable Prolog Specialisation, Ph.D. thesis, University of Southampton, U.K. (June 2005).

[13] S.-J. Craig, J. Gallagher, M. Leuschel, K. Henriksen, Fully Automatic Binding Time Analysis for Prolog, in: Proc. of the Int'l Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04), Springer LNCS 3573, 2005, pp. 53–68.

[14] S. Decorte, D. De Schreye, M. Leuschel, B. Martens, K. Sagonas, Termination Analysis for Tabled Logic Programming, in: Proc. of LOPSTR'97, Springer LNCS 1463, 1998, pp. 111–127.

[15] N. Dershowitz, Termination of rewriting, Journal of Symbolic Computation 3 (1&2) (1987) 69–115.

[16] M. Falaschi, G. Levi, M. Martelli, C. Palamidessi, Declarative Modeling of the Operational Behavior of Logic Languages, Theoretical Computer Science 69 (3) (1989) 289–318.

[17] S. Fogarty, M. Y. Vardi, Efficient Büchi universality checking, in: J. Esparza, R. Majumdar (eds.), Proc. of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010), vol. 6015 of Lecture Notes in Computer Science, Springer, 2010, pp. 205–220.

[18] Y. Futamura, Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler, Higher-Order and Symbolic Computation 12 (4) (1999) 381–391, reprint of article in *Systems, Computers, Controls* 1971.

[19] M. Gabbrielli, R. Giacobazzi, Goal Independency and Call Patterns in the Analysis of Logic Programs, in: Proc. of the 1994 ACM Symposium on Applied Computing, ACM Press, 1994, pp. 394–399.

[20] J. Gallagher, Tutorial on Specialisation of Logic Programs, in: Proc. of the ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93), ACM, New York, 1993, pp. 88–98.

[21] J. Giesl, P. Schneider-Kamp, R. Thiemann, AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework, in: Proc. of Int'l Joint Conf. on Automated Reasoning (IJCAR'06), Springer LNCS 4130, 2006, pp. 281–286.

[22] J. Giesl, R. Thiemann, P. Schneider-Kamp, S. Falke, Mechanizing and Improving Dependency Pairs, Journal of Automated Reasoning 37 (3) (2006) 155–203.

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

[24] A. J. Glenstrup, N. D. Jones, BTA algorithms to ensure termination of off-line partial evaluation, in: Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference, LNCS 1181, Springer-Verlag, 1996, pp. 273–284.

[25] M. Gómez-Zamalloa, E. Albert, G. Puebla, Improving the Decompilation of Java Bytecode to Prolog by Partial Evaluation, Electr. Notes Theor. Comput. Sci. 190 (1) (2007) 85–101.

[26] K. S. Henriksen, J. P. Gallagher, Abstract interpretation of PIC programs through logic programming, in: SCAM, IEEE Computer Society, 2006, pp. 184–196.

[27] P. Hill, J. W. Lloyd, The Gödel Programming Language, MIT Press, 1994.

[28] C. Holst, Finiteness Analysis, in: Proc. of Functional Programming Languages and Computer Architecture, Springer LNCS 523, 1991, pp. 473–495.

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

[30] C. Lee, Finiteness Analysis in Polynomial Time, in: Proc. of the 9th Int'l Symposium on Static Analysis (SAS'02), Springer LNCS 2477, 2002, pp. 493–508.

[31] C. Lee, N. Jones, A. Ben-Amram, The Size-Change Principle for Program Termination, SIGPLAN Notices (Proc. of POPL'01) 28 (2001) 81–92.

[32] M. Leuschel, The DPPD library of benchmarks, obtainable via http://www.stups.uni-duesseldorf.de/systems/dppd.html (1996-2012).

[33] M. Leuschel, M. Bruynooghe, Logic Program Specialisation through Partial Deduction: Control Issues, Theory and Practice of Logic Programming 2 (4-5) (2002) 461–515.

[34] M. Leuschel, S.-J. Craig, M. Bruynooghe, W. Vanhoof, Specialising Interpreters Using Offline Partial Deduction, in: Program Development in Computational Logic, Springer LNCS 3049, 2004, pp. 340–375.

[35] M. Leuschel, S.-J. Craig, D. Elphick, Supervising offline partial evaluation of logic programs using online techniques, in: G. Puebla (ed.), LOPSTR, vol. 4407 of Lecture Notes in Computer Science, Springer-Verlag, 2006, pp. 43–59.

[36] M. Leuschel, J. Jørgensen, W. Vanhoof, M. Bruynooghe, Offline Specialisation in Prolog using a Hand-Written Compiler Generator, Theory and Practice of Logic Programming 4 (1-2) (2004) 139–191.

[37] M. Leuschel, B. Martens, D. De Schreye, Controlling generalisation and polyvariance in partial deduction of normal logic programs, ACM Transactions on Programming Languages and Systems 20 (1) (1998) 208–258.

[38] M. Leuschel, B. Martens, K. 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), Springer LNCS 1463, 1998, pp. 189–205.

[39] M. Leuschel, T. Massart, Infinite state model checking by abstract interpretation and program specialisation, in: A. Bossi (ed.), Proceedings LOPSTR'99, LNCS 1817, Venice, Italy, 2000, pp. 63–82.

[40] M. Leuschel, S. Tamarit, G. Vidal, Improving Size-Change Analysis in Offline Partial Evaluation, in: P. Arenas, D. Zanardini (eds.), Proc. of the 18th Workshop on Logic-based methods in Programming Environments, 2008.

[41] M. Leuschel, S. Tamarit, G. Vidal, Fast and Accurate Size-Change Strong Termination Analysis with an Application to Partial Evaluation, in: S. Escobar (ed.), Proc. of the 18th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP'09), Springer LNCS 5979, 2009, pp. 111–127.

[42] M. Leuschel, G. Vidal, Fast Offline Partial Evaluation of Large Logic Programs, in: Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR'08), Springer LNCS 5438, 2009, pp. 119–134.

[43] N. Lindenstrauss, Y. Sagiv, Automatic Termination Analysis of Logic Programs, in: Proc. of Int'l Conf. on Logic Programming (ICLP'97), The MIT Press, 1997, pp. 63–77.

[44] J. Lloyd, Foundations of Logic Programming, Springer-Verlag, Berlin, 1987, second edition.

[45] J. Lloyd, J. Shepherdson, Partial Evaluation in Logic Programming, Journal of Logic Programming 11 (1991) 217–242.

[46] F. Ramsey, On a problem of formal logic, in: Proc. of the London Mathematical Society, vol. 30, 1930, pp. 264–286.

[47] R. Thiemann, J. Giesl, The Size-Change Principle and Dependency Pairs for Termination of Term Rewriting, Applicable Algebra in Engineering, Communication and Computing 16 (4) (2005) 229–270.

[48] W. Vanhoof, Binding-Time Analysis by Constraint Solving: a modular and higher-order approach for Mercury, in: M. Parigot, A. Voronkov (eds.), Proceedings of LPAR'2000, LNAI 1955, Springer-Verlag, 2000, pp. 399–416.

[49] W. Vanhoof, M. Bruynooghe, Binding-time analysis for Mercury, in: D. De Schreye (ed.), Pro-

ceedings of the International Conference on Logic Programming ICLP'99, MIT Press, 1999, pp. 500–514.

[50] W. Vanhoof, M. Bruynooghe, Binding-time annotations without binding-time analysis, in: R. Nieuwenhuis, A. Voronkov (eds.), Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LNCS 2250, Springer-Verlag, 2001, pp. 707–722.

[51] W. Vanhoof, M. Bruynooghe, M. Leuschel, Binding-time analysis for Mercury, in: M. Bruynooghe, K.-K. Lau (eds.), Program Development in Computational Logic, LNCS 3049, Springer-Verlag, 2004, pp. 189–232.

[52] S. Verbaeten, K. Sagonas, D. De Schreye, Termination Proofs for Logic Programs with Tabling, ACM Transactions on Computational Logic 2 (1) (2001) 57–92.

[53] G. Vidal, Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation, in: Proc. of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation (PEPM'07), ACM Press, 2007, pp. 51–60.