

Termination Analysis of Tabled Logic Programs Using Mode and Type Information

Sofie Verbaeten Danny De Schreye

Report CW 277, January 1999



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Termination Analysis of Tabled Logic Programs Using Mode and Type Information

Sofie Verbaeten *Danny De Schreye*

Report CW 277, January 1999

Department of Computer Science, K.U.Leuven

Abstract

Tabled logic programming is receiving increasing attention in our community. It avoids many of the shortcomings of SLD(NF) execution and provides a more flexible and often extremely efficient execution mechanism for logic programs. In particular, tabled execution of logic programs terminates more often than execution based on SLD-resolution. So, if a program can be proven to terminate under SLD-resolution, then the program will trivially also terminate under SLG-resolution, the resolution principle of tabulation. But, since there are SLG-terminating programs which are not SLD-terminating, better proof techniques can be found. One of the few approaches studying termination of tabled logic programs was developed by Decorte et al. They present necessary and sufficient conditions for two notions of termination under LG-resolution, i.e. SLG-resolution with left-to-right selection rule: quasi-termination and (the stronger notion of) LG-termination. Starting from these necessary and sufficient conditions, we introduce sufficient conditions for quasi-termination and LG-termination which reason fully at the clause level and are easy to automatize. To this end, we use mode and type information: we consider simply moded, well-moded/well-typed programs and queries. We point out how our termination conditions can be automatized, by extending the recently developed constraint-based automatic termination analysis for SLD-resolution by Decorte and De Schreye.

Keywords : Logic Programming, Termination Analysis.

CR Subject Classification : I.2.3, I.2.2, F.3.1.

1 INTRODUCTION

Tabled logic programming [6, 10, 25, 26] is receiving increasing attention in our community. It avoids many of the shortcomings of SLD(NF) execution and provides a more flexible and often extremely efficient execution mechanism for logic programs. In particular, tabled execution of logic programs terminates more often than execution based on SLD-resolution. So, if a program can be proven to terminate under SLD-resolution (by one of the existing automated techniques surveyed in [11]), then the program will trivially also terminate under SLG-resolution, the resolution principle of tabulation [10]. But, since there are SLG-terminating programs which are not SLD-terminating, better proof techniques can be found. One of the few works studying termination of tabled logic programs is [14], in which necessary and sufficient conditions are given for two notions of termination under LG-resolution, i.e. SLG-resolution with left-to-right selection rule; namely quasi-termination and (the stronger notion of) LG-termination. This paper is based on [14]. Starting from the necessary and sufficient conditions of [14], we present sufficient conditions for quasi-termination and LG-termination which reason fully at the clause level. These conditions form the basis for extending the automatic termination analysis for LD-resolution (SLD-resolution with left-to-right selection rule) of [13], in order to automatically prove quasi-termination and LG-termination.

The ideas underlying *tabling* are very simple. Essentially, under tabled execution mechanism, answers for selected atoms are stored in a table. When a variant of such an atom is recursively called, the selected atom is not resolved against program clauses, instead, all corresponding answers computed so far are looked up in the table and the corresponding answer substitutions are applied to the atom. This process is repeated for all subsequent computed answer substitutions that correspond to the atom.

We study *universal* termination of definite tabled logic programs executed under SLG using a fixed left-to-right selection rule (we drop the “S” in SLD and SLG whenever we refer to the left-to-right selection rule). Note that, whenever SLD-computation from an initial goal leads to infinitely many different, non-variant calls, the SLG-computation from this initial goal will also be infinite. This leads to the first basic notion of termination under tabled execution mechanism: *quasi-termination*. A program P is said to quasi-terminate w.r.t. a set of queries S iff for all $A \in S$ there are only finitely many different calls in the LD-computation of A . Even when tabling, quasi-termination only partially corresponds to our intuitive notion of a “terminating computation”. This is because an atom can have infinitely many computed answers (note that this does not have to lead to infinitely many new calls). Therefore, the stronger notion of *LG-termination* is introduced: P LG-terminates w.r.t. S iff P quasi-terminates w.r.t. S and for all $A \in S$ the set of all computed answers for calls in the LD-computation of A is finite.

In [14], these two kinds of termination under tabled execution mechanism are introduced and necessary and sufficient conditions for both notions are given. A key notion in the termination conditions of [14] is that of *finitely partitioning level mapping*, i.e. a level mapping (a mapping from the set of atoms to \mathbb{N}) such that no infinite set of atoms is mapped to the same natural number. The level mappings which are used in the termination conditions of [14] are required to be finitely partitioning on the call set of a set of queries S w.r.t. a program P , this is the set of all atoms which occur as selected atom in an LD-derivation starting from a query of S . As opposed to termination conditions for (S)LD-resolution (see for instance [12]), where a strict decrease on the calls w.r.t. a level mapping is required, in the termination condition for LG-resolution of

[14], a *non-strict decrease* on the calls w.r.t. a level mapping is required, *if* the level mapping is finitely partitioning on the call set. In most automatic approaches to termination analysis, a level mapping $|\cdot|$ is defined as a positive linear combinations of *norms*:

$$|p(t_1, \dots, t_n)| = \sum_{i=1}^n l_i \|t_i\|$$

with $l_1, \dots, l_n \in \mathbb{N}$ depending on p/n , and where $\|\cdot\|$ is a norm, i.e. a mapping from the set of terms to \mathbb{N} , defined 0 on constants and variables, and recursively on terms of the form $f(s_1, \dots, s_m)$ as a positive linear combination of the norms of the arguments s_1, \dots, s_m plus a constant c_f , depending on the functor f/m :

$$\|f(s_1, \dots, s_m)\| = c_f + \sum_{i=1}^m k_i \|s_i\|$$

with $c_f, k_1, \dots, k_m \in \mathbb{N}$ depending on f/m . Hence, in order for such a level mapping to be finitely partitioning on the call set, the level mapping must take into account *enough* argument positions of predicates and functors.

In order to *automatize* termination proofs, it is important to have a termination condition which reasons fully at the clause level (and not on “calls” as in the termination condition of [14]). In most automatic approaches, like e.g. in [13], where a new strategy for automatically proving LD-termination is developed, this is obtained by requiring that the level mapping is *rigid* on the call set. A level mapping is rigid on the call set if the value of an atom in the call set is invariant under substitutions. If a level mapping is rigid on the call set, the atoms in the call set can be considered as ground w.r.t. the level mapping. In this way, the problem of backpropagation of bindings in the calls is dealt with, and the termination condition can be stated at the clause level. As was shown in [7], in order for a level mapping to be rigid on the call set, the level mapping may *not* take into account *too many* argument positions of predicates and functors (more precisely, the positions of variables in the call set may not be taken into account).

In this paper, we present easy to automatize, sufficient conditions for proving termination of definite logic programs under tabled execution mechanism. As follows from the above informal presentation of finitely partitioning and rigid level mappings, we first show that a straightforward combination of the automatic approach towards LD-termination of [13], which relies on rigidity, with the approach of [14] for tabled logic programs, which relies on finitely partitioning level mappings, offers some difficulties. But, for *simply moded well-moded* programs and queries, we will be able to combine these two requirements and we present easy to automatize, sufficient conditions for termination under LG-resolution in this case. We will also present another approach; instead of trying to combine the rigidity requirement on the level mapping with the requirement to be finitely partitioning, we will abandon the rigidity requirement and search for syntactical conditions on the program and set of queries which allow us to reason fully at the clause level in the termination condition. We present a solution for *simply moded well-typed* programs and queries, which generalizes our result for simply moded well-moded programs and queries.

The rest of the paper is organised as follows. In the following section 2 of preliminaries, we first recall the notion of SLG-resolution [10] for definite programs. Next, we introduce modes and define the notions of well-modedness and simply modedness. Also types and the notions of well-typedness and generic expressions for types are discussed. We end section 2 with some definitions and properties of norms and level mappings. In the following section 3, the

notion of quasi-termination and the necessary and sufficient condition for quasi-termination of [14], namely quasi-acceptability, is defined. We present easy to automatize, sufficient conditions for quasi-termination of simply moded well-moded programs and queries in subsection 3.1, and of simply moded well-typed programs and queries in subsection 3.2. In subsection 3.3, we point out how the constraint-based approach of [13] (where a new strategy for automatically proving LD-termination of programs w.r.t. sets of queries is developed) needs to be changed in order to automatically prove quasi-termination of the kind of programs and queries considered in the subsections 3.1 and 3.2. In section 4, the stronger notion of LG-termination is studied. We present an optimisation of the (necessary and sufficient) condition of [14] for definite programs, and state easy to automatize conditions for simply moded well-moded programs and queries in subsection 4.1 and simply moded well-typed programs and queries in subsection 4.2. Again, in subsection 4.3, we point out how the approach of [13] needs to be changed in order to automatically prove LG-termination of the kinds of programs and queries considered in the subsections 4.1 and 4.2. We end with a conclusion and discussion on related works in section 5.

2 PRELIMINARIES

We assume familiarity with the basic concepts of logic programming [20, 3]. Throughout the paper, P will denote a definite logic program. The *extended Herbrand Universe*, U_P^E , and the *extended Herbrand Base*, B_P^E , associated with a program P , were introduced in [16]. They are defined as follows. Let $Term_P$ and $Atom_P$ denote the set of respectively all terms and atoms that can be constructed from the alphabet underlying P . The variant relation, denoted \approx , defines an equivalence. U_P^E and B_P^E are respectively the quotient sets $Term_P/\approx$ and $Atom_P/\approx$. For any term t (or atom A), we denote its class in U_P^E (B_P^E) as \tilde{t} (\tilde{A}). However, when no confusion is possible, we omit the tildes. By $Pred_P$ and Fun_P , we denote the sets of predicate and function symbols (including the constant symbols, which are considered as nullary function symbols) of P respectively. We assume that these sets are finite.

Let P be a program and p, q two predicate symbols of P . We say that p refers to q in P if there is a clause in P with p in the head and q occurring in the body. We say that p depends on q in P , and write $p \sqsupseteq q$, if (p, q) is in the reflexive, transitive closure of the relation refers to. Note that, by definition, each predicate depends on itself. We write $p \simeq q$ iff $p \sqsupseteq q$, $q \sqsupseteq p$ (p and q are mutually recursive or $p = q$). If $A = p(t_1, \dots, t_n)$, then we denote by $Rel(A)$ the predicate symbol p of A ; i.e. $Rel(A) = p$.

In analogy with [5], we will refer to SLD-derivations (see [20]) following the left-to-right selection rule as LD-derivations. Other concepts will adopt this naming accordingly.

Definition 2.1 (call set) *Given $S \subseteq B_P^E$, by $Call(P, S)$ we denote the subset of B_P^E such that $B \in Call(P, S)$ whenever an element of B is a selected literal in an LD-derivation for some $P \cup \{\leftarrow A\}$, with $\tilde{A} \in S$.*

Throughout the paper we assume that in any derivation of a query w.r.t. a program, representatives of equivalence classes are systematically provided with fresh variables, to avoid the necessity of renaming apart.

In sections 3 and 4, we study termination of SLG-resolution (see [10]), using a fixed left-to-right selection rule, for a given set of atomic (top level) queries with atoms in $S \subseteq B_P^E$. We will abbreviate SLG-resolution under left-to-right selection rule by LG-resolution (which for definite programs is similar to OLDT-resolution [25, 18], modulo the fact that OLDT specifies a more fixed control

strategy and uses subsumption checking and term-depth abstraction instead of variant checking). In the next subsection, we formalise these notions.

2.1 SLG-Resolution for Definite Programs

This subsection is based on a subsection in [14]. We present a non-constructive definition of SLG-resolution that is sufficient for our purposes, and refer to [6, 10, 25, 26] for more constructive formulations of (variants) of tabled resolution.

Definition 2.2 (pseudo SLG-tree, pseudo LG-tree) *Let P be a definite program, \mathcal{R} a selection rule and A an atom. A pseudo SLG-tree for $P \cup \{\leftarrow A\}$ under \mathcal{R} is a tree τ_A such that:*

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

If \mathcal{R} is the leftmost selection rule, τ_A is called a pseudo LG-tree for $P \cup \{\leftarrow A\}$.

We say that a pseudo SLG-tree τ_A for $P \cup \{\leftarrow A\}$ is smaller than another pseudo SLG-tree τ'_A for $P \cup \{\leftarrow A\}$ iff τ'_A can be obtained from τ_A by attaching new sub-branches to nodes in τ_A .

A (computed) answer clause of a pseudo SLG-tree τ_A for $P \cup \{\leftarrow A\}$ is a clause of the form $A\theta \leftarrow$ where θ is the composition of the answer substitutions found on a branch of τ_A whose leaf is labeled by the empty goal.

Intuitively, a pseudo SLG-tree (in an SLG-forest, see Definition 2.3 below) represents the tabled computation of all answers for a given subgoal labeling the root node of the tree. The trees in the above definition are called *pseudo* SLG-trees because there is no condition yet on which clauses $B\theta \leftarrow$ exactly are to be used for resolution in point (5). These clauses represent the answers found (possibly in another tree of the forest) for the selected atom. This interaction between the trees in an SLG-forest is captured in the following definition.

Definition 2.3 (SLG-forest, LG-forest) *Let P be a definite program, \mathcal{R} be a selection rule and S be a (possibly infinite) set of atoms such that no two different atoms in S are variants of each other. \mathcal{F} is an SLG-forest for P and S under \mathcal{R} iff \mathcal{F} is a set of minimal pseudo SLG-trees $\{\tau_A \mid A \in S\}$ where*

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

Let $Q = \leftarrow A$ be an atomic query. An SLG-forest for $P \cup \{Q\}$ under \mathcal{R} is an SLG-forest for a minimal set S with $A \in S$.

An LG-forest is an SLG-forest containing only pseudo LG-trees.

Point (2) of Definition 2.3, together with the imposed minimality of trees in a forest, now uniquely determines these trees. So we can drop the designation “pseudo” and refer to (S)LG-trees in an (S)LG-forest.

Example 2.1 Let P be the following program defining the natural numbers:

$$\begin{cases} \text{nat}(0) & \leftarrow \\ \text{nat}(s(X)) & \leftarrow \text{nat}(X) \end{cases}$$

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

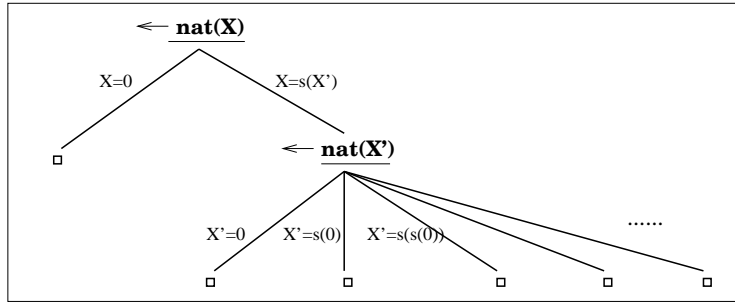


Figure 1: SLG-forest for $P \cup \{\leftarrow \text{nat}(X)\}$.

Example 2.2 Let P^a be the following program (the meaning of this program, called the a -transform (Definition 4.2) of the program P of Example 2.1, will become clear in section 4, where the notion of LG-termination is introduced):

$$\begin{cases} \text{nat}(0) & \leftarrow \\ \text{nat}(s(X)) & \leftarrow \text{nat}(X), \text{nat}^a(X) \\ \text{nat}^a(X) & \leftarrow \end{cases}$$

Let $S = \{\text{nat}(X)\}$. Then the LG-forest for P^a and S contains an infinite number of LG-trees, see Figure 2.

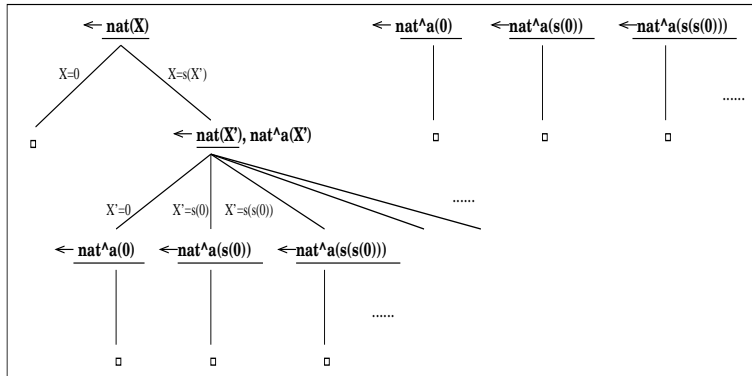


Figure 2: The (infinite) LG-forest for Example 2.2.

Note that we can use the notions of LD-derivation and LD-computation (as they appear in the definition of the call set $Call(P, S)$) even in the context of SLG-resolution, as the set of call patterns and the set of computed answer substitutions are not influenced by tabling; see e.g. [18, Theorem 2.1].

2.2 Modes, Well-Modedness and Simply Modedness

Definition 2.4 (mode) Let p be an n -ary predicate symbol. A mode for p is a function $m_p : \{1, \dots, n\} \rightarrow \{In, Out\}$. If $m_p(i) = In$ (resp. Out), then we say that i is an input (resp. output) position of p (w.r.t. m_p).

We assume that each predicate symbol has a unique mode. Multiple modes can be obtained by simply renaming the predicates. In examples, we will write the mode m_p for the predicate p as follows: $p(m_p(1), \dots, m_p(n))$.

For an atom A , we denote by $Var(A)$ the set of variables occurring in A . By $InVar(A)$ we denote the set of variables occurring in input positions of A and by $OutVar(A)$ the set of variables occurring in output positions of A . Similar notation is used for sequences of atoms. For a term t , we denote by $Var(t)$ the set of variables occurring in t . Similar notation is used for sequences of terms.

The following concept of well-modedness, is essentially due to [15], we give the formulation of [22]. For simplifying the notation, when writing an atom as $p(\mathbf{u}, \mathbf{v})$, we assume that \mathbf{u} is the sequence of terms filling in the input positions of p and \mathbf{v} is the sequence of terms filling in the output positions of p .

Definition 2.5 (well-modedness)

A clause $p_0(\mathbf{t}_0, \mathbf{s}_{\mathbf{n}+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is called well-moded if for $i \in [1, n+1]$

$$Var(\mathbf{s}_i) \subseteq \bigcup_{j=0}^{i-1} Var(\mathbf{t}_j).$$

A program is called well-moded iff every clause of it is well-moded.

A query $\leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is well-moded iff the clause dummy $\leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is well-moded, where dummy is any nullary predicate.

For instance the clause $p(X, f(Y)) \leftarrow q(X, g(Z)), q(Z, g(Y))$ with moding $p(In, Out)$, $q(In, Out)$ is well-moded. Note that in a well-moded query, the terms that occur in the input positions of the leftmost atom are all ground.

The next lemma shows the persistence of the notion of well-modedness [4].

Lemma 2.1 An LD-resolvent of a well-moded goal and a well-moded clause that is variable-disjoint with it, is well-moded.

The following lemma states that well-moded programs are data driven [15].

Lemma 2.2 Let P be a well-moded program and Q be a well-moded query. All atoms selected in an LD-derivation of $P \cup \{Q\}$ contain ground terms in their input positions.

For a proof of the following lemma we refer to [2].

Lemma 2.3 Let P be a well-moded program and Q be a well-moded query. For every computed answer substitution σ of Q in P , $Q\sigma$ is ground.

Next, we introduce the notion of simply-modedness [1]. A family of terms is called *linear* if every variable occurs at most once in it.

Definition 2.6 (simply modedness)

A clause $p_0(\mathbf{s}_0, \mathbf{t}_{\mathbf{n}+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is called simply moded if $\mathbf{t}_1, \dots, \mathbf{t}_n$ is a linear family of variables and for $i \in [1, n]$

$$Var(\mathbf{t}_i) \cap \left(\bigcup_{j=0}^i Var(\mathbf{s}_j) \right) = \emptyset.$$

A program is called *simply moded* iff every clause of it is simply moded.

A query $\leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is simply moded iff the clause $\text{dummy} \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is simply moded, where *dummy* is any nullary predicate.

For instance the clauses $r(X, Y, f(Z)) \leftarrow q(g(X), Y), q(g(Y), Z)$ and $q(X, Y) \leftarrow q(Z, Y)$ with moding $r(In, Out, Out)$, $q(In, Out)$ are simply moded. Note that this last clause is not well-moded. The well-moded clause $p(X, f(Y)) \leftarrow q(X, g(Z)), q(Z, g(Y))$ (with moding $p(In, Out)$, $q(In, Out)$) from above is not simply moded.

The following persistence lemma was given in [1].

Lemma 2.4 *An LD-resolvent of a simply moded goal and a simply moded clause that is variable-disjoint with it, is simply moded.*

An atom is called *input/output disjoint* if the family of terms occurring in its input positions has no variable in common with the family of terms occurring in its output positions. The following is an easy consequence of Lemma 2.4.

Lemma 2.5 *Let P be a simply moded program and Q be a simply moded query. All atoms selected in an LD-derivation of $P \cup \{Q\}$ are input/output disjoint and such that each of the output positions is filled in by a distinct variable.*

In [1], it is argued that most programs are simply moded, and that often non-simply moded programs can be naturally transformed into simply moded ones. In [1], the class of simply moded, well-moded programs and queries is shown to be unification free, i.e. in the execution, unification can be replaced by iterated matching.

We conclude with a definition which will be useful in the termination conditions (Propositions 3.1, 3.2 in section 3 and Propositions 4.1, 4.2 in section 4). Recall that when writing an atom as $p(\mathbf{u}, \mathbf{v})$, we assume that \mathbf{u} is the sequence of terms filling in the input positions of p and \mathbf{v} is the sequence of terms filling in the output positions of p .

Definition 2.7 (input-correct w.r.t. S) *Let $S \subseteq B_P^E$ and $A = p(\mathbf{u}, \mathbf{v})$ an atom. We call A input-correct w.r.t. S , denoted by $A^{In} \in S^{In}$, iff there is an atom $B = p(\mathbf{u}, \mathbf{w})$ such that $\tilde{B} \in S$.*

2.3 Types, Well-typedness and Generic Expressions

In the sequel, we also use types. A *type* is defined as a decidable set of terms closed under substitution. A type T is called *ground* if all its elements are ground, and *non-ground* otherwise. A *typed term* is a construct of the form $s : S$, where s is a term and S is a type. Given a sequence $\mathbf{s} : \mathbf{S} = s_1 : S_1, \dots, s_n : S_n$ of typed terms, we write $\mathbf{s} \in \mathbf{S}$ iff for any $i \in [1, n]$, we have $s_i \in S_i$.

Example 2.3 *We give some examples of types.*

- *U : the set of all terms,*
- *Ground: the set of all ground terms,*
- *List: the set of all (possibly non-ground) nil-terminated lists (built on the empty list $[\]$ and the list constructor $[\cdot, \]$),*
- *Nat: the set of all natural numbers $\{0, s(0), s(s(0)), \dots\}$.*

Throughout the paper, we fix a specific set of types, denoted by $Types$. We also associate types with predicate symbols.

Definition 2.8 (type of a predicate) *Let p be an n -ary predicate symbol. A type for p is a function $t_p : \{1, \dots, n\} \rightarrow Types$. If $t_p(i) = T$, we call T the type associated with position i of p .*

Definition 2.9 (correctly typed) *Assuming a type t_p for the predicate p , we say that an atom $p(s_1, \dots, s_n)$ is correctly typed in position i iff $s_i \in t_p(i)$. We say that $p(s_1, \dots, s_n)$ is correctly typed iff it is correctly typed in all its positions.*

When every considered predicate has a mode and a type associated with it, we can talk about types of input positions and of output positions of an atom. An n -ary predicate p with a mode m_p and type t_p will be denoted by $p(\langle m_p(1) : t_p(1) \rangle, \dots, \langle m_p(n) : t_p(n) \rangle)$. To simplify the notation, when writing an atom as $p(\mathbf{u} : \mathbf{S}, \mathbf{v} : \mathbf{T})$ we assume that $\mathbf{u} : \mathbf{S}$ is a sequence of typed terms filling in the input positions of p and $\mathbf{v} : \mathbf{T}$ is a sequence of typed terms filling in the output positions of p . We call $p(\mathbf{u} : \mathbf{S}, \mathbf{v} : \mathbf{T})$ a *typed atom*.

Next, we introduce the notion of well-typedness [9]. First, we need the following concept of a type judgement.

Definition 2.10 ((true) type judgement)

A statement of the form $\mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$ is called a type judgement.

A type judgement $\mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$ is true, notated $\models \mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$, if for all substitutions θ , $\mathbf{s}\theta \in \mathbf{S}$ implies $\mathbf{t}\theta \in \mathbf{T}$.

Definition 2.11 (well-typedness)

A clause $p_0(\mathbf{o}_0 : \mathbf{O}_0, \mathbf{i}_{n+1} : \mathbf{I}_{n+1}) \leftarrow p_1(\mathbf{i}_1 : \mathbf{I}_1, \mathbf{o}_1 : \mathbf{O}_1), \dots, p_n(\mathbf{i}_n : \mathbf{I}_n, \mathbf{o}_n : \mathbf{O}_n)$ is called well-typed if for $j \in [1, n+1]$

$$\models \mathbf{o}_0 : \mathbf{O}_0, \dots, \mathbf{o}_{j-1} : \mathbf{O}_{j-1} \Rightarrow \mathbf{i}_j : \mathbf{I}_j.$$

A program is called well-typed iff every clause of it is well-typed.

A query $\leftarrow p_1(\mathbf{i}_1 : \mathbf{I}_1, \mathbf{o}_1 : \mathbf{O}_1), \dots, p_n(\mathbf{i}_n : \mathbf{I}_n, \mathbf{o}_n : \mathbf{O}_n)$ is well-typed iff the clause dummy $\leftarrow p_1(\mathbf{i}_1 : \mathbf{I}_1, \mathbf{o}_1 : \mathbf{O}_1), \dots, p_n(\mathbf{i}_n : \mathbf{I}_n, \mathbf{o}_n : \mathbf{O}_n)$ is well-typed, where dummy is any nullary predicate.

Example 2.4 *Let P be the following program computing all the lists which have a certain term as last element.*

$$\begin{aligned} last(X, [X]) &\leftarrow \\ last(X, [Y|R]) &\leftarrow last(X, R) \end{aligned}$$

Then, P is well-typed w.r.t. the typing $last(\langle In : U \rangle, \langle Out : List \rangle)$. This is because the following type judgements are true:

$$\begin{aligned} X : U &\Rightarrow [X] : List \quad , \\ X : U &\Rightarrow X : U \quad , \\ X : U \wedge R : List &\Rightarrow [Y|R] : List \quad . \end{aligned}$$

Note that in a well-typed query, the left-most atom is correctly typed in its input positions. The following lemma shows the persistence of the notion of well-typedness [9].

Lemma 2.6 *An LD-resolvent of a well-typed query and a well-typed clause that is variable-disjoint with it, is well-typed.*

Corollary 2.1 *Let P be a well-typed program and Q be a well-typed query. All atoms selected in an LD-derivation of $P \cup \{\leftarrow Q\}$ are correctly typed in their input positions.*

In [1] the notion of generic expression for a type was introduced. Intuitively, a term t is a generic expression for a type T if it is more general than all elements of T which unify with t . This notion is the main tool in the approach of [1] towards replacing unification by iterated matching. It turns out that surprisingly often the input positions of the heads of program clauses are filled in by generic expressions for appropriate types (see [1]). We recall the definition and basic properties, as stated in [1].

Definition 2.12 (generic expression for a type) *Let T be a type. A term t is a generic expression for T iff for every $s \in T$ variable-disjoint with t , if s unifies with t then s is an instance of t .*

Example 2.5 *Recall the types of Example 2.3.*

- $[], [X], [X|Y], [X, Y|Z], \dots$ are generic expressions for the type *List*,
- $0, s(X), s(s(X)), \dots$ are generic expressions for the type *Nat*.

A term of the form $f(X_1, \dots, X_m)$ with X_1, \dots, X_m a sequence of different variables is called a *pure variable term*.

Lemma 2.7 [1, Lemma 3.7] *Let T be a type. Then*

- *variables are generic expressions for T ,*
- *the only generic expressions for type U are variables,*
- *if T does not contain variables, then every pure variable term is a generic expression for T ,*
- *if T is ground, then every term is a generic expression for T .*

In [1], the notion of input safe atom was introduced as follows.

Definition 2.13 (input safe atom) *An atom is called input safe if*

- *each of its input positions is filled in with a generic expression for this position's type,*
- *either the types of all input positions are ground or the terms filling in the input positions are mutually variable-disjoint.*

In particular, an atom is input safe if the types of all input positions are ground.

The following lemma follows from [1, Lemma 3.13]. It models a one-step derivation of a simply moded well-typed query A with a simply moded well-typed clause with input safe head H . The parameter passing mechanism in this case is as follows. First the input values are passed from the atom A to the head H . Then the output values are passed from H to A .

Lemma 2.8 *Consider two variable-disjoint atoms $A = p(\mathbf{u}, \mathbf{v})$ and $H = p(\mathbf{s}, \mathbf{t})$ with the same predicate symbol p . Suppose that*

- *A is correctly typed in its input positions and simply moded,*
- *H is input safe.*

If there is a mgu ϕ for the set of equations $\{\mathbf{u} = \mathbf{s}, \mathbf{v} = \mathbf{t}\}$, then a mgu $\theta = \theta_1\theta_2$ exists with

- θ_1 is a mgu for $\{\mathbf{u} = \mathbf{s}\}$,
 $Dom(\theta_1) \subseteq InVar(H) = Var(\mathbf{s})$,
 $Ran(\theta_1) \subseteq InVar(A) = Var(\mathbf{u})$,
- θ_2 is a mgu for $\{\mathbf{v} = \mathbf{t}\theta_1\}$,
 $Dom(\theta_2) \subseteq OutVar(A) = Var(\mathbf{v})$,
 $Ran(\theta_2) \subseteq Var(\mathbf{t}\theta_1) \subseteq OutVar(H) \cup InVar(A) = Var(\mathbf{t}) \cup Var(\mathbf{u})$,
- $A\theta = p(\mathbf{u}, \mathbf{v})\theta = p(\mathbf{u}, \mathbf{v}\theta_2) = p(\mathbf{s}\theta_1, \mathbf{t}\theta_1) = p(\mathbf{s}, \mathbf{t})\theta = H\theta$,

and θ is equal to ϕ modulo variable renaming.

The fact that, in the previous lemma, θ is equal to ϕ modulo variable renaming follows from the following lemma proven in [19]: if θ and θ' are two mgu's of a set of equations, then for some renaming η , $\theta' = \theta\eta$.

Proposition 2.1 *Let P be a simply moded well-typed program and $\leftarrow B_1, \dots, B_n$ be a simply moded well-typed query. Suppose that the heads of the clauses of P are input safe.*

If $\sigma = \theta^1 \dots \theta^k$ is the substitution of a partial LD-derivation of $\leftarrow B_1, \dots, B_n$ in P , then the input arguments of B_1 are equal to the input arguments of $B_1\sigma$ modulo variable renaming.

Proof Recall that the notions of simply modedness and well-typedness are persistent (Lemma 2.4 and 2.6). Because two mgu's of the same set of equations are equal modulo renaming [19], it is sufficient to prove the proposition in the case that each θ^i , $i = 1, \dots, k$, is of the form given in Lemma 2.8 (note that the conditions of that lemma are satisfied in each step). We prove that in this case, the input arguments of B_1 are equal to the input arguments of $B_1\sigma = B_1\theta^1 \dots \theta^k$, i.e. no substitution θ^i , $i = 1, \dots, k$, is ever applied on variables of $InVar(B_1)$. For this, it is sufficient to prove the following two statements for every $i = 1, \dots, k$:

- the variables of $InVar(B_1)$ do not occur in the output positions of the i 'th query in the partial LD-derivation of $\leftarrow B_1, \dots, B_n$ in P ,
- in the i 'th step of the partial LD-derivation of $\leftarrow B_1, \dots, B_n$ in P , the domain of the mgu θ^i does not include variables of $InVar(B_1)$.

We will prove these two statements by induction on the length of the partial LD-derivation.

If $i = 1$, the first statement follows from the fact that the (first) query $\leftarrow B_1, \dots, B_n$ is simply moded. The second statement follows from Lemma 2.8.

Suppose $i > 1$ and suppose that the two statements are true for $i - 1$. That is, variables of $InVar(B_1)$ do not occur in the output positions of the $(i - 1)$ 'th query and the mgu θ^{i-1} is not applied on variables of $InVar(B_1)$. We prove the two statements for i .

- We prove that the variables of $InVar(B_1)$ do not occur in the output positions of the resulting i 'th query. Let $\leftarrow C_1, \dots, C_m$ be the $(i - 1)$ 'th query and $H \leftarrow B'_1, \dots, B'_r$ be the clause with which it is resolved. So, $\theta^{i-1} = mgu(C_1, H)$ and the resulting

i 'th query is $\leftarrow (B'_1, \dots, B'_r, C_2, \dots, C_m)\theta^{i-1}$. Because we assumed that θ^{i-1} is of the form given in Lemma 2.8, we can write $\theta^{i-1} \equiv \theta_1^{i-1}\theta_2^{i-1}$ (see Lemma 2.8), and so the i 'th query is equal to $\leftarrow (B'_1, \dots, B'_r)\theta_1^{i-1}, (C_2, \dots, C_m)\theta_2^{i-1}$. From Lemma 2.8 we know that $Dom(\theta_1^{i-1}) \subseteq InVar(H)$. From the fact that $H \leftarrow B'_1, \dots, B'_n$ is simply moded (more specifically from the fact that output positions in the body don't share variables with the input positions of the head), we can conclude that $(B'_1, \dots, B'_n)\theta_1^{i-1}$ does not contain variables of $InVar(B_1)$ in the output positions. From Lemma 2.8 we also know that $Dom(\theta_2^{i-1}) \subseteq OutVar(C_1)$. From the induction hypothesis, we know that $\leftarrow C_1, \dots, C_m$ does not contain variables of $InVar(B_1)$ in the output positions. Because $\leftarrow C_1, \dots, C_m$ is simply moded, more specifically because the output arguments form a linear family of variables, we hence have that $(C_2, \dots, C_m)\theta_2^{i-1}$ does not contain variables of $InVar(B_1)$ in the output positions.

- Next we prove that the domain of the *mg*u of the i 'th step, θ^i , does not include variables of $InVar(B_1)$. This follows from Lemma 2.8; the domain of θ^i is included in the set of input variables of the head of the clause used in the i 'th step union the set of output variables of $B_1\theta^{i-1}$ (which, as we already proved, doesn't contain variables of $InVar(B_1)$).

□

Corollary 2.2 *Let P be a simply moded well-typed program and $A \in B_P^E$ be a simply moded well-typed query. Suppose that the heads of the clauses of P are input safe.*

If $\sigma = \theta^1 \dots \theta^k$ is the substitution of a partial LD-derivation of $\leftarrow A$ in P , then the input arguments of A are equal to the input arguments of $A\sigma$ modulo variable renaming.

In particular, this holds when σ is an LD-computed answer substitution for $P \cup \{\leftarrow A\}$.

This corollary is interesting because it shows that if P and A are simply moded well-typed and if the heads of the clauses in P are input safe, then in every LD-derivation of $P \cup \{\leftarrow A\}$ the input arguments of A get never instantiated. This will be useful in the termination condition for simply moded well-typed programs and queries (Propositions 3.2 and 4.2). Namely, if the heads of the clauses of the program are input safe, we are no longer forced to reason on “calls” in the termination condition, instead, the termination condition can be easily stated at the clause level, which is useful in the context of an automatic termination analysis.

In [1] it was shown that if a program P and query Q are well-typed and simply moded and if the heads of the clauses in P are input safe, $P \cup \{\leftarrow Q\}$ is unification free. We refer to the list of program examples (taken from the book of Sterling and Shapiro [24]) in the discussion section of [1], where appropriate modings and typings for the considered predicates are given such that the programs are simply moded well-typed and the heads of their clauses are input safe. Note that for surprisingly many programs appropriate modings and typings can be found.

2.4 Norms and Level Mappings

We recall the definitions of norm and level mapping.

Definition 2.14 (norm) A norm is a function $\| \cdot \| : U_P^E \rightarrow \mathbb{N}$.

Definition 2.15 (level mapping) A level mapping is a function $|\cdot| : B_P^E \rightarrow \mathbb{N}$.

A level mapping or norm is said to be *trivial* if it is the constant 0-mapping. We introduce the following notations and concepts.

Definition 2.16 (functor and predicate coefficients) The set of functor coefficients, $FC(P)$, respectively predicate coefficients, $PC(P)$, associated to a program P are the sets of symbols

- $FC(P) = \{f_i \mid f/m \in Fun_P \wedge i \in \{0, 1, \dots, m\}\}$,
- $PC(P) = \{p_i \mid p/n \in Pred_P \wedge i \in \{1, \dots, n\}\}$.

Here, all norms and level mappings will be of a specified form (for the norms this is a slight variant of the semi-linear norms [8]). We first introduce these *symbolic* forms for the norm and level mapping, with symbols in $FC(P) \cup PC(P)$. A concrete norm or level mapping is then obtained by giving values to the symbols in $FC(P) \cup PC(P)$ in \mathbb{N} . Let Var_P denote the set of variables in a program and $Const_P$ denote the set of constant symbols in a program.

Definition 2.17 (symbolic norm and level mapping)

The symbolic norm $\| \cdot \| ^S$ is defined as:

$$\begin{cases} \| X \| ^S = 0, & X \in Var_P, \\ \| c \| ^S = 0, & c \in Const_P, \\ \| f(t_1, \dots, t_m) \| ^S = f_0 + \sum_{i=1}^m f_i \| t_i \| ^S \end{cases}$$

with $f_i \in FC(P)$, for $i \in \{0, \dots, m\}$.

The symbolic level mapping $|\cdot| ^S$ is defined as:

$$|p(t_1, \dots, t_n)| ^S = \sum_{i=1}^n p_i \| t_i \| ^S$$

with $p_i \in PC(P)$, for $i \in \{1, \dots, n\}$.

Definition 2.18 (symbol mapping) A symbol mapping is a mapping $s : FC(P) \cup PC(P) \rightarrow \mathbb{N}$.

A symbol mapping s induces in a natural way a norm and a level mapping, by mapping the coefficient symbols in the symbolic norm and level mapping to their actual values under s .

Definition 2.19 (norm and level mapping induced by s) Let s be a symbol mapping.

The norm $\| \cdot \|_s$ induced by s is defined in the following natural way:

$$\begin{cases} \| X \|_s = 0, & X \in Var_P, \\ \| c \|_s = 0, & c \in Const_P, \\ \| f(t_1, \dots, t_m) \|_s = s(f_0) + \sum_{i=1}^m s(f_i) \| t_i \|_s \end{cases}.$$

The level mapping $|\cdot|_s$ induced by s is defined in the following natural way:

$$|p(t_1, \dots, t_n)|_s = \sum_{i=1}^n s(p_i) \| t_i \|_s.$$

For example, the *term-size norm* is induced by the symbol mapping s with $s(f_i) = 1$, $i = 0, \dots, n$, for all $f/n \in Fun_P$, $n > 0$.

In the rest of this subsection, we introduce the notions of rigid and finitely partitioning level mapping. As we will see, these notions turn out to be very useful in the context of a termination analysis. In particular, the notion of *rigid* level mapping [8] deals with the problem of backpropagation of bindings in the calls. Instead of reasoning on “calls” in the termination condition, a level mapping which is rigid on the call set allows us to reason fully at the clause level, which is important in the context of an automatic termination analysis [13]. In [14], the notion of *finitely partitioning* level mapping was shown to be crucial in the context of a termination condition of tabled logic programs. Namely, in the quasi-acceptability condition (Definition 3.2 further on), which was shown to be equivalent with quasi-termination of tabled logic programs (Theorem 3.1), a non-strict decrease on the calls is required w.r.t. a level mapping which is finitely partitioning on the call set.

Definition 2.20 (rigid level mapping) *Let $|\cdot|$ be a level mapping and $A \in B_P^E$. The level mapping $|\cdot|$ is called rigid on A iff its value on A is invariant under substitutions, i.e. $\forall \sigma : |A\sigma| = |A|$.*

The level mapping $|\cdot|$ is rigid on a set of atoms $S \subseteq B_P^E$ iff $|\cdot|$ is rigid on all $A \in S$.

Following the syntactic characterisation of rigidity of terms under semi-linear norms of [7], we have the following necessary and sufficient condition for rigidity of a level mapping on a set of atoms.

Lemma 2.9 (rigid level mapping: necessary and sufficient condition)

Let $|\cdot|_s$ be a non-trivial level mapping (induced by the symbol mapping s) and $S \subseteq B_P^E$. Then, $|\cdot|_s$ is rigid on S iff

for any atom $p(t_1, \dots, t_i, \dots, t_n)$ in S and any occurrence of a variable X , occurring within some argument t_i of $p(t_1, \dots, t_i, \dots, t_n)$, either $s(p_i) = 0$ or there is a subterm of t_i (possibly t_i itself), $f(s_1, \dots, s_j, \dots, s_m)$, with s_j containing the occurrence of X , such that $s(f_j) = 0$.

(i.e. no occurrence of a variable in an atom of S is measured by $|\cdot|_s$)

Proof \Leftarrow : Suppose the condition on the level mapping is satisfied. Then $|\cdot|_s$ is rigid on S . This can easily be seen because the positions of all the variables occurring in the atoms of S are not taken into account by the level mapping, so instantiating an atom in S does not change its value under $|\cdot|_s$.

\Rightarrow : Suppose $|\cdot|_s$ is rigid on S . We prove that $|\cdot|_s$ satisfies the condition of the lemma. Suppose that there is an atom $p(t_1, \dots, t_i, \dots, t_n)$ with t_i containing a free variable X , in S and $s(p_i) \neq 0$ and all the functorarguments f_j leading to this positions of X are such that $s(f_j) \neq 0$ (possibly there are no functorarguments leading to the position of X if $t_i = X$). We prove a contradiction with the rigidity of $|\cdot|_s$ on S . Let t be a term such that $\|t\|_s \neq 0$, with $\|\cdot\|_s$ the norm underlying the level mapping $|\cdot|_s$ (note that such a term t exists, since $|\cdot|_s$ is non-trivial). Consider the substitution $\sigma = \{X/t\}$. Then it is easy to see that $|p(t_1, \dots, t_i, \dots, t_n)|_s \neq |p(t_1, \dots, t_i, \dots, t_n)\sigma|_s$ and this gives a contradiction. \square

Next, we introduce the notion of finitely partitioning level mapping, which is crucial in the context of termination of tabled logic programs [14].

Definition 2.21 (finitely partitioning level mapping) Let $|\cdot|$ be a level mapping and $S \subseteq B_P^E$. The level mapping $|\cdot|$ is called finitely partitioning on S iff $\forall n \in \mathbb{N} : \sharp(|\cdot|^{-1}(n) \cap S) < \infty$.

It is easy to see that if the set S is finite, all level mappings are finitely partitioning on S (in particular this is the case if the Herbrand Universe is finite). For an infinite set of atoms which is closed under substitution, we give a necessary condition on a level mapping to be finitely partitioning on that set.

Lemma 2.10 (finitely partitioning level mapping: necessary condition) Assume the Herbrand Universe is infinite. Let $|\cdot|_s$ be a level mapping (induced by the symbol mapping s) and $S \subseteq B_P^E$ be an infinite set closed under substitution. If $|\cdot|_s$ is finitely partitioning on S , then for any atom $p(t_1, \dots, t_i, \dots, t_n)$ in S and any variable X , occurring in $p(t_1, \dots, t_i, \dots, t_n)$, there is a term t_i in which X occurs such that $s(p_i) \neq 0$ and for all subterms of t_i (including t_i itself), $f(s_1, \dots, s_j, \dots, s_m)$, with s_j containing the occurrence of X , $s(f_j) \neq 0$. (i.e. some occurrence of a variable in an atom of S is measured by $|\cdot|_s$)

Proof \Rightarrow : Suppose $|\cdot|_s$ is finitely partitioning on S . We prove that the condition on $|\cdot|_s$ is satisfied. Suppose that there is an atom $p(t_1, \dots, t_n)$ in S and a variable X occurring in $p(t_1, \dots, t_n)$ such that no occurrence of X in $p(t_1, \dots, t_n)$ is measured by the level mapping $|\cdot|_s$ (i.e. the negation of the condition in the lemma is true). Consider the following infinite set of substitutions $\{\sigma_t = \{X/t\} \mid t \in HU\}$ (with HU the (infinite) Herbrand Universe). Because S is closed under substitution, the infinite set of atoms $\{p(t_1, \dots, t_n)\sigma_t \mid t \in HU\}$ is a subset of S . It is easy to see that $|p(t_1, \dots, t_n)\sigma_t|_s = |p(t_1, \dots, t_n)\sigma_{t'}|_s, \forall t, t' \in HU$. This gives a contradiction with the fact that $|\cdot|_s$ is finitely partitioning on S . \square

As one can see from Lemma 2.9 and Lemma 2.10, for infinite sets closed under substitution the notions of rigid and finitely partitioning level mapping are in a sense contradictory. This suggests that combining the constraint-based approach of [13], which relies on rigidity, with the approach of [14] for tabled logic programs, which relies on finitely partitioning level mappings, will not be feasible -at least not in the context of sets which are closed under substitution. We will deal with this problem in two ways.

First, we will consider simply moded well-moded programs and queries (subsections 3.1 and 4.1). For well-moded programs and sets of queries, we will give a sufficient condition on a level mapping in order for it to be rigid on the call set of the set of queries w.r.t. the program (see Proposition 2.2 below). We will also give a sufficient condition on a level mapping in order for it to be finitely partitioning on the call set of a set of simply moded queries w.r.t. a simply moded program (see Proposition 2.3 below). As we will see, we will be able to combine these sufficient conditions on a level mapping in order for it to be rigid *and* finitely partitioning on the call set of a set of simply moded well-moded queries w.r.t. a simply moded well-moded program.

In the second approach, we consider simply moded well-typed programs and queries (subsections 3.2 and 4.2). Again, we will use Proposition 2.3 (see below) which gives a sufficient condition on a level mapping in order for it to be finitely partitioning on the call set of a set of simply moded queries w.r.t. a simply moded program. But, we will abandon the rigidity requirement on the level mapping. Instead, we will use Proposition 2.1 and Corollary 2.2 which give a sufficient condition on a simply moded well-typed program such that the input arguments of a simply moded well-typed query get never instantiated during a

derivation (namely, that the heads of the clauses are input safe). This will give us a way to reason at the clause level (and not on the calls) in the termination condition.

In the rest of this subsection, we will use mode-information, in particular well-modedness and simply modedness, to obtain sufficient conditions on a level mapping in order for it to be rigid or finitely partitioning on the call set. First, we need the following definition.

Definition 2.22 (measuring only/all input) *Let $|\cdot|_s$ be a level mapping (induced by the symbol mapping s) and $S \subseteq B_P^E$.*

We say that $|\cdot|_s$ measures only input positions in S iff

- *for every predicate p/n occurring in S : if for $i \in \{1, \dots, n\}$ $s(p_i) \neq 0$, then $m_p(i) = In$,*

We say that $|\cdot|_s$ measures all input positions in S iff

- *for every predicate p/n occurring in S : if $m_p(i) = In$, then $s(p_i) \neq 0$, and*
- *for every functor f/m , $m > 0$, occurring in an input position of an atom in S : $s(f_i) \neq 0$ for all $i \in \{0, \dots, m\}$.*

Proposition 2.2 *Let P be a well-moded program and S be a set of well-moded queries. Let $|\cdot|_s$ be a level mapping which measures only input positions in $Call(P, S)$. Then, $|\cdot|_s$ is rigid on $Call(P, S)$.*

Proof Because well-moded programs are data-driven (Lemma 2.2), all atoms in $Call(P, S)$ contain ground terms in their input positions. So variables can only occur in the output positions of atoms in $Call(P, S)$. Since $|\cdot|_s$ measures only input positions in $Call(P, S)$, it follows from Lemma 2.9 that $|\cdot|_s$ is rigid on $Call(P, S)$. \square

Proposition 2.3 *Let P be a simply moded program and S be a set of simply moded queries. Let $|\cdot|_s$ be a level mapping which measures all input positions in $Call(P, S)$. Then, $|\cdot|_s$ is finitely partitioning on $Call(P, S)$.*

Proof Take $n \in \mathbb{N}$. We prove that $\sharp((|\cdot|_s)^{-1}(n) \cap Call(P, S)) < \infty$. Since P and S are simply moded, all atoms in $Call(P, S)$ contain a linear family of variables in their output positions (Lemma 2.5). Hence, since $|\cdot|_s$ measures all input positions in $Call(P, S)$ (and there are only a finite number of function symbols), there are only a finite number of atoms in $Call(P, S)$ with value n under $|\cdot|_s$. \square

Obviously, this proposition can be stated more general (and we will use also this formulation in the sequel), namely: *if $|\cdot|_s$ is a level mapping which measures all input positions in $S' \subseteq Call(P, S)$, then $|\cdot|_s$ is finitely partitioning on S' .*

It follows from Proposition 2.2 and 2.3 that, if P is a simply moded well-moded program, S a set of simply moded well-moded queries and $|\cdot|_s$ a level mapping which measures all and only input positions in $Call(P, S)$, then $|\cdot|_s$ is rigid and finitely partitioning on $Call(P, S)$. This will be used in the termination conditions in subsection 3.1 (Proposition 3.1) and subsection 4.1 (Proposition 4.1).

3 QUASI-TERMINATION

A first basic notion of termination under tabled execution mechanism is quasi-termination (see [14]). It is defined as follows.

Definition 3.1 (quasi-termination) *Let P be a program and $S \subseteq B_P^E$. P quasi-terminates w.r.t. S iff for all A which have a representant in S , the LG-forest for $P \cup \{\leftarrow A\}$ consists of a finite number of LG-trees.*

Note that the program P of Example 2.1 (subsection 2.1) which defines the natural numbers, quasi-terminates w.r.t. $\{nat(X)\}$; the LG-forest of $P \cup \{\leftarrow nat(X)\}$ consists of one LG-tree. The program P^a of Example 2.2 does not quasi-terminate w.r.t. $\{nat(X)\}$.

Lemma 3.1 [14, Lemma 3.1] *Let P be a program and $S \subseteq B_P^E$. P quasi-terminates w.r.t. S iff for every A in S , $Call(P, \{A\})$ is finite.*

It follows from Lemma 2.5 that if P and S are simply moded, all atoms in $Call(P, S)$ are such that the output positions are filled in by a linear family of variables. Hence, for simply moded programs and queries, we can state Lemma 3.1 as follows.

Corollary 3.1 *Let P be a simply moded program and $S \subseteq B_P^E$ be a set of simply moded queries. P quasi-terminates w.r.t. S iff for every A in S , the set of all terms that occur in input positions of the atoms in $Call(P, \{A\})$ is finite.*

It is easy to see that LD-termination of P w.r.t. S implies quasi-termination of P w.r.t. S . Note that when a program is quasi-terminating w.r.t. a query Q , there are a finite number of trees in the LG-forest, all of them have finite branches, but, possibly, they are infinitely branching. In the next section 4, we introduce and provide conditions for the stronger notion of *LG-termination*, which requires that the LG-forest consists of a finite number of finite LG-trees.

In [14], the quasi-acceptability condition was introduced and it was shown to be equivalent with quasi-termination.

Definition 3.2 (quasi-acceptability w.r.t. S) *Let P be a program and $S \subseteq B_P^E$. If there is a level mapping $|\cdot|$ which is finitely partitioning on $Call(P, S)$ such that*

- *for every atom A such that $\tilde{A} \in Call(P, S)$,*
- *for every clause $H \leftarrow B_1, \dots, B_n$ in P such that $mgu(A, H) = \theta$ exists,*
- *for every B_i , $i \in \{1, \dots, n\}$,*
- *for every LD-computed answer substitution θ_{i-1} for $\leftarrow (B_1, \dots, B_{i-1})\theta$,*

$$|A| \geq |B_i\theta\theta_{i-1}|,$$

then, we say that P is quasi-acceptable w.r.t. S and $|\cdot|$.

Theorem 3.1 [14, Theorem 3.1] *Let P be a program and $S \subseteq B_P^E$. P is quasi-acceptable w.r.t. S and some level mapping $|\cdot|$ which is finitely partitioning on $Call(P, S)$ iff P is quasi-terminating w.r.t. S .*

In order to prove quasi-termination in an automatic way, we need (1) a termination condition which reasons fully at the clause level (and not on calls), and (2) a syntactical condition on a level mapping in order to be finitely partitioning. Concerning (2), we will use Proposition 2.3, and consider simply moded programs in the sequel. In the next subsection 3.1, we will consider simply moded well-moded programs. In order to deal with non-ground input, we will consider simply moded well-typed programs in subsection 3.2.

3.1 Simply Moded Well-Moded Programs and Queries

In order to be able to reason fully at the clause level in the termination condition, the notion of rigid level mapping was shown to be useful (see for instance the rigid acceptability condition of [13] in the context of LD-termination). For well-moded programs P and sets of queries S , we know that a level mapping is rigid on $\text{Call}(P, S)$ if it measures *only* the input positions in $\text{Call}(P, S)$ (Proposition 2.2). Proposition 2.3 provides us with a sufficient condition on a level mapping to be finitely partitioning on $\text{Call}(P, S)$, in case the program P and set of queries S are simply moded. Namely, the level mapping has to measure *all* input positions in $\text{Call}(P, S)$. Combined together, we have the following result.

Proposition 3.1 *Let P be a simply moded well-moded program and $S \subseteq B_P^E$ be a set of simply moded well-moded queries. Let $|\cdot|$ be a level mapping which measures all and only the input positions in $\text{Call}(P, S)$ and such that*

- for any clause $H \leftarrow B_1, \dots, B_n$ in P ,
- for any atom B_i , $i \in \{1, \dots, n\}$,
- for any correct answer substitution ψ for $\leftarrow B_1, \dots, B_{i-1}$,
such that $(H\psi)^{In}$, $(B_1\psi)^{In}, \dots, (B_i\psi)^{In} \in \text{Call}(P, S)^{In}$,

$$|H\psi| \geq |B_i\psi|,$$

then P is quasi-acceptable w.r.t. S . Hence, P quasi-terminates w.r.t. S .

Proof Suppose the above condition is satisfied for P . Note that, because P and S are simply moded well-moded, it follows from Propositions 2.2 and 2.3 that $|\cdot|$ is rigid and finitely partitioning on $\text{Call}(P, S)$. Let A be an atom such that $\tilde{A} \in \text{Call}(P, S)$. Let $H \leftarrow B_1, \dots, B_n$ be a clause in P such that $\text{mgu}(A, H) = \theta$ exists. Let θ_{i-1} be an LD-computed answer substitution for $\leftarrow (B_1, \dots, B_{i-1})\theta$ (in the case $i = 1$, θ_{i-1} is the empty substitution). Then, $\theta\theta_{i-1}$ is a correct answer substitution for $\leftarrow B_1, \dots, B_{i-1}$. We have to prove that $|A| \geq |B_i\theta\theta_{i-1}|$ (Theorem 3.1). Because $(H\theta\theta_{i-1})^{In} = (A\theta\theta_{i-1})^{In} = A^{In}$ (A is a well-moded query, so ground in its input positions), we have that $(H\theta\theta_{i-1})^{In} \in \text{Call}(P, S)^{In}$. Also, it is easy to see that $(B_1\theta\theta_{i-1})^{In}, \dots, (B_i\theta\theta_{i-1})^{In} \in \text{Call}(P, S)^{In}$. Hence, we know that $|H\theta\theta_{i-1}| \geq |B_i\theta\theta_{i-1}|$. Because $A\theta = H\theta$, $|A\theta\theta_{i-1}| = |H\theta\theta_{i-1}|$. Now, since $A \in \text{Call}(P, S)$ and $|\cdot|$ is rigid on $\text{Call}(P, S)$, $|A\theta\theta_{i-1}| = |A|$. Thus, $|A| = |H\theta\theta_{i-1}|$, therefore $|A| \geq |B_i\theta\theta_{i-1}|$. \square

In the following examples, we will prove quasi-termination of a simply moded well-moded program w.r.t. a set of simply moded well-moded queries, by applying Proposition 3.1. Note that we can remove the restriction on ψ in the third condition of the proposition (namely that $(H\psi)^{In}, (B_1\psi)^{In}, \dots, (B_i\psi)^{In} \in \text{Call}(P, S)^{In}$) and prove the condition for a bigger class of substitutions ψ (this removal might simplify the termination proof). Obviously, we can still conclude quasi-termination.

Example 3.1 *Let P be the program of Example 2.1 of subsection 2.1 defining the natural numbers. Let S be the set $\{\text{nat}(X)\}$. Consider the moding $\text{nat}(\text{Out})$. Then, P and S are simply moded well-moded.*

We prove that P quasi-terminates w.r.t. S by applying Proposition 3.1. Let the level mapping $|\cdot|$ be the trivial level mapping (which maps everything to 0). Then, $|\cdot|$ measures all and only the input positions in $\text{Call}(P, S)$ (there are no

input positions). And trivially, $|\text{nat}(s(X))| = 0 \geq 0 = |\text{nat}(X)|$. Hence, P quasi-terminates w.r.t. S . Note that P doesn't LD-terminate w.r.t. S . Note that P also doesn't LG-terminate w.r.t. S . The LG-forest has one infinitely branching LG-tree.

Example 3.2 Let P be the following program

$$\begin{cases} \text{even}(0) & \leftarrow \\ \text{even}(s(X)) & \leftarrow \text{odd}(X) \\ \text{odd}(s(X)) & \leftarrow \text{even}(X) \end{cases}$$

with moding $\text{even}(\text{Out}), \text{odd}(\text{Out})$ and $S = \{\text{even}(X), \text{odd}(X)\}$. Then, P and S are simply moded well-moded. Similarly as in the previous example, P quasi-terminates w.r.t. S .

Example 3.3 Consider the following program P computing all the paths in a (cyclic) graph from one node to the reachable nodes.

$$\begin{cases} \text{edge}(a, b) & \leftarrow \\ \text{edge}(b, c) & \leftarrow \\ \text{edge}(c, a) & \leftarrow \\ \text{edge}(c, d) & \leftarrow \\ \text{path}(X, X, []) & \leftarrow \\ \text{path}(X, Y, [Y]) & \leftarrow \text{edge}(X, Y) \\ \text{path}(X, Z, [Y|L]) & \leftarrow \text{edge}(X, Y), \text{path}(Y, Z, L) \end{cases}$$

with moding $\text{edge}(\text{In}, \text{Out})$ and $\text{path}(\text{In}, \text{Out}, \text{Out})$. Then it is easy to see that P is simply moded well-moded. Consider the (simply moded well-moded) query $\text{path}(a, Z, L)$.

P quasi-terminates w.r.t. $\{\text{path}(a, Z, L)\}$. To prove this, we apply Proposition 3.1. Let $|\cdot|$ be the following level mapping (which measures all and only input positions in $\text{Call}(P, \{\text{path}(a, Z, L)\})$):

$$\begin{aligned} |\text{edge}(t_1, t_2)| &= \|t_1\|, \\ |\text{path}(t_1, t_2, t_3)| &= \|t_1\|, \end{aligned}$$

where $\|\cdot\|$ is the term-size norm.

Then, for every substitution ψ , $|\text{path}(X, Y, [Y])\psi| = \|X\psi\| \geq \|X\psi\| = |\text{edge}(X, Y)\psi|$ and $|\text{path}(X, Z, [Y|L])\psi| = \|X\psi\| \geq \|X\psi\| = |\text{edge}(X, Y)\psi|$. Now, because every correct answer substitution ψ for $\leftarrow \text{edge}(X, Y)$, is of the form $\{X/c_1, Y/c_2\}$, with c_1 and c_2 constants, also $|\text{path}(X, Z, [Y|L])\psi| = \|X\psi\| \geq \|Y\psi\| = |\text{path}(Y, Z, L)\psi|$.

This proves that P quasi-terminates w.r.t. $\{\text{path}(a, Z, L)\}$. Note that P does not LD-terminate w.r.t. $\{\text{path}(a, Z, L)\}$. Also, P does not LG-terminate w.r.t. $\{\text{path}(a, Z, L)\}$ (there are infinitely many (cyclic) paths from a to itself).

In the previous three examples, the set of all terms that occur in input positions of S is finite. In the next example, the set of simply moded well-moded queries S contains an infinite number of atoms with an infinite number of different terms in the input positions.

Example 3.4 Similarly as the program of Example 3.3, the following program P computes the paths from a given node to the reachable nodes. But here the graph is given as an input parameter; it is represented as a list of terms $e(x, y)$, indicating that there is an edge from node x to node y .

$$\begin{cases} \text{path}(X, Ed, X, []) & \leftarrow \\ \text{path}(X, Ed, Y, [Y]) & \leftarrow \text{member}(X, Ed, Y) \\ \text{path}(X, Ed, Z, [Y|L]) & \leftarrow \text{member}(X, Ed, Y), \text{path}(Y, Ed, Z, L) \\ \text{member}(X, [e(X, Y)|L], Y) & \leftarrow \\ \text{member}(X, [e(X_1, X_2)|L], Y) & \leftarrow \text{member}(X, L, Y) \end{cases}$$

with moding $\text{path}(In, In, Out, Out)$ and $\text{member}(In, In, Out)$. Then P is simply moded well-moded. Let $C = \{a, b, c, d\}$ be a set of constant symbols. Consider the following (infinite) set of simply moded well-moded queries

$S = \{\text{path}(x, ed, Y, L) \mid x \in C \text{ and } ed \text{ is a nil-terminated list of terms } e(y, z) \text{ with } y, z \in C\}$.

We prove that P quasi-terminates w.r.t. S by applying Proposition 3.1. Let $|\cdot|$ be the following level mapping (satisfying the conditions of Proposition 3.1):

$$\begin{aligned} |\text{path}(t_1, t_2, t_3, t_4)| &= \|t_1\| + \|t_2\|, \\ |\text{member}(t_1, t_2, t_3)| &= \|t_1\| + \|t_2\|, \end{aligned}$$

with $\|\cdot\|$ the term-size norm.

Consider the second clause. Let ψ be a substitution, then it is easy to see that $|\text{path}(X, Ed, Y, [Y])\psi| = \|X\psi\| + \|Ed\psi\| \geq \|X\psi\| + \|Ed\psi\| = |\text{member}(X, Ed, Y)\psi|$.

For the third clause, we can apply the same reasoning for the first body atom. Concerning the second body atom, let ψ be a correct answer substitution for $\leftarrow \text{member}(X, Ed, Y)$, such that $\text{path}(X, Ed, Z, [Y|L])\psi$, $\text{member}(X, Ed, Y)\psi$ and $\text{path}(Y, Ed, Z, L)\psi$ are input-correct w.r.t. $\text{Call}(P, S)$. It is easy to see that ψ binds Ed to a nil-terminated list of $e(x, y)$ -terms, with $x, y \in C$ and that ψ binds X and Y to a constant in C . Hence, $|\text{path}(X, Ed, Z, [Y|L])\psi| = \|X\psi\| + \|Ed\psi\| \geq \|Y\psi\| + \|Ed\psi\| = |\text{path}(Y, Ed, Z, L)\psi|$.

Finally, consider the non-unit clause for member . Let ψ be a substitution, then $|\text{member}(X, [e(X_1, X_2)|L], Y)\psi| = \|X\psi\| + \| [e(X_1, X_2)|L]\psi \| \geq \|X\psi\| + \|L\psi\| = |\text{member}(X, L, Y)\psi|$.

Note again that P does not LG-terminate w.r.t. S (for the same reason as in Example 3.3). But, if S' is the subset of S consisting of atoms $\text{path}(x, ed, Y, L)$, where $x \in C$ and ed is a nil-terminated list of $e(y, z)$ -terms, $y, z \in C$, representing an acyclic graph, then P LG-terminates w.r.t. S' . We even have that P LD-terminate w.r.t. S' .

3.2 Simply Moded Well-Typed Programs and Queries

In the previous subsection, we considered well-moded programs and queries, and hence we only dealt with ground input. In this subsection, we want to extend this in order to be able to deal with non-ground input as well. We will consider simply moded well-typed programs and queries. Again, we will use the fact that for a simply moded program P and a set of simply moded queries S , a level mapping is finitely partitioning on the call set $\text{Call}(P, S)$, if it measures all input positions. In order to be able to state a condition for quasi-termination which reasons fully at the clause level, we use Proposition 2.1 and Corollary 2.2. In Corollary 2.2, we proved that for a simply moded well-typed program P and query A , such that the heads of the clauses of P are input safe, the input arguments of A get never instantiated during an LD-derivation.

Proposition 3.2 *Let P be a simply moded well-typed program and $S \subseteq B_P^E$ be a set of simply moded well-typed queries. Suppose that the heads of the clauses in P are input safe. Let $|\cdot|$ be a level mapping which measures all and only the input positions in $\text{Call}(P, S)$ and such that*

- for any clause $H \leftarrow B_1, \dots, B_n$ in P ,
- for any atom B_i , $i \in \{1, \dots, n\}$,
- for any correct answer substitution ψ for $\leftarrow B_1, \dots, B_{i-1}$,
such that $(H\psi)^{In}, (B_1\psi)^{In}, \dots, (B_i\psi)^{In} \in \text{Call}(P, S)^{In}$,

$$|H\psi| \geq |B_i\psi|,$$

then P is quasi-acceptable w.r.t. S . Hence, P quasi-terminates w.r.t. S .

Proof Let P , S and $|\cdot|$ satisfy the condition of the proposition. Because P and S are simply moded and $|\cdot|$ measures all input positions, $|\cdot|$ is finitely partitioning on $\text{Call}(P, S)$ (Lemma 2.3). Let A be an atom such that $A \in \text{Call}(P, S)$ and $H \leftarrow B_1, \dots, B_n$ be a clause in P such that $\text{mgu}(A, H) = \theta$ exists and let $i \in \{1, \dots, n\}$ and θ_{i-1} be an LD-computed answer substitution for $\leftarrow (B_1, \dots, B_{i-1})\theta$. We have to prove that $|A| \geq |B_i\theta\theta_{i-1}|$. The substitution $\theta\theta_{i-1}$ is a correct answer substitution for $\leftarrow B_1, \dots, B_{i-1}$. From Corollary 2.2, we know that the input arguments of A are equal to the input arguments of $A\theta\theta_{i-1}$ modulo variable renaming. Because $H\theta\theta_{i-1} = A\theta\theta_{i-1}$, we have that $(H\theta\theta_{i-1})^{In} \in \text{Call}(P, S)^{In}$. Because of Proposition 2.1, we also know that $(B_1\theta\theta_{i-1})^{In}, \dots, (B_{i-1}\theta\theta_{i-1})^{In} \in \text{Call}(P, S)^{In}$. Trivially, $(B_i\theta\theta_{i-1})^{In} \in \text{Call}(P, S)^{In}$. Hence, $|H\theta\theta_{i-1}| \geq |B_i\theta\theta_{i-1}|$. Since $|\cdot|$ measures only input positions, we have that $|A| = |H\theta\theta_{i-1}|$ and therefore, $|A| \geq |B_i\theta\theta_{i-1}|$. \square

Note that, because P and S are well-typed, it follows that, if the conditions of the proposition are satisfied, i.e. if ψ is a correct answer substitution for $\leftarrow B_1, \dots, B_{i-1}$ such that $(H\psi)^{In}, (B_1\psi)^{In}, \dots, (B_i\psi)^{In} \in \text{Call}(P, S)^{In}$, then $H\psi$ is correctly typed in its input positions, $(B_1, \dots, B_{i-1})\psi$ is correctly typed and $B_i\psi$ is correctly typed in its input positions.

Notice the similarity with Proposition 3.1. Actually, Proposition 3.1 is a corollary of the previous proposition where the input types of the predicates are all defined as *Ground*. This follows from the fact that every term is a generic expression for the *Ground* type (Lemma 2.7), so the heads of all clauses are trivially input safe in this case, and from results proven in [4]. Namely, it is proven in [4] that the notion of well-moded program (resp. query) is a special case of the notion of well-typed program (resp. query) ([4, Theorem 3.7]). We refer to [4] for more details.

In the next example, we use Proposition 3.2 to prove quasi-termination of a program w.r.t. a set of queries.

Example 3.5 Let P be the program of Example 2.4 of subsection 2.3, computing all the lists which have a certain term as last element. Consider the query $Q = \text{last}(X, L)$, with X, L variables. We prove that P quasi-terminates w.r.t. $\{Q\}$.

We already proved in Example 2.4 that P is well-typed w.r.t. the typing $\text{last}(\langle In : U \rangle, \langle Out : List \rangle)$. It is trivial to see that $\leftarrow Q$ is also well-typed w.r.t. this typing. Also, P and Q are simply moded and the heads of the clauses in P are input safe.

Let $|\cdot|$ be the following level mapping (measuring all and only the input positions in $\text{Call}(P, S)$):

$$|\text{last}(t_1, t_2)| = \|t_1\|,$$

where $\|\cdot\|$ is the term-size norm.

It is trivial to see that for any substitution ψ , $|\text{last}(X, [Y|R])\psi| = \|X\psi\| \geq \|X\psi\| = |\text{last}(X, R)\psi|$.

Hence, P quasi-terminates w.r.t. $\{Q\}$. Note that P does not LG-terminate w.r.t. $\{Q\}$.

3.3 Constraint-based Approach for Automatically Proving Quasi-Termination

In this subsection, we point out how the constraint-based approach of [13] for automatically proving LD-termination of definite programs w.r.t. sets of queries needs to be changed in order to prove quasi-termination.

We first recall the main ideas of [13]. In [13], a new strategy for automatically proving LD-termination, i.e. left-termination or termination of SLD-resolution w.r.t. the left-to-right selection rule, of logic programs w.r.t. a set of queries is developed. A symbolic termination condition is introduced, by parametrising the concepts of norm, level mapping and model. The condition is translated into a system of constraints on the values of the introduced symbols only. With each solution of the constraint system, there corresponds one termination proof. The solving of constraint sets enables the different components of a termination proof to communicate with one another and to direct the proof towards success (if there is). The method of [13] is both efficient and precise. The basis of the approach in [13] is the following proposition, which gives sufficient conditions such that a program LD-terminates w.r.t. a set of queries. Note that the condition is fully at the clause level.

Proposition 3.3 (rigid acceptability w.r.t. S) *Let P be a program and $S \subseteq B_P^E$. If there exists a level mapping $|\cdot|$ which is rigid on $Call(P, S)$ and a model I for P , such that*

- for any clause $H \leftarrow B_1, \dots, B_n \in P$,
- for any atom B_i such that $Rel(B_i) \simeq Rel(H)$,
- for any substitution θ such that $I \models (B_1 \wedge \dots \wedge B_{i-1})\theta$:

$$|H\theta| > |B_i\theta|, \tag{1}$$

then P LD-terminates w.r.t. S .

In [13], this rigid acceptability condition is translated into a system of constraints on the values of the introduced symbols, which parametrize the notions of norm, level mapping and model. The symbolic versions of norm and level mapping are as in Definition 2.17 of subsection 2.4, but with this difference that in [13] the symbolic norm of a variable is the variable itself, i.e. $\|X\|^S = X$, in order to include information about the instantiation level of the term and to take into account those parts whose size may still change under instantiation. In this subsection, we will not go into detail about this, and we refer to [13] where a motivation and lots of examples are given. In order to symbolize the notion of model, symbolic versions of interargument relations are introduced. Interargument relations are abstractions of models by specifying relations which hold between the norms of certain arguments of their member atoms. In [13], interargument relations express an inequality relation. The symbolic versions of norms, level mappings and interargument relations form the basis for a symbolic termination condition. The condition is formulated as constraints on the introduced symbols. As follows from the rigid acceptability condition of Proposition 3.3, there are three conditions:

- (i) The level mapping must be rigid on $Call(P, S)$.
- (ii) Any introduced interargument relation must be valid.
- (iii) The rigid acceptability condition (1), w.r.t. this level mapping and interargument relations, must hold.

These conditions are translated into a system of constraints on the symbols. A system of constraints identifies sets of suitable norms, level mappings and interargument relations which can be used in the original termination condition of Proposition 3.3. In other words, if a solution for the constraint system exists, termination can be proved.

We adapt the approach of [13] to prove quasi-termination of simply moded well-moded programs w.r.t. sets of simply moded well-moded queries by using Proposition 3.1, and to prove quasi-termination of simply moded well-typed programs w.r.t. sets of simply moded well-typed queries, such that the clauses of the programs have input safe heads, by using Proposition 3.2. Note that these two propositions reason fully at the clause level. In the case of quasi-termination, we have the following conditions on the introduced symbols for norms, level mappings and interargument relations (note the difference with the above listed 3 conditions of [13]).

- (i') The level mapping has to measure all and only input positions in $Call(P, S)$ (to make sure it is finitely partitioning on the call set and to be able to reason fully at the clause level).
- (ii') Any introduced interargument relation must be valid.
- (iii') The weak inequality $|H\psi| \geq |B_i\psi|$ (see Propositions 3.1 and 3.2) must hold (note that we consider *all* body atoms and not only the recursive ones).

Note that in the termination conditions of Propositions 3.1 and 3.2, we worked with correct answer substitutions and not with models as in Proposition 3.3. This difference is however not relevant. With respect to the modifications needed to transform the constraints of [13] into constraints for quasi-termination, the main difference between the two lists of conditions above, lies in the first items; (i) and (i'). The differences between the other two items (namely, (ii) versus (ii') and (iii) versus (iii')) will pose no problems in the modification of the constraint set; the modifications are rather straightforward, and we will not elaborate on them. We show in the following example how the first conditions (i) (for LD-termination) and (i') (for quasi-termination) are translated into symbolic constraints.

Example 3.6 *Let P be the following program*

$$\begin{cases} p(0, 0) & \leftarrow \\ p(f(X), Y) & \leftarrow p(X, Y) \end{cases}$$

with moding $p(In, Out)$ and let S be the set of queries $\{p(0, Y), p(f(0), Y), p(f(f(0)), Y), \dots\}$. Then, P and S are simply moded well-moded. It is easy to see that $Call(P, S) = S$.

First, we introduce symbolic versions for the norm and level mapping. As we already noted above, the symbolic norms and level mappings of [13], differ with the symbolic norms and level mappings of Definition 2.17, because the former ones take into account variables (a variable X is mapped to itself and not to 0). With respect to conditions (i) and (i'), this difference is however not relevant, and we will work with our Definition 2.17 of symbolic norm and level mapping. Let $t \in U_P^E$, then the symbolic norm on t is defined as:

$$\begin{cases} \|t\| = f_0 + f_1 \|t_1\|^S & \text{if } t = f(t_1), \\ \|t\|^S = 0 & \text{if } t = \tilde{X}, \text{ with } X \in Var_P, \\ \|t\|^S = 0 & \text{if } t = c \in Const_P, \end{cases}$$

with $\{f_0, f_1\} = FC(P)$. Let $p(t_1, t_2) \in B_P^E$, then the symbolic level mapping on $p(t_1, t_2)$ is defined as:

$$|p(t_1, t_2)|^S = p_1 \|t_1\|^S + p_2 \|t_2\|^S,$$

with $\{p_1, p_2\} = PC(P)$.

The condition (i) on the level mapping to be rigid on $Call(P, S)$, which is equivalent with the condition on it to measure only input positions in $Call(P, S)$ (as it appears in condition (i')), is expressed as the following constraint on the symbols of $FC(P) \cup PC(P)$:

$$p_2 = 0.$$

Condition (i') also requires that the level mapping is finitely partitioning on $Call(P, S)$, or equivalently that the level mapping measures all input positions. This gives rise to the following constraints on the symbols of $FC(P) \cup PC(P)$:

$$p_1 \neq 0 \quad , \quad f_0 \neq 0 \quad , \quad f_1 \neq 0 \quad .$$

As opposed to Proposition 3.3, Propositions 3.1 and 3.2 take into account some information about the input positions of the call set in their termination condition. Namely, the decreases $|H\psi| \geq |B_i\psi|$ only have to hold for correct answer substitutions ψ such that $(H\psi)^{In}, (B_1\psi)^{In}, \dots, (B_i\psi)^{In} \in Call(P, S)^{In}$. In the following we show that this restriction on ψ is important if we want to prove quasi-termination of a program P w.r.t. a set of queries S , and P is not quasi-terminating w.r.t. the set of all queries.

Example 3.7 Let P be the following program

$$\begin{cases} p(0, 0) & \leftarrow \\ p(f(X), Y) & \leftarrow p(X, Y) \\ p(g(X), Y) & \leftarrow p(g(g(X)), Y) \end{cases}$$

with moding $p(In, Out)$ and let S be the set of queries $\{p(0, Y), p(f(0), Y), p(f(f(0)), Y), \dots\}$. Then, P and S are simply moded well-moded.

It is obvious that P does not quasi-terminate w.r.t. all simply moded well-moded queries, e.g. P doesn't quasi-terminate for the query $p(g(0), Y)$. But, P quasi-terminates w.r.t. the set S (P even LD-terminates w.r.t. S). To prove this, we can apply Proposition 3.1. P has two non-unit clauses, but actually, we only have to consider the first one $p(f(X), Y) \leftarrow p(X, Y)$. This is because there is no substitution ψ such that $p(g(X), Y)\psi$ is input-correct w.r.t. $Call(P, S)$, so it follows from the third condition in Proposition 3.1 that we don't have to consider the last clause $p(g(X), Y) \leftarrow p(g(g(X)), Y)$. Quasi-termination of P w.r.t. S can be proven easily now. Let $|\cdot|$ be the level mapping $|p(t_1, t_2)| = \|t_1\|$, with $\|\cdot\|$ the term-size norm. Then, for every substitution ψ , $|p(f(X), Y)\psi| = \|f(X)\psi\| \geq \|X\psi\| = |p(X, Y)\psi|$. Hence, P quasi-terminates w.r.t. S .

The example suggests that the constraint-based approach of [13] can be refined by taking into account information about the call set (as the example shows this is also important in the context of LD-termination). For instance type-information about the call set can be propagated in the constraints in order to systematically simplify (or even remove) the constraints. This is a topic for future research.

4 LG-TERMINATION

The notion of LG-termination was defined in [14] as follows.

Definition 4.1 (LG-termination) Let P be a program and $S \subseteq B_P^E$. P is LG-terminating w.r.t. S iff for every atom A such that $\tilde{A} \in S$, the LG-forest for $P \cup \{\leftarrow A\}$ is a finite set of finite LG-trees.

It is easy to see that LG-termination of a program P w.r.t. a set of queries S implies quasi-termination of P w.r.t. S .

In [14], in order to characterize LG-termination, the *sol*-transformation was introduced. Here, we present an optimisation of this transformation, namely the *a*-transformation, which will also be used to characterize LG-termination. But, proving LG-termination will require less work using this *a*-transformation than using the *sol*-transformation of [14]. For a program P , we denote with Rec_P the set of recursive predicates of P .

Definition 4.2 (a-transformation)

Let P be a program. Define the *a*-transform P^a of P as follows:

- For a clause $C = H \leftarrow B_1, \dots, B_n$ in P , we define

$$C^a = H \leftarrow B_1, B_1^*, \dots, B_n, B_n^*$$

with $B_i^* = p^a(t_1, \dots, t_m)$ if $B_i = p(t_1, \dots, t_m)$ and $p \simeq Rel(H)$; and $B_i^* = \emptyset$ otherwise.

Here, p^a is a new predicate with the same arity as p . Let $Rec_P^a = \{p^a \mid p \in Rec_P\}$.

- For the program P , we define

$$P^a = \{C^a \mid C \in P\} \cup \{p^a(\overline{X}) \leftarrow \mid p \in Rec_P, \overline{X} \text{ is a linear family of variables}\}.$$

Note that the program of Example 2.2 (subsection 2.1) is the *a*-transform of the program of Example 2.1 (subsection 2.1). We give another simple example.

Example 4.1 Let P be the following program

$$\begin{cases} even(s(X)) & \leftarrow odd(X) \\ odd(s(X)) & \leftarrow even(X) \end{cases}$$

Then, the *a*-transform of P is the program P^a :

$$\begin{cases} even(s(X)) & \leftarrow odd(X), odd^a(X) \\ odd(s(X)) & \leftarrow even(X), even^a(X) \\ even^a(X) & \leftarrow \\ odd^a(X) & \leftarrow \end{cases}$$

It is easy to see that $Call(P, S) = Call(P^a, S) \cap B_P^E$. Also, if we denote with $cas(P, \{p(\overline{t})\})$ the set of computed answer substitutions of $P \cup \{\leftarrow p(\overline{t})\}$, then $cas(P, \{p(\overline{t})\}) = cas(P^a, \{p(\overline{t})\})$ for all $p(\overline{t}) \in B_P^E$. It is important to note that if we have a query $p(\overline{t}) \in B_{Rec_P}^E$ to the program P , then $p(\overline{t})\sigma$ is a computed answer if $p^a(\overline{t})\sigma \in Call(P^a, \{p(\overline{t})\})$. This is in fact the main purpose of the transformation.

Definition 4.3 (LG-acceptability w.r.t. S) Let P be a program and $S \subseteq B_P^E$. P is LG-acceptable w.r.t. S iff there is a level mapping $|\cdot|$ which is finitely partitioning on $Call(P^a, S) \cap B_{Rec_P \cup Rec_P^a}^E$ such that

- for every atom A such that $\tilde{A} \in Call(P, S)$,

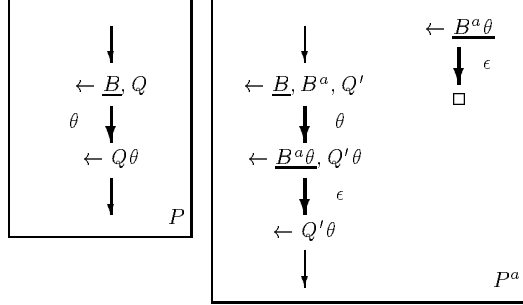


Figure 3: Relating answer clause resolution in P and P^a .

- for every clause $H \leftarrow B_1, \dots, B_n$ in P^a such that $mgu(A, H) = \theta$ exists,
- for every B_i such that $Rel(B_i) \simeq Rel(A)$ or such that $Rel(B_i) \in Rec_P^a$,
- for every LD-computed answer substitution θ_{i-1} in P^a for $\leftarrow (B_1, \dots, B_{i-1})\theta$,

$$|A| \geq |B_i\theta\theta_{i-1}|.$$

In the next Theorem we prove that LG-acceptability w.r.t. S is equivalent with LG-termination w.r.t. S . Here, we will only give a sketch of the proof, which gives the intuition behind this. The full proof together with the necessary notions can be found in Appendix A.

Theorem 4.1 *Let P be a program and $S \subseteq B_P^E$. P is LG-acceptable w.r.t. S iff P is LG-terminating w.r.t. S .*

Proof (sketch).

Consider a query with selected atom $B = p(t_1, \dots, t_n)$ in some derivation in P . Suppose that the atom $B = p(t_1, \dots, t_n)$ was introduced in this query by means of resolution of some atom in an earlier query in the derivation, with a clause $C = H \leftarrow B_1, \dots, B_n$ of P , such that $p = Rel(B_j) \simeq Rel(H)$ and B is descending from B_j . Note that the a -transform P^a of P contains the clause C^a and that on the right of the atom B_j in C^a , there is the atom B_j^a . As can be seen in Figure 3 every answer clause resolution for $B = p(t_1, \dots, t_n)$ in P (with the answer clause $B\theta = p(t_1, \dots, t_n)\theta$) translates into two answer clause resolutions (for $B = p(t_1, \dots, t_n)$ and $B^a = p^a(t_1, \dots, t_n)\theta$) and one new LG-tree (for $B^a\theta = p^a(t_1, \dots, t_n)\theta$) in P^a . Let $A \in S$.

\Rightarrow : Suppose P does not LG-terminate w.r.t. $\{A\}$. We will prove a contradiction with the LG-acceptability of P w.r.t. $\{A\}$. Let \mathcal{F} be the LG-forest for $P \cup \{\leftarrow A\}$. Then one of the following cases holds (both cases will result in a contradiction with the LG-acceptability of P w.r.t. $\{A\}$).

- There is an LG-tree in \mathcal{F} which is infinitely branching.
This means that there is a recursive atom with infinitely many computed answer substitutions. This translates into an infinite number of LG-trees with a query of the form $\leftarrow p^a(t_1, \dots, t_n)$ as root in the LG-forest \mathcal{F}^a for $P^a \cup \{\leftarrow A\}$. But then, there is no level mapping which is finitely partitioning on $Call(P^a, S) \cap B_{Rec_P \cup Rec_P^a}^E$ such that P is LG-acceptable w.r.t. $\{A\}$ and this level mapping.

- \mathcal{F} consists of an infinite number of LG-trees.

A first possible reason for an infinite number of LG-trees in \mathcal{F} is an infinitely branching LG-tree in \mathcal{F} . But we already proved that this does not occur. The other possibility is that there exists an infinite LD-derivation of $\leftarrow A$ in P with an infinite number of different selected atoms of $B_{Rec_P}^E$. But then, there does not exist a level mapping which is finitely partitioning on $Call(P^a, S) \cap B_{Rec_P \cup Rec_P^a}^E$, such that P is LG-acceptable w.r.t. $\{A\}$ and this level mapping.

\Leftarrow : Suppose P is LG-terminating w.r.t. $\{A\}$.

Then, the LG-forest for $P \cup \{\leftarrow A\}$ consists of a finite number of finite LG-trees. Thus, in P^a only finitely many answer clause resolutions and finitely many LG-trees are added. A level mapping can be found which is finitely partitioning on $Call(P^a, S) \cap B_{Rec_P \cup Rec_P^a}^E$, such that P is LG-acceptable w.r.t. S and this level mapping. The construction of this level mapping will be given in the appendix. \square

We have a similar result as Corollary 3.1 of the previous section. First, note that if P is a simply moded program and we define $m_{p^a}(i) = In$ for all $i \in \{1, \dots, n\}$ with $p^a/n \in Rec_P^a$, then, P^a is also simply moded.

Corollary 4.1 *Let P be a simply moded program and $S \subseteq B_P^E$ be a set of simply moded queries. P LG-terminates w.r.t. S iff for every A in S , the set of all terms occurring in input positions of the atoms in $Call(P^a, \{A\})$ is finite.*

In [25] the following result is proven: *if the minimum Herbrand model of a program P is finite, then P SLG-terminates under any search strategy w.r.t. any set of queries.* Note that this result applies to *all* programs P (under any search strategy), but the finiteness of the minimum Herbrand model of P is required.

4.1 Simply Moded Well-Moded Programs and Queries

We reformulate Theorem 4.1 in order to reason fully at the clause level. Recall that all arguments of atoms $p^a \in Rec_P^a$ are defined as input in P^a . Then, if P is a simply moded well-moded program, also P^a is simply moded well-moded. We hence have the following proposition.

Proposition 4.1 *Let P be a simply moded well-moded program and $S \subseteq B_P^E$ be a set of simply moded well-moded queries. Let $|\cdot|$ be a level mapping which measures only the input positions in $Call(P, S)$ and all the input positions in $Call(P^a, S) \cap B_{Rec_P \cup Rec_P^a}^E$, such that*

- for any clause $H \leftarrow B_1, \dots, B_n$ in P^a ,
- for any atom B_i , such that $Rel(B_i) \simeq Rel(H)$ or such that $Rel(B_i) \in Rec_P^a$,
- for any correct answer substitution ψ in P^a for $\leftarrow B_1, \dots, B_{i-1}$, such that $(H\psi)^{In}, (B_1\psi)^{In}, \dots, (B_i\psi)^{In} \in Call(P^a, S)^{In}$,

$$|H\psi| \geq |B_i\psi|,$$

then P is LG-acceptable w.r.t. S . Hence, P LG-terminates w.r.t. S .

Proof Suppose the above condition is satisfied for P . Note that, because P and S are simply moded well-moded, it follows from Propositions 2.2 and 2.3 that $|\cdot|$ is rigid on $Call(P, S)$ and finitely partitioning on $Call(P^a, S) \cap B_{Rec_P \cup Rec_P^a}^E$. Let A be an atom such that $A \in Call(P, S)$ and let $H \leftarrow B_1, \dots, B_n$ be a clause in P^a such that $mgu(A, H) = \theta$ exists. Suppose B_i is such that $Rel(B_i) \simeq Rel(A)$ or such that $Rel(B_i) \in Rec_P^a$. Let θ_{i-1} be an LD-computed answer substitution in P^a for $\leftarrow (B_1, \dots, B_{i-1})\theta$ (in the case $i = 1$, θ_{i-1} is the empty substitution). Then, $\theta\theta_{i-1}$ is a correct answer substitution in P^a for $\leftarrow B_1, \dots, B_{i-1}$. We have to prove that $|A| \geq |B_i\theta\theta_{i-1}|$. Because $(H\theta\theta_{i-1})^{In} = (A\theta\theta_{i-1})^{In} = A^{In}$ (A is a well-moded query, so ground in its input positions), we have that $(H\theta\theta_{i-1})^{In} \in Call(P, S)^{In}$. Also, it is easy to see that $(B_1\theta\theta_{i-1})^{In}, \dots, (B_i\theta\theta_{i-1})^{In} \in Call(P^a, S)^{In}$. Hence, we know that $|H\theta\theta_{i-1}| \geq |B\theta\theta_{i-1}|$. Because $A\theta = H\theta$, $|A\theta\theta_{i-1}| = |H\theta\theta_{i-1}|$. Now, since $A \in Call(P, S)$ and $|\cdot|$ is rigid on $Call(P, S)$, $|A\theta\theta_{i-1}| = |A|$. Thus, $|A| = |H\theta\theta_{i-1}|$, therefore $|A| \geq |B_i\theta\theta_{i-1}|$. \square

Again, we can remove the restriction on ψ in the third condition of the proposition statement (saying that $(H\psi)^{In}, (B_1\psi)^{In}, \dots, (B_i\psi)^{In} \in Call(P, S)^{In}$), when this is not needed in the termination proof.

Example 4.2 Let P be the program of Example 4.1. Consider the moding $even(Out), odd(Out)$ and let S be the set $\{even(X), odd(X)\}$. Then, P and S are simply moded well-moded. We prove that P LG-terminates w.r.t. S . Therefore, we consider the a -transform of P , the program P^a (shown in Example 4.1), with moding $even^a(In), odd^a(In)$. Because there is no correct answer substitution for the queries $odd(X)$ and $even(X)$, we only have to define the level mapping on $even/1$ and $odd/1$. Let $|\cdot|$ be the trivial level mapping (mapping everything to 0). Then $|even(s(X))| = 0 \geq 0 = |odd(X)|$ and $|odd(s(X))| = 0 \geq 0 = |even(X)|$. Hence, P LG-terminates w.r.t. S . Note that P does not LD-terminate w.r.t. S .

Example 4.3 Consider the following program P computing all the reachable nodes from one node in a (cyclic) graph. Notice the similarity with Example 3.3, but here the program P does not compute the paths.

$$\left\{ \begin{array}{ll} edge(a, b) & \leftarrow \\ edge(b, c) & \leftarrow \\ edge(c, a) & \leftarrow \\ edge(c, d) & \leftarrow \\ reach(X, Y) & \leftarrow edge(X, Y) \\ reach(X, Y) & \leftarrow edge(X, Z), reach(Z, Y) \end{array} \right.$$

with moding $edge(In, Out), reach(In, Out)$. Then P is simply moded well-moded. Consider the simply moded well-moded query $reach(a, Y)$.

We prove that P LG-terminates w.r.t this query. Consider the program P^a :

$$\left\{ \begin{array}{ll} edge(a, b) & \leftarrow \\ edge(b, c) & \leftarrow \\ edge(c, a) & \leftarrow \\ edge(c, d) & \leftarrow \\ reach(X, Y) & \leftarrow edge(X, Y) \\ reach(X, Y) & \leftarrow edge(X, Z), reach(Z, Y), reach^a(Z, Y) \\ reach^a(X_1, X_2) & \leftarrow \end{array} \right.$$

with $\text{reach}^a(\text{In}, \text{In})$. Let $|\cdot|$ be the following level mapping (satisfying the conditions of Proposition 4.1):

$$\begin{aligned} |\text{edge}(t_1, t_2)| &= \|t_1\|, \\ |\text{reach}(t_1, t_2)| &= \|t_1\|, \\ |\text{reach}^a(t_1, t_2)| &= \|t_1\| + \|t_2\|, \end{aligned}$$

with $\|\cdot\|$ the term-size norm.

A correct answer substitution ψ for $\leftarrow \text{edge}(X, Z)$ is of the form $\{X/c_1, Z/c_2\}$ with c_1, c_2 constants, so $|\text{reach}(X, Y)\psi| = \|X\psi\| \geq \|Z\psi\| = |\text{reach}(Z, Y)\psi|$. And a correct answer substitution ψ for $\leftarrow \text{edge}(X, Z)$, $\text{reach}(Z, Y)$ is of the form $\{X/c_1, Z/c_2, Y/c_3\}$ with c_1, c_2, c_3 constants, hence $|\text{reach}(X, Y)\psi| = \|X\psi\| \geq \|Z\psi\| + \|Y\psi\| = |\text{reach}^a(Z, Y)\psi|$.

Note that P does not LD-terminate w.r.t. $\{\text{reach}(a, Y)\}$.

We end with an example for which the set of all terms occurring in input positions of S is infinite.

Example 4.4 The following program P computes all the reachable nodes from a given node. As opposed to Example 4.3, here the graph is given as an input parameter. The graph is represented as a list of terms $e(x, y)$, indicating that there is an edge from node x to node y . Notice the similarity with Example 3.4, but here P does not compute the paths.

$$\left\{ \begin{array}{ll} \text{reach}(X, \text{Ed}, X) & \leftarrow \\ \text{reach}(X, \text{Ed}, Y) & \leftarrow \text{member}(X, \text{Ed}, Y) \\ \text{reach}(X, \text{Ed}, Z) & \leftarrow \text{member}(X, \text{Ed}, Y), \text{reach}(Y, \text{Ed}, Z) \\ \text{member}(X, [e(X, Y)|L], Y) & \leftarrow \\ \text{member}(X, [e(X_1, X_2)|L], Y) & \leftarrow \text{member}(X, L, Y) \end{array} \right.$$

with moding $\text{reach}(\text{In}, \text{In}, \text{Out})$ and $\text{member}(\text{In}, \text{In}, \text{Out})$. Then P is simply moded well-moded. Let $C = \{a, b, c, d\}$ be a set of constant symbols. Consider the following (infinite) set of simply moded well-moded queries $S = \{\text{reach}(x, \text{ed}, Y) \mid x \in C \text{ and } \text{ed} \text{ is a nil-terminated list of terms } e(y, z) \text{ with } y, z \in C\}$.

We prove that P LG-terminates w.r.t. S by applying Proposition 4.1. Consider the program P^a :

$$\left\{ \begin{array}{ll} \text{reach}(X, \text{Ed}, X) & \leftarrow \\ \text{reach}(X, \text{Ed}, Y) & \leftarrow \text{member}(X, \text{Ed}, Y) \\ \text{reach}(X, \text{Ed}, Z) & \leftarrow \text{member}(X, \text{Ed}, Y), \text{reach}(Y, \text{Ed}, Z), \\ & \text{reach}^a(Y, \text{Ed}, Z) \\ \text{member}(X, [e(X, Y)|L], Y) & \leftarrow \\ \text{member}(X, [e(X_1, X_2)|L], Y) & \leftarrow \text{member}(X, L, Y), \text{member}^a(X, L, Y) \\ \text{reach}^a(X_1, X_2, X_3) & \leftarrow \\ \text{member}^a(X_1, X_2, X_3) & \leftarrow \end{array} \right.$$

with moding $\text{reach}^a(\text{In}, \text{In}, \text{In})$ and $\text{member}^a(\text{In}, \text{In}, \text{In})$.

Let $|\cdot|$ be the following level mapping:

$$\begin{aligned} |\text{reach}(t_1, t_2, t_3)| &= \|t_1\| + \|t_2\|, \\ |\text{reach}^a(t_1, t_2, t_3)| &= \|t_1\| + \|t_2\| + \|t_3\|, \\ |\text{member}(t_1, t_2, t_3)| &= \|t_1\| + \|t_2\|, \\ |\text{member}^a(t_1, t_2, t_3)| &= \|t_1\| + \|t_2\| + \|t_3\|, \end{aligned}$$

with $\|\cdot\|$ the term-size norm.

Consider the third clause for $\text{reach}/3$. Let ψ be a correct answer substitution for $\leftarrow \text{member}(X, \text{Ed}, Y)$, such that $\text{reach}(X, \text{Ed}, Z)\psi$, $\text{member}(X, \text{Ed}, Y)\psi$

and $\text{reach}(Y, Ed, Z)\psi$ are input-correct w.r.t. $\text{Call}(P^a, S)$. It is easy to see that ψ binds X and Y to a constant in C and Ed to a nil-terminated list of $\epsilon(x, y)$ -terms with $x, y \in C$. So, $|\text{reach}(X, Ed, Z)\psi| = \|X\psi\| + \|Ed\psi\| \geq \|Y\psi\| + \|Ed\psi\| = |\text{reach}(Y, Ed, Z)\psi|$.

Now, concerning the reach^a -atom in the body, let ψ be a correct answer substitution for $\leftarrow \text{member}(X, Ed, Y)$, $\text{reach}(Y, Ed, Z)$, such that $\text{reach}(X, Ed, Z)\psi$, $\text{member}(X, Ed, Y)\psi$, $\text{reach}(Y, Ed, Z)\psi$ and $\text{reach}^a(Y, Ed, Z)\psi$ are input-correct w.r.t. $\text{Call}(P^a, S)$. It is easy to see that ψ binds X , Y and Z to a constant in C and Ed to a nil-terminated list of $\epsilon(x, y)$ -terms with $x, y \in C$. So, $|\text{reach}(X, Ed, Z)\psi| = \|X\psi\| + \|Ed\psi\| \geq \|Y\psi\| + \|Ed\psi\| + \|Z\psi\| = |\text{reach}^a(Y, Ed, Z)\psi|$.

Consider the recursive clause for $\text{member}/3$. Let ψ be a substitution. Then, $|\text{member}(X, [e(X_1, X_2)|L], Y)\psi| = \|X\psi\| + \|[e(X_1, X_2)|L]\psi\| \geq \|X\psi\| + \|L\psi\| = |\text{member}(X, L, Y)\psi|$.

Next, let ψ be a correct answer substitution for $\leftarrow \text{member}(X, L, Y)$, such that $\text{member}(X, [e(X_1, X_2)|L], Y)\psi$ and $\text{member}(X, L, Y)\psi$ are input-correct w.r.t. $\text{Call}(P^a, S)$. It is easy to see that ψ binds X, X_1, X_2, Y to constants in C and L to a nil-terminated list of $\epsilon(x, y)$ -terms with $x, y \in C$. Hence, we have that $|\text{member}(X, [e(X_1, X_2)|L], Y)\psi| = \|X\psi\| + \|[e(X_1, X_2)|L]\psi\| \geq \|X\psi\| + \|L\psi\| + \|Y\psi\| = |\text{member}^a(X, L, Y)\psi|$.

Note that P does not LD-terminate w.r.t. S .

4.2 Simply Moded Well-Typed Programs and Queries

In order to deal with non-ground input, we consider in this subsection simply moded well-typed programs and queries. Again, we put for predicates $p^a/n \in \text{Rec}_P^a$, $m_{p^a}(i) = In$, $i \in \{1, \dots, n\}$. Concerning the typing of p^a , we put $t_{p^a}(i) = t_p(i)$, $i \in \{1, \dots, n\}$. Then, it is easy to see that if P^a is simply moded well-typed, also P is simply moded well-typed. Also, it is easy to see that if the heads of the clauses in P are input safe, then the heads of the clauses in P^a are input safe.

Proposition 4.2 *Let P be a simply moded well-typed program and $S \subseteq B_P^E$ be a set of simply moded well-typed queries. Suppose that the heads of the clauses in P are input safe. Let $|\cdot|$ be a level mapping which measures only the input positions in $\text{Call}(P, S)$ and all the input positions in $\text{Call}(P^a, S) \cap B_{\text{Rec}_P \cup \text{Rec}_P^a}^E$ and such that*

- for any clause $H \leftarrow B_1, \dots, B_n$ in P^a ,
- for any atom B_i , such that $\text{Rel}(B_i) \simeq \text{Rel}(H)$ or such that $\text{Rel}(B_i) \in \text{Rec}_P^a$,
- for any correct answer substitution ψ in P^a for $\leftarrow B_1, \dots, B_{i-1}$, such that $(H\psi)^{In}, (B_1\psi)^{In}, \dots, (B_i\psi)^{In} \in \text{Call}(P^a, S)^{In}$,

$$|H\psi| \geq |B_i\psi|,$$

then P is LG-acceptable w.r.t. S . Hence, P LG-terminates w.r.t. S .

Proof Let P, S and $|\cdot|$ satisfy the condition of the proposition. Because P and S are simply moded, and $|\cdot|$ measures all input positions in $\text{Call}(P^a, S) \cap B_{\text{Rec}_P \cup \text{Rec}_P^a}^E$, $|\cdot|$ is finitely partitioning on $\text{Call}(P^a, S) \cap B_{\text{Rec}_P \cup \text{Rec}_P^a}^E$ (Proposition 2.3). We prove that P is LG-acceptable w.r.t. S . Let A be an atom such that $\tilde{A} \in \text{Call}(P, S)$. Let $H \leftarrow B_1, \dots, B_n$ be a clause in P^a such

that $\text{mgu}(A, H) = \theta$ exists. Let B_i be an atom such that $\text{Rel}(B_i) \simeq \text{Rel}(H)$ or such that $\text{Rel}(B_i) \in \text{Rec}_P^a$. Let θ_{i-1} be an LD-computed answer substitution in P^a for $\leftarrow (B_1, \dots, B_{i-1})\theta$. We have to prove that $|A| \geq |B_i\theta\theta_{i-1}|$. The substitution $\theta\theta_{i-1}$ is a correct answer substitution for $\leftarrow B_1, \dots, B_{i-1}$ in P^a . From Corollary 2.2, we know that the input arguments of A are equal to the input arguments of $A\theta\theta_{i-1}$ modulo variable renaming. Because $A\theta\theta_{i-1} = H\theta\theta_{i-1}$, we have that $(H\theta\theta_{i-1})^{In} \in \text{Call}(P, S)^{In}$. Because of Proposition 2.1, we know that $(B_1\theta\theta_{i-1})^{In}, \dots, (B_{i-1}\theta\theta_{i-1})^{In} \in \text{Call}(P^a, S)^{In}$. Trivially, $(B_i\theta\theta_{i-1})^{In} \in \text{Call}(P^a, S)^{In}$. Hence, $|H\theta\theta_{i-1}| \geq |B_i\theta\theta_{i-1}|$. Since $|\cdot|$ measures only input positions in $\text{Call}(P, S)$, $|A| = |H\theta\theta_{i-1}|$, hence $|A| \geq |B_i\theta\theta_{i-1}|$. \square

Again, Proposition 4.1 of the previous subsection, which gives a condition for LG-termination of simply moded well-moded programs w.r.t. a set of simply moded well-moded queries, is a corollary of the previous Proposition 4.2.

Example 4.5 *Let P be the following program computing the cyclic permutations of a list.*

$$\begin{array}{ll} \text{splitlast}([X], [], X) & \leftarrow \\ \text{splitlast}([X|L], [X|R], Y) & \leftarrow \text{splitlast}(L, R, Y) \\ \text{cycperm}(L, L) & \leftarrow \\ \text{cycperm}(L, R) & \leftarrow \text{splitlast}(L, T, X), \text{cycperm}([X|T], R) \end{array}$$

with the typing $\text{splitlast}(\langle In : List \rangle, \langle Out : List \rangle, \langle Out : U \rangle)$ and $\text{cycperm}(\langle In : List \rangle, \langle Out : List \rangle)$. Let S be the set of queries $\{\text{cycperm}(l, X) \mid l \in List\}$. We prove that P LG-terminates w.r.t. S .

First, note that P and S are well-typed. This is trivial to see for the set S . The program P is well-typed because the following typed judgements are true:

$$\begin{array}{ll} [X] : List & \Rightarrow [] : List \wedge X : U \\ [X|L] : List & \Rightarrow L : List \\ [X|L] : List \wedge R : List \wedge Y : U & \Rightarrow [X|R] : List \wedge Y : U \\ L : List & \Rightarrow L : List \\ L : List & \Rightarrow L : List \\ L : List \wedge T : List \wedge X : U & \Rightarrow [X|T] : List \\ L : List \wedge T : List \wedge X : U \wedge R : List & \Rightarrow R : List \end{array}$$

Also, P and S are simply moded and the heads of the clauses in P are input safe. Consider the program P^a :

$$\begin{array}{ll} \text{splitlast}([X], [], X) & \leftarrow \\ \text{splitlast}([X|L], [X|R], Y) & \leftarrow \text{splitlast}(L, R, Y), \text{splitlast}^a(L, R, Y) \\ \text{cycperm}(L, L) & \leftarrow \\ \text{cycperm}(L, R) & \leftarrow \text{splitlast}(L, T, X), \text{cycperm}([X|T], R), \\ & \text{cycperm}^a([X|T], R) \\ \text{splitlast}^a(X_1, X_2, X_3) & \leftarrow \\ \text{cycperm}^a(X_1, X_2) & \leftarrow \end{array}$$

with $\text{splitlast}^a(\langle In : List \rangle, \langle In : List \rangle, \langle In : U \rangle)$ and $\text{cycperm}^a(\langle In : List \rangle, \langle In : List \rangle)$. As already mentioned at the beginning of this subsection, with this moding and typing, P^a is also simply moded well-typed, and the heads of the clauses of P^a are also input safe.

Let $|\cdot|$ be the following level mapping (measuring all and only the input positions in $\text{Call}(P^a, S)$):

$$\begin{array}{ll} |\text{splitlast}(t_1, t_2, t_3)| & = 2\|t_1\|, \\ |\text{splitlast}^a(t_1, t_2, t_3)| & = \|t_1\| + \|t_2\| + \|t_3\|, \\ |\text{cycperm}(t_1, t_2)| & = 2\|t_1\| \\ |\text{cycperm}^a(t_1, t_2)| & = \|t_1\| + \|t_2\|, \end{array}$$

with $\| \cdot \|$ the term-size norm.

Consider the recursive clause for *splitlast* in P^a . Obviously, for every ψ , $|\text{splitlast}([X|L], [X|R], Y)\psi| = 2\| [X|L]\psi \| \geq 2\| L\psi \| = |\text{splitlast}(L, R, Y)\psi|$.

Next, suppose ψ is a correct answer substitution for $\leftarrow \text{splitlast}(L, R, Y)$ such that $\text{splitlast}([X|L], [X|R], Y)\psi$, $\text{splitlast}(L, R, Y)\psi$ and $\text{splitlast}^a(L, R, Y)\psi$ are input-correct w.r.t. $\text{Call}(P^a, S)$. One can see that then, $\| L\psi \| \geq 1 + \| Y\psi \| + \| R\psi \|$. Hence, $|\text{splitlast}([X|L], [X|R], Y)\psi| = 2\| [X|L]\psi \| \geq \| L\psi \| + \| R\psi \| + \| Y\psi \| = |\text{splitlast}^a(L, R, Y)\psi|$. Note that the program consisting of the two clauses for *splitlast* even LD-terminates w.r.t. any set of simply moded well-typed queries.

Consider the recursive clause for *cycperm*/2. Let ψ be a correct answer substitution for $\leftarrow \text{splitlast}(L, T, X)$ such that $\text{cycperm}(L, R)\psi$, $\text{splitlast}(L, T, X)\psi$ and $\text{cycperm}([X|T], R)\psi$ are input-correct w.r.t. $\text{Call}(P^a, S)$. Again, because in that case $\| L\psi \| \geq 1 + \| X\psi \| + \| T\psi \| = \| [X|T]\psi \|$, $|\text{cycperm}(L, R)\psi| = 2\| L\psi \| \geq 2\| [X|T]\psi \| = |\text{cycperm}([X|T], R)\psi|$.

Next, suppose that ψ is a correct answer substitution for $\leftarrow \text{splitlast}(L, T, X)$, $\text{cycperm}([X|T], R)$ such that the atoms $\text{cycperm}(L, R)\psi$, $\text{splitlast}(L, T, X)\psi$, $\text{cycperm}([X|T], R)\psi$, $\text{cycperm}^a([X|T], R)\psi$ are input-correct w.r.t. $\text{Call}(P^a, S)$. Again we know that then $\| L\psi \| \geq 1 + \| X\psi \| + \| T\psi \| = \| [X|T]\psi \|$, we can also see that $\| [X|T]\psi \| \geq \| R\psi \|$. Hence, $|\text{cycperm}(L, R)\psi| = 2\| L\psi \| \geq \| [X|T]\psi \| + \| R\psi \| = |\text{cycperm}^a([X|T], R)\psi|$.

Hence, P LG-terminates w.r.t. S . Note that P does not LD-terminate w.r.t. S .

4.3 Constraint-based Approach for Automatically Proving LG-Termination

Similar as in subsection 3.3 in the context of quasi-termination, we can also modify the constraint-based approach of [13] in order to automatically prove LG-termination. We refer to subsection 3.3 where the main ideas of [13] are given.

The approach of [13] can be modified to prove LG-termination of simply moded well-moded programs w.r.t. sets of simply moded well-moded queries by using Proposition 4.1, and to prove LG-termination of simply moded well-typed programs w.r.t. sets of simply moded well-typed queries, such that the clauses of the programs have input safe heads, by using Proposition 4.2. Note that these two propositions reason fully at the clause level. In the case of LG-termination, we have the following constraints on the introduced symbols for norms, level mappings and interargument relations. Note that the a -transform P^a of P is considered in the analysis.

(i'') The level mapping has to measure only input positions in $\text{Call}(P, S)$ (to be able to reason fully at the clause level) and all input positions in $\text{Call}(P^a, S) \cap B_{Rec_P \cup Rec_P^a}^E$ (so that the level mapping is finitely partitioning on this set).

(ii'') Any introduced interargument relation must be valid.

(iii'') The weak inequality $|H\psi| \geq |B_i\psi|$ (see Propositions 4.1 and 4.2) must hold (note that we only have to consider recursive body atoms and body atoms which belong to $B_{Rec_P^a}^E$).

Similar as in subsection 3.3 in the context of quasi-termination, also here, the only non-straightforward modification of the symbolic constraints for LD-termination into constraints for LG-termination, lies in the first condition (i'')

(versus (i) , see subsection 3.3). We refer to subsection 3.3, where an example is given which illustrates the modifications needed in the case of quasi-termination. The modifications for LG-termination are of similar nature.

As we already noted in subsection 3.3, it remains to be studied how to propagate information about the call set into the constraints in order to simplify them.

5 CONCLUSION AND RELATED WORKS

In this paper, we studied termination under tabled execution mechanism (as introduced in [14]) for simply moded well-moded programs and simply moded well-typed programs.

The basis of this paper is the work of [14]. There, the notions of quasi-termination and LG-termination are introduced and necessary and sufficient conditions (quasi-acceptability) are given. However, the rigidity condition on the level mapping, as it appears in automated termination analysis, is difficult to combine with the condition on it to be finitely partitioning, as it appears in the quasi-acceptability condition. In this paper, we showed that for simply moded well-moded programs, these two conditions can be combined easily and can be fit in the constraint-based automatic termination analysis of [13]. Also, we showed that for simply moded well-typed programs such that the heads of the clauses are input-safe, a sufficient condition for quasi-termination and for LG-termination can be given, which is fully at the clause level and easy to automatize.

Since all programs that terminate under LD-resolution, are quasi-terminating and LG-terminating as well, verification of termination under LD-resolution using an existing automated termination analysis (such as those surveyed in e.g. [11]) is a sufficient proof of the programs quasi-termination and LG-termination. In the recent paper [23], Etalle et al. study how mode information can be used for characterizing properties of LD-termination. They define and study the class of well-terminating programs, i.e. programs for which all well-moded queries have finite LD-derivations. They introduce the notion of well-acceptability and show that for well-moded programs, well-acceptability implies well-termination. They prove that for simply moded well-moded programs, the notions of well-acceptability and well-termination are equivalent.

Termination proofs for (S)LD-resolution are sufficient to prove termination under tabled execution mechanism, but, since there are quasi-terminating and LG-terminating programs, which are not LD-terminating, better proof techniques can be found. There are only relatively few works studying termination under tabled execution mechanism. We already discussed the work of [14], which forms the basis of this paper. In [21], in the context of well-moded programs, a sufficient condition is given for the bounded term-size property, which implies LG-termination. [17] provides another sufficient condition for quasi-termination in the context of functional programming.

A topic for future research is to implement the constraint-based technique for automatically proving quasi-termination and LG-termination. Also, it is worth investigating how and which information about the call set can be propagated into the constraints in order to simplify them and hence, enabling us to automatically prove termination in more cases (than with the original (stronger) constraints). It should also be investigated how the rather strong condition on a level mapping in order for it to be finitely partitioning on a call set (namely that it should measure *all* input positions) can be weakened in case we have for instance some type-information about the call set at our disposal. Also, it remains to be studied how our results can be extended to automatically prove

quasi-termination and LG-termination for a larger class of programs and queries (i.e. for programs and queries which are not simply moded well-typed). Finally, the study of termination of *normal* logic programs under tabled execution mechanism is an interesting topic for future research.

A PROOF OF THEOREM 4.1

The following concepts will be useful for proving the termination condition of Theorem 4.1.

Definition A.1 (direct descendant) Let P be a program and $\tilde{A}, \tilde{B} \in B_P^E$. We call \tilde{B} a direct descendant of \tilde{A} iff

- there exists a clause $H \leftarrow B_1, \dots, B_n$ in P such that $\text{mgu}(A, H) = \theta$ exists,
- there is an $i \in [1, n]$ such that there is an LD-refutation for $\leftarrow (B_1, \dots, B_{i-1})\theta$ with computed answer substitution θ_{i-1} and $B \approx B_i\theta_{i-1}$.

Definition A.2 (directed subsequence of an LD-derivation) Let P be a program and $\tilde{A} \in B_P^E$. Let $\leftarrow A = G_0, G_1, \dots$ be an LD-derivation of $\leftarrow A$ in P . A subsequence G_{i_0}, G_{i_1}, \dots , with $G_{i_j} = \leftarrow A_{i_j}, \mathbf{B}_{i_j}$, is called a directed subsequence iff for all $j \geq 0$, $\tilde{A}_{i_{j+1}}$ is a direct descendant of \tilde{A}_{i_j} in the LD-derivation.

Definition A.3 (call graph associated to S) Let P be a program and $S \subseteq B_P^E$. The call graph $\text{Call-Gr}(P, S)$ associated to P and S is a graph such that:

- its set of nodes is $\text{Call}(P, S)$,
- there exists a directed arc from \tilde{A} to \tilde{B} iff \tilde{B} is a direct descendant of \tilde{A} .

In [14, Proposition A.1], it was noted that the notion of a call graph has a particularly interesting property, which is useful in the study of termination. We recall this proposition together with its proof.

Proposition A.1 (paths and selected atoms) Let P be a program, $S \subseteq B_P^E$, $\text{Call}(P, S)$ and $\text{Call-Gr}(P, S)$ be defined as above. Let p be any directed path in $\text{Call-Gr}(P, S)$. Then there exists an LG-derivation for some element of $\text{Call}(P, S)$, such that all the nodes in p occur as selected atoms in the derivation.

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

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

We can now prove Theorem 4.1.

Proof (of Theorem 4.1).

\Rightarrow : Suppose P is LG-acceptable w.r.t. S and a level mapping $|\cdot|$. Let A be an atom such that $\tilde{A} \in S$. Let \mathcal{F} be the LG-forest of $P \cup \{\leftarrow A\}$. We prove that \mathcal{F} consists of a finite number of finite LG-trees.

- The LG-trees in \mathcal{F} are finitely branching.
 Suppose this is not the case, i.e. there is an LG-tree in \mathcal{F} which is infinitely branching. Then, there is an LG-tree \mathcal{T} in \mathcal{F} which is infinitely branching in a non-root node, which is a query with left-most atom $p(t_1, \dots, t_n)$, with $p \in \text{Rec}_P$, which is directly descending from an atom $q(s_1, \dots, s_m)$, with $p \simeq q$, via a recursive clause $C = q(u_1, \dots, u_m) \leftarrow \dots, p(v_1, \dots, v_n), \dots$. Now, consider the LG-forest \mathcal{F}^a of $P^a \cup \{\leftarrow A\}$. Let \mathcal{T}^a be the LG-tree in \mathcal{F}^a corresponding to \mathcal{T} . Note that the clause C^a instead of C is used in \mathcal{T}^a . Because of this, the atom on the right of $p(t_1, \dots, t_n)$ in the infinitely branching node is $p^a(t_1, \dots, t_n)$. Thus, \mathcal{F}^a consists of a infinite number of LG-trees (there are an infinite number of LG-trees with predicate p^a in the root). But, all these p^a -atoms directly descend from the node $q(s_1, \dots, s_m)$ via the clause C^a in P^a and hence, their value under the level mapping $|\cdot|$ is smaller or equal to $|q(s_1, \dots, s_m)|$. Because $|\cdot|$ is finitely partitioning on $\text{Call}(P^a, S) \cap B_{\text{Rec}_P^a}^E$, this gives a contradiction.
- \mathcal{F} consists of a finite number of LG-trees, i.e. $\#\text{Call}(P, \{A\}) < \infty$.
 Suppose this is not the case. A first possible reason for an infinite number of LG-trees in \mathcal{F} is an infinitely branching LG-tree in \mathcal{F} . But we already proved that this does not occur. The other possibility is that there exists an infinite LD-derivation of $\leftarrow A$ in P which contains an infinite directed subsequence, such that this infinite directed subsequence has a tail G_n, G_{n+1}, \dots with $G_i = \leftarrow A_i, \mathbf{B}_i, i \geq n$, such that $\{A_i \mid i \geq n\}$ is an infinite set and $\text{Rel}(A_i) \simeq \text{Rel}(A_{i+1})$ for all $i \geq n$. Since $A_i \in \text{Call}(P, S) \cap B_{\text{Rec}_P}^E \subseteq \text{Call}(P^a, S) \cap B_{\text{Rec}_P \cup \text{Rec}_P^a}^E$, and since $|\cdot|$ is finitely partitioning on this set and $|A_i| \geq |A_{i+1}|$ for all $i \geq n$ (by the LG-acceptability condition), this gives a contradiction.

\Leftarrow : Suppose P is LG-terminating w.r.t. S . We need to design a level mapping $|\cdot|$, which is finitely partitioning on $\text{Call}(P^a, S) \cap B_{\text{Rec}_P \cup \text{Rec}_P^a}^E$ and such that P is LG-acceptable w.r.t. S and $|\cdot|$. The proof of this part is based on a similar proof in [14, Proof of Theorem 3.1].

In order to define $|\cdot|$ on $\text{Call}(P^a, S)$, consider the $\text{Call-Gr}(P^a, S)$ -graph.

First, note that this graph is finitely branching. To see this, suppose this is not so. Then, there is an atom in $\text{Call}(P, S) \cap B_{\text{Rec}_P}^E$ which has infinitely many answers. This contradicts LG-termination of P w.r.t. S .

Another crucial point in this part of the proof is that the strongly connected components of $\text{Call-Gr}(P^a, S)$ are necessarily all finite. To see this, suppose this is not the case. Then there is an infinitely long path p , starting from an element in S , through elements of this infinite strongly connected component. Note that this infinitely long path p contains only atoms from B_P^E (since atoms from $B_{\text{Rec}_P^a}^E$ don't depend on any other atom). By Proposition A.1, this means that there exists a derivation for that element in S , such that an infinite number of different atoms from $\text{Call}(P^a, S) \cap B_P^E = \text{Call}(P, S)$ are selected. Obviously, this contradicts LG-termination.

So, all strongly connected components of $Call-Gr(P^a, S)$ are finite. Define $Call-Gr(P^a, S)/reduced$ as the graph obtained from $Call-Gr(P^a, S)$ by replacing any strongly connected component by a single contracting node and replacing any arc from $Call-Gr(P^a, S)$ pointing to (resp. from) any node in that strongly connected component by an arc to (resp. from) that contracting node. $Call-Gr(P^a, S)/reduced$ does not have any (non-trivial) strongly connected components. Moreover, any strongly connected component from $Call-Gr(P^a, S)$ that was collapsed into a contracting node of $Call-Gr(P^a, S)/reduced$ necessarily consists of only a finite number of nodes.

Note now that each path in $Call-Gr(P^a, S)/reduced$ which is not cyclic (note that there are only trivial cycles in $Call-Gr(P^a, S)/reduced$) is finite. This also follows from Proposition A.1.

So, we can consider the layers of $Call-Gr(P^a, S)/reduced$, and there are only a finite number of layers. We define $|\cdot|$ as follows. Let layer-0 be the set of leaves in $Call-Gr(P^a, S)/reduced$. We assign to these nodes a number within \mathbb{N} such that no infinite set of atoms from this layer is assigned the same number.

Then, we move up to the next layer in $Call-Gr(P^a, S)/reduced$. This layer, layer-1, consists of all nodes $Node$ such that $\max(\{length(p) \mid p \text{ is a path starting from } Node \text{ in } Call-Gr(P^a, S)/reduced\}) = 1$. To any element of layer-1, we assign a natural number n , such that n is larger than all the natural numbers assigned to descendants of the element (note that the graph is finitely branching).

We continue this process layer by layer. The value of the level mapping $|\cdot|$ on elements of $Call(P^a, S)$ is defined as follows: all calls in the same strongly connected components of $Call(P^a, S)$ receive the number assigned to their representative in $Call-Gr(P^a, S)/reduced$.

We prove that P is LG-acceptable w.r.t. S and this level mapping $|\cdot|$.

- $|\cdot|$ is finitely partitioning on $Call(P^a, S) \cap B_{Rec_P \cup Rec_P^a}^E$.

We even have that $|\cdot|$ is finitely partitioning on $Call(P^a, S)$. This is because each strongly connected component of $Call(P^a, S)$ is finite and because of the construction of $|\cdot|$ (there are only a finite number of layers in $Call-Gr(P^a, S)/reduced$ and no infinite set of nodes in the same layer is assigned the same natural number).

- Let A be an atom such that $\tilde{A} \in Call(P, S)$, let $H \leftarrow B_1, \dots, B_n$ be a clause in P^a , such that $mgu(A, H) = \theta$ exists, let B_i be such that $Rel(B_i) \simeq Rel(A)$ or $Rel(B_i) \in Rec_P^a$, let θ_{i-1} be an LD-computed answer substitution for $\leftarrow (B_1, \dots, B_{i-1})\theta$, then we have to prove that $|A| \geq |B_i\theta\theta_{i-1}|$. This follows immediately because there is a directed arc from A to $B_i\theta\theta_{i-1}$ in $Call-Gr(P^a, S)$ and because of the construction of $|\cdot|$.

□

References

- [1] K. R. Apt and S. Etalle. On the unification free prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS 93)*, pages 1–19. Lecture Notes in Computer Science, Springer-Verlag, 1993.
- [2] K. R. Apt and A. Pellegrini. On the occur-check free prolog programs. *ACM Toplas*, 16(3):687–726, 1994.
- [3] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of theoretical computer science, Vol. B*. Elsevier Science Publishers, 1990.
- [4] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.
- [5] K.R. Apt and D. Pedreschi. Studies in pure Prolog: termination. In *Proceedings Esprit symposium on computational logic*, pages 150–176, Brussels, november 1990. Springer-Verlag.
- [6] R. Bol and L. Degerstedt. The underlying search for magic templates and tabulation. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 793–811, Budapest, Hungary, june 1993. The MIT Press.
- [7] A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. In *Proc. CCPSD-TAPSOFT'91*, pages 153–180. Springer-Verlag, LNCS 494, 1991.
- [8] A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science*, 124(2):297–328, februari 1994.
- [9] F. Bronsard, T.K. Lakshman, and U.S. Reddy. A framework of directionality for proving termination of logic programs. In Krzysztof Apt, editor, *Proc. JICSLP '92*, pages 321–335. MIT Press, 1992.
- [10] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, january 1996.
- [11] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19 & 20:199–260, may/july 1994.
- [12] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In *Proc. FGCS'92*, pages 481–488, ICOT Tokyo, 1992. ICOT.
- [13] S. Decorte and D. De Schreye. Demand-driven and constraint-based automatic termination analysis for logic programs. In L. Naish, editor, *Proc. 14th International Conference on Logic Programming*, pages 78–92, Leuven, Belgium, july 1997.
- [14] S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K. Sagonas. Termination analysis for tabled logic programs. In *Proceedings of LOPSTR'97: Logic Program Synthesis and Transformation*, Leuven, Belgium, july 1997.
- [15] P. Dembinski and J. Maluszynski. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the International Symposium on Logic Programming*, pages 29–38, Boston, 1985.
- [16] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modelling of the operational behaviour of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
- [17] C. K. Holst. Finiteness Analysis. In John Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA)*, number 523 in LNCS, pages 473–495. Springer-Verlag, august 1991.
- [18] T. Kanamori and T. Kawamura. OLDT-based abstract interpretation. *Journal of Logic Programming*, 15(1 & 2):1–30, januari 1993.

- [19] J.-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of deductive databases and logic programming*, pages 587–625. Morgan-Kaufman, 1988.
- [20] J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, 1987.
- [21] L. Plümer. *Termination proofs for logic programs*. Number 446 in LNAI. Springer-Verlag, 1990.
- [22] D. Rosenblueth. Chart parsers as proof procedures for fixed-mode logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1125–1132, ICOT, Japan, 1992. Association for Computing Machinery.
- [23] A. Bossi S. Etalle and N. Cocco. Termination of well-moded programs. *Journal of Logic Programming*, 38(2):243–257, 1998.
- [24] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.
- [25] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proceedings ICLP'86*, Lecture Notes in Computer Science 225, pages 84–98. Springer Verlag, 1986.
- [26] L. Vieille. Recursive query processing: the power of logic. *Theoretical computer science*, 69(1):1–53, 1989.