Modular Termination Proofs for Prolog with Tabling

Sofie Verbaeten[1], Konstantinos Sagonas*[2], and Danny De Schreye[1]

[1] Department of Computer Science, K.U.Leuven, Belgium
e-mail: {sofie,dannyd}@cs.kuleuven.ac.be
[2] Computing Science Department, Uppsala Universitet, Sweden
e-mail: kostis@csd.uu.se

Abstract. Tabling avoids many of the shortcomings of SLD(NF) execution and provides a more flexible and efficient execution mechanism for logic programs. In particular, tabled execution of logic programs terminates more often than execution based on SLD-resolution. One of the few works studying termination under a tabled execution mechanism is that of Decorte *et al.* They introduce and characterise two notions of universal termination of logic programs w.r.t. sets of queries executed under SLG-resolution, using the left-to-right selection rule; namely the notion of quasi-termination and the (stronger) notion of LG-termination. This paper extends the results of Decorte *et al* in two ways: (1) we consider a mix of tabled and Prolog execution, and (2) besides a characterisation of the two notions of universal termination under such a mixed execution, we also give modular termination conditions. From both practical and efficiency considerations, it is important to allow tabled and non-tabled predicates to be freely intermixed. This motivates the first extension. Concerning the second extension, it was already noted in the literature in the context of termination under SLD-resolution (by e.g. Apt and Pedreschi), that it is important for programming in the large to have modular termination proofs, i.e. proofs that are capable of combining termination proofs of separate programs to obtain termination proofs of combined programs.

1 Introduction

The extension of SLD-resolution with a tabling mechanism [4, 15, 18], avoids many of the shortcomings of SLD(NF) execution and provides a more flexible and often considerably more efficient execution mechanism for logic programs. In particular, tabled execution terminates more often than execution based on SLD. So, if a program and query can be proven terminating under SLD-resolution (by one of the existing techniques surveyed in [5]), then they will also trivially terminate under SLG-resolution, the resolution principle of tabulation [4]. However, since there are programs and queries which terminate under SLG-resolution and not under SLD-resolution, more effective proof techniques can be found. This paper is one of the few works studying termination of tabled logic programs.

* This research was conducted during the author's stay at the K.U.Leuven, Belgium.

We base our approach on the work of Decorte *et al* [8]. There, two notions of universal termination of a tabled logic program w.r.t. a set of queries executed under SLG-resolution using the left-to-right selection rule (called LG-resolution in the sequel) are introduced and characterised. Namely, the notion of quasi-termination and the (stronger) notion of LG-termination. We extend the results of [8] in two ways: (1) we consider a mix of LG-resolution and LD-resolution, i.e. a mix of tabled and Prolog execution, and (2) besides a characterisation of the two notions of universal termination under such a mixed execution schema, we also give modular termination conditions, i.e. conditions on two programs P and R, where P extends R, ensuring termination of the union $P \cup R$. The motivation for extension (1) will be given in the next section. There, examples from context-free grammar recognition and parsing are given, which show that, from the point of view of efficiency, it is important to allow tabled and non-tabled predicates to be freely intermixed. Extension (2) was already motivated in the literature in the context of termination under SLD-resolution (see for instance [3]). Indeed, it is important for programming in the large, to have modular termination proofs, i.e. proofs that are capable of combining termination proofs of separate programs to obtain termination proofs of combined programs.

The rest of the paper is structured as follows: In the next section we present examples which motivate the need to freely mix tabled and Prolog execution. In Section 3, we introduce some necessary concepts and present a definition of SLG-resolution. Section 4 introduces a first notion of termination under tabled evaluation: *quasi-termination*. In addition, a characterisation for quasi-termination is given which generalises the characterisation given in [8] to the case where Prolog and tabled execution are intermixed. Then, a modular termination condition is given for the quasi-termination of the union $P \cup R$ of two programs P and R, where P extends R. In Section 5, the stronger notion of *LG-termination* is defined and characterized, and a method for obtaining modular proofs for LG-termination is presented. We conclude with discussing related and future work. We refer to the full version of the paper, [17], for more results and examples and for the proofs of all the theorems, propositions and statements made here.

2 Mixing Tabled and Prolog Execution: some motivation

It has long been noted in the literature that tabled evaluation can be used for context-free grammar recognition and parsing: tabling eliminates redundancy and handles grammars that would otherwise infinitely loop under Prolog-style execution (e.g. left recursive ones). The program of Fig. 1 where all predicates are tabled, provides such an example. This grammar, recognizing arithmetic expressions containing additions and multiplications over the integers, is left recursive — left recursion is used to give the arithmetic operators their proper associativity — and would be non-terminating for Prolog-style execution. Under tabled execution, left recursion is handled correctly. In fact, one only needs to table predicates $expr/2$ and $term/2$ to get the desired termination behaviour; we can and will safely drop the tabling of $primary/2$ in the sequel.

$$
\begin{array}{ll}
expr(Si, So) & \leftarrow expr(Si, S1), S1 = ['+'|S2], term(S2, So) \\
expr(Si, So) & \leftarrow term(Si, So) \\
term(Si, So) & \leftarrow term(Si, S1), S1 = ['*'|S2], primary(S2, So) \\
term(Si, So) & \leftarrow primary(Si, So) \\
primary(Si, So) & \leftarrow Si = ['('|S1], expr(S1, S2), S2 = [')'|So] \\
primary(Si, So) & \leftarrow Si = [I|So], integer(I)
\end{array}
$$

Fig. 1. A tabled program recognizing simple arithmetic expressions.

To see why a mix of tabled with Prolog execution is desirable in practice, suppose that we want to extend the above recognition grammar to handle exponentiation. The most natural way to do so is to introduce a new nonterminal, named *factor*, for handling exponentiation and make it right recursive, since the exponentiation operator is right associative. The resulting grammar is as below where only the predicates $expr/2$ and $term/2$ are tabled. Note that, at least

$$
\begin{array}{ll}
expr(Si, So) & \leftarrow expr(Si, S1), S1 = ['+'|S2], term(S2, So) \\
expr(Si, So) & \leftarrow term(Si, So) \\
term(Si, So) & \leftarrow term(Si, S1), S1 = ['*'|S2], factor(S2, So) \\
term(Si, So) & \leftarrow factor(Si, So) \\
factor(Si, So) & \leftarrow primary(Si, S1), S1 = ['\wedge'|S2], factor(S2, So) \\
factor(Si, So) & \leftarrow primary(Si, So) \\
primary(Si, So) & \leftarrow Si = ['('|S1], expr(S1, S2), S2 = [')'|So] \\
primary(Si, So) & \leftarrow Si = [I|So], integer(I)
\end{array}
$$

as far as termination is concerned, there is no need to table the new nonterminal. Indeed, Prolog's evaluation strategy handles right recursion in grammars finitely. In fact, Prolog-style evaluation of right recursion is more efficient than its tabled-based evaluation: Prolog has linear complexity for a simple right recursive grammar, but with tabling implemented as in XSB the evaluation could be quadratic as calls need to be recorded in the tables using explicit copying. Thus, it is important to allow tabled and non-tabled predicates to be freely intermixed, and be able to choose the strategy that is most efficient for the situation at hand.

By using tabling in context-free grammars, one gets a recognition algorithm that is a variant of Early's algorithm (also known as active chart recognition algorithm) whose complexity is polynomial in the size of the input expression/string [9]. However, often one wants to construct the parse tree(s) for a given input string. The usual approach is to introduce an extra argument to the nonterminals of the input grammar — representing the portion of the parse tree that each rule generates — and naturally to also add the necessary code that constructs the parse tree. This approach is straightforward, but as noticed in [19], using the same program for recognition as well as parsing may be extremely unsatisfactory from a complexity standpoint: in context-free grammars recognition is polynomial while parsing is exponential since there can be exponentially many parse trees for a given input string. The obvious solution is to use two interleaved versions of the grammar as in the example program of Fig. 2. Note that only $a/2$, i.e. the recursive predicate of the 'recognition' part,

$$R : \begin{cases} s(Si, So) \leftarrow a(Si, S), S = [b|So] \\ a(Si, So) \leftarrow a(Si, S), a(S, So) \\ a(Si, So) \leftarrow Si = [a|So] \end{cases}$$

$$P : \begin{cases} s(Si, So, PT) \leftarrow a(Si, S), S = [b|So], PT = spt(Pa, b), a(Si, S, Pa) \\ a(Si, So, PT) \leftarrow a(Si, S), a(S, So), PT = apt(P1, P2), a(Si, S, P1), a(S, So, P2) \\ a(Si, So, PT) \leftarrow Si = [a|So], PT = a \end{cases}$$

Fig. 2. A tabled program recognizing and parsing the language $a^n b$.

R, of the program (consisting of predicates $s/2$ and $a/2$), needs to be tabled. This action allows recognition to terminate and to have polynomial complexity. Furthermore, the recognizer can now be used as a filter for the parsing process in the following way: only after knowing that a particular part of the input belongs to the grammar and having computed the exact substring that each nonterminal spans, do we invoke the parsing routine on the nonterminal to construct its (possibly exponentially many) parse trees. Doing so, avoids e.g. cases where it may take exponential time to fail on an input string that does not belong in the given language: an example for the grammar under consideration is the input string a^n. On the other hand, tabling the 'parsing' part of the program (consisting of predicates $s/3$ and $a/3$) does not affect the efficiency of the process complexity-wise and incurs a small performance overhead due to the recording of calls and their results in the tables. Finally, note that the construction is modular in the sense that the 'parsing' part of the program, P, depends on the 'recognition' part, R, but not vice versa — we say that P *extends* R.

3 Preliminaries

We assume familiarity with the basic concepts of logic programming [13, 1]. Throughout the paper, P will denote a definite logic program and we restrict ourselves in this class. By $Pred_P$, we denote the set of predicates occurring in P, and by Def_P we denote the set of predicates defined in P (i.e. predicates occurring in the head of a clause of P). By Rec_P, resp. $NRec_P$, we denote the set of (directly or indirectly) recursive, resp. non-recursive, predicates of the program P (so $NRec_P = Pred_P \setminus Rec_P$). If $A = p(t_1, \ldots, t_n)$, then we denote by $Rel(A)$ the predicate symbol p of A; i.e. $Rel(A) = p$.

The *extended Herbrand Universe*, U_P^E, and the *extended Herbrand Base*, B_P^E, associated with a program P, were introduced in [10]. 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. If $\Pi \subseteq Pred_P$, we denote with B_Π^E the subset of B_P^E consisting of (equivalence classes of) atoms based on the predicate symbols of Π. So B_P^E can be seen as an abbreviation of $B_{Pred_P}^E$.

Let P be a program and p, q two predicate symbols of P. We say that p *refers to* q in P iff 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 $p \sqsupseteq q$, iff (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 $p = q$). The *dependency graph* G of a program P is a graph where the nodes are labeled with the predicates of $Pred_P$. There is a *directed arc* from p to q in G iff p refers to q. Finally, we will say that a program P *extends* a program R iff no predicate defined in P occurs in R.

In analogy to [2], we will refer to SLD-derivations (see [13]) following the left-to-right selection rule as LD-derivations. Other concepts will adopt this naming accordingly. For a program P and set $S \subseteq B_P^E$, we denote by $Call(P, S)$ the subset of B_P^E such that $B \in Call(P, S)$ whenever an element of B is a selected atom in an LD-derivation for some $P \cup \{\leftarrow A\}$, with $\tilde{A} \in S$. Throughout, we will assume that in any derivation of a query w.r.t. a program, representants of equivalence classes are systematically provided with fresh variables, to avoid the necessity of renaming apart. In the sequel, we abbreviate computed answer substitution with c.a.s. Our termination conditions are based on the following concept of finitely partitioning level mapping.

Definition 1. ((finitely partitioning) level mapping)
Let P be a program and $L \subseteq B_P^E$. A level mapping *on L is a function $l : L \to \mathbb{N}$. A level mapping l on L is* finitely partitioning *on $C \subseteq L$ iff for all $n \in \mathbb{N}$: $\sharp(l^{-1}(n) \cap C) < \infty$, where $\sharp$ is the cardinality function.*

3.1 SLG-Resolution

We present a non-constructive definition of SLG-resolution that is sufficient for our purposes, and refer to [4, 15] for more constructive formulations of (variants) of tabled resolution.

By fixing a *tabling* for a program P, we mean choosing a set of predicates of P which are tabled. We denote this tabling as Tab_P. The complement of this set of tabled predicates is denoted as $NTab_P = Pred_P \setminus Tab_P$.

Definition 2. (pseudo SLG-tree, pseudo LG-tree) *Let P be a definite program, $Tab_P \subseteq Pred_P$, $\mathcal{R}$ a selection rule and A an atom. A* pseudo SLG-tree w.r.t. Tab_P for $P \cup \{\leftarrow A\}$ under $\mathcal{R}$ *is a tree τ_A such that:*

1. *the nodes of τ_A are labeled with goals along with an indication of the selected atom according to $\mathcal{R}$,*
2. *the arcs are labeled with 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, the arcs are labeled with the corresponding mgu used in the resolution step,*
5. *the children of a non-root node labeled with the goal $\mathbf{Q}$ where $\mathcal{R}(\mathbf{Q}) = B$ are obtained as follows:*

(a) *if* $Rel(B) \in Tab_P$*, then the (possibly infinitely many) children of the node can only be obtained by resolving the selected atom* B *of the node with clauses of the form* $B\theta \leftarrow$ *(not necessarily in* P*), the arcs are labeled with the corresponding mgu used in the resolution step (i.e.* θ*),*
(b) *if* $Rel(B) \in NTab_P$*, then the children of the node are obtained by resolution of* B *against all matching program clauses in* P*, and the arcs are labeled with the corresponding mgu used in the resolution step.*

If $\mathcal{R}$ *is the leftmost selection rule,* τ_A *is called a* pseudo LG-tree *w.r.t.* Tab_P *for* $P \cup \{\leftarrow A\}$.
We say that a pseudo SLG-tree τ_A *w.r.t.* Tab_P *for* $P \cup \{\leftarrow A\}$ *is* smaller *than another pseudo SLG-tree* $\tau_A^{'}$ *w.r.t.* Tab_P *for* $P \cup \{\leftarrow A\}$ *iff* $\tau_A^{'}$ *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 *w.r.t.* Tab_P *for* $P \cup \{\leftarrow A\}$ *is a clause of the form* $A\theta \leftarrow$ *where* θ *is the composition of the substitutions found on a branch of* τ_A *whose leaf is labeled with the empty goal.*

Intuitively, a pseudo SLG-tree (in an SLG-forest, see Definition 3 below) represents the tabled computation (w.r.t. Tab_P) 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 5a. These clauses represent the answers found (possibly in another tree of the forest) for the selected tabled atom. This interaction between the trees in an SLG-forest is captured in the following definition.

Definition 3. (SLG-forest, LG-forest) *Let* P *be a definite program,* $Tab_P \subseteq Pred_P$*,* $\mathcal{R}$ *be a selection rule and* T *be a (possibly infinite) set of atoms such that no two different atoms in* T *are variants of each other.* $\mathcal{F}$ *is an* SLG-forest w.r.t. Tab_P for P and T under $\mathcal{R}$ *iff* $\mathcal{F}$ *is a set of minimal pseudo SLG-trees* $\{\tau_A \mid A \in T\}$ *w.r.t.* Tab_P *where*

1. τ_A *is a pseudo SLG-tree w.r.t.* Tab_P *for* $P \cup \{\leftarrow A\}$ *under* $\mathcal{R}$,
2. *every selected tabled atom* B *of each node in every* $\tau_A \in \mathcal{F}$ *is a variant of an element* $B^{'}$ *of* T*, such that every clause resolved with* B *is a variant of an answer clause of* $\tau_{B'}$ *and vice versa, for every answer clause of* $\tau_{B'}$ *there is a variant of this answer clause which is resolved with* B.

Let S *be a set of atoms. An* SLG-forest for P and S w.r.t. Tab_P under $\mathcal{R}$ *is an SLG-forest w.r.t.* Tab_P *for a minimal set* T *with* $S \subseteq T$*. If* $S = \{A\}$*, then we also talk about the SLG-forest for* $P \cup \{\leftarrow A\}$.
An LG-forest *is an SLG-forest containing only pseudo LG-trees.*

Point 2 of Definition 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.

Note that, if $Tab_P = \emptyset$, the (S)LG-forest of $P \cup \{\leftarrow A\}$ consists of one tree: the (S)LD-tree of $P \cup \{\leftarrow A\}$. We use the following artificial tabled program to illustrate the concepts that we introduced.

Example 1. Let P be the following program, with $Tab_P = \{member/2\}$. Let

$$
\begin{array}{lll}
intersection(Xs, Ys, Z) & \leftarrow & member(Xs, Z), member(Ys, Z) \\
member([Z|Zs], Z) & \leftarrow & \\
member([X|Zs], Z) & \leftarrow & member(Zs, Z)
\end{array}
$$

$S = \{intersection(Xs, Ys, a)\}$. Then, $Call(P, S) = S \cup \{member(Zs, a)\}$ and the LG-forest for P and S is shown in Fig. 3. Note that there is a finite number of LG-trees, all with finite branches, but the trees have infinitely branching nodes.

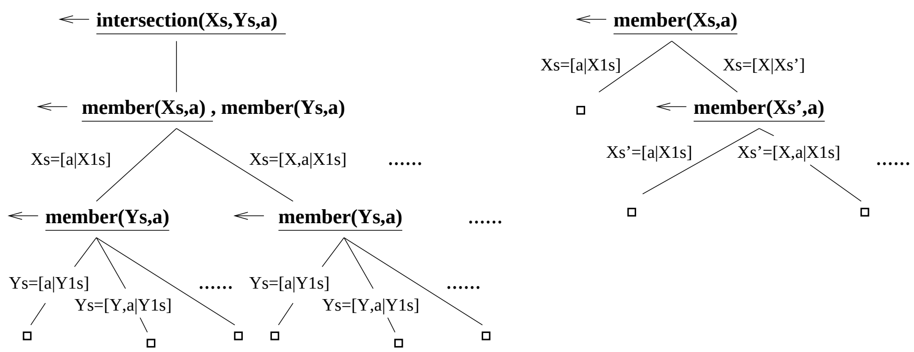

Fig. 3. The LG-forest for $P \cup \{\leftarrow intersection(Xs, Ys, a)\}$.

Note that we can use the notions of LD-derivation and LD-computation (as they appear for instance 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. [12, Theorem 2.1].

4 Quasi-Termination

A first basic notion of (universal) termination under a tabled execution mechanism is quasi-termination. It is defined as follows (see [8, Definition 3.1] for the case $Tab_P = Pred_P$).

Definition 4. (quasi-termination) *Let P be a program, $Tab_P \subseteq Pred_P$ and $S \subseteq B_P^E$. P* quasi-terminates w.r.t. *Tab_P* and *S iff for all A such that $\tilde{A} \in S$, the LG-forest w.r.t. Tab_P for $P \cup \{\leftarrow A\}$ consists of a finite number of LG-trees without infinite branches. Also, P* quasi-terminates w.r.t. *S iff P quasi-terminates w.r.t. $Pred_P$ and S.*

Note that it is not required that the LG-trees are finitely branching in their nodes. In the next section, 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 trees (i.e. trees with finite branches *and* which are finitely branching). Recall the program P and set S of Example 1. P quasi-terminates w.r.t. $\{member/2\}$ and S. Note that P does not LG-terminate w.r.t. $\{member/2\}$ and S.

Many works (see [5] for a survey) address the problem of LD-termination: A program P is said to be *LD-terminating* w.r.t. a set $S \subseteq B_P^E$ iff for all A such that $\tilde{A} \in S$, the LD-tree of $P \cup \{\leftarrow A\}$ is finite (see for instance [6]). It can be easily shown that, if P LD-terminates w.r.t. S, then P quasi-terminates w.r.t. Tab_P and S (for every $Tab_P \subseteq Pred_P$). As shown in Example 1, the notion of LD-termination is strictly stronger than the notion of quasi-termination.

Since quasi-termination requires that there are only finitely many LG-trees in the LG-forest of a query, there can only be a finite number of tabled atoms in the call set of that query. More formally: *If a program P quasi-terminates w.r.t. Tab_P and S, then, for every $A \in S$, $Call(P, \{A\}) \cap B^E_{Tab_P}$ is finite.*

In [8], the special case where $Tab_P = Pred_P$, i.e. where all predicates occurring in P are tabled, is considered. If $Tab_P = Pred_P$, an LG-tree cannot have infinite branches. So, P quasi-terminates w.r.t. a set S iff for all A such that $\tilde{A} \in S$, the LG-forest for $P \cup \{\leftarrow A\}$ consists of a finite number of LG-trees. In [8, Lemma 3.1] the following equivalence was proven: *P quasi-terminates w.r.t. S iff for every $A \in S$, $Call(P, \{A\})$ is finite.* We want to note that none of the directions (if nor only-if) hold in case that the tabled predicates of the program P are a strict subset of $Pred_P$. We refer to [17] for two counterexamples, one for each direction.

4.1 Characterisation of Quasi-Termination

In order to state a necessary and sufficient condition for quasi-termination, we need to make an assumption on the set Tab_P of tabled predicates of the program P. If the assumption is not satisfied, the condition is sufficient (but not necessary).

Definition 5. (well-chosen tabling (w.r.t. a program)) *Let P be a program, $Pred_P = Tab_P \sqcup NTab_P$ and G be the predicate dependency graph of P. The tabling Tab_P is called* well-chosen w.r.t. *the program P iff for every $p, q \in NTab_P$ such that $p \simeq q$, exactly one of the following two conditions holds:*

$C_1(p, q)$: *no cycle of directed arcs in G containing p and q contains a predicate from Tab_P.*

$C_2(p, q)$: *all cycles of directed arcs in G containing p and q contain at least one predicate from Tab_P.*

In particular, if $NTab_P \subseteq \{p \in Pred_P \mid p$ is a non-recursive or only directly recursive predicate$\}$ or if $NTab_P = \emptyset$ (i.e. $Tab_P = Pred_P$), then the tabling Tab_P is well-chosen w.r.t. P.

The next theorem provides a necessary and sufficient condition for quasi-termination of a program P w.r.t. a tabling and a set of atoms, in case the tabling is well-chosen w.r.t. P.

Theorem 1. (characterisation of quasi-termination in case the tabling is well-chosen) *Let P be a program, $Tab_P \subseteq Pred_P$ and $S \subseteq B_P^E$. Suppose the tabling Tab_P is well-chosen w.r.t. P. Then, P quasi-terminates w.r.t. Tab_P and S iff there is a level mapping l on B_P^E such that for every $A \in S$, l is finitely partitioning on $Call(P,\{A\}) \cap B_{Tab_P}^E$, and such that*

- *for every atom A such that $\tilde{A} \in Call(P,S)$,*
- *for every clause $H \leftarrow B_1,\ldots,B_n$ in P, such that $mgu(A,H) = \theta$ exists,*
- *for every $1 \le i \le n$ and for every LD-c.a.s. θ_{i-1} for $\leftarrow (B_1,\ldots,B_{i-1})\theta$:*

$$\begin{array}{l} l(A) \ge l(B_i\theta\theta_{i-1}) \\ \textit{and} \\ l(A) > l(B_i\theta\theta_{i-1}) \textit{ if } Rel(A) \simeq Rel(B_i) \in NTab_P \\ \qquad\qquad\qquad \textit{and } C_2(Rel(A), Rel(B_i)) \textit{ does not hold.} \end{array}$$

Note that [8, Theorem 3.1] is an instance of this theorem, with $Tab_P = Pred_P$. We illustrate the intuition behind Theorem 1 with the following example.

Example 2. Consider the following three propositional programs P, P' and P'':

$$P : \begin{cases} p \leftarrow q \\ q \leftarrow r \\ r \leftarrow q \end{cases} \qquad P' : \begin{cases} p \leftarrow q \\ q \leftarrow r \\ r \leftarrow p \end{cases} \qquad P'' : \begin{cases} p \leftarrow q \\ q \leftarrow r \\ r \leftarrow p \\ r \leftarrow q \end{cases}$$

with $Tab_P = Tab_{P'} = Tab_{P''} = \{p\}$ and $S = \{p\}$. The LG-forests for $P \cup \{\leftarrow p\}$, $P' \cup \{\leftarrow p\}$ and $P'' \cup \{\leftarrow p\}$ are shown in Fig. 4. P and P'' do not quasi-terminate

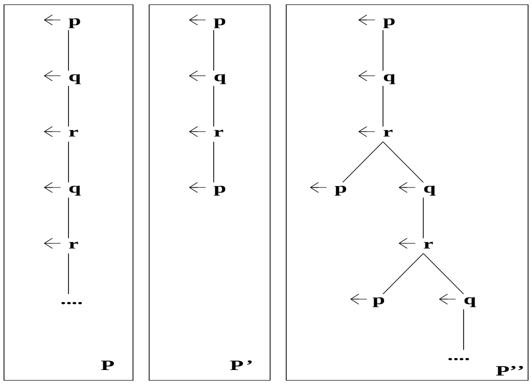

Fig. 4. The LG-forests for $P \cup \{\leftarrow p\}$, $P' \cup \{\leftarrow p\}$, and $P'' \cup \{\leftarrow p\}$.

w.r.t. $\{p\}$, whereas P' does.
Note that for programs P and P', the tablings are well-chosen. Also note that, because the programs are propositional, every level mapping will be finitely partitioning on the whole Herbrand base.
Let's first consider program P. For this program condition $C_1(q,r)$ holds. Also note that there is no level mapping l such that $l(q) > l(r)$ and $l(r) > l(q)$. Hence,

the condition in Theorem 1 can not be satisfied and P does not quasi-terminate w.r.t. $\{p\}$.
Consider next program $P^{'}$ for which condition $C_2(q,r)$ holds. Let l be the following level mapping $l(p) = l(q) = l(r) = 0$. With this level mapping, $P^{'}$ satisfies the condition of Theorem 1 and hence, $P^{'}$ quasi-terminates w.r.t. $\{p\}$.
Finally, note that for the program $P^{''}$, the tabling is not well-chosen w.r.t. $P^{''}$.

In case the tabling for a program is not well-chosen (like for the program $P^{''}$ of Example 2), the condition of Theorem 1 is sufficient (but not necessary) for proving quasi-termination.

4.2 Modular Proofs for Quasi-Termination

We now present a proposition which gives a modular proof for the quasi-termination of the union $P \cup R$ of two programs P and R, such that P extends R. If $Pred_{P\cup R} = Tab_{P\cup R} \sqcup NTab_{P\cup R}$, let

$$Tab_P = Tab_{P\cup R} \cap Pred_P \ , NTab_P = NTab_{P\cup R} \cap Pred_P,$$
$$Tab_R = Tab_{P\cup R} \cap Pred_R \ , NTab_R = NTab_{P\cup R} \cap Pred_R.$$

Proposition 1. *Suppose P and R are two programs, such that P extends R. Let $S \subseteq B^E_{P\cup R}$. If*

- *R quasi-terminates w.r.t. Tab_R and $Call(P \cup R, S)$,*
- *there is a level mapping l on B^E_P such that for every $A \in S$, l is finitely partitioning on $Call(P \cup R, \{A\}) \cap B^E_{Tab_P}$, and such that*
  - *for every atom A such that $\tilde{A} \in Call(P \cup R, S)$,*
  - *for every clause $H \leftarrow B_1, \ldots, B_n$ in P such that $mgu(A, H) = \theta$ exists,*
  - *for every $1 \leq i \leq n$ and for every LD-c.a.s. θ_{i-1} in $P \cup R$ for $\leftarrow (B_1, \ldots, B_{i-1})\theta$:*

$$l(A) \geq l(B_i\theta\theta_{i-1})$$
and
$$l(A) > l(B_i\theta\theta_{i-1}) \text{ if } Rel(A) \simeq Rel(B_i) \in NTab_P \text{ and } C_2(Rel(A), Rel(B_i)) \text{ does not hold.}$$

then, $P \cup R$ quasi-terminates w.r.t. $Tab_{P\cup R}$ and S.

The program and query of Example 1 can be proven to quasi-terminate by applying Proposition 1 (see [17]). In [17] we give three more propositions for proving quasi-termination in a modular way. They all consider special cases of Proposition 1: in the first of these propositions, no defined predicate of P is tabled, in the second one all the defined predicates in P are tabled, and in the last one, P and R extend each other.

Note that the above modular termination conditions prove the quasi-termination of $P \cup R$ without constructing an appropriate level mapping which satisfies the condition of Theorem 1. We refer to [17] where modular termination conditions for quasi-termination are given which construct (from simpler level mappings) a level mapping such that $P \cup R$ and this level mapping satisfy the condition for quasi-termination of Theorem 1.

5 LG-Termination

The notion of quasi-termination only partially corresponds to our intuitive notion of a terminating execution of a query against a tabled program. This notion only requires that the LG-forest consists of only a finite number of LG-trees, without infinite branches, yet these trees can have infinitely branching nodes. To capture this source of non-termination for a tabled computation, the following stronger notion is introduced (see [8, Def. 4.1] for the special case where $Tab_P = Pred_P$).

Definition 6. (LG-termination) *Let P be a program, $Tab_P \subseteq Pred_P$ and $S \subseteq B_P^E$. P* LG-terminates w.r.t. Tab_P and S *iff for every atom A such that $\tilde{A} \in S$, the LG-forest w.r.t. Tab_P for $P \cup \{\leftarrow A\}$ consists of a finite number of finite LG-trees.*

As already noted, the program P of Example 1 does not LG-terminate w.r.t. $\{member/2\}$ and $\{intersection(Xs, Ys, a)\}$. Obviously, the notion of LG-termination is (strictly) stronger than the notion of quasi-termination. Also, LD-termination implies LG-termination.

Consider two tablings for a program P; one with set of tabled predicates equal to $Tab_1 \subseteq Pred_P$, the other with set of tabled predicates equal to $Tab_2 \subseteq Pred_P$. Suppose $Tab_1 \subseteq Tab_2$ (hence $NTab_1 \supseteq NTab_2$). The next proposition studies the relationship between the LG-termination of P w.r.t. these two tablings. We note that it does not hold for quasi-termination; see [17] for a counterexample.

Proposition 2. *Let P be a program. Let $Pred_P = Tab_1 \sqcup NTab_1$ and $Pred_P = Tab_2 \sqcup NTab_2$. Suppose $Tab_1 \subseteq Tab_2$. Let $S \subseteq B_P^E$. If P LG-terminates w.r.t. Tab_1 and S, then P LG-terminates w.r.t. Tab_2 and S.*

We now relate the notions of quasi-termination and LG-termination in a more detailed way: By definition, quasi-termination only corresponds to part of the LG-termination notion; it fails to capture non-termination caused by an infinitely branching node in an LG-tree. Note that if an LG-forest contains a tree with an infinitely branching node, then there is an LG-tree in the forest which is infinitely branching in a node which contains a goal with a recursive, tabled atom at the leftmost position. This observation leads to the following lemma. Let us denote the set of tabled, recursive predicates in a program P with TR_P:

$$TR_P = Tab_P \cap Rec_P.$$

Lemma 1. *Let P be a program, $Tab_P \subseteq Pred_P$ and $S \subseteq B_P^E$. P LG-terminates w.r.t. Tab_P and S iff P quasi-terminates w.r.t. Tab_P and S and for all $A \in S$ the set of (LD-)computed answers for atoms in $Call(P, \{A\}) \cap B_{TR_P}^E$ is finite.*

It follows from the proof of this lemma (see [17]) that, if P LG-terminates w.r.t. Tab_P and S, the set of computed answers for atoms in $Call(P, \{A\})$ is finite for all $A \in S$. We now present a characterisation of LG-termination.

5.1 Characterisation of LG-Termination

First (in Theorem 2), we will characterise LG-termination of a program P in terms of quasi-termination of the program P^a, which is obtained by applying the answer-transformation (Definition 7) on P. However, we will also characterise LG-termination in a more direct way (Theorem 3).

Lemma 1 gives the intuition behind the answer-transformation that we are about to present. The answer-transformation forms the basis of the characterisation of LG-termination in Theorem 2; LG-termination of a program P will be shown to be equivalent with quasi-termination of the program P^a, obtained by applying the answer-transformation to P.

Definition 7. (a(nswer)-transformation) *Let P be a program with tabling $Tab_P \subseteq Pred_P$. The* a-transformation *is defined as follows:*

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

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

with B_i^* defined as follows (suppose $B_i = p(t_1, \ldots, t_n)$):
if $p \simeq Rel(H)$ and $p \in Tab_P$, then $B_i^* = p^a(t_1, \ldots, t_n)$, where p^a/n is a new predicate, else $B_i^* = \emptyset$.
Let $TR_P^a = \{p^a/n \mid p/n \in TR_P\}$ (recall that $TR_P = Tab_P \cap Rec_P$).
- *For the program P, we define*

$$P^a = \{C^a \mid C \in P\} \cup \{p^a(X_1, \ldots, X_n) \leftarrow \ \mid p^a/n \in TR_P^a\}.$$

- *The set of tabled predicates of the program P^a is defined as*

$$Tab_{P^a} = Tab_P \cup TR_P^a.$$

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

We want to mention that a similar transformation, namely the solution-transformation, is introduced in [8, Definition 4.2] in order to relate the concepts of LG-termination and quasi-termination. But, as opposed to the answer-transformation, the solution-transformation introduces much more "overhead" in the sense that a clause $C = H \leftarrow B_1, \ldots, B_n$ is transformed into a clause $C_{sol} = H \leftarrow B_1, sol(B_1), \ldots, B_n, sol(B_n)$ where $sol/1$ is a new tabled predicate. Notice that in contrast, the answer-transformation only keeps track of the computed answers of *recursive, tabled* body atoms (and not of all body atoms).

Example 3. Let P be the program of Example 1, with $Tab_P = \{member/2\}$. The a-transformation, P^a, of P is shown in Fig. 5; $Tab_{P^a} = \{member/2, member^a/2\}$.

$$
\begin{array}{lll}
intersection(Xs, Ys, Z) & \leftarrow & member(Xs, Z), member(Ys, Z) \\
member([Z|Zs], Z) & \leftarrow & \\
member([X|Zs], Z) & \leftarrow & member(Zs, Z), member^a(Zs, Z) \\
member^a(L, Z) & \leftarrow &
\end{array}
$$

Fig. 5. The a-transformation of P from Example 1.

The following theorem is a generalisation of [8, Theorem 4.1] (there, the previously mentioned solution-transformation of [8, Definition 4.2] is used to relate LG-termination and quasi-termination in case $Tab_P = Pred_P$).

Theorem 2. (characterisation of LG-termination in terms of quasi-termination) *Let P be a program, $Tab_P \subseteq Pred_P$ and $S \subseteq B_P^E$. P LG-terminates w.r.t. Tab_P and S iff P^a quasi-terminates w.r.t. Tab_{P^a} and S.*

Example 4. (Ex. 3 continued). The LG-forest of $P \cup \{\leftarrow intersection(Xs, Ys, a)\}$ w.r.t. Tab_P was shown in Fig. 3. Note that the trees are infinitely branching and hence, P does not LG-terminate w.r.t. Tab_P and $\{intersection(Xs, Ys, a)\}$. In Fig. 6, the LG-forest of the program P^a and $\{intersection(Xs, Ys, a)\}$ w.r.t. Tab_{P^a} is shown. Note that there are infinitely many LG-trees in the forest; P^a doesn't quasi-terminate w.r.t. Tab_{P^a} and $\{intersection(Xs, Ys, a)\}$.

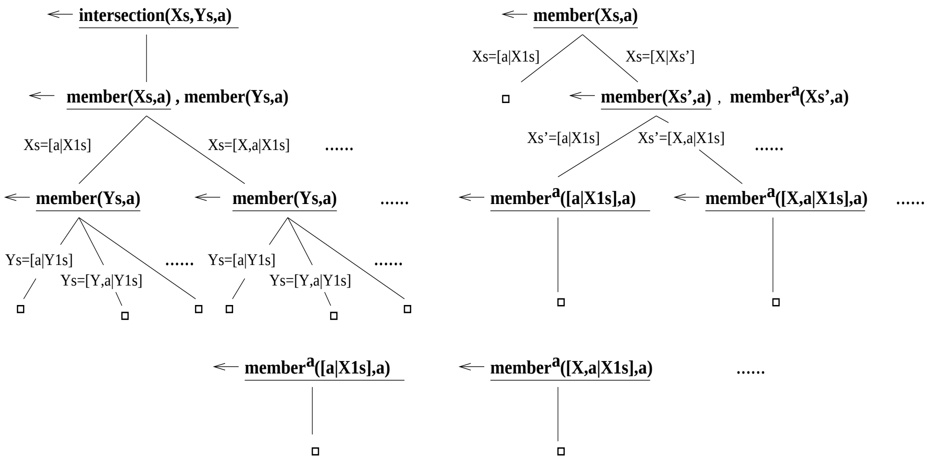

Fig. 6. The LG-forest for $P^a \cup \{\leftarrow intersection(Xs, Ys, a)\}$.

Theorem 2 provides a way to prove LG-termination of a program w.r.t. a set of queries. Namely, it is sufficient to prove quasi-termination of the a-transformation of the program w.r.t. the set of queries. To prove quasi-termination, we can use the results of Section 4.1: the condition of Theorem 1, which is necessary and sufficient in case the tabling is well-chosen[1], and which is sufficient in the general case. However, the condition of this theorem on P^a can be weakened; i.e. some of the decreases "$l(A) \geq l(B_i\theta\theta_{i-1})$" need not be checked because they can always

[1] Note that if Tab_P is well-chosen w.r.t. P, then also Tab_{P^a} is well-chosen w.r.t. P^a.

be fulfilled. In particular, we only have to require the non-strict decrease for recursive, tabled body atoms B_i (to obtain an LG-forest with only finitely many LG-trees) or for body atoms B_i of the form $p^a(t_1, \ldots, t_n)$ (to obtain LG-trees which are finitely branching); the conditions on non-tabled predicates remain the same. The following theorem presents these optimised conditions and characterises LG-termination of a program in case the tabling is well-chosen. If the tabling is not well-chosen, the condition is sufficient (but not necessary).

Theorem 3. (characterisation of LG-termination in case the tabling is well-chosen) *Let P be a program, $Tab_P \subseteq Pred_P$ and $S \subseteq B_P^E$. Suppose the tabling Tab_P is well-chosen w.r.t. P. Then, P LG-terminates w.r.t. Tab_P and S iff there is a level mapping l on $B_{P^a}^E$ such that for every $A \in S$, l is finitely partitioning on $Call(P^a, \{A\}) \cap B_{TR_P \cup TR_P^a}^E$, and such that*

- *for every atom A such that $\tilde{A} \in Call(P^a, S)$,*
- *for every clause $H \leftarrow B_1, \ldots, B_n$ in P^a, such that $mgu(A, H) = \theta$ exists,*
- *for every B_i such that $Rel(B_i) \simeq Rel(H)$ or $Rel(B_i) \in TR_P^a$,*
- *for every LD-c.a.s. θ_{i-1} in P^a for $\leftarrow (B_1, \ldots, B_{i-1})\theta$:*

$$\begin{array}{l} l(A) \geq l(B_i\theta\theta_{i-1}) \\ \textit{and} \\ l(A) > l(B_i\theta\theta_{i-1}) \textit{ if } Rel(A) \simeq Rel(B_i) \in NTab_P \\ \qquad\qquad\qquad\qquad \textit{and } C_2(Rel(A), Rel(B_i)) \textit{ does not hold.} \end{array}$$

Example 5. Recall the recognition part of the grammar program of Fig. 2, where

$$R : \begin{cases} s(Si, So) \leftarrow a(Si, S), S = [b|So] \\ a(Si, So) \leftarrow a(Si, S), a(S, So) \\ a(Si, So) \leftarrow Si = [a|So] \end{cases}$$

$Tab_R = \{a/2\}$. We show that R LG-terminates w.r.t. $\{a/2\}$ and $S = \{s(si, So)\}$ where si is a ground list consisting of atoms and So is a variable. Consider the following a-transformation, R^a, of R ($Tab_{R^a} = \{a/2, a^a/2\}$) shown below. When

$$R^a : \begin{cases} s(Si, So) \leftarrow a(Si, S), S = [b|So] & \qquad a^a(Si, So) \leftarrow \\ a(Si, So) \leftarrow a(Si, S), a^a(Si, S), a(S, So), a^a(S, So) \\ a(Si, So) \leftarrow Si = [a|So] \end{cases}$$

applying Theorem 3, we only have to consider the second clause of R^a. Note that, for all $a(t1, t2) \in Call(R^a, \{s(si, S0)\})$, $t1$ is a sublist of si and $t2$ is a variable. Also, for all $a^a(v1, v2) \in Call(R^a, \{s(si, So)\})$, $v1$ is a sublist of si and $v2$ is a (strict) sublist of $v1$. Consider the trivial level mapping l (mapping everything to 0) on $Call(R^a, \{s(si, So)\}) \cap B_{\{a/2, a^a/2\}}^E$. Since this set is finite, l is obviously finitely partitioning on this set. R and S, together with l, satisfy the conditions of Theorem 3. Hence, R LG-terminates w.r.t. $\{a/2\}$ and S.

5.2 Modular Proofs for LG-Termination

Similarly to the case of quasi-termination (Section 4.2), we want to be able to obtain modular termination proofs for LG-termination of the union $P \cup R$ of two programs P and R, where P extends R. Note that, because of Theorem 2 and because $(P \cup R)^a = P^a \cup R^a$ (if P extends R), we can use the modular conditions for quasi-termination of Section 4.2. However, as we already noted in Section 5.1, we can give optimised conditions which require less checking for decreases between the values under the level mapping of the head and body atoms. Due to space limitations, in this paper we will only consider the case of two programs P and R, where P extends R, and no defined predicate symbol of P is tabled. Proposition 3 will give modular conditions for LG-termination of $P \cup R$ in that case, without using Theorem 2. The general case, the case in which all defined predicate symbols of P are tabled and the case in which the two programs extend each other are treated in the full version of the paper [17].

Proposition 3. *Let P and R be two programs, such that P extends R and such that $Def_P \subseteq NTab_P$. Let $S \subseteq B^E_{P \cup R}$. If*

- *R LG-terminates w.r.t. Tab_R and $Call(P \cup R, S)$,*
- *there is a level mapping l on B^E_P such that*
  - *for every atom A with $\tilde{A} \in Call(P \cup R, S)$,*
  - *for every clause $H \leftarrow B_1, \ldots, B_n$ in P such that $mgu(A, H) = \theta$ exists,*
  - *for every B_i, $i \in \{1, \ldots, n\}$, with $Rel(B_i) \simeq Rel(A)$,*
  - *for every LD-c.a.s. θ_{i-1} in $P \cup R$ for $\leftarrow (B_1, \ldots, B_{i-1})\theta$:*

$$l(A) > l(B_i\theta\theta_{i-1})$$

then, $P \cup R$ LG-terminates w.r.t. $Tab_{P \cup R}$ and S.

Example 6. Recall program R of Example 5. Let P be the parsing part of the grammar program of Fig. 2 which is also shown below. As already noted in Section 2, P extends R, and the only tabled predicate in $P \cup R$ is $a/2$ — see Section 2 for why this tabling is sufficient.

$$P : \begin{cases} s(Si, So, PT) \leftarrow a(Si, S), S = [b|So], PT = spt(Pa, b), a(Si, S, Pa) \\ a(Si, So, PT) \leftarrow a(Si, S), a(S, So), PT = apt(P1, P2), a(Si, S, P1), a(S, So, P2) \\ a(Si, So, PT) \leftarrow Si = [a|So], PT = a \end{cases}$$

Let $S = \{s(si, So, PT)\}$ where si is a ground list of atoms, and So, PT are distinct variables. We show, by using Proposition 3, that $P \cup R$ LG-terminates w.r.t. $\{a/2\}$ and S.

- R LG-terminates w.r.t. $\{a/2\}$ and $Call(P \cup R, S)$.
  Note that, if $a(t1, t2) \in Call(P \cup R, S)$, then either $t1$ is a sublist of si and $t2$ is a variable, or $t1$ and $t2$ are both sublists of si. In Example 5, we proved that R LG-terminates w.r.t. this first kind of queries. To prove that R LG-terminates w.r.t. the second kind of queries, we can again apply Theorem 3. Due to space limitations, we omit the proof here.
- Note first that, if $a(t1, t2, PT) \in Call(P \cup R, S)$, then $t2$ is a (strict) sublist of $t1$, $t1$ is a sublist of si and PT is a variable. Let l be the following level mapping on $Call(P \cup R, S) \cap B^E_{\{a/3\}}$: $l(a(t1, t2, PT)) = \| t1 \|_l - \| t2 \|_l$, where $\| \ \|_l$ is the list-length norm. Because of the remark above, l is well-defined. Note that we only have to consider the recursive clause for $a/3$ in the analysis.

- Consider the fourth body atom in the recursive clause for $a/3$. If this clause is called with $a(ti, to, PT)$, with to a (strict) sublist of ti, then the fourth body atom is called as $a(ti, t, P1)$ where to is a (strict) sublist of t and t is a (strict) sublist of ti. Hence, $l(a(ti, to, PT)) = \parallel ti \parallel_l - \parallel to \parallel_l > \parallel ti \parallel_l - \parallel t \parallel_l = l(a(ti, t, P1))$.
- Consider the last body atom. If the recursive clause is called with $a(ti, to, PT)$, with to a (strict) sublist of ti, then the last body atom is called as $a(t, to, P2)$ where to is a (strict) sublist of t and t is a (strict) sublist of ti. Hence, $l(a(ti, to, PT)) = \parallel ti \parallel_l - \parallel to \parallel_l > \parallel t \parallel_l - \parallel to \parallel_l = l(a(t, to, P2))$.

We conclude that $P \cup R$ and S satisfy the condition of Proposition 3, so $P \cup R$ LG-terminates w.r.t. $\{a/2\}$ and S.

6 Related and Future Work

Our work is based on, and significantly extends, the results of [8]. In [8], the two notions of (universal) termination under tabled execution, namely quasi-termination and LG-termination, are introduced and characterised. As opposed to [8], where it is assumed that all predicates in the program are tabled, we here consider programs with a mix of tabled and Prolog execution, thereby providing a termination framework for 'real' tabled programs. We further extend the applicability of this framework by presenting modular termination conditions: conditions ensuring termination of the union $P \cup R$ of two programs P and R, where P extends R.

Termination proofs for (S)LD-resolution (such as e.g. those surveyed in [5]) are sufficient to prove termination under a tabled execution mechanism, but, since there are quasi-terminating and LG-terminating programs, which are not LD-terminating, more effective proof techniques can and need to be found. Besides [8], there are only relatively few works studying termination under tabling. In the context of well-moded programs, [14] presents a sufficient condition for a program to have the bounded term-size property, which implies LG-termination. [11] provides another sufficient condition for quasi-termination in the context of functional programming. In parallel with the work reported on in this paper, an orthogonal extension of the work of [8] was investigated in [16]. Namely, in [16] the constraint-based approach of [7] for automatically proving LD-termination was extended to the case of quasi-termination and LG-termination. More specifically, in the context of simply moded, well-typed programs and queries, sufficient conditions for quasi-termination and LG-termination (in the case that $Tab_P = Pred_P$) are given. These conditions allow reasoning fully at the clause level, contrary to those in the current paper which are stated for sets of calls. An integration of these two extensions of [8] is straightforward.

A topic for future research is to extend our results to *normal* logic programs executed under a mix of Prolog and tabled execution. Another, with an arguably more practical flavour, is to investigate how the termination conditions presented here can form the basis of a compiler that automatically decides on — or at least guides a programmer in choosing — a tabling (i.e. a set of tabled predicates) for an input program such that quasi-termination of the program is ensured.

Acknowledgements

Sofie Verbaeten is Research Assistant of the Fund for Scientific Research - F.W.O. Flanders, Belgium. Konstantinos Sagonas was supported by GOA "LP+, a second generation logic programming language". Danny De Schreye is Senior Research Associate of F.W.O. Flanders. Thanks to the anonymous referees for valuable comments.

References

1. K. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of theoretical computer science, Vol. B*. Elsevier Science Publishers, 1990.
2. K. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. In *Information and Computation*, 106(1):109–157, 1993.
3. K. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In *Advances in Logic Programming Theory*, pages 183–229. 1994.
4. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.
5. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19 & 20:199–260, 1994.
6. 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.
7. S. Decorte, D. De Schreye and H. Vandecasteele. Constraint-based automatic termination analysis for logic programs. *ACM TOPLAS*. To appear.
8. S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K. Sagonas. Termination Analysis for Tabled Logic Programming. In N. Fuchs, editor, *Proceedings of LOPSTR'97*, number 1463 in LNCS, pages 107–123. Springer, 1997.
9. J. Early. An efficient context-free parsing algorithm. *CACM*, 13(2):94–102, 1970.
10. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modelling of the operational behaviour of logic languages. *Theor. Comp. Sc.*, 69(3):289–318, 1989.
11. C. K. Holst. Finiteness Analysis. In J. 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, 1991.
12. T. Kanamori and T. Kawamura. OLDT-based abstract interpretation. *Journal of Logic Programming*, 15(1 & 2):1–30, 1993.
13. J. Lloyd. *Foundations of logic programming*. Springer-Verlag, 1987.
14. L. Plümer. *Termination proofs for logic programs*. Number 446 in LNAI. Springer-Verlag, 1990.
15. H. Tamaki and T. Sato. OLD Resolution with Tabulation. In E. Y. Shapiro, editor, *Proceedings ICLP'86*, LNCS 225, pages 84–98. Springer Verlag, 1986.
16. S. Verbaeten and D. De Schreye. Termination analysis of tabled logic programs using mode and type information. Technical Report 277, Department of Computer Science, K.U.Leuven. Available at http://www.cs.kuleuven.ac.be/∼sofie.
17. S. Verbaeten, K. Sagonas, and D. De Schreye. Modular termination proofs for Prolog with tabling. Technical Report, Department of Computer Science, K.U.Leuven, 1999. Available at http://www.cs.kuleuven.ac.be/∼sofie.
18. L. Vieille. Recursive query processing: the power of logic. *Theor. Comp. Sc.*, 69(1):1–53, 1989.
19. D. S. Warren. Notes for "Programming in Tabled Prolog". Early draft available at http://www.cs.sunysb.edu/∼warren/, 1998.