PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Framework for Analysing the Termination of Definite Logic Programs with respect to Call Patterns

Danny De Schreye* Kristof Verschaetse† Maurice Bruynooghe*

Department of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium.
e-mail: {dannyd,kristof,maurice}@cs.kuleuven.ac.be

Abstract

We extend the notions *'recurrency'* and *'acceptability'* of a logic program, which were respectively defined in the work of M. Bezem and the work of K. R. Apt and D. Pedreschi, and which were shown to be equivalent to respectively termination under an arbitrary computation rule and termination under the Prolog computation rule. We show that these equivalences still hold for the extended definitions. The main idéa is that instead of measuring ground instances of atoms, all possible calls are measured (which are not necessarily ground). By doing so, a more practical technique is obtained, in the sense that "more natural" measures can be used, which can easily be found automatically.

1 Introduction

In the last few years, a strong research effort in the field of logic programming has addressed the issue of termination. From the more theoretical point of view, the results obtained by Vasak and Potter [1986], Baudinet [1988], Bezem [1989], Cavedon [1989], Apt and Pedreschi [1990], and Bossi *et al.* [1991] have provided several frameworks and basic techniques to formulate and solve questions regarding the termination of logic programs in semantically clear and general terms. Other researchers, such as Ullman and Van Gelder [1988], Plümer [1990], Wang and Shyamasundar [1990], Verschaetse and De Schreye [1991], and Sohn and Van Gelder [1991] have provided practical and automatable techniques for proving the termination of logic programs with respect to certain classes of queries at compile time.

In this paper, we propose an extension of the theoretical frameworks for the characterisation of terminating programs and queries proposed in [Bezem 1989] and [Apt and Pedreschi 1990]. The framework does not only provide slightly more general results, but also increases the practicality of the techniques in view of automation.

*Supported by the National Fund for Scientific Research.
†Supported by ESPRIT BRA COMPULOG project nr. 3012.

Let us recall some definitions from [Bezem 1989] in order to explain our motivation and the intuition behind our approach.

Definition 1.1 (see [Bezem 1989]; Definition 2.1) A *level mapping* for a definite logic program P is a mapping $|.| : B_P \rightarrow I\!N$.

Definition 1.2 (see [Bezem 1989]; Definition 2.2) A definite logic program P is *recurrent* if there exists a level mapping $|.|$, such that for each ground instance $A \leftarrow B_1, \ldots, B_n$ of a clause in P, $|A| > |B_i|$, for each $i = 1, \ldots, n$.

Definition 1.3 (see [Bezem 1989]; Definition 2.7) A definite logic program P is *terminating* if all SLD-derivations for $(P, \leftarrow G)$, where G is a ground goal, are finite.

One of the basic results of [Bezem 1989] is that a program is recurrent if and only if it is terminating. Although this result is very interesting from a theoretical perspective, it is not a very practical one in terms of automated detection of terminating programs and queries. The problem comes from the fact that the definition of recurrency requires that the level mapping "compares" the head of each ground instance of a clause with *every* corresponding atom in the body and imposes a decrease. Intuitively, what would be preferable is to obtain a well-founding based on a measure function (or level mapping), which only decreases on each recursive call to a same predicate. This corresponds better to our intuition, since nontermination (for pure logic programs) can only be caused by infinite recursion.

As we stated above, the problem is not merely related to our intuition on the cause of nontermination, but more importantly to the practicality of level mappings. Consider the following example.

Example 1.4

$$\begin{array}{l} p([]). \\ p([H|T]) \leftarrow q([H|T]), p(T). \end{array}$$

$$\begin{array}{l} q([]). \\ q([H|T]) \leftarrow q(T). \end{array}$$

It is not possible to take as level mapping a function that maps ground instances $p(x)$ and $q(x)$ to the same level, namely *list-length*(x) if x is a ground list, and 0 otherwise. Instead, the definition of recurrency obliges us to take a level mapping that has a "unnatural" offset (1 in this case).

$$|p(x)| = \textit{list-length}(x) + 1$$
$$|q(x)| = \textit{list-length}(x).$$

In a naive attempt to improve on the results of [Bezem 1989], one could try to start from an adapted definition for a recurrent program, in which the relation $|A| > |B_i|$ would only be required if A and B_i are atoms with the same predicate symbol. However, the equivalence with termination would immediately be lost — even for programs having only direct recursion — as the following example shows.

Example 1.5

$$\texttt{append}([], L, L).$$
$$\texttt{append}([H|S], T, [H|U]) \leftarrow \texttt{append}(S, T, U).$$

$$\texttt{p}([H|T]) \leftarrow \texttt{append}(X, Y, Z), \texttt{p}(T).$$

An "extended" notion of recurrency, where the level mapping only relates the measure of ground instances of the recursive calls, would hold with respect to the level mapping:

$$|\texttt{p}(x)| = \textit{list-length}(x)$$
$$|\texttt{append}(x, y, z)| = \textit{list-length}(x).$$

On the other hand, the program is clearly not terminating — if it would be terminating, then we would have shown that append/3 terminates for a call with all three arguments free.

The heart of the problem is that in the definition of recurrency, the level mapping is used for two quite distinct purposes at the same time. First, the level mapping does ensure that on each derivation step, the measure of a recursive descending call is smaller than the measure of the ancestor call (or at least: for each ground instance of such a derivation step). Second, since we are only given that the top level goal is ground (or, in a more general version of the theorem, bounded) — but we have no information on the instantiation of any of the descending calls — the level mapping is also used to ensure that we have some upper limit on the measures for the calls of the (independent) recursive subcomputation evoked by the original call. In the current definition, this is done by imposing that the level also decreases between a call and its descendants that are not related through recursion.

The way in which we address the problem here, differs from the approach in [Bezem 1989] in three ways:

1. We first compute all atoms that can occur as calls during any SLD-derivation for the top-level goal(s) under consideration.
2. We use an extended notion of level mapping, defined on all such atoms — not only the ground ones.
3. We have an adapted definition of recurrency, with as its most important features:
   (a) the condition $|A| > |B_i|$ is not imposed on ground instances of a clause, but instead, on each instance obtained after unification with a (possible) call,
   (b) the decrease $|A| > |B_i|$ is only imposed if A and B_i are calls to the same predicate symbol. (This is for direct recursion — in the context of indirect recursion, the condition is more complex).

One of the side effects of taking this approach is that there is no more necessity to start the analysis for one ground or bounded goal. The technique works equally well when we start from any general set of atoms. The additional advantage that we gain here is that in practice, we are usually interested in the termination properties of a program with respect to some call pattern. Such call patterns can always be specified in terms of abstract properties of the arguments in the goals through mode information, type information or combined (rigid or integrated) mode and type information (see [Janssens and Bruynooghe 1990]). Any such call pattern corresponds to a set of atoms in the concrete domain, and can therefore be analysed with our approach.

The paper is organised as follows. In the next section we extend the equivalence theorem of [Bezem 1989] in the way described above. In section 3 we take a completely similar approach to extend results of [Apt and Pedreschi 1990] on left termination. In section 4, we illustrate the improved practicality of the new framework. We also indicate how some simple extensions are likely to provide full theoretical support for the automated technique proposed in [Verschaetse and De Schreye 1991].

All proofs have been omitted from the paper. They can be found in [De Schreye and Verschaetse 1992].

2 Recurrency with respect to a set of atoms

We first introduce some conventions and recall some basic terminology. 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 to a program P, were introduced in

[Falaschi *et al.* 1989]. They are defined as follows. Let $Term_P$ and $Atom_P$ denote the sets of respectively all terms and all atoms that can be constructed from the alphabet underlying to 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 $\bar{t}$ ($\bar{A}$). There is a natural partial order on U_P^E (and B_P^E), defined as: $\bar{s} \leq \bar{t}$ if there exist representants s' of $\bar{s}$ and t' of $\bar{t}$ in $Term_P$ and a substitution θ, such that $s' = t'\theta$. Throughout the paper, S will denote a subset of B_P^E. We define its closure under $\leq$ as: $S^c = \{\bar{A} \in B_P^E \mid \exists \bar{B} \in S : \bar{A} \leq \bar{B}\}$.

Definition 2.1 P is *terminating with respect to* S if for any representant A' of any element $\bar{A}$ of S, every SLD-tree for $(P, \leftarrow A')$ is finite.

Denoting the classical notion of a Herbrand Base (of ground atoms) over P as B_P, then with the terminology of [Bezem 1989] we have:

Lemma 2.2 P is terminating if and only if it is terminating with respect to B_P.

Lemma 2.3 If all SLD-derivations for $(P, \leftarrow A)$ are finite, and θ is any substitution, then all SLD-derivations for $(P, \leftarrow A\theta)$ are finite.

From lemma 2.3 it follows that in order to verify definition 2.1 for a set $S \subseteq B_P^E$, it suffices to verify the finiteness of the SLD-trees for $(P, \leftarrow A)$ for only one representant of each element in $\bar{A}$. It also follows that P is terminating with respect to a set $S \subseteq B_P^E$ if and only if it is terminating with respect to S^c. In fact, given that P terminates with respect to S, it will in general be terminating with respect to a larger set of atoms than those in S^c. It is clear that if all SLD-trees for $(P, \leftarrow A)$ are finite, and if $H \leftarrow B_1, \ldots, B_n$ is a clause in P, such that A and H unify, then all SLD-trees for $(P, \leftarrow B_i\theta), i = 1, \ldots, n$, where $\theta = mgu(A, H)$, are finite. We can characterise the complete set of terminating atoms associated to a given set S as follows.

Definition 2.4 For any $T \subseteq B_P^E$, define $T_P^{-1}(T) = \{\overline{B_i\theta} \in B_P^E \mid A'$ is a representant of $\bar{A} \in T$, $H \leftarrow B_1, \ldots, B_n$ is a clause in P, $\theta = mgu(A', H)$ and $1 \leq i \leq n\}$.

Denote $\mathcal{H}_S = \{T \in 2^{B_P^E} \mid S^c \subseteq T\}$. $\mathcal{H}_S$ is a complete lattice with bottom element S^c.

Definition 2.5 $R_S : \mathcal{H}_S \to \mathcal{H}_S : R_S(T) = T \cup T_P^{-1}(T)^c$.

Lemma 2.6 R_S is continuous.

As a result, the least fix-point for R_S is $R_S\uparrow\omega$.

Lemma 2.7 P is terminating with respect to S if and only if P is terminating with respect to $R_S\uparrow\omega$.

As a result of our construction (in fact: as the very purpose of it), $R_S\uparrow\omega$ contains every call in every SLD-tree for any atomic goal of S. Formally:

Proposition 2.8 Let $call(P, S)$ denote the set of all atoms B, such that B is the subgoal selected by the computation rule in some goal of some SLD-tree for a pair $(P, \leftarrow A)$, with A the representant of an element of S. Then, $call(P, S) \subseteq R_S\uparrow\omega$.

We now introduce a variant of the definition of a level mapping, where the mapping is defined on equivalence classes of calls.

Definition 2.9 (level mapping)
A *level mapping with respect to a set* $S \subseteq B_P^E$ is a function $|.| : R_S\uparrow\omega \to I\!N$. A level mapping $|.|$ is called *rigid* if for all $\bar{A} \in R_S\uparrow\omega$ and for any substitution θ, $|\bar{A}| = |\overline{A\theta}|$, i.e. the level of an atom remains invariant under substitution.

With slight abuse of notation, we will often write $|A|$, where A is a representant of $\bar{A} \in B_P^E$. The associated notion of recurrency with respect to S will not be defined on ground instances of clauses, but instead on all instances $(H \leftarrow B_1, \ldots, B_n)\theta$ of clauses $H \leftarrow B_1, \ldots, B_n$ of P, such that $\theta = mgu(A, H)$, where A is a representant of an element of $R_S\uparrow\omega$. The definition in [Bezem 1989] does not explicitly impose a decrease of the level mapping at each inference step. The level mapping's values should only decrease for ground instances of clauses. By considering more general instances of clauses (as above), we can explicitly impose a decrease of the level mapping's value during (recursive) inference steps. As a result, the adapted level mapping no longer needs to perform different functionalities at once, and we can concentrate on the real structure of the recursion.

Now, concerning this recursive structure, there are a number of different possibilities for a new definition of recurrency, depending on how we aim to deal with indirect recursion. In order not to confuse all issues involved, we first provide a definition for programs P, relying only on direct recursion.

Definition 2.10 A (directly recursive) program P is *recurrent with respect to* S, if there exists a level mapping $|.|$ with respect to S, such that:

- for any A' representant of $\bar{A} \in R_S\uparrow\omega$,
- for any clause $H \leftarrow B_1, \ldots, B_n$ in P, such that $mgu(A', H) = \theta$ exists,
- for any atom $B_i, 1 \leq i \leq n$, with the same predicate symbol as H: $|A'| > |B_i\theta|$.

What is expressed in this definition is that for any two recursively descending calls with a same predicate symbol in any SLD-tree for (representants of) atoms in S, the level mapping's value should decrease. This condition has the advantage of being perfectly natural and therefore, of being easy to verify in an automated way. The only possible problem in view of automation is that it requires the computation of $R_S\uparrow\omega$. But, this problem is precisely the type of problem that can easily be solved (or approximated) through abstract interpretation (see section 4).

In the presence of indirect recursion, we need a more complex definition, that deals with the problem that a recursive call with a same predicate symbol as an ancestor call may only appear after a finite number of inference steps (instead of in the body of the particular instance of the applied clause). This can be done in several ways. We first provide a definition related to the concept of a resultant of a finite (incomplete) derivation. Based on this definition, we prove the equivalence with termination. After that, we provide a more practical condition, of which definition 2.10 is an obvious instance for the case of direct recursion.

First, we need some additional terminology.

Definition 2.11 Let A be an atom and $(G_0 = \leftarrow A)$, $G_1, G_2, \ldots, G_n$, $(n > 0)$, a finite, incomplete SLD-derivation for $(P, \leftarrow A)$. Let $\theta_1, \ldots, \theta_n$ be the corresponding sequence of substitutions, and let $\theta = \theta_1\theta_2\cdots\theta_n$ and $G_n = \leftarrow B_1, \ldots, B_m$. With the terminology of [Lloyd and Shepherdson 1991] we say that $A\theta \leftarrow B_1, \ldots, B_m$ is the *resultant* of the derivation.

Definition 2.12 A resultant $A\theta \leftarrow B_1, \ldots, B_m$ of a derivation $(G_0 = \leftarrow A), G_1, \ldots, G_n$, is a *recursive resultant for* A if there exists i $(1 \leq i \leq m)$, such that B_i has the same predicate symbol as A.

Definition 2.13 (recurrency wrt a set of atoms)
A program P is *recurrent with respect to* S, if there exists a level mapping, $|.|$, with respect to S, such that:

- for any A' representant of $\tilde{A} \in R_S\uparrow\omega$,
- for any recursive resultant $A'\theta \leftarrow B_1, \ldots, B_m$, for A',
- for any atom B_i, $1 \leq i \leq m$, with the same predicate symbol as A': $|A'| > |B_i|$.

Proposition 2.14 If P is recurrent with respect to S, then P terminates with respect to S.

Just as in the framework of Bezem, the converse statement holds as well.

Theorem 2.15
P is recurrent with respect to S if and only if it is terminating with respect to S.

One of the nice consequences of this result is that we can now relate the concept of a recurrent program in the sense of [Bezem 1989] to recurrency with respect to a set of (ground) atoms.

Corollary 2.16 P is recurrent if and only if it is recurrent with respect to B_P.

It may seem surprising to the reader that two apparently very different notions such as recurrency and recurrency with respect to B_P coincide. It is our experience from our work in termination of unfolding in the context of partial deduction ([Bruynooghe *et al.* 1991]) that this is not unusual. The reason is that conditions occurring in these contexts require the *"existence"* of some well-founded measure. The specific properties of such measures can take totally different form without loosing the termination property. The only real difference lies in the practicality.

We conclude the section by introducing a condition that implies definition 2.13. This condition has the advantage over definition 2.13 that it does not rely on the verification of some property for each of a potentially infinite number of recursive resultants. Instead it only requires such a verification for a finite number of clauses, which can be characterised through the minimal, cyclic collections of P.

Definition 2.17 (minimal cyclic collection)
A minimal cyclic collection of P is a finite sequence of clauses of P:

$$\begin{array}{lcl} A_1 & \leftarrow & B_1^1, \ldots, A_2', \ldots, B_{n_1}^1 \\ & \vdots & \\ A_m & \leftarrow & B_1^m, \ldots, A_{m+1}', \ldots, B_{n_m}^m \end{array}$$

such that:

- for each pair $(i \neq j)$, the heads of the clauses, A_i and A_j, are atoms with distinct predicate symbols,
- A_i and A_i' have the same predicate symbols $(1 < i \leq m)$,
- A_{m+1}' has the same predicate symbol as A_1.

Only a finite number of minimal cyclic collections exists. They can easily be characterised and computed from the predicate dependency graph for P.

Proposition 2.18
Let $S \subseteq B_P^g$ and $|.|$ a *rigid* level mapping with respect to S, such that for any minimal cyclic collection of P (after standardizing apart),

$$\begin{array}{lcl} A_1 & \leftarrow & B_1^1, \ldots, A_2', \ldots, B_{n_1}^1 \\ & \vdots & \\ A_m & \leftarrow & B_1^m, \ldots, A_{m+1}', \ldots, B_{n_m}^m \end{array}$$

and for any $\tilde{A}_1, \ldots, \tilde{A}_m \in R_S\uparrow\omega$, with $A_1'', \ldots, A_m''$ as their respective representants, and $\theta_i = mgu(A_i, A_i'')$, $(1 \leq i \leq m)$, the following condition holds:

$$\left\{ \begin{array}{lcl} |A_2'\theta_1| & \geq & |A_2''| \\ & \vdots & \\ |A_m'\theta_{m-1}| & \geq & |A_m''| \end{array} \right\}$$
$$\Downarrow$$
$$|A_1''| > |A_{m+1}'\theta_m|.$$

Then, P is recurrent with respect to S.

The conditions in proposition 2.18 seem rather unnatural at first sight and need some clarification. First, observe that in the case of direct recursion — except for the rigidity of the level mapping — the conditions coincide with those of definition 2.10.

For the case of indirect recursion, the conditions that one would intuitively expect, are that for each minimal cyclic collection

$$\begin{array}{rcl} A_1 & \leftarrow & B_1^1, \ldots, A_2', \ldots, B_{n_1}^1 \\ & \vdots & \\ A_m & \leftarrow & B_1^m, \ldots, A_{m+1}', \ldots, B_{n_m}^m \end{array}$$

and each A_1'' representant of $\tilde{A}_1 \in R_S\uparrow\omega$, such that $\theta = mgu(A_1'', A_1)$ and $\theta_i = mgu(A_i', A_i)$, $1 < i \leq m$, exist and are consistent, we have

$$|A_1''| > |A_{m+1}'\theta\theta_1 \cdots \theta_m|.$$

The problem is that such a condition is not correct. Consider the clauses:

$$\begin{array}{lcll} p(a,[\_|X]) & \leftarrow & p(b,X). & (cl1) \\ p(b,X) & \leftarrow & q(a,[\_|X]). & (cl2) \\ q(b,X) & \leftarrow & p(a,[\_|X]). & (cl3) \\ q(a,[\_|X]) & \leftarrow & q(b,X). & (cl4) \end{array}$$

There are 4 associated minimal collections: (cl1), (cl2,cl3), (cl3,cl2) and (cl4). Consider for instance the derivation $\leftarrow p(a,[\_,\_])$, $\leftarrow p(b,[\_])$, $\leftarrow q(a,[\_,\_])$, $\leftarrow q(b,[\_])$, $\leftarrow p(a,[\_,\_])$.

The problem is caused by resultants associated to derivations that start with a clause from one minimal cyclic collection — say (cl2) in the collection (cl2,cl3) — then shift to applying another collection, (cl4), and only after this resume the first collection and apply clause (cl3). The head of the third clause, $q(b,X)$, does not unify with $q(a,[\_|X'])$, and therefore, the condition on the cycle (cl2,cl3) can not be applied.

So, we have to impose the condition in proposition 2.18. It states that, even if the next call in the traversal of a minimal collection (A_i'') is not really related — as an instance — to a call we obtained earlier ($A_i'\theta_{i-1}$), but if — through the intermediate computation in another minimal collection — the level between these two has decreased anyway, then the final conclusion between the original call to the collection and the indirectly depending one must still hold. We will not discuss the condition any further here, but we will return to its practicality in section 4.

3 Acceptability with respect to a set of atoms

All definitions and propositions from the previous section can be specialised for the Prolog computation rule. Following [Apt and Pedreschi 1990], we call an SLD-derivation that uses Prolog's left-to-right computation rule, an *LD-derivation*.

Definition 3.1 (left termination wrt S) Let S be a subset of B_P^E. A program P is left-terminating with respect to S if for any representant A of any element of S, every LD-derivation is finite.

Recall definitions 2.4 and 2.5. The motivation behind these definitions was finding an overestimation of all calls that are possible in any SLD-derivation using an arbitrary computation rule. The fact that no fixed computation rule is used, forces us to take the closure under all possible instantiations in definition 2.5, and hence $R_S\uparrow\omega$ contains in general a lot more calls than can really occur when a particular computation rule is chosen.

In this section, we focus our analysis on computations that use Prolog's left-to-right computation rule. Therefore, adapted definitions of the T_P^{-1} and R_S functions are needed.

Definition 3.2 For any $T \subseteq B_P^E$, define: $\mathcal{P}_P^{-1}(T) = \{\overline{B_i\theta\sigma_1 \cdots \sigma_{i-1}} \in B_P^E \mid A'$ is a representant of $\tilde{A} \in T$, $H \leftarrow B_1, \ldots, B_n$ is a clause in P, $\theta = mgu(A', H)$, $1 \leq i \leq n$, $\exists\sigma_1, \ldots, \sigma_{i-1}$, such that $\forall j = 1, \ldots, i-1$: σ_j is an answer for $(P, \leftarrow B_j\theta\sigma_1 \cdots \sigma_{j-1})\}$.

The answer substitutions σ_j are computed using LD-resolution. Let $\mathcal{H}_S^{l-r}$ denote $\{T \in 2^{B_P^E} \mid S \subseteq T\}$.

Definition 3.3 $R_S^{l-r} : \mathcal{H}_S^{l-r} \rightarrow \mathcal{H}_S^{l-r} : R_S^{l-r}(T) = T \cup \mathcal{P}_P^{-1}(T)$

In a completely analogous way as in the previous section, we find that R_S^{l-r} is continuous. Hence, the least fix point $R_S^{l-r}\uparrow\omega$ contains all atoms that can possibly occur as a call when P is executed under the Prolog computation rule, and when a representant of an element from S is used as query.

Level mappings are now defined on R_S^{l-r}. Recursive resultants are constructed using the left-to-right computation rule. This allows us to consider only recursive resultants of the form $p(s_1, \ldots, s_n) \leftarrow p(t_1, \ldots, t_n), B_2, \ldots, B_m$. The analogue of recurrency with respect to a set S of atoms, is *acceptability with respect to S*.

Definition 3.4 (acceptability wrt a set of atoms) A program P is *acceptable with respect to S*, if there exists a level mapping $|.|$ with respect to S, such that for any $p(s_1, \ldots, s_n)$, representant of an element in $R_S^{l-r}\uparrow\omega$, and for any recursive resultant $p(s_1, \ldots, s_n)\theta \leftarrow p(t_1, \ldots, t_n), B_2, \ldots, B_m$: $|p(s_1, \ldots, s_n)| > |p(t_1, \ldots, t_n)|$.

Theorem 3.5
P is acceptable with respect to S if and only if it is left-terminating with respect to S.

As in section 2, we provide a more practical, sufficient condition. The result is completely analogous to proposition 2.18.

Proposition 3.6
Let $S \subseteq B_P^E$ and $|.|$ a level mapping with respect to S, such that for any minimal cyclic collection of P (after standardizing apart),

$$\begin{array}{rcl} A_1 & \leftarrow & B_1^1, \ldots, B_{i_1}^1, A_2', \ldots, B_{n_1}^1 \\ & \vdots & \\ A_m & \leftarrow & B_1^m, \ldots, B_{i_m}^m, A_{m+1}', \ldots, B_{n_m}^m \end{array}$$

and for any $\tilde{A}_1, \ldots, \tilde{A}_m \in R_S^{l-r}\uparrow\omega$, with $A_1'', \ldots, A_m''$ as their respective representants, and with $\theta_j = mgu(A_j, A_j'')$ $(1 \leq j \leq m)$ and σ_k^j is a computed answer substitution for $(P, \leftarrow B_k^j\theta_j\sigma_1^j \cdots \sigma_{k-1}^j)$ $(1 \leq k \leq i_j)$, the following condition holds:

$$\left\{ \begin{array}{lcl} |A_2'\theta_1\sigma_1^1 \cdots \sigma_{i_1}^1| & \geq & |A_2''| \\ & \vdots & \\ |A_m'\theta_{m-1}\sigma_1^{m-1} \cdots \sigma_{i_{m-1}}^{m-1}| & \geq & |A_m''| \end{array} \right\}$$
$$\Downarrow$$
$$|A_1''| > |A_{m+1}'\theta_m\sigma_1^m \cdots \sigma_{i_m}^m|,$$

Then, P is acceptable with respect to S.

4 Practicality and automation

A fully automated technique needs to address the following issues:

- safe approximations of $R_S\uparrow\omega$ and $R_S^{l-r}\uparrow\omega$ must be computed,
- precise and natural level mappings are needed, and
- the conditions in propositions 2.18 and 3.6 must be automatically verifiable.

For left termination, there is one extra issue:

- some properties of the answer substitutions for the atoms in $R_S^{l-r}\uparrow\omega$ are needed; in particular, after application of a computed answer substitution we want an estimation of the relationship between the sizes of the arguments of the atoms in $R_S^{l-r}\uparrow\omega$.

Concerning the first issue, observe that in practice, the sets of atoms S in the framework are likely to be specified in terms of call patterns over some abstract domain. The framework contains no implicit restriction on the kind of abstractions that are used for this purpose. They could be either expressing mode or type information, or even combined mode and type information — as in the rigid or integrated types of [Janssens and Bruynooghe 1990]. Abstract interpretation can be applied to automatically infer a safe approximation of $R_S\uparrow\omega$ or $R_S^{l-r}\uparrow\omega$ (see [Janssens and Bruynooghe 1990]).

Automated techniques for proving termination use various types of *norms*. A *norm* is a mapping $||.|| : U_P^E \rightarrow \mathbb{N}$. Several examples of norms can be found in the literature. When dealing with lists, it is often appropriate to use *list-length*, which gives the depth of the rightmost branch in the tree representation of the term. A more general norm is *term-size*, which counts the number of function symbols in a term. Another frequently used norm is *term-depth*, which gives the maximum depth of (the tree representation of) a term.

However, we restrict ourselves to *semi-linear* norms, which were defined in [Bossi *et al.* 1991].

Definition 4.1 (semi-linear norm)
A norm $||.||$ is *semi-linear* if it satisfies the folowing conditions:

- $||V|| = 0$ if V is a variable, and
- $||f(t_1, \ldots, t_n)|| = c + ||t_{i_1}|| + \cdots + ||t_{i_m}||$ where $c \in \mathbb{N}$, $1 \leq i_1 < \cdots < i_m \leq n$ and $c, i_1, \ldots, i_m$ depend only on f/n.

Examples of semi-linear norms are *list-length* and *term-size*.

As was pointed out in [Bossi *et al.* 1991], proving termination is significantly facilitated if the norm of a term remains invariant under substitution. Such terms are called *rigid*.

Definition 4.2 (rigid term; see [Bossi *et al.* 1991])
Let $||.||$ be a (semi-linear) norm. A term t is *rigid* with respect to $||.||$ if for any substitution σ, $||t\sigma|| = ||t||$.

Rigidity is a generalisation of groundness; by using this concept it is possible to avoid restricting the definition of a norm to ground terms only, a restriction that is often found in the literature.

Given a semi-linear norm and a set of atoms S, a very natural level mapping with respect to S can be associated to them.

Definition 4.3 (natural level mapping)
Given is a semi-linear norm $||.||$ and a set of atoms S. $|.|_{nat}$, the *natural level mapping induced by* S, is defined as follows: $\forall p(t_1, \ldots, t_n) \in R_S\uparrow\omega$:

$$\begin{array}{rcll} |p(t_1, \ldots, t_n)|_{nat} & = & \sum_{i \in I} ||t_i||, & \text{if } I \neq \emptyset \\ & = & 0 & \text{otherwise,} \end{array}$$

with $I = \{i \mid \forall p(u_1, \ldots, u_n) \in R_S\uparrow\omega : u_i \text{ is rigid}\}$.

Let us illustrate the practicality of such mappings — and of the framework itself — with some examples.

Example 4.4
Reconsider example 1.4 from the introduction. Assume that $S = \{p(x) \mid x \text{ is a nil-terminated list}\}$. Let $\|.\|_l$ be the *list-length* norm. The argument positions of all atoms in $R_S\uparrow\omega$ are rigid under this norm. So, $|p(x)|_{nat} = \|x\|_l$ and $|q(x)|_{nat} = \|x\|_l$. The program is directly recursive, so that it suffices to verify the conditions of definition 2.10.

For the clause $p([H|T]) \leftarrow q([H|T]), p(T)$ and for each call $p(x) \in R_S\uparrow\omega$, with $\theta = mgu(x, [H|T])$, we have $|p(x)|_{nat} > |p(T)\theta|_{nat}$. By the same argument, the condition on the clause $q([H|T]) \leftarrow q(T)$ holds as well. Thus, the program is recurrent with respect to S under the natural, *list-length* level mapping with respect to S.

As a second example, we take a program with indirect recursion. It defines some form of well-formed expressions built from integers and the function symbols $+/2$, $*/2$ and $-/1$.

Example 4.5

$$\begin{array}{lcll} e(X+Y) & \leftarrow & f(X), e(Y). & (cl1) \\ e(X) & \leftarrow & f(X). & (cl2) \\ \\ f(X*Y) & \leftarrow & g(X), f(Y). & (cl3) \\ f(X) & \leftarrow & g(X). & (cl4) \\ \\ g(-(X)) & \leftarrow & e(X). & (cl5) \\ g(X) & \leftarrow & integer(X). & (cl6) \end{array}$$

The obvious choice for a level mapping for this program is *term-size*. However, the program is not recurrent in the sense of [Bezem 1989] with respect to this norm. Since it is clearly terminating, a level mapping exists. The most natural mapping (in the sense of [Bezem 1989]) we were able to come up with is:

$$\begin{array}{lcl} |e(x)| & = & 3 \times term\text{-}size(x) + 2 \\ |f(x)| & = & 3 \times term\text{-}size(x) + 1 \\ |g(x)| & = & 3 \times term\text{-}size(x). \end{array}$$

In the context of our framework, consider the set $S = \{e(x) \mid x \text{ is ground}\}$. Through abstract interpretation, we can find that $R_S\uparrow\omega \subseteq B_P$.

Let $\|.\|_t$ be the *term-size* norm. Again, the argument positions of all atoms in $R_S\uparrow\omega$ are rigid (even ground) under this norm. Thus, $|e(x)|_{nat} = \|x\|_t$, $|f(x)|_{nat} = \|x\|_t$ and $|g(x)|_{nat} = \|x\|_t$. The program contains essentially[1] 6 minimal, cyclic collections: (cl1), (cl3), (cl1, cl3, cl5), (cl1, cl4, cl5), (cl2, cl3, cl5), (cl2, cl4, cl5).

Let us consider, as an example, the third collection:

$$\begin{array}{lcl} e(X+Y) & \leftarrow & f(X), e(Y). \\ f(X'*Y') & \leftarrow & g(X'), f(Y'). \\ g(-(X'')) & \leftarrow & e(X''). \end{array}$$

[1]Since collections are sequences of clauses, cyclic permutations should be considered as well.

Assume that $e(x)$, $f(y)$ and $g(z)$ are any atoms with ground terms x, y and z, and that:

$$\begin{array}{l} \theta_1 = mgu(e(x), e(X+Y)) \\ \theta_2 = mgu(f(y), f(X'*Y')) \\ \theta_3 = mgu(g(z), g(-(X''))). \end{array}$$

Also assume that $|f(X)\theta_1| \geq |f(y)|$ and $|g(X')\theta_2| \geq |g(z)|$. We then have $|e(x)| > |f(X)\theta_1| \geq |f(y)| > |g(X')\theta_2| \geq |g(z)| > |e(X'')\theta_3|$, so that $|e(x)| > |e(X'')\theta_3|$, and the conditions of proposition 2.18 (for the third cycle) are fulfilled. All other cycles can be verified in a similar way. The conclusion is that the program is recurrent with respect to S and the very natural *term-size* level mapping.

In the context of left termination, definition 4.3 can be adapted to produce equally natural level mappings with respect to a set S. Obviously, $R_S\uparrow\omega$ should be replaced by $R_S^{l-r}\uparrow\omega$. In the context of left termination there is an extra issue, namely, (an approximation of) the set of possible answer substitutions for an atom is needed. The next example illustrates how this is handled.

Example 4.6

$$\begin{array}{l} p([],[]). \\ p([H|T],[G|S]) \leftarrow d(G,[H|T],U), p(U,S). \\ \\ d(H,[H|T],T). \\ d(G,[H|T],[H|U]) \leftarrow d(G,T,U). \end{array}$$

Assume that $S = \{p(x,y) \mid x \text{ is a nil-terminated list and } y \text{ is free}\}$. Notice that $R_S\uparrow\omega$ contains the set $\{p(x,y) \mid x \text{ and } y \text{ are free variables}\}$. We are not able to define a level mapping on $R_S\uparrow\omega$ that can be used to prove recurrency with respect to S. This is not surprising, since P is not terminating with respect to S.

However, program P is left terminating with respect to S. We prove this by showing that P is acceptable with respect to S. The set $R_S^{l-r}\uparrow\omega$ is the union of $\{p(x,y) \mid x \text{ is a nil-terminated list and } y \text{ is free}\}$ and $\{d(x,y,z) \mid x \text{ and } z \text{ are free variables and } y \text{ is a nil-terminated list}\}$. This can be found by using abstract interpretation. Since there is only direct recursion in program P, it suffices to show that: (1) for any $p(x,y) \in R_S^{l-r}\uparrow\omega$, $|p(x,y)| > |p(U,S)\theta\sigma|$, where $\theta = mgu(p(x,y), p([H|T],[G|S]))$ and σ is a computed answer substitution for $(P, \leftarrow d(G,[H|T],U)\theta)$, and (2) for any $d(x,y,z) \in R_S^{l-r}\uparrow\omega$, $|d(x,y,z)| > |d(G,T,U)\theta|$, where $\theta = mgu(d(x,y,z), d(G,[H|T],[H|U]))$.

Now, in practice, the statement "σ is a computed answer substitution for $(P, \leftarrow d(G,[H|T],U)\theta)$" can be replaced by "$\|[H|T]\theta\sigma\|_t = \|U\theta\sigma\|_t + 1$". This latter statement is a so-called *linear size relation*, which expresses a relation between the norms of the arguments of the atoms in the success set of the program. Alternatively, it can also be interpreted as a (non-Herbrand)

model of the program. For more details we refer to [Verschaetse and De Schreye 1992], where we describe an automated technique for deriving linear size relations.

By taking this information into account, and by taking $|p(x,y)| = ||x||_l$ for any $p(x,y) \in R_S^{l-r} \uparrow \omega$ — notice that x is rigid with respect to $||.||_l$ — we find: $|p(x,y)| = ||x||_l = ||[H|T]\theta||_l = ||[H|T]\theta\sigma||_l = ||U\theta\sigma||_l + 1 > ||U\theta\sigma||_l = |p(U,S)\theta\sigma|$.

The second inequality, $|d(x,y,z)| > |d(G,T,U)\theta|$, is more easy to prove. This time, the *list-length* of the second argument can be taken as level mapping. Since both inequalities hold, we can conclude that the program is acceptable with respect to the set of atoms that is considered.

Automatic verification of the conditions for recurrency and acceptability is handled by reformulating them into a problem of checking the solvability of a linear system of inequalities. This part of the work is described in more detail in [De Schreye and Verschaetse 1992].

References

[Apt and Pedreschi 1990] K. R. Apt and D. Pedreschi. Studies in pure Prolog: termination. In *Proceedings Esprit symposium on computational logic*, pages 150–176, Brussels, November 1990.

[Baudinet 1988] M. Baudinet. Proving termination properties of Prolog programs: a semantic approach. In *Proceedings of the 3rd IEEE symposium on logic in computer science*, pages 336–347, Edinburgh, July 1988. Revised version to appear in Journal of Logic Programming.

[Bezem 1989] M. Bezem. Characterizing termination of logic programs with level mappings. In *Proceedings NACLP'89*, pages 69–80, 1989.

[Bossi *et al.* 1991] A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. Technical Report 4/29, CNR, Department of Mathematics, University of Padova, March 1991.

[Bruynooghe *et al.* 1991] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction of logic programs. In *Proceedings ILPS'91*, pages 117–131, San Diego, October 1991. MIT Press.

[Cavedon 1989] L. Cavedon. Continuity, consistency, and completeness properties for logic programs. In *Proceedings ICLP'89*, pages 571–584, June 1989.

[De Schreye and Verschaetse 1992] D. De Schreye and K. Verschaetse. Termination analysis of definite logic programs with respect to call patterns. Technical Report CW 138, Department Computer Science, K.U.Leuven, January 1992.

[Falaschi *et al.* 1989] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behaviour of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[Janssens and Bruynooghe 1990] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. Technical Report CW 107, Department of Computer Science, K.U.Leuven, March 1990. To appear in Journal of Logic Programming, in print.

[Lloyd and Shepherdson 1991] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3 & 4):217–242, October/November 1991.

[Plümer 1990] L. Plümer. *Termination proofs for logic programs*. Lecture Notes in Artificial Intelligence 446. Springer-Verlag, 1990.

[Sohn and Van Gelder 1991] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *Proceedings 10th symposium on principles of database systems*, pages 216–226. Acm Press, May 1991.

[Ullman and Van Gelder 1988] J. D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal ACM*, 35(2):345–373, April 1988.

[Vasak and Potter 1986] T. Vasak and J. Potter. Characterisation of terminating logic programs. In *Proceedings 1986 symposium on logic programming*, pages 140–147, Salt Lake City, 1986.

[Verschaetse and De Schreye 1991] K. Verschaetse and D. De Schreye. Deriving termination proofs for logic programs, using abstract procedures. In *Proceedings ICLP'91*, pages 301–315, Paris, June 1991. MIT Press.

[Verschaetse and De Schreye 1992] K. Verschaetse and D. De Schreye. Automatic derivation of linear size relations. Technical Report CW 139, Department Computer Science, K.U.Leuven, January 1992.

[Wang and Shyamasundar 1990] B. Wang and R. K. Shyamasundar. Towards a characterization of termination of logic programs. In *Proceedings of international workshop PLILP'90*, Lecture Notes in Computer Science 456, pages 204–221, Linköping, August 1990. Springer-Verlag.