On Modular Termination Proofs of General Logic Programs

ANNALISA BOSSI, NICOLETTA COCCO, SABINA ROSSI
Dipartimento di Informatica, Università Ca' Foscari di Venezia
via Torino 155, 30172 Venezia, Italy

SANDRO ETALLE
Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
and
CWI – Center for Mathematics and Computer Science,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

We propose a modular method for proving termination of general logic programs (i.e., logic programs with negation). It is based on the notion of acceptable programs, but it allows us to prove termination in a truly modular way. We consider programs consisting of a hierarchy of modules and supply a general result for proving termination by dealing with each module separately. For programs which are in a certain sense well-behaved, namely well-moded or well-typed programs, we derive both a simple verification technique and an iterative proof method. Some examples show how our system allows for greatly simplified proofs.

1 Introduction

It is standard practice to tackle a large proof by decomposing it into more manageable pieces (lemmata or modules) and proving them separately. By composing appropriately these simpler results, one can then obtain the final proof. This methodology has been recognized an important one also when proving termination of logic programs. Moreover most practical logic programs are engineered by assembling different modules and libraries, some of which might be pre-compiled or written in a different programming language. In such a situation, a compositional methodology for proving termination is of crucial importance.

The first approach to modular termination proofs of logic programs has been proposed by Apt and Pedreschi in (Apt and Pedreschi 1994). It extends the seminal work on *acceptable* programs (Apt and Pedreschi 1993) which provides an algebraic characterization of programs terminating under Prolog left-to-right selection rule. The class of acceptable programs contains programs which terminate on ground queries. To prove acceptability one needs to determine a measure on literals (*level mapping*) such that, in any clause, the measure of the head is greater than the measure of each body literal. This implies the decreasing of the measure of the

literals resolved during any computation starting from a ground or *bounded* query and hence termination.

The significance of a modular approach to termination of logic programs has been recognized also by other authors; more recent proposals can be found in (Pedreschi and Ruggieri 1996, Marchiori 1996, Verbaeten, Sagonas and De Schreye 1999, Etalle, Bossi and Cocco 1999, Verbaeten, Sagonas and De Schreye 2001).

All previous proposals (with the exception of (Verbaeten et al. 1999, Etalle et al. 1999)) require the existence of a relation between the level mappings used to prove acceptability of distinct modules. This is not completely satisfactory: it would be nice to be able to put together modules which were independently proved terminating, and be sure that the resulting program is still terminating.

We propose a modular approach to termination which allows one to reason independently on each single module and get a termination result on the whole program. We consider general logic programs, i.e., logic programs with negation, employing SLDNF-resolution together with the leftmost selection rule (also called *LDNF-resolution*) as computational mechanism. We consider programs which can be divided into modules in a hierarchical way, so that each module is an extension of the previous ones. We show that in this context the termination proof of the entire program can be given in terms of separate proofs for each module, which are naturally much simpler than a proof for the whole program. While assuming a hierarchy still allows one to tackle most real-life programs, it leads to termination proofs which, in most cases, are extremely simple.

We characterize the class of queries terminating for the whole program by introducing a new notion of boundedness, namely *strong boundedness*. Intuitively, strong boundedness captures the queries which preserve (standard) boundedness through the computation. By proving acceptability of each module wrt. a level mapping which measures only the predicates defined in that module, we get a termination result for the whole program which is valid for any strongly bounded query. Whenever the original program is decomposed into a hierarchy of small modules, the termination proof can be drastically simplified with respect to previous modular approaches. Moreover strong boundedness can be naturally guaranteed by common persistent properties of programs and queries, namely properties preserved through LDNF-resolution such as *well-modedness* (Dembiński and Maluszyński 1985) or *well-typedness* (Bronsard, Lakshman and Reddy 1992).

The paper is organized as follows. Section 2 contains some preliminaries. In particular we briefly recall the key concepts of LDNF-resolution, acceptability, boundedness and program extension. Section 3 contains our main results which show how termination proofs of separate programs can be combined to obtain proofs of larger programs. In particular we define the concept of strongly bounded query and we prove that for general programs composed by a hierarchy of n modules, each one independently acceptable wrt. its own level mapping, any strongly bounded query terminates. In Section 4 we show how strong boundedness is naturally ensured by some program properties which are preserved through LDNF-resolution such as well-modedness and well-typedness. In Section 5 we show how these properties allow us to apply our general results also for proving termination of modular programs

in an iterative way. In Section 6 we compare our work with Apt and Pedreschi's approach. Other related works and concluding remarks are discussed in Section 7.

2 Preliminaries

We use standard notation and terminology of logic programming (Lloyd 1987, Apt 1990, Apt 1997). Just note that general logic programs are called in (Lloyd 1987) normal logic programs.

2.1 General Programs and LDNF-Resolution

A *general clause* is a construct of the form

$$H \leftarrow L_1, \ldots, L_n$$

with $(n \geq 0)$, where H is an atom and $L_1, \ldots, L_n$ are literals (i.e., either atoms or the negation of atoms). In turn, a *general query* is a possibly empty finite sequence of literals $L_1, \ldots, L_n$, with $(n \geq 0)$. A *general program* is a finite set of general clauses[1]. Given a query $Q := L_1, \ldots, L_n$, a *non-empty prefix of* Q is any query $L_1, \ldots, L_i$ with $i \in \{1, \ldots, n\}$. For a literal L, we denote by $rel(L)$ the predicate symbol of L.

Following the convention adopted in (Apt 1997), we use bold characters to denote sequences of objects (so that $\mathbf{L}$ indicates a sequence of literals $L_1, \ldots, L_n$, while $\mathbf{t}$ indicates a sequence of terms $t_1, \ldots, t_n$).

For a given program P, we use the following notations: B_P for the Herbrand base of P, $ground(P)$ for the set of all ground instances of clauses from P, $comp(P)$ for the Clark's completion of P (Clark 1978).

Since in this paper we deal with general queries, clauses and programs, we omit from now on the qualification "general", unless some confusion might arise.

We consider *LDNF-resolution*, and following Apt and Pedreschi's approach in studying the termination of general programs (Apt and Pedreschi 1993), we view LDNF-resolution as a top-down interpreter which, given a general program P and a general query Q, attempts to build a search tree for $P \cup \{Q\}$ by constructing its branches in parallel. The branches in this tree are called *LDNF-derivations* of $P \cup \{Q\}$ and the tree itself is called *LDNF-tree* of $P \cup \{Q\}$. Negative literals are resolved using the *negation-as-failure* rule which calls for the construction of a *subsidiary LDNF-tree*. If during this subsidiary construction the interpreter diverges, the (main) LDNF-derivation is considered to be infinite. An LDNF-derivation is finite also if during its construction the interpreter encounters a query with the first literal being negative and non-ground. In such a case we say that the LDNF-derivation *flounders*.

[1] In the examples through the paper, we will adopt the syntactic conventions of Prolog so that each query and clause ends with the period "." and "←" is omitted in the unit clauses.

By termination of a general program we actually mean termination of the underlying interpreter. Hence in order to ensure termination of a query Q in a program P, we require that all LDNF-derivations of $P \cup \{Q\}$ are finite.

By an *LDNF-descendant* of $P \cup \{Q\}$ we mean any query occurring during the LDNF-resolution of $P \cup \{Q\}$, including Q and all the queries occurring during the construction of the subsidiary LDNF-trees for $P \cup \{Q\}$.

For a non-empty query Q, we denote by $\mathit{first}(Q)$ the first literal of Q. Moreover we define $\mathit{Call}_P(Q) = \{\mathit{first}(Q') \mid Q' \text{ is an LDNF-descendant of } P \cup \{Q\}\}$. It is worth noting that if $\neg A \in \mathit{Call}_P(Q)$ and A is a ground atom, then $A \in \mathit{Call}_P(Q)$ too. Notice that, for definite programs, the set $\mathit{Call}_P(Q)$ coincides with the call set $\mathit{Call}(P, \{Q\})$ in (De Schreye, Verschaetse and Bruynooghe 1992, Decorte, De Schreye and Vandecasteele 1999).

The following trivial proposition holds.

Proposition 1
Let P be a program and Q be a query. All LDNF-derivations of $P \cup \{Q\}$ are finite iff for all positive literals $A \in \mathit{Call}_P(Q)$, all LDNF-derivations of $P \cup \{A\}$ are finite.

2.2 Acceptability and Boundedness

The method we are going to use for proving termination of modular programs is based on the concept of *acceptable* program (Apt and Pedreschi 1993). In order to introduce it, we start by the following definition, originally due to (Bezem 1993) and (Cavedon 1989).

Definition 2 (*Level Mapping*)
A *level mapping* for a program P is a function $|\ | : B_P \to \mathbf{N}$ of ground atoms to natural numbers. By convention, this definition is extended in a natural way to ground literals by putting $|\neg A| = |A|$. For a ground literal L, $|L|$ is called the *level* of L.

We will use the following notations. Let P be a program and p and q be relations. We say that p *refers to* q if there is a clause in P that uses p in its head and q in its body; p *depends on* q if (p, q) is in the reflexive, transitive closure of the relation *refers to*. We say that p and q are *mutually recursive* and write $p \simeq q$, if p depends on q and q depends on p. We also write $p \sqsupset q$, when p depends on q but q does not depend on p.

We denote by Neg_P the set of relations in P which occur in a negative literal in a clause of P and by Neg_P^* the set of relations in P on which the relations in Neg_P depend. P^- denotes the set of clauses in P defining a relation of Neg_P^*.

In the sequel we refer to the standard definition of model of a program and model of the completion of a program, see (Apt 1990, Apt 1997) for details. In particular we need the following notion of *complete model* for a program.

Definition 3 (*Complete Model*)
A model M of a program P is called *complete* if its restriction to the relations from Neg_P^* is a model of $\mathit{comp}(P^-)$.

Notice that if I is a model of $comp(P)$ then its restriction to the relations in Neg_P^* is a model of $comp(P^-)$; hence I is a complete model of P.

The following notion of acceptable program was introduced in (Apt and Pedreschi 1993). Apt and Pedreschi proved that such a notion fully characterizes left-termination, namely termination wrt. any ground query, both for definite programs and for general programs which have no LDNF-derivations which flounder.

Definition 4 (*Acceptable Program*)
Let P be a program, $|\ |$ be a level mapping for P and M be a complete model of P. P is called *acceptable wrt.* $|\ |$ *and* M if for every clause $A \leftarrow \mathbf{A}, B, \mathbf{B}$ in $ground(P)$ the following implication holds:

$$\text{if } M \models \mathbf{A} \text{ then } |A| > |B|.$$

Note that if P is a definite program, then both P^- and Neg_P^* are empty and M can be any model of P.

We also need the notion of bounded atom.

Definition 5 (*Bounded Atom*)
Let P be a program and $|\ |$ be a level mapping for P. An atom A is called *bounded wrt.* $|\ |$ if the set of all $|A'|$, where A' is a ground instance of A, is finite. In this case we denote by $max|A|$ the maximum value in this set.

Notice that if an atom A is bounded then, by definition of level mapping, also the corresponding negative literal, $\neg A$, is bounded.

Note also that, for atomic queries, this definition coincides with the definition of bounded query introduced in (Apt and Pedreschi 1993) in order to characterize terminating queries for acceptable programs. In fact, in case of atomic queries the notion of boundedness does not depend on a model.

2.3 Extension of a Program

In this paper we consider a hierarchical situation where a program uses another one as a subprogram. The following definition formalizes this situation.

Definition 6 (*Extension*)
Let P and R be two programs. A relation p is *defined in* P if p occurs in a head of a clause of P; a literal L is *defined in* P if $rel(L)$ is defined in P; P *extends* R, denoted $P \sqsupset R$, if no relation defined in P occurs in R.

Informally, P extends R if P defines new relations with respect to R. Note that P and R are independent if no relation defined in P occurs in R and no relation defined in R occurs in P, i.e. $P \sqsupset R$ and $R \sqsupset P$.

In the sequel we will study termination in a hierarchy of programs.

Definition 7 (*Hierarchy of Programs*)
Let $P_1, \ldots, P_n$ be programs such that for all $i \in \{1, \ldots, n-1\}$, $P_{i+1} \sqsupset (P_1 \cup \cdots \cup P_i)$. Then we call $P_n \sqsupset \cdots \sqsupset P_1$ a *hierarchy of programs*.

3 Hierarchical Termination

This section contains our main results which show how termination proofs of separate programs can be combined to obtain proofs of larger programs. We start with a technical result, dealing with the case in which a program consists of a hierarchical combination of two modules. This is the base both of a generalization to a hierarchy of n programs and of an iterative proof method for termination presented in Section 5. Let us first introduce the following notion of *P-closed* class of queries.

Definition 8 (*P-closed Class*)
Let $\mathcal{C}$ be a class of queries and P be a program. We say that $\mathcal{C}$ is *P-closed* if it is closed under non-empty prefix (i.e., it contains all the non-empty prefixes of its elements) and for each query $Q \in \mathcal{C}$, every LDNF-descendant of $P \cup \{Q\}$ is contained in $\mathcal{C}$.

Note that if $\mathcal{C}$ is P-closed, then for each query $Q \in \mathcal{C}$, $Call_P(Q) \subseteq \mathcal{C}$.

We can now state our first general theorem. Notice that if P extends R and P is acceptable wrt. some level mapping $|\ |$ and model M, then P is acceptable also wrt. the level mapping $|\ |'$ and M, where $|\ |'$ is defined on the Herbrand base of the union of the two programs $B_{P \cup R}$ and it takes the value 0 on the literals which are not defined in P (and hence, in particular, on the literals which occur in P but are defined in R). This shows that in each module it is sufficient to compare only the level of the literals defined inside it, while we can ignore literals defined outside the module. In the following we make use of this observation in order to associate to each module in a hierarchy a level mapping which is independent from the context.

Theorem 9
Let P and R be two programs such that P extends R, M be a complete model of $P \cup R$ and $\mathcal{C}$ be a $(P \cup R)$-closed class of queries. Suppose that

- P is acceptable wrt. a level mapping $|\ |$ and M,
- for all queries $Q \in \mathcal{C}$, all LDNF-derivations of $R \cup \{Q\}$ are finite,
- for all atoms $A \in \mathcal{C}$, if A is defined in P then A is bounded wrt. $|\ |$.

Then for all queries $Q \in \mathcal{C}$, all LDNF-derivations of $(P \cup R) \cup \{Q\}$ are finite.

Proof
By the fact that $\mathcal{C}$ is $(P \cup R)$-closed and Proposition 1, it is sufficient to prove that for all positive literals $A \in \mathcal{C}$, all LDNF-derivations of $(P \cup R) \cup \{A\}$ are finite. Let us consider an atom $A \in \mathcal{C}$.

If A is defined in R, then the thesis trivially holds by hypothesis.

If A is defined in P, A is bounded wrt. $|\ |$ by hypothesis and thus $max|A|$ is defined. The proof proceeds by induction on $max|A|$.

Base. Let $max|A| = 0$. In this case, by acceptability of P, there are no clauses in P whose head unifies with A and whose body is non-empty. Hence, the thesis holds.

Induction step. Let $max|A| > 0$. It is sufficient to prove that for all direct descendants $(L_1, \ldots, L_n)$ in the LDNF-tree of $(P \cup R) \cup \{A\}$, if θ_i is a computed answer for $P \cup \{L_1, \ldots, L_{i-1}\}$ then all LDNF-derivations of $(P \cup R) \cup \{L_i\theta_i\}$ are finite.

Let $c : H' \leftarrow L'_1, \ldots, L'_n$ be a clause of P such that $\sigma = mgu(H', A)$. Let $H = H'\sigma$ and for all $i \in \{1, \ldots, n\}$, let $L_i = L'_i\sigma$ and θ_i be a substitution such that θ_i is a computed answer of $L_1, \ldots, L_{i-1}$ in $P \cup R$.

We distinguish two cases. If L_i is defined in R then the thesis follows by hypothesis.

Suppose that L_i is defined in P. We prove that $L_i\theta_i$ is bounded and $max|A| > max|L_i\theta_i|$. The thesis will follow by the induction hypothesis.

Let γ be a substitution such that $L_i\theta_i\gamma$ is ground. By soundness of LDNF-resolution (Clark 1978), there exists γ' such that $M \models (L_1, \ldots, L_{i-1})\gamma'$ and $c\sigma\gamma'$ is a ground instance of c and $L_i\gamma' = L_i\theta_i\gamma$. Therefore

$$\begin{array}{rcll} |L_i\theta_i\gamma| & = & |L_i\gamma'| & \\ & = & |L'_i\sigma\gamma'| & (\text{since } L_i = L'_i\sigma) \\ & < & |H'\sigma\gamma'| & (\text{since } P \text{ is acceptable}) \\ & = & |A\sigma\gamma'| & (\text{since } \sigma = mgu(H', A)). \end{array}$$

Since A is bounded, we can conclude that $L_i\theta_i$ is bounded and also that $max|A| > max|L_i\theta_i|$. □

We are going to extend the above theorem in order to handle the presence of more than two modules. We need to introduce more notation. Let us consider the case of a program P consisting of a hierarchy $R_n \sqsupset \ldots \sqsupset R_1$ of distinct modules, and satisfying the property that each module, R_i, is acceptable wrt. a distinct level mapping, $|\ |_i$, and a complete model, M, of the whole program. Under these assumptions we identify a specific class of queries which terminate in the whole program. We characterize the class of terminating queries in terms of the following notion of strong boundedness. This class enjoys the property of being P-closed.

Definition 10 (*Strongly Bounded Query*)
Let the program $P := R_1 \cup \ldots \cup R_n$ be a hierarchy $R_n \sqsupset \ldots \sqsupset R_1$ and $|\ |_1, \ldots, |\ |_n$ be level mappings for $R_1, \ldots, R_n$, respectively. A query Q is called *strongly bounded wrt. P and $|\ |_1, \ldots, |\ |_n$* if

- for all atoms $A \in Call_P(Q)$, if A is defined in R_i (with $i \in \{1, \ldots, n\}$) then A is bounded wrt. $|\ |_i$.

Notice that the notion of boundedness for an atom (see Definition 5) does not depend on the choice of a particular model of P. As a consequence, also the definition of strong boundedness does not refer to any model of P; however, it refers to the LDNF-derivations of P. For this reason, a ground atom is always bounded but not necessarily strongly bounded. On the other hand, if A is strongly bounded then it is bounded too.

The following remark follows immediately.

Remark 11
Let the query Q be strongly bounded wrt. P and $|\ |_1, \ldots, |\ |_n$, where P is a hierarchy $R_n \sqsupset \cdots \sqsupset R_1$. Let $i \in \{1, \ldots, n\}$. If Q is defined in $R_1 \cup \ldots \cup R_i$ then Q is strongly bounded wrt. $R_1 \cup \ldots \cup R_i$ and $|\ |_1, \ldots, |\ |_i$.

In order to verify whether a query Q is strongly bounded wrt. a given program P one can perform a call-pattern analysis (Janssen and Bruynooghe 1992, Gabbrielli and Giacobazzi 1994, Codish and Demoen 1995) which allows us to infer information about the form of the call-patterns, i.e., the atoms that will be possibly called during the execution of $P \cup \{Q\}$. However this is not the only way for guaranteeing strong boundedness. There are classes of programs and queries for which strong boundedness can be proved in a straightforward way. This is shown in the following section.

Let us illustrate the notion of strong boundedness through an example.

Example 12
Let `LIST01` be the following program which defines the proper lists of `0`'s and `1`'s, i.e. lists containing only `0`'s and `1`'s and at least two distinct elements, as follows:

```
r1: list01([ ],0,0).
r2: list01([0|Xs],s(N0),N1)  ← list01(Xs,N0,N1).
r3: list01([1|Xs],N0,s(N1))  ← list01(Xs,N0,N1).

r4: length([ ],0).
r5: length([X|Xs],s(N))  ← length(Xs,N).

r6: plist01(Ls)  ← list01(Ls,N0,N1),
        ¬length(Ls,N0), ¬length(Ls,N1).
```

Let us distinguish two modules in `LIST01`: $R_1 = \{\mathtt{r_1}, \mathtt{r_2}, \mathtt{r_3}, \mathtt{r_4}, \mathtt{r_5}\}$ and $R_2 = \{\mathtt{r_6}\}$ (R_2 extends R_1). Let $|\ |_1$ be the natural level mapping for R_1 defined by:

$|\mathtt{list01}(ls, n0, n1)|_1 = |ls|_{\mathtt{length}}$
$|\mathtt{length}(ls, n)|_1 = |n|_{\mathtt{size}}$

where for a term t, if t is a list then $|t|_{\mathtt{length}}$ is equal to the length of the list, otherwise it is 0, while $|t|_{\mathtt{size}}$ is the number of function symbols occurring in the term t. Let also $|\ |_2$ be the trivial level mapping for R_2 defined by:

$|\mathtt{plist01}(ls)|_2 = \mathtt{1}$

and assume that $|L|_2 = 0$, if L is not defined in R_2.

Let us consider the following sets of atomic queries for $\mathtt{LIST01} := R_1 \cup R_2$:

$\mathcal{Q}_1$ = {$\mathtt{list01}(ls, n0, n1)|$ ls is a list, possibly non-ground, of a fixed length};
$\mathcal{Q}_2$ = {$\mathtt{length}(ls, n)|$ n is a ground term of the form either 0 or $\mathtt{s(s(...(0)))}$};
$\mathcal{Q}_3$ = {$\mathtt{plist01}(ls)|$ ls is a list, possibly non-ground, of a fixed length}.

By definition of $|\ |_1$, all the atoms in $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are bounded wrt. $|\ |_1$. Analogously, all the atoms in $\mathcal{Q}_3$ are bounded wrt. $|\ |_2$. Notice that for all atoms $A \in Call_P(\mathcal{Q}_j)$, with $j \in \{1, 2, 3\}$, there exists $k \in \{1, 2, 3\}$ such that $A \in \mathcal{Q}_k$. Hence, if A is defined in R_i then A is bounded wrt. $|\ |_i$. This proves that the set of queries $\mathcal{Q}_1$, $\mathcal{Q}_2$ and $\mathcal{Q}_3$ are strongly bounded wrt. `LIST01` and $|\ |_1$, $|\ |_2$.

Here we introduce our main result.

Theorem 13
Let $P := R_1 \cup \ldots \cup R_n$ be a program such that $R_n \sqsupset \ldots \sqsupset R_1$ is a hierarchy, $|\ |_1, \ldots, |\ |_n$ be level mappings for $R_1, \ldots, R_n$, respectively, and M be a complete model of P. Suppose that

- R_i is acceptable wrt. $|\ |_i$ and M, for all $i \in \{1, \ldots, n\}$.
- Q is a query strongly bounded wrt. P and $|\ |_1, \ldots, |\ |_n$.

Then all LDNF-derivations of $P \cup \{Q\}$ are finite.

Proof
Let Q be a query strongly bounded wrt. P and $|\ |_1, \ldots, |\ |_n$. We prove the theorem by induction on n.

Base. Let $n = 1$. This case follows immediately by Theorem 9, where $P = R_1$, R is empty and $\mathcal{C}$ is the class of strongly bounded queries wrt. R_1 and $|\ |_1$, and the fact that a strongly bounded atom is also bounded.

Induction step. Let $n > 1$. Also this case follows by Theorem 9, where $P = R_n$, $R = R_1 \cup \ldots \cup R_{n-1}$ and $\mathcal{C}$ is the class of strongly bounded queries wrt. $R_1 \cup \ldots \cup R_n$ and $|\ |_1, \ldots, |\ |_n$. In fact,

- R_n is acceptable wrt. $|\ |_n$ and M;
- for all queries $Q \in \mathcal{C}$, all LDNF-derivations of $(R_1 \cup \ldots \cup R_{n-1}) \cup \{Q\}$ are finite, by Remark 11 and the inductive hypothesis;
- for all atoms $A \in \mathcal{C}$, if A is defined in R_n then A is bounded wrt. $|\ |_n$, by definition of strong boundedness.

□

Here are a few examples applying Theorem 13.

Example 14
Let us reconsider the program of Example 12. In the program `LIST01`, R_1 and R_2 are acceptable wrt. any complete model and the level mappings $|\ |_1$ and $|\ |_2$, respectively. We already showed that $\mathcal{Q}_1, \mathcal{Q}_2$ and $\mathcal{Q}_3$ are strongly bounded wrt. `LIST01` and $|\ |_1, |\ |_2$. Hence, by Theorem 13, all LDNF-derivations of `LIST01` $\cup \{Q\}$, where Q is a query in $\mathcal{Q}_1, \mathcal{Q}_2$ or $\mathcal{Q}_3$, are finite.

Notice that in the previous example the top module in the hierarchy, R_2, contains no recursion. Hence it is intuitively clear that any problem for termination cannot depend on it. This is reflected by the fact that the level mapping for R_2 is completely trivial. This shows how the hierarchical decomposition of the program can simplify the termination proof.

Example 15
Consider the sorting program `MERGESORT` (Apt 1997):

```
c1: mergesort([ ],[ ]).
c2: mergesort([X],[X]).
c3: mergesort([X,Y|Xs],Ys) ←
        split([X,Y|Xs],X1s,X2s),
```

```
          mergesort(X1s,Y1s),
          mergesort(X2s,Y2s),
          merge(Y1s,Y2s,Ys).

c4: split([ ],[ ],[ ]).
c5: split([X|Xs],[X|Ys],Zs) ← split(Xs,Zs,Ys).

c6: merge([ ],Xs,Xs).
c7: merge(Xs,[ ],Xs).
c8: merge([X|Xs],[Y|Ys],[X|Zs]) ← X<=Y, merge(Xs,[Y|Ys],Zs).
c9: merge([X|Xs],[Y|Ys],[Y|Zs]) ← X>Y, merge([X|Xs],Ys,Zs).
```

Let us divide the program MERGESORT into three modules, R_1, R_2, R_3, such that $R_3 \sqsupset R_2 \sqsupset R_1$ as follows:

- $R_3 := \{\texttt{c1}, \texttt{c2}, \texttt{c3}\}$, it defines the relation `mergesort`,
- $R_2 := \{\texttt{c4}, \texttt{c5}\}$, it defines the relation `split`,
- $R_1 := \{\texttt{c6}, \texttt{c7}, \texttt{c8}, \texttt{c9}\}$, it defines the relation `merge`.

Let us consider the natural level mappings

$$|\texttt{merge}(xs, ys, zs)|_1 = |xs|_{\texttt{length}} + |ys|_{\texttt{length}}$$

$$|\texttt{split}(xs, ys, zs)|_2 = |xs|_{\texttt{length}}$$

$$|\texttt{mergesort}(xs, ys)|_3 = |xs|_{\texttt{length}}$$

and assume that for all $i \in \{1, 2, 3\}$, $|L|_{\texttt{i}} = 0$ if L is not defined in R_i.

All ground queries are strongly bounded wrt. the program MERGESORT and the level mappings $|\ |_1, |\ |_2, |\ |_3$. Moreover, since the program is a definite one, R_1 and R_2 are acceptable wrt. any model and the level mappings $|\ |_1$ and $|\ |_2$, respectively, while R_3 is acceptable wrt. the level mapping $|\ |_3$ and the model M below:

$$\begin{aligned} M = &[\texttt{mergesort}(\texttt{Xs}, \texttt{Ys})] \cup [\texttt{merge}(\texttt{Xs}, \texttt{Ys}, \texttt{Zs})] \cup \\ &\{\texttt{split}([\,], [\,], [\,])\} \cup \\ &\{\texttt{split}([x], [\,], [x]) |\ \ x \text{ is any ground term}\} \cup \\ &\{\texttt{split}([x], [x], [\,]) |\ \ x \text{ is any ground term}\} \cup \\ &\{\texttt{split}(xs, ys, zs) |\ \ xs, ys, zs \text{ are ground terms and} \\ &|xs|_{length} \geq 2,\ |xs|_{length} > |ys|_{length}, |xs|_{\texttt{length}} > |zs|_{\texttt{length}}\} \end{aligned}$$

where we denote by $[A]$ the set of all ground instances of an atom A.

Hence, by Theorem 13, all LDNF-derivations of MERGESORT $\cup \{Q\}$, where Q is a ground query, are finite.

Note that by exchanging the roles of R_1 and R_2 we would obtain the same result. In fact the definition of `merge` and `split` are independent from each other.

4 Well-Behaving Programs

In this section we consider the problem of how to prove that a query is strongly bounded. In fact one could argue that checking strong boundedness is more difficult and less abstract than checking boundedness itself in the sense of (Apt and Pedreschi

1993): we have to refer to all LDNF-derivations instead of referring to a model, which might well look like a step backwards in the proof of termination of a program. This is only partly true: in order to check strong boundedness we can either employ tools based on abstract interpretation or concentrate our attention only on programs which exhibit useful persistence properties wrt. LDNF-resolution.

We now show how the well-established notions of well-moded and well-typed programs can be employed in order to verify strong boundedness and how they can lead to simple termination proofs.

4.1 Well-Moded Programs

The concept of a well-moded program is due to (Dembiński and Maluszyński 1985). The formulation we use here is from (Rosenblueth 1991), and it is equivalent to that in (Drabent 1987). The original definition was given for definite programs (i.e., programs without negation), however it applies to general programs as well, just by considering literals instead of atoms. It relies on the concept of *mode*, which is a function that labels the positions of each predicate in order to indicate how the arguments of a predicate should be used.

Definition 16 (*Mode*)
Consider an n-ary predicate symbol p. By a *mode* for p we mean a function m_p from $\{1,\ldots,n\}$ to the set $\{+,-\}$. If $m_p(i) = +$ then we call i an *input position* of p; if $m_p(i) = -$ then we call i an *output position* of p. By a *moding* we mean a collection of modes, one for each predicate symbol.

In a moded program, we assume that each predicate symbol has a unique mode associated to it. Multiple moding may be obtained by simply renaming the predicates. We use the notation $p(m_p(1),\ldots,m_p(n))$ to denote the moding associated with a predicate p (e.g., `append`$(+,+,-)$). Without loss of generality, we assume, when writing a literal as $p(\mathbf{s},\mathbf{t})$, that we are indicating with $\mathbf{s}$ the sequence of terms filling in the input positions of p and with $\mathbf{t}$ the sequence of terms filling in the output positions of p. Moreover, we adopt the convention that $p(\mathbf{s},\mathbf{t})$ could denote both negative and positive literals.

Definition 17 (*Well-Moded*)

- A query $p_1(\mathbf{s}_1,\mathbf{t}_1),\ldots,p_n(\mathbf{s}_n,\mathbf{t}_n)$ is called *well-moded* if for all $i \in \{1,\ldots,n\}$
$$Var(\mathbf{s}_i) \subseteq \bigcup_{j=1}^{i-1} Var(\mathbf{t}_j).$$
- A clause $p(\mathbf{t}_0,\mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1,\mathbf{t}_1),\ldots,p_n(\mathbf{s}_n,\mathbf{t}_n)$ is called *well-moded* if for all $i \in \{1,\ldots,n+1\}$
$$Var(\mathbf{s}_i) \subseteq \bigcup_{j=0}^{i-1} Var(\mathbf{t}_j).$$
- A program is called *well-moded* if all of its clauses are well-moded.

Note that well-modedness can be syntactically checked in a time which is linear wrt. the size of the program (query).

Remark 18
If Q is a well-moded query then all its prefixes are well-moded.

The following lemma states that well-moded queries are closed under LDNF-resolution. This result has been proved in (Apt and Pellegrini 1994) for LD-derivations and definite programs.

Lemma 19
Let P and Q be a well-moded program and query, respectively. Then all LDNF-descendants of $P \cup \{Q\}$ are well-moded.

Proof
It is sufficient to extend the proof in (Apt and Pellegrini 1994) by showing that if a query $\neg A, L_1, \ldots, L_n$ is well-moded and A is ground then both A and $L_1, \ldots, L_n$ are well-moded. This follows immediately by definition of well-modedness. If A is non-ground then the query above has no descendant. □

When considering well-moded programs, it is natural to measure atoms only in their input positions (Etalle et al. 1999).

Definition 20 (*Moded Level Mapping*)
Let P be a moded program. A function $|\ |$ is a *moded level mapping for* P if it is a level mapping for P such that

- for any $\mathbf{s}$, $\mathbf{t}$ and $\mathbf{u}$, $|p(\mathbf{s}, \mathbf{t})| = |p(\mathbf{s}, \mathbf{u})|$.

Hence in a moded level mapping the level of an atom is independent from the terms in its output positions.

The following Remark and Proposition allow us to exploit well-modedness for applying Theorem 13.

Remark 21
Let P be a well-moded program. If Q is well-moded, then *first*(Q) is ground in its input position and hence it is bounded wrt. any moded level mapping for P. Moreover, by Lemma 19, every well-moded query is strongly bounded wrt. P and any moded level mapping for P.

Proposition 22
Let $P := R_1 \cup \ldots \cup R_n$ be a *well-moded* program and $R_n \sqsupset \ldots \sqsupset R_1$ a hierarchy, and $|\ |_1, \ldots, |\ |_n$ be *moded* level mappings for $R_1, \ldots, R_n$, respectively.
Then every well-moded query is strongly bounded wrt. P and $|\ |_1, \ldots, |\ |_n$.

Example 23
Let `MOVE` be the following program which defines a permutation between two lists such that only one element is moved. We introduce modes and we distinguish the two uses of `append` by renaming it as `append1` and `append2`.

```
mode delete(+,−,−).
mode append1(−,−,+).
mode append2(+,+,−).
mode move(+,−).

r1: delete([X|Xs],X,Xs).
r2: delete([X|Xs],Y,[X|Ys]) ← delete(Xs,Y,Ys).

r3: append1([ ],Ys,Ys).
r4: append1([X|Xs],Ys,[X|Zs]) ← append1(Xs,Ys,Zs).

r5: append2([ ],Ys,Ys).
r6: append2([X|Xs],Ys,[X|Zs]) ← append2(Xs,Ys,Zs).

r7: move(Xs,Ys) ← append1(X1s,X2s,Xs),
        delete(X1s,X,Y1s), append2(Y1s,[X|X2s],Ys).
```

Let us partition `MOVE` into the modules $R_1 = \{\mathtt{r}_1, \mathtt{r}_2, \mathtt{r}_3, \mathtt{r}_4, \mathtt{r}_5, \mathtt{r}_6\}$ and $R_2 = \{\mathtt{r}_7\}$ (R_2 extends R_1). Let $|\ |_1$ be the natural level mapping for R_1 defined by:

$|\mathtt{append1}(xs, ys, zs)|_1 = |zs|_{\mathtt{length}}$
$|\mathtt{append2}(xs, ys, zs)|_1 = |xs|_{\mathtt{length}}$.
$|\mathtt{delete}(xs, x, ys)|_1 = |xs|_{\mathtt{length}}$.

R_2 does not contain any recursive definition hence let $|\ |_2$ be the trivial level mapping defined by:

$|\mathtt{move}(xs, ys)|_2 = \mathtt{1}$

and assume that $|L|_2 = 0$, if L is not defined in R_2.

The program $\mathtt{MOVE} := R_1 \cup R_2$ is well-moded and hence by Proposition 22 every well-moded query is strongly bounded wrt. `MOVE` and $|\ |_1$, $|\ |_2$.

Example 24
Let R_1 be the program which defines the relations `member` and `is`, R_2 be the program defining the relation `count` and R_3 be the program defining the relation `diff` with the moding and the definitions below.

```
mode member(+,+).
mode is(−,+).
mode diff(+,+,+,−).
mode count(+,+,−).

r1: member(X,[X|Xs]).
r2: member(X,[Y|Xs]) ← member(X,Xs).

r3: diff(Ls,I1,I2,N) ← count(Ls,I1,N1), count(Ls,I2,N2),
        N is N1-N2.

r4: count([ ],I,0).
r5: count([H|Ts],I,M) ← member(H,I), count(Ts,I,M1),
        M is M1+1.
r6: count([H|Ts],I,M) ← ¬ member(H,I), count(Ts,I,M).
```

The relation $\texttt{diff}(ls, i1, i2, n)$, given a list ls and two check-lists $i1$ and $i2$, defines the difference n between the number of elements of ls occurring in $i1$ and the number of elements of ls occurring in $i2$. Clearly $R_3 \sqsupset R_2 \sqsupset R_1$. It is easy to see that R_1 is acceptable wrt. any complete model and the moded level mapping

$$|\texttt{member}(e, ls)|_1 = |ls|_{\texttt{length}}$$

R_2 is acceptable wrt. any complete model and the moded level mapping:

$$|\texttt{count}(ls, i, n)|_2 = |ls|_{\texttt{length}}$$

and R_3 is acceptable wrt. any complete model and the trivial moded level mapping:

$$|\texttt{diff}(ls, i1, i2, n)|_3 = \texttt{1}$$

where $|L|_{\texttt{i}} = \texttt{0}$, if L is not defined in R_i.

The program $\texttt{DIFF} := R_1 \cup R_2 \cup R_3$ is well-moded. Hence, by Proposition 22, every well-moded query is strongly bounded wrt. DIFF and $|\ |_1$, $|\ |_2$, $|\ |_3$.

Note that the class of strongly bounded queries is generally larger than the class of well-moded queries. Consider for instance the program MOVE and the query $Q := \texttt{move}([\texttt{X1}, \texttt{X2}], \texttt{Ys}), \texttt{delete}(\texttt{Ys}, \texttt{Y}, \texttt{Zs})$ which is not well-moded since it is not ground in the input position of the first atom. However Q can be easily recognized to be strongly bounded wrt. MOVE and $|\ |_1$, $|\ |_2$ defined in Example 23. We will come back to this query later.

4.2 Well-Typed Programs

A more refined well-behavior property of programs, namely well-typedness, can also be useful in order to ensure the strong boundedness property.

The notion of well-typedness relies both on the concepts of *mode* and *type*. The following very general definition of a type is sufficient for our purposes.

Definition 25 (*Type*)
A *type* is a set of terms closed under substitution.

Assume as given a specific set of types, denoted by *Types*, which includes *Any*, the set of all terms, and *Ground* the set of all ground terms.

Definition 26 (*Type Associated with a Position*)
A *type for an n-ary predicate symbol p* is a function t_p from $\{1, \ldots, n\}$ to the set *Types*. If $t_p(i) = T$, we call T *the type associated with the position i of p*. Assuming a type t_p for the predicate p, we say that a literal $p(s_1, \ldots, s_n)$ is *correctly typed in position i* if $s_i \in t_p(i)$.

In a typed program we assume that every predicate p has a fixed mode m_p and a fixed type t_p associated with it and we denote it by

$$p(m_p(1) : t_p(1), \ldots, m_p(n) : t_p(n)).$$

So, for instance, we write

$$\texttt{append}(+ : List, + : List, - : List)$$

to denote the moded atom $\texttt{append}(+, +, -)$ where the type associated with each argument position is *List*, i.e., the set of all lists.

We can then talk about types of input and of output positions of an atom.

The notion of well-typed queries and programs relies on the following concept of type judgement.

Definition 27 (*Type Judgement*)
By a *type judgement* we mean a statement of the form $\mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$. We say that a type judgement $\mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$ *is true*, and write $\models \mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$, if for all substitutions θ, $\mathbf{s}\theta \in \mathbf{S}$ implies $\mathbf{t}\theta \in \mathbf{T}$.

For example, the type judgements $(x : Nat,\ l : ListNat) \Rightarrow ([x|l] : ListNat)$ and $([x|l] : ListNat) \Rightarrow (l : ListNat)$ are both true.

A notion of well-typed program has been first introduced in (Bronsard et al. 1992) and also studied in (Apt and Etalle 1993) and in (Apt and Luitjes 1995). Similarly to well-moding, the notion was developed for definite programs. Here we extend it to general programs.

In the following definition, we assume that $\mathbf{i}_s : \mathbf{I}_s$ is the sequence of typed terms filling in the input positions of L_s and $\mathbf{o}_s : \mathbf{O}_s$ is the sequence of typed terms filling in the output positions of L_s.

Definition 28 (*Well-Typed*)

- A query $L_1, \ldots, L_n$ is called *well-typed* if for all $j \in \{1, \ldots, n\}$

  $$\models \mathbf{o}_{j_1} : \mathbf{O}_{j_1}, \ldots, \mathbf{o}_{j_k} : \mathbf{O}_{j_k} \Rightarrow \mathbf{i}_j : \mathbf{I}_j$$

  where $L_{j_1}, \ldots, L_{j_k}$ are all the positive literals in $L_1, \ldots, L_{j-1}$.
- A clause $L_0 \leftarrow L_1, \ldots, L_n$ is called *well-typed* if for all $j \in \{1, \ldots, n\}$

  $$\models \mathbf{i}_0 : \mathbf{I}_0, \mathbf{o}_{j_1} : \mathbf{O}_{j_1}, \ldots, \mathbf{o}_{j_k} : \mathbf{O}_{j_k} \Rightarrow \mathbf{i}_j : \mathbf{I}_j$$

  where $L_{j_1}, \ldots, L_{j_k}$ are all the positive literals in $L_1, \ldots, L_{j-1}$, and

  $$\models \mathbf{i}_0 : \mathbf{I}_0, \mathbf{o}_{j_1} : \mathbf{O}_{j_1}, \ldots, \mathbf{o}_{j_h} : \mathbf{O}_{j_h} \Rightarrow \mathbf{o}_0 : \mathbf{O}_0$$

  where $L_{j_1}, \ldots, L_{j_h}$ are all the positive literals in $L_1, \ldots, L_n$.
- A program is called *well-typed* if all of its clauses are well-typed.

Note that an atomic query is well-typed iff it is correctly typed in its input positions and a unit clause $p(\mathbf{s} : \mathbf{S}, \mathbf{t} : \mathbf{T}) \leftarrow$ is well-typed if $\models \mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$.

The difference between Definition 28 and the one usually given for definite programs is that the correctness of the terms filling in the output positions of negative literals cannot be used to deduce the correctness of the terms filling in the input positions of a literal to the right (or the output positions of the head in a clause). The two definitions coincide either for definite programs or for general programs whose negative literals have only input positions.

As an example, let us consider the trivial program

```
p(− : List).
q(+ : List).

p([]).
q([]).
```

By adopting a straightforward extension of well-typedness to normal programs which considers also the outputs of negative literals, we would have that the query $\neg \mathtt{p}(\mathtt{a}), \mathtt{q}(\mathtt{a})$ is well-typed even if `a` is not a list. Moreover well-typedness would not be persistent wrt. LDNF-resolution since `q(a)`, which is the first LDNF-resolvent of the previous query, is no more well-typed. Our extended definition and the classical one coincide either for definite programs or for general programs whose negative literals have only input positions.

For definite programs, well-modedness can be viewed as a special case of well-typedness if we consider only one type: *Ground*. With our extended definitions of well-moded and well-typed general programs this is no more true. We could have given a more complicated definition for well-typedness in order to capture also well-modedness as a special case. For the sake of simplicity, we prefer to give two distinct and simpler definitions.

Remark 29
If Q is a well-typed query, then all its non-empty prefixes are well-typed. In particular, $\mathit{first}(Q)$ is well-typed.

The following Lemma shows that well-typed queries are closed under LDNF-resolution. It has been proved in (Bronsard et al. 1992) for definite programs.

Lemma 30
Let P and Q be a well-typed program and query, respectively. Then all LDNF-descendants of $P \cup \{Q\}$ are well-typed.

Proof
Similarly to the case of well-moded programs, to extend the result to general programs it is sufficient to show that if a query $Q := \neg A, L_1, \ldots, L_n$ is well-typed then both A and $L_1, \ldots, L_n$ are well-typed. In fact, by Remark 29, $\neg A = \mathit{first}(Q)$ is well-typed and by Definition 28, if the first literal in a well-typed query is negative, then it is not used to deduce well-typedness of the rest of the query. □

It is now natural to exploit well-typedness in order to check strong boundedness. Analogously to well-moded programs, there are level mappings that are more natural in presence of type information. They are the level mappings for which every well-typed atom is bounded. By Lemma 30 we have that a well-typed query Q is strongly bounded wrt. a well-typed program P and any such level mapping. This is stated by the next proposition.

Proposition 31
Let $P := R_1 \cup \ldots \cup R_n$ be a *well-typed* program and $R_n \sqsupset \ldots \sqsupset R_1$ be a hierarchy, and $|\ |_1, \ldots, |\ |_n$ be level mappings for $R_1, \ldots, R_n$, respectively. Suppose that for every well-typed atom A, if A is defined in R_i then A is bounded wrt. $|\ |_i$, for $i \in \{1, \ldots, n\}$. Then every well-typed query is strongly bounded wrt. P and $|\ |_1, \ldots, |\ |_n$.

Example 32
Let us consider again the modular proof of termination for $\texttt{MOVE} := R_1 \cup R_2$, where R_1 defines the relations `append1`, `append2` and `delete`, while R_2, which extends R_1, defines the relation `move`. We consider the moding of Example 23 with the following types:

$\texttt{delete}(+ : List, - : Any, - : List)$
$\texttt{append1}(- : List, - : List, + : List)$
$\texttt{append2}(+ : List, + : List, - : List)$
$\texttt{move}(+ : List, - : List)$.

Program `MOVE` is *well-typed* in the assumed modes and types.

Let us consider the same level mappings as used in Example 23. We have already seen that R_2 is acceptable wrt. $|\ |_2$ and any model, and R_1 is acceptable wrt. $|\ |_1$ and any model. By definition of $|\ |_2$ and $|\ |_1$, one can easily see that

- every well-typed atom A defined in R_i is bounded wrt. $|\ |_i$.

Hence, by Proposition 31,

- every well-typed query is strongly bounded wrt. `MOVE` and $|\ |_1$, $|\ |_2$.

Let us consider again the query $Q := \texttt{move}([\texttt{X1}, \texttt{X2}], \texttt{Ys}), \texttt{delete}(\texttt{Ys}, \texttt{Y}, \texttt{Zs})$ which is not well-moded but it is well-typed. We have that Q is strongly bounded wrt. `MOVE` and $|\ |_1$, $|\ |_2$, and consequently, by Theorem 13, that all LDNF-derivations of $\texttt{MOVE} \cup \{Q\}$ are finite.

Example 33
Consider the program `COLOR_MAP` from (Sterling and Shapiro 1986) which generates a coloring of a map in such a way that no two neighbors have the same color. The map is represented as a list of regions and colors as a list of available colors. In turn, each region is determined by its name, color and the colors of its neighbors, so it is represented as a term `region(name,color,neighbors)`, where `neighbors` is a list of colors of the neighboring regions.

```
c1: color_map([ ],Colors).
c2: color_map([Region|Regions],Colors) ←
        color_region(Region,Colors),
        color_map(Regions,Colors).

c3: color_region(region(Name,Color,Neighbors),Colors) ←
        select(Color,Colors,Colors1)
        subset(Neighbors,Colors1).

c4: select(X,[X|Xs],Xs).
c5: select(X,[Y|Xs],[Y|Zs]) ← select(X,Xs,Zs).

c6: subset([ ],Ys).
c7: subset([X|Xs],Ys) ← member(X,Ys), subset(Xs,Ys).

c8: member(X,[X|Xs]).
c9: member(X,[Y|Xs]) ← member(X,Xs).
```

Consider the following modes and types for the program COLOR_MAP:

color_map$(+ : \mathit{ListRegion}, + : \mathit{List})$
color_region$(+ : \mathit{Region}, + : \mathit{List})$
select$(+ : \mathit{Any}, + : \mathit{List}, - : \mathit{List})$
subset$(+ : \mathit{List}, + : \mathit{List})$
member$(+ : \mathit{Any}, + : \mathit{List})$

where

- *Region* is the set of all terms of the form region(name,color,neighbors) with name, color $\in$ *Any* and neighbors $\in$ *List*,
- *ListRegion* is the set of all lists of regions.

We can check that COLOR_MAP is well-typed in the assumed modes and types.

We can divide the program COLOR_MAP into four distinct modules, R_1, R_2, R_3, R_4, in the hierarchy $R_4 \sqsupset R_3 \sqsupset R_2 \sqsupset R_1$ as follows:

- $R_4 := \{\mathtt{c1}, \mathtt{c2}\}$ defines the relation color_map,
- $R_3 := \{\mathtt{c3}\}$ defines the relation color_region,
- $R_2 := \{\mathtt{c4}, \mathtt{c5}, \mathtt{c6}, \mathtt{c7}\}$ defines the relations select and subset,
- $R_1 := \{\mathtt{c8}, \mathtt{c9}\}$ defines the relation member.

Each R_i is trivially acceptable wrt. any model M and the simple level mapping $|\ |_i$ defined below:

$$|\mathtt{color\_map}(xs, ys)|_4 = |xs|_{\mathtt{length}}$$

$$|\mathtt{color\_region}(x, xs)|_3 = \mathtt{1}$$

$$|\mathtt{select}(x, xs, ys)|_2 = |xs|_{\mathtt{length}}$$
$$|\mathtt{subset}(xs, ys)|_2 = |xs|_{\mathtt{length}}$$

$$|\mathtt{member}(x, xs)|_1 = |xs|_{\mathtt{length}}$$

where for all $i \in \{1, 2, 3, 4\}$, $|L|_i = 0$, if L is not defined in R_i.

Moreover, for every well-typed atom A and $i \in \{1, 2, 3, 4\}$, if A is defined in R_i then A is bounded wrt. $|\ |_i$. Hence, by Proposition 31,

- every well-typed query is strongly bounded wrt. the program COLOR_MAP and $|\ |_1, \ldots, |\ |_4$.

This proves that all LDNF-derivations of the program COLOR_MAP starting in a well-typed query are finite. In particular, all the LDNF-derivations starting in a query of the form color_map(xs, ys), where xs is a list of regions and ys is a list, are finite. Note that in proving termination of such queries the choice of a model is irrelevant. Moreover, since such queries are well-typed, their input arguments are required to have a specified structure, but they are not required to be ground terms as in the case of well-moded queries. Hence, well-typedness allows us to reason about a larger class of queries with respect to well-modedness.

This example is also discussed in (Apt and Pedreschi 1994). In order to prove its termination they define a particular level mapping $|\ |$, obtained by combining

the level mappings of each module, and a special model M wrt. which the whole program `COLOR_MAP` is acceptable. Both the level mapping $|\ |$ and the model M are non-trivial.

5 Iterative Proof Method

In the previous section we have seen how we can exploit properties which are preserved by LDNF-resolution, such as well-modedness and well-typedness, for developing a modular proof of termination in a hierarchy of programs. In this section we show how these properties allow us to apply our general result, i.e., Theorem 9, also in an iterative way.

Corollary 34
Let P and R be two programs such that $P \cup R$ is well-moded and P extends R, and M be a complete model of $P \cup R$. Suppose that

- P is acceptable wrt. a moded level mapping $|\ |$ and M,
- for all well-moded queries Q, all LDNF-derivations $R \cup \{Q\}$ are finite.

Then for all well-moded queries Q, all LDNF-derivations of $(P \cup R) \cup \{Q\}$ are finite.

Proof
Let $\mathcal{C}$ be the class of well-moded queries of $P \cup R$. By Remark 18 and Lemma 19, $\mathcal{C}$ is $(P \cup R)$-closed. Moreover

- P is acceptable wrt. a moded level mapping $|\ |$ and M, by hypothesis;
- for all well-moded queries Q, all LDNF-derivations of $R \cup \{Q\}$ are finite, by hypothesis;
- for all well-moded atoms A, if A is defined in P then A is bounded wrt. $|\ |$, by Remark 21, since $|\ |$ is a moded level mapping.

Hence by Theorem 9 we get the thesis. □

Note that this result allows one to incrementally prove well-termination for general programs thus extending the result given in (Etalle et al. 1999) for definite programs.

A similar result can be stated also for well-typed programs and queries, provided that there exists a level mapping for P implying boundedness of atomic well-typed queries.

Corollary 35
Let P and R be two programs such that $P \cup R$ is well-typed and P extends R, and M be a complete model of $P \cup R$. Suppose that

- P is acceptable wrt. a level mapping $|\ |$ and M,
- every well-typed atom defined in P is bounded wrt. $|\ |$,
- for all well-typed queries Q, all LDNF-derivations of $R \cup \{Q\}$ are finite.

Then for all well-typed queries Q, all LDNF-derivations of $(P \cup R) \cup \{Q\}$ are finite.

Proof
Let $\mathcal{C}$ be the class of well-typed queries of $P \cup R$. By Remark 29 and Lemma 30, $\mathcal{C}$ is $(P \cup R)$-closed. Moreover

- P is acceptable wrt. a level mapping $|\ |$ and M, by hypothesis;
- for all well-typed queries Q, all LDNF-derivations of $R \cup \{Q\}$ are finite, by hypothesis;
- for all well-typed atoms A, if A is defined in P then A is bounded wrt. $|\ |$, by hypothesis.

Hence by Theorem 9 we have the thesis. □

Example 36
Let us consider again the program `COLOR_MAP` with the same modes and types as in Example 33. We apply the iterative termination proof given by Corollary 35 to `COLOR_MAP`.

First step. We can consider at first two trivial modules, $R_1 := \{\texttt{c8}, \texttt{c9}\}$ which defines the relation `member`, and $R_0 := \emptyset$. We already know that

- R_1 is acceptable wrt. any model M and the level mapping $|\ |_1$ already defined;
- all well-typed atoms A, defined in R_1, are bounded wrt. $|\ |_1$;
- for all well-typed queries Q, all LDNF-derivations of $R_0 \cup \{Q\}$ are trivially finite.

Hence, by Corollary 35, for all well-typed queries Q, all LDNF-derivations of $(R_1 \cup R_0) \cup \{Q\}$ are finite.

Second step. We can now iterate the process one level up. Let us consider the two modules, $R_2 := \{\texttt{c4}, \texttt{c5}, \texttt{c6}, \texttt{c7}\}$ which defines the relations `select` and `subset`, and $R_1 := \{\texttt{c8}, \texttt{c9}\}$ which defines the relation `member` and it is equal to $(R_1 \cup R_0)$ of the previous step. We already showed in Example 33 that

- R_2 is acceptable wrt. any model M and the level mapping $|\ |_2$ already defined;
- all well-typed atoms A, defined in R_2, are bounded wrt. $|\ |_2$;
- for all well-typed queries Q, all LDNF-derivations of $R_1 \cup \{Q\}$ are finite.

Hence, by Corollary 35, for all well-typed queries Q, all LDNF-derivations of $(R_2 \cup R_1) \cup \{Q\}$ are finite.

By iterating the same reasoning for two steps more, we can prove that all LDNF-derivations of the program `COLOR_MAP` starting in a well-typed query are finite.

Our iterative method applies to a hierarchy of programs where on the lowest module, R, we require termination wrt. a particular class of queries. This can be a weaker requirement on R than acceptability as shown in the following contrived example.

Example 37
Let R define the predicate `lcount` which counts the number of natural numbers in a list.

```
lcount(+ : List, − : Nat)
nat(+ : Any).

r1: lcount([ ],0).
r2: lcount([X|Xs],s(N)) ← nat(X), lcount(Xs,N).
r3: lcount([X|Xs],N) ← ¬ nat(X), lcount(Xs,N).
r4: lcount(0,N) ← lcount(0,s(N)).

r5: nat(0).
r6: nat(s(N)) ← nat(N).
```

R is well-typed wrt. the specified modes and types. Note that R cannot be acceptable due to the presence of clause `r4`. On the other hand, the program terminates for all well-typed queries.

Consider now the following program P which extends R. The predicate `split`, given a list of lists, separates the list elements containing more than `max` natural numbers from the other lists:

```
split(+ : ListList, − : ListList, − : ListList)
>(+ : Nat, + : Nat)
<=(+ : Nat, + : Nat)

p1: split([ ],[ ],[ ]).
p2: split([L|Ls],[L|L1],L2) ← lcount(L,N), N > max,
        split(Ls,L1,L2).
p3: split([L|Ls],L1,[L|L2]) ← lcount(L,N), N <= max,
        split(Ls,L1,L2).
```

where *ListList* denotes the set of all lists of lists, and `max` is a natural number. The program $P \cup R$ is well-typed. Let us consider the simple level mapping $|\ |$ for P defined by:

$$|\mathtt{split}(ls, l1, l2)| = |ls|_{\mathtt{length}}$$

which assigns level 0 to any literal not defined in P. Note that

- P is acceptable wrt. the level mapping $|\ |$ and any complete model M,
- all well-typed atoms defined in P are bounded wrt. $|\ |$,
- for all well-typed queries Q, all LDNF-derivations of $R \cup \{Q\}$ are finite.

Hence, by Corollary 35, for all well-typed queries Q, all LDNF-derivations of $(P \cup R) \cup \{Q\}$ are finite.

This example shows that well-typedness could be useful to exclude what might be called "dead code".

6 Comparing with Apt and Pedreschi's Approach

Our work can be seen as an extension of a proposal in (Apt and Pedreschi 1994). Hence we devote this section to a comparison with their approach.

On one hand, since our approach applies to general programs, it clearly covers

cases which cannot be treated with the method proposed in (Apt and Pedreschi 1994), which was developed for definite programs. On the other hand, for definite programs the classes of queries and programs which can be treated by Apt and Pedreschi's approach are properly included in those which can be treated by our method as we show in this section.

We first recall the notions of *semi-acceptability* and *bounded query* used in (Apt and Pedreschi 1994).

Definition 38 (*Semi-acceptable Program*)
Let P be a definite program, $|\ |$ be a level mapping for P and M be a model of P. P is called *semi-acceptable wrt.* $|\ |$ *and* M if for every clause $A \leftarrow \mathbf{A}, B, \mathbf{B}$ in $ground(P)$ such that $M \models \mathbf{A}$

- $|A| > |B|$, if $rel(A) \simeq rel(B)$,
- $|A| \geq |B|$, if $rel(A) \sqsupset rel(B)$.

Definition 39 (*Bounded Query*)
Let P be a definite program, $|\ |$ be a level mapping for P, and M be a model of P.

- With each query $Q := L_1, \ldots, L_n$ we associate n sets of natural numbers defined as follows: For $i \in \{1, \ldots, n\}$,

  $|Q|_i^M = \{|L_i'| \mid L_1', \ldots, L_n'$ is a ground instance of Q and $M \models L_1', \ldots, L_{i-1}'\}$.

- A query Q is called *bounded wrt.* $|\ |$ *and* M if $|Q|_i^M$ is finite (i.e., if $|Q|_i^M$ has a maximum in $\mathbf{N}$) for all $i \in \{1, \ldots, n\}$.

Lemma 40
Let P be a definite program which is semi-acceptable wrt. $|\ |$ and M. If Q is a query bounded wrt. $|\ |$ and M then all LD-descendants of $P \cup \{Q\}$ are bounded wrt. $|\ |$ and M.

Proof
It is a consequence of Lemma 3.6 in (Apt and Pedreschi 1994) and (the proof of) Lemma 5.4 in (Apt and Pedreschi 1994). □

We can always decompose a definite program P into a hierarchy of $n \geq 1$ programs $P := R_1 \cup \ldots \cup R_n$, where $R_n \sqsupset \ldots \sqsupset R_1$ in such a way that for every $i \in \{1, \ldots, n\}$ if the predicate symbols p_i and q_i are both defined in R_i then neither $p_i \sqsupset q_i$ nor $q_i \sqsupset p_i$ (either they are mutually recursive or independent). We call such a hierarchy a *finest decomposition* of P.

The following property has two main applications. First it allows us to compare our approach with (Apt and Pedreschi 1994), then it provides an extension of Theorem 13 to hierarchies of semi-acceptable programs.

Proposition 41
Let P be a semi-acceptable program wrt. a level mapping $|\ |$ and a model M and Q be a query strongly bounded wrt. P and $|\ |$. Let $P := R_1 \cup \ldots \cup R_n$ be a finest decomposition of P into a hierarchy of modules. Let $|\ |_i$, with $i \in \{1, \ldots, n\}$, be defined in the following way: if A is defined in R_i then $|A|_i = |A|$ else $|A|_i = 0$. Then

- every R_i is acceptable wrt. $|\ |_i$ and M (with $i \in \{1, \ldots, n\}$),
- Q is strongly bounded wrt. $R_1 \cup \ldots \cup R_n$ and $|\ |_1, \ldots, |\ |_n$.

Proof
Immediate by the definitions of semi-acceptability and strongly boundedness, since we are considering a finest decomposition. □

In order to compare our approach to the one presented in (Apt and Pedreschi 1994) we consider only Theorem 5.8 in (Apt and Pedreschi 1994), since this is their most general result which implies the other ones, namely Theorem 5.6 and Theorem 5.7.

Theorem 42 (*Theorem 5.8 in (Apt and Pedreschi 1994)*)
Let P and R be two definite programs such that P extends R, and let M be a model of $P \cup R$. Suppose that

- R is semi-acceptable wrt. $|\ |_R$ and $M \cap B_R$,
- P is semi-acceptable wrt. $|\ |_P$ and M,
- there exists a level mapping $\|\ \|_P$ such that for every ground instance of a clause from P, $A \leftarrow \mathbf{A}, B, \mathbf{B}$, such that $M \models \mathbf{A}$
  - $\|A\|_P \geq \|B\|_P$, if $rel(B)$ is defined in P,
  - $\|A\|_P \geq |B|_R$, if $rel(B)$ is defined in R.

Then $P \cup R$ is semi-acceptable wrt. $|\ |$ and M, where $|\ |$ is defined as follows:

$|A| = |A|_P + \|A\|_P$, if $rel(A)$ is defined in P,
$|A| = |A|_R$, if $rel(A)$ is defined in R.

The following remark follows from Lemma 5.4 in (Apt and Pedreschi 1994) and Corollary 3.7 in (Apt and Pedreschi 1994). Together with Theorem 42, it implies termination of bounded queries in (Apt and Pedreschi 1994).

Remark 43
If $P \cup R$ is semi-acceptable wrt. $|\ |$ and M and Q is bounded wrt. $|\ |$ and M then all LD-derivations of $(P \cup R) \cup \{Q\}$ are finite.

We now show that whenever Theorem 42 can be applied to prove termination of all the queries bounded wrt. $|\ |$ and M, then also our method can be used to prove termination of the same class of queries with no need of $\|\ \|_P$ for relating the proofs of the two modules.

In the following theorem for the sake of simplicity we assume that $P \sqsupset R$ is a finest decomposition of $P \cup R$. We discuss later how to extend the result to the general case.

Theorem 44
Let P and R be two programs such that P extends R, and let M be a model of $P \cup R$. Suppose that

- R is semi-acceptable wrt. $|\ |_R$ and $M \cap B_R$,
- P is semi-acceptable wrt. $|\ |_P$ and M,

- there exists a level mapping $\| \ \|_P$ defined as in Theorem 42.

Let $|\ |$ be the level mapping defined by Theorem 42. Moreover, suppose $P \sqsupset R$ is a finest decomposition of $P \cup R$. If Q is bounded wrt. $|\ |$, then Q is strongly bounded wrt. $P \cup R$ and $|\ |_P$ and $|\ |_R$.

Proof
Since we are considering a finest decomposition of $P \cup R$, by Proposition 41, R is acceptable wrt. $|\ |_R$, while P is acceptable wrt. $|\ |'_P$ such that if A is defined in P then $|A|'_P = |A|_P$ else $|A|'_P = 0$.

By Lemma 40 all LD-descendants of $(P \cup R) \cup \{Q\}$ are bounded wrt. $|\ |$ and M. By definition of boundedness, for all LD-descendants Q' of $(P \cup R) \cup \{Q\}$, $\mathit{first}(Q')$ is bounded wrt. $|\ |$. By definition of $|\ |$, for all atoms A bounded wrt. $|\ |$ we have that: if A is defined in R then A is bounded wrt. $|\ |_R$, while if A is defined in P then A is bounded wrt. $|\ |_P$ and hence wrt. $|\ |'_P$ (since $|A|'_P = |A|_P$). Hence the thesis follows. □

If the hierarchy $P \sqsupset R$ is not a finest one and $|\ |_P$ and $|\ |_R$ are the level mappings corresponding to P and R respectively, then we can decompose P into a finest decomposition, $P := P_n \sqsupset \ldots \sqsupset P_1$, and consider instead of $|\ |_P$ the derived level mappings $|\ |_{P_i}$ defined in the following way: if A is defined in P_i then $|A|_{P_i} = |A|_P$ else $|A|_{P_i} = 0$. Similarly we can decompose $R := R_n \sqsupset \ldots \sqsupset R_1$ and define the corresponding level mappings. The derived level mappings satisfy all the properties we need for proving that if Q is bounded wrt. $|\ |$, then Q is strongly bounded wrt. $P \cup R$ and $|\ |_{P_1}, \ldots, |\ |_{P_n}, |\ |_{R_1}, \ldots, |\ |_{R_n}$.

To complete the comparison with (Apt and Pedreschi 1994), we can observe that our method is applicable also for proving termination of queries in modular programs which are not (semi-)acceptable. Such programs clearly cannot be dealt with Apt and Pedreschi's method. The program of Example 37 is a non-acceptable program for which we proved termination of all well-typed queries by applying Corollary 35. The following is a simple example of a non-acceptable program to which we can apply the general Theorem 13.

Example 45
Let R be the following trivial program:

```
r1: q(0).
r2: q(s(Y)) ← q(Y).
```

The program R is acceptable wrt. the following natural level mapping $|\ |_R$ and any model M:

$|\mathtt{q}(t)|_R = |t|_{size}$.

Let P be a program, which extends R, defined as follows:

```
p1: r(0,0).
p2: r(s(X),Y).
p3: p(X) ← r(X,Y), q(Y).
```

The program P is acceptable wrt. the following trivial level mapping $|\ |_P$ and any model M:

$|\mathtt{q}(y)|_P = 0$,
$|\mathtt{r}(x,y)|_P = 0$,
$|\mathtt{p}(x)|_P = 1$.

Note that, even if each module is acceptable, $P \cup R$ cannot be acceptable wrt. any level mapping and model. In fact $P \cup R$ is not left-terminating: for example it does not terminate for the ground query `p(s(0))`. As a consequence Apt and Pedreschi's method does not apply to $P \cup R$. On the other hand, there are ground queries, such as `p(0)`, which terminate in $P \cup R$. We can prove it as follows.

- By Theorem 13, for all strongly bounded queries Q wrt. $P \cup R$ and $|\ |_R$, $|\ |_P$, all LD-derivations of $(P \cup R) \cup \{Q\}$ are finite.
- `p(0)` is strongly bounded wrt. $P \cup R$ and $|\ |_R$, $|\ |_P$. In fact, $Call_{(P \cup R)}(\mathtt{p(0)}) = \{\mathtt{p(0)}, \mathtt{r(0,Y)}, \mathtt{q(0)}\}$ and all these atoms are bounded wrt. their corresponding level mapping.

7 Conclusions

In this paper we propose a modular approach to termination proofs of general programs by following the proof style introduced by Apt and Pedreschi. Our technique allows one to give simple proofs in hierarchically structured programs, namely programs which can be partitioned into n modules, $R_1 \cup \ldots \cup R_n$, such that for all $i \in \{1, \ldots, n-1\}$, R_{i+1} extends $R_1 \cup \ldots \cup R_i$.

We supply the general Theorem 9 which can be iteratively applied to a hierarchy of two programs and a class of queries enjoying persistence properties through LDNF-resolution. We then use such a result to deal with a general hierarchy of acceptable programs, by introducing an extension of the concept of boundedness for hierarchical programs, namely strong boundedness. Strong boundedness is a property on queries which can be easily ensured for hierarchies of programs behaving well, such as well-moded or well-typed programs. We show how specific and simple hierarchical termination proofs can be derived for such classes of programs and queries. We believe this is a valuable result since realistic programs are typically well-moded and well-typed.

The simplifications in the termination proof derive from the fact that for proving the termination of a modular program, we simply prove acceptability of each module by choosing a level mapping which focuses only on the predicates defined in it, with no concern of the module context. Generally this can be done by using very simple and natural level mappings which are completely independent from one module to another. A complicated level mapping is generally required when we prove the termination of a program as a whole and we have to consider a level mapping which appropriately relates all the predicates defined in the program. Hence the finer the modularization of the program the simpler the level mappings. Obviously we cannot completely ignore how predicates defined in different modules relate to each other.

On one hand, when we prove acceptability for each module, we consider a model for the whole program. This guarantees the compatibility among the definitions in the hierarchy. On the other hand, for queries we use the notion of strong boundedness. The intuition is that we consider only what may influence the evaluation of queries in the considered class.

The proof method which derives from our general result, Theorem 9, can be applied also to programs which are not acceptable. In fact, the condition on the lower module is just that it terminates on all the queries in the considered class and not on all ground queries as required for acceptable programs. From our general result we could also derive a method to deal with pre-compiled modules (or even modules written in a different language) provided that we already know termination properties and we have a complete specification.

For sake of simplicity, in the first part of the paper we consider the notion of acceptability instead of the less requiring notion of semi-acceptability. This choice makes proofs of our results much simpler. On the other hand, as we show in Section 6, our results can be applied also to hierarchies of semi-acceptable programs.

We have compared our proposal with the one in (Apt and Pedreschi 1994). They propose a modular approach to left-termination proofs in a hierarchy of two definite programs $P \sqsupset R$. They require both the (semi)-acceptability of the two modules R and P wrt. their respective level mappings and a condition relating the two level mappings which is meant to connect the two termination proofs.

Our method is more powerful both because we consider also general programs and because we capture definite programs and queries which cannot be treated by the method developed in (Apt and Pedreschi 1994). In fact there are non-acceptable programs for which we can single out a class of terminating queries.

For the previous reasons our method improves also with respect to (Pedreschi and Ruggieri 1996, Pedreschi and Ruggieri 1999) where hierarchies of modules are considered. In (Pedreschi and Ruggieri 1996, Pedreschi and Ruggieri 1999) a unifying framework for the verification of total correctness of logic programs is provided. The authors consider modular termination by following the approach in (Apt and Pedreschi 1994).

In (Marchiori 1996) a methodology for proving termination of general logic programs is proposed which is based on modularization. In this approach, the *acyclic* modules, namely modules that terminate independently from the selection rule, play a distinctive role. For such modules, the termination proof does not require a model. In combination with appropriate notions of *up-acceptability* and *low-acceptability* for the modules which are not acyclic, this provides a practical technique for proving termination of the whole program. Analogously to (Apt and Pedreschi 1994), also in (Marchiori 1996) a relation between the level mappings of all modules is required. It is interesting to note that the idea of exploiting acyclicity is completely orthogonal to our approach: we could integrate it into our framework.

Another related work is (Decorte et al. 1999), even if it does not aim explicitly at modularity. In fact they propose a technique for automatic termination analysis of definite programs which is highly efficient also because they use a rather operational notion of acceptability with respect to a set of queries, where decreasing levels are

required only on (mutually) recursive calls as in (De Schreye et al. 1992). Effectively, this corresponds to considering a finest decomposition of the program and having independent level mappings for each module. However, their notion of acceptability is defined and verified on call-patterns instead of program clauses. In a sense, such an acceptability with respect to a set of queries combines the concepts of strongly boundedness and (standard) acceptability. They start from a class of queries and try to derive automatically a termination proof for such a class, while we start from the program and derive a class of queries for which it terminates.

In (Verbaeten et al. 1999) termination in the context of tabled execution is considered. Also in this case modular results are inspired by (De Schreye et al. 1992) by adapting the notion of acceptability wrt. call-patterns to tabled executions. This work is further developed in (Verbaeten et al. 2001) where their modular termination conditions are refined following the approach by (Apt and Pedreschi 1994).

In (Etalle et al. 1999) a method for modular termination proofs for well-moded definite programs is proposed. Our present work generalizes such result to general programs.

Our method may help in designing more powerful automatic systems for verifying termination (De Schreye et al. 1992, Speirs, Somogyi and Søndergaard. 1997, Decorte et al. 1999, Codish and Taboch 1999). We see two directions which could be pursued for a fruitful integration with existing automatic tools. The first one exploits the fact that in each single module it is sufficient to synthesize a level mapping which does not need to measure atoms defined in other modules. The second one concerns tools based on call-patterns analysis (De Schreye et al. 1992, Gabbrielli and Giacobazzi 1994, Codish and Demoen 1995). They can take advantage of the concept of strong boundedness which, as we show, can be implied by well-behavior of programs (Debray and Warren 1988, Debray 1989).

Acknowledgements. This work has been partially supported by MURST with the National Research Project "Certificazione automatica di programmi mediante interpretazione astratta".

References

Apt, K. R. (1990). Introduction to Logic Programming, *in* J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. B: Formal Models and Semantics, Elsevier, Amsterdam and The MIT Press, Cambridge, pp. 495–574.

Apt, K. R. (1997). *From Logic Programming to Prolog*, Prentice Hall.

Apt, K. R. and Etalle, S. (1993). On the unification free Prolog programs, *in* A. Borzyszkowski and S. Sokolowski (eds), *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS 93)*, Vol. 711 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–19.

Apt, K. R. and Luitjes, I. (1995). Verification of logic programs with delay declarations, *in* A. Borzyszkowski and S. Sokolowski (eds), *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, (AMAST'95)*, Vol. 936 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–19.

Apt, K. R. and Pedreschi, D. (1993). Reasoning about termination of pure Prolog programs, *Information and Computation* **106**(1): 109–157.

Apt, K. R. and Pedreschi, D. (1994). Modular termination proofs for logic and pure Prolog programs, *in* G. Levi (ed.), *Advances in Logic Programming Theory*, Oxford University Press, pp. 183–229.

Apt, K. R. and Pellegrini, A. (1994). On the occur-check free Prolog programs, *ACM Transactions on Programming Languages and Systems* **16**(3): 687–726.

Bezem, M. (1993). Strong termination of logic programs, *Journal of Logic Programming* **15**(1&2): 79–97.

Bronsard, F., Lakshman, T. K. and Reddy, U. S. (1992). A framework of directionality for proving termination of logic programs, *in* K. R. Apt (ed.), *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, pp. 321–335.

Cavedon, L. (1989). Continuity, consistency and completeness properties for logic programs, *in* G. Levi and M. Martelli (eds), *Proceedings of the Sixth International Conference on Logic Programming*, The MIT press, pp. 571–584.

Clark, K. L. (1978). Negation as failure rule, *in* H. Gallaire and G. Minker (eds), *Logic and Data Bases*, Plenum Press, pp. 293–322.

Codish, M. and Demoen, B. (1995). Analyzing logic programs using "prop"-ositional logic programs and a magic wand, *Journal of Logic Programming* **25**(3): 249–274.

Codish, M. and Taboch, C. (1999). A semantic basis for the termination analysis of logic programs, *Journal of Logic Programming* **41**(1): 103–123.

De Schreye, D., Verschaetse, K. and Bruynooghe, M. (1992). A Framework for Analyzing the Termination of Definite Logic programs with respect to Call Patterns, *in* I. Staff (ed.), *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'92), Tokio*, ICOT, pp. 481–488.

Debray, S. K. (1989). Static inference of modes and data dependencies in logic programs, *ACM Transactions on Programming Languages and Systems* **11**(3): 418–450.

Debray, S. K. and Warren, D. S. (1988). Automatic mode inference for logic programs, *Journal of Logic Programming* **5**(3): 207–229.

Decorte, S., De Schreye, D. and Vandecasteele, H. (1999). Constraint-based termination analysis of logic programs, *ACM Transactions on Programming Languages and Systems* **21**(6): 1137–1195.

Dembiński, P. and Maluszyński, J. (1985). AND-parallelism with intelligent backtracking for annotated logic programs, *Proceedings of the International Symposium on Logic Programming*, Boston, pp. 29–38.

Drabent, W. (1987). Do logic programs resemble programs in conventional languages?, *in* E. Wada (ed.), *Proceedings International Symposium on Logic Programming*, IEEE Computer Society, pp. 389–396.

Etalle, S., Bossi, A. and Cocco, N. (1999). Termination of well-moded programs, *Journal of Logic Programming* **38**(2): 243–257.

Gabbrielli, M. and Giacobazzi, R. (1994). Goal independency and call patterns in the analysis of logic programs, *Proceedings of the Ninth ACM Symposium on Applied Computing*, ACM Press, pp. 394–399.

Janssen, G. and Bruynooghe, M. (1992). Deriving descriptions of possible values of program variables by means of abstract interpretation, *Journal of Logic Programming* **13**(2–3): 205–258.

Lloyd, J. W. (1987). *Foundations of Logic Programming*, Symbolic Computation – Artificial Intelligence, Springer-Verlag. Second edition.

Marchiori, E. (1996). Practical methods for proving termination of general logic programs, *Journal of Artificial Intelligence Research* **4**: 179–208.

Pedreschi, D. and Ruggieri, S. (1996). Modular verification of Logic Programs, *in* F. de Boer and M. Gabbrielli (eds), *Proceedings of the W2 Post-Conference Workshop of the 1996 JLCSLP,Bonn, Germany*. http://www.di.unipi.it/∼gabbri/w2.html.

Pedreschi, D. and Ruggieri, S. (1999). Verification of Logic Programs, *Journal of Logic Programming* **39**(1–3): 125–176.

Rosenblueth, D. (1991). Using program transformation to obtain methods for eliminating backtracking in fixed-mode logic programs, *Technical Report 7*, Universidad Nacional Autonoma de Mexico, Instituto de Investigaciones en Matematicas Aplicadas y en Sistemas.

Speirs, C., Somogyi, Z. and Søndergaard., H. (1997). Termination analysis for Mercury, *in* P. V. Hentenryck (ed.), *Proceedings of the Fourth International Static Analysis Symposium, (SAS'97)*, Vol. 1302 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 157–171.

Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*, The MIT Press.

Verbaeten, S., Sagonas, K. F. and De Schreye, D. (1999). Modular termination proofs for Prolog with tabling, *in* G. Nadathur (ed.), *Proceedings of International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, Vol. 1702 of *Lecture Notes in Computer Science*, pp. 342–359.

Verbaeten, S., Sagonas, K. F. and De Schreye, D. (2001). Termination proofs for logic programs with tabling, *ACM Transactions on Computational Logic* **2**(1): 57–92.