Termination of Well-Typed Logic Programs

Annalisa Bossi, Nicoletta Cocco, Sabina Rossi

Dipartimento di Informatica
Università di Venezia - Cà Foscari
via Torino, 155, 30172 Mestre-Venezia, Italy
{bossi,cocco,srossi}@dsi.unive.it

Abstract

We consider an extended definition of *well-typed programs* to general logic programs, i.e., logic programs with negated literals in the body of the clauses. This is a quite large class of programs which properly includes all the well-moded ones. We study termination properties of well-typed general logic programs while employing the Prolog's left-to-right selection rule. We introduce the notion of *typed acceptable program* and provide an algebraic characterization for the class of well-typed programs which terminate on all well-typed queries.

1 Introduction

In studying termination of logic programs two main directions can be recognized as clearly described in [18]. The first one is intended to algebraically characterize classes of programs and queries terminating wrt. a specific interpreter, such as termination wrt. SLD-resolution [3, 11], LD-resolution [10, 22], LDNF-resolution [9, 12], SLD-resolution with dynamic scheduling [14, 25]. The second one is intended to automatize the verification by defining sufficient conditions for termination wrt. the standard Prolog interpreter [27, 20, 13, 21, 19].

In this paper we follow the first approach: we define and characterize the class of well-typed *typed terminating* programs, namely well-typed general programs terminating wrt. LDNF-resolution for any well-typed general query. These programs and queries may contain negated literals; they are moded and typed and they satisfy some correctness conditions relating the types of input arguments to the types of output arguments.

Our work is in the style of Apt and Pedreschi in [9] for characterizing left termination of general programs. We introduce the notion of *typed acceptability* and prove that it is both a necessary and a sufficient condition for typed termination. Our proposal exploits the well-behavior properties of well-typed programs and queries similarly to what has be done in [22] for well-moded definite programs. Actually, our present proposal can also be interpreted as an extension of [22] to general programs. In fact, when we consider definite programs and the set of ground terms as the only possible type, the class of well-typed programs and queries coincides with the class of well-moded ones. Hence, in this paper, we give also a full characterization of well-terminating programs.

Well-typed definite programs and queries has been introduced by Bronsard *et al.* in [15] and studied also by Apt *et al.* in [6, 7]. The extension of this notion to general logic programs has been introduced in [12] where we study modular and incremental techniques for proving termination properties of general programs wrt. LDNF-resolution. In that paper we already observe how well-behavior properties of programs can greatly simplify such verification proofs. These ideas have been further developed in the present work and are used in the proofs.

Well-typed programs form an interesting class of programs, since they include the majority of the programs used in practice. In fact modes and types can be viewed as an abstract specification of the intended meaning of the defined predicates, while well-typedness guarantees that the correctness wrt. such a specification is preserved through computations [8]. Both notions of well-moded and well-typed programs are largely exploited in the development of logic programs and are incorporated in the most recent proposals of logic languages such as Mercury [26].

The class of typed terminating programs is included neither in the class of left terminating programs, i.e., programs terminating for any ground query, nor in the class of well-terminating programs, i.e., programs terminating for any well-moded query. As an example let us consider the following program `ROTATE`. Given a list l containing at least one ground element different from 0, it computes a permutation of l with a non-zero element as the first element.

```
rotate([0|Xs],Ys) ← append(Xs,[0],Zs), rotate(Zs,Ys).
rotate([X|Xs],[X|Xs]) ← ¬zero(X).

zero(0).

append([ ],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).
```

The intended use of `rotate` is to give the first argument in input and to obtain the second one in output. It is easy to see that the program `ROTATE` terminates for all queries of the form `rotate`(s,t), where s is a list containing at least one ground element different from 0. Moreover, `ROTATE` is neither left terminating nor well-terminating since it does not terminate for all ground queries whose first argument is a list of zero's. The intended and correct use of this program can be captured by mode and type specifications formalizing the fact that the program is intended to be called with an appropriate list in input. Intuitively, the program `ROTATE` is well-typed wrt. such specifications since, whenever we call it with a query respecting the intended use, all the subcalls will also respect such an intended use.

The paper is organized as follows. In Section 2 a few preliminary definitions are given, in particular we briefly recall the notion of LDNF-resolution, and the concepts of complete model, level mapping and bounded atom. In Section 3 the definition of well-typedness, extended to general programs and queries, is recalled and its properties are proved. Typed termination is also defined in this section. In Section 4 the concepts of typed level mappings and typed acceptability are introduced. We prove that a well-typed program, typed acceptable wrt. a typed level mapping and some complete model, is typed terminating. In Section 5 we prove that typed acceptability is also a necessary condition for typed termination. Section 6 briefly compares our proposal with other approaches. In the Appendix the proofs of some technical results used in the paper are given.

2 Preliminaries

We use standard notation and terminology of logic programming (see [1, 2, 23]). Just note that general logic programs are called normal logic programs in [23].

A *general clause* is a construct of the form $H \leftarrow L_1, \dots, L_n$ with $(n \geq 0)$, where H is an atom and $L_1, \dots, L_n$ are literals (i.e., either atoms or the negation of atoms). In turn, a *general query* is a possibly empty finite sequence of literals $L_1, \dots, L_n$, with $(n \geq 0)$. A *general program* is a finite set of general clauses[1]. As in the paper we deal with general queries, clauses and programs, we omit from now on the qualification "general", unless some confusion might arise.

For a literal L, we denote by $rel(L)$ the predicate symbol of L.

Following the convention adopted by Apt in [2], we use bold characters to denote sequences of objects (so that $\mathbf{L}$ indicates a sequence of literals $L_1, \dots, L_n$, while $\mathbf{t}$ indicates a sequence of terms $t_1, \dots, t_n$).

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 [17].

We consider the *LDNF-resolution*, and following Apt and Pedreschi's approach in studying the termination of general programs [9], we view the 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.

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.

For 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 $P \cup \{Q\}$.

[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 "$\leftarrow$" is omitted in the unit clauses.

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 [1, 2] for details). In particular we use the following notion of *complete model* for a program.

Definition 2.1 (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 $comp(P^-)$.*

The notion of bounded atom that we will use in the sequel is based on the following definition of level mapping, originally due to Bezem [11] and Cavedon [16].

Definition 2.2 (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.*

Definition 2.3 (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 this definition is equivalent to the definition of bounded query introduced in [9] when atomic queries are considered. In fact, in case of atomic queries the notion of boundedness does not depend on a model.

In this paper we also use the following notion of extension of a program which formalizes the situation where a program uses another one as a subprogram.

Definition 2.4 (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 by $P \sqsupset R$, 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 $R \sqsupset P$.

We consider also hierarchies of programs, namely chains of extensions.

Definition 2.5 (Hierarchy of Programs) *Let $P_1, \dots, P_n$ be programs such that for all $i \in \{1, \dots, n-1\}$, $P_{i+1} \sqsupset (P_1 \cup \dots \cup P_i)$. Then we call $P_n \sqsupset \dots \sqsupset P_1$ a* hierarchy of programs.

3 Well-Typed Programs

In this section, we recall the definition of well-typed general program given in [12] and show some properties of the programs in this class.

The notion of well-typedness relies both on the concepts of *mode* and *type*.

Definition 3.1 (Mode) *Consider an n-ary predicate symbol p. By a* mode *for p we mean a function m_p from $\{1,\dots,n\}$ 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.*

The following very general definition of a type is sufficient for our purposes.

Definition 3.2 (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 3.3 (Type Associated with a Position of an Atom) *A* type *for an n-ary predicate symbol p is a function t_p from $\{1,\dots,n\}$ 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 $p(s_1,\dots,s_n)$ 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

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

So, for instance, we write $\texttt{append}(+ : List, + : List, - : List)$ to denote the common use of `append` where the first two argument positions are input positions, the last one is an output position, and the type associated with each argument position is *List*, i.e., the set of all lists.

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

Definition 3.4 (Type Judgment) *By a* type judgment *we mean a statement of the form $\mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$. We say that a type judgment $\mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$* 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 judgments $(x : Nat,\ l : ListNat) \Rightarrow ([x|l] : ListNat)$ and $([x|l] : ListNat) \Rightarrow (l : ListNat)$ are both true.

A notion of well-typed program has been first introduced by Bronsard *et al.* in [15] and studied also by Apt and Etalle in [6] and by Apt and Luitjes in [7]. This notion was developed for definite programs. In [12] we extend it to general programs as defined below.

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 3.5 (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.*

The difference between this definition 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 rightmost literal (or the output positions of the head in a clause). The two definitions coincide either for definite programs or general programs whose negative literals have all argument positions being input positions.

Example 3.6 *Consider again the program* `ROTATE` *of the introduction: it is well-typed wrt. the modes and types specified below*

`rotate`$(+ : List*, - : List*)$
`zero`$(+ : Any)$
`append`$(+ : List*, + : List*, - : List*)$

where List denotes the set of all (possibly non-ground) lists containing at least one ground element different from* 0.

Note that well-typedness does not imply correct typedness in all argument positions: an atomic query is well-typed if 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 $\models \mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$.

Definition 3.7 (Correct Typedness) *Let* P *be a typed program. We say that an* atom is correctly typed *if it is correctly typed in all its argument positions. A* query is correctly typed *if all its positive literals are correctly typed and all its negative literals are correctly typed in all their input positions. A* clause is correctly typed *if both the body and the head are correctly typed.*

Note that correct typedness of a well-typed query is ensured just by requiring correct typedness of the output positions of the positive literals, while correct typedness of a well-typed clause is ensured just by requiring correct typedness of the input positions of the head and of the output positions of the positive literals in the body.

In the literature we find many properties of well-typed definite programs which hold also for general programs. Here we recall some of them we will use in the rest of the paper.

Remark 3.8 *If $Q := L_1, \ldots, L_n$ is a non-empty well-typed query, then all prefixes, $L_1, \ldots, L_i$ with $i \in \{1, \ldots, n\}$, of it are well-typed too. In particular, its first literal L_1 is well-typed.*

The next Lemma states that well-typed queries are closed under LDNF-resolution. It has been proved by Bronsard *et. al.* in [15] for definite programs and extended to general programs in [12].

Lemma 3.9 *Let P and Q be a well-typed program and a well-typed query, respectively. Then all LDNF-descendants of $P \cup \{Q\}$ are well-typed.*

Lemma 3.10 *Let P and Q be a well-typed program and a well-typed query, respectively. Let θ be a computed answer substitution of a successful LDNF-derivation of $P \cup \{Q\}$. Then $Q\theta$ is correctly typed.*

Proof. The proof follows by a straightforward generalization of Corollary 10.9 and Corollary 10.10 in [2] to LDNF-resolution. ∎

In what follows we denote by $ground_\tau(P)$ the set of all correctly typed ground instances of clauses of P. The proof of the following result is reported in the Appendix.

Theorem 3.11 *Let P and Q be a well-typed program and a well-typed query, respectively, and M be a complete model of $ground_\tau(P)$. If there is a successful LDNF-derivation of $P \cup \{Q\}$ with computed answer substitution θ then $M \models Q\theta$.*

We now define the termination property we focus on.

Definition 3.12 (Typed Termination) *A program P is called* typed terminating *if all LDNF-derivations of P starting in a well-typed query Q are finite.*

The following property holds.

Lemma 3.13 *Let P be a well-typed program. P is typed terminating iff for all well-typed positive literals A, all LDNF-derivations of $P \cup \{A\}$ are finite.*

Proof. Clearly, if P is typed terminating then for all well-typed positive literals A, all LDNF-derivations of $P \cup \{A\}$ are finite.

Suppose now that for all well-typed positive literals A, all LDNF-derivations of $P \cup \{A\}$ are finite. By Lemma 3.9 and Remark 3.8 all selected literals in all LDNF-derivations of P starting in a well-typed query Q are well-typed. Moreover, if all LDNF-derivations of $P \cup \{A\}$ are finite then also all LDNF-derivations of $P \cup \{\neg A\}$ are finite. Then P is typed terminating. ∎

4 Typed Acceptable Programs

In order to prove typed termination of well-typed programs we introduce the concept of *typed acceptable program.*

We first define the concept of *typed level mapping.*

Definition 4.1 (Typed Level Mapping) *Let P be a typed program and $|\ |$ be a level mapping for P. We say that $|\ |$ is a* typed level mapping for P *if*

- *every well-typed atom defined in P is bounded wrt. $|\ |$.*

Example 4.2 *Consider the program* `ROTATE` *of the introduction. The following is a typed level mapping for* `ROTATE`*.*

$|\mathtt{rotate}(l1, l2)| = |l1|_{\mathtt{length0}}$
$|\mathtt{zero}(x)| = 0$
$|\mathtt{append}(l1, l2, l3)| = |l1|_{\mathtt{length}}$

where for a term t, if t is a list then $|t|_{\mathtt{length0}}$ is the length of the maximal prefix of t made by zero's, otherwise it is 0, while $|t|_{\mathtt{length}}$ is equal to the length of the list, otherwise it is 0.

For well-typed programs, we introduce the following notion of typed acceptability. It is in the same style of the notion of well-acceptability introduced in [22], but as we discuss later on there is a main difference in the requirement on the level mapping.

Definition 4.3 (Typed Acceptable Program) *Let P be a well-typed program, $|\ |$ be a typed level mapping for P and M be a complete model of $ground_\tau(P)$.*

- *A clause of P is called* typed acceptable wrt. $|\ |$ and M *if for every ground instance $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of it such that A is correctly typed in its input positions,*

$$\text{if } M \models \mathbf{A} \text{ and } rel(A) \approx rel(B) \text{ then } |A| > |B|.$$

- *P is called* typed acceptable wrt. $|\ |$ and M *if all its clauses are.*

Notice that in the definition of typed acceptability we only require to compare the level of the head with the level of the "reachable" *mutually recursive* literals in clause bodies. This is a much weaker requirement than the one given in both the notions of acceptability and of semi-acceptability, introduced in [9, 10] for proving left termination. In fact, in [9, 10], all the "reachable" literals in the bodies have to be measured.

We first prove a result which provides an incremental method for proving typed termination.

Theorem 4.4 *Let P and R be two programs such that P extends R and $P \cup R$ is well-typed. Let M be a complete model of $ground_\tau(P \cup R)$. Suppose that* (*i*) *if the predicate symbols* p *and* q *are both defined in* P *then neither* $p \sqsupset q$ *nor* $q \sqsupset p$ *(i.e., either they are mutually recursive or independent),*

(*ii*) P *is typed acceptable wrt. a typed level mapping* $|\ |$ *and* M*,*

(*iii*) R *is typed terminating.*

Then $P \cup R$ *is typed terminating.*

Proof. By Lemma 3.13, it is sufficient to prove that for all well-typed positive literals A, all LDNF-derivations of $(P \cup R) \cup \{A\}$ are finite. Let us consider a well-typed atom A.

If A is defined in R, then the thesis trivially holds by (iii).

If A is defined in P, by definition of typed level mapping, A is bounded wrt. $|\ |$ and then $max|A|$ is defined. The proof proceeds by induction on $max|A|$.

Base. Let $max|A| = 0$. In this case, by (i) and (ii), if $c : H \leftarrow \mathbf{L}$ is a clause of P such that H unifies with A and $\mathbf{L}$ is non-empty, then all literals in $\mathbf{L}$ are defined in R. The thesis follows by (iii).

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 $(P \cup R) \cup \{(L_1, \ldots, L_{i-1})\}$ 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 $\sigma = mgu(H, A)$. For all $i \in \{1, \ldots, n\}$, let $L_i = L'_i\sigma$ and θ_i be a computed answer for $(P \cup R) \cup \{(L_1, \ldots, L_{i-1})\}$. By Remark 3.8 and Lemma 3.9, each literal $L_i\theta_i$ is well-typed. We distinguish two cases.

If $L_i\theta_i$ is defined in R then the thesis follows by (iii).

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

First of all, by hypothesis (i), $rel(L_i\theta_i) \approx rel(H')$.

Let γ be a substitution such that $L_i\theta_i\gamma$ is a ground instance of $L_i\theta_i$. Then there exists γ' such that $(L_1, \ldots, L_{i-1})\gamma'$ is a ground instance of $(L_1, \ldots, L_{i-1})\theta_i$, $c\sigma\gamma'$ is a ground instance of c and $L_i\gamma' = L_i\theta_i\gamma$. By the facts that A is well-typed and $L_1, \ldots, L_i$ is a prefix of an LDNF-descendant of $(P \cup R) \cup \{A\}$, it follows that $L_1, \ldots, L_i$ is well-typed. Hence, by Theorem 3.11, $M \models (L_1, \ldots, L_{i-1})\gamma'$. Moreover, since A is correctly typed in its input positions and $A\sigma = H\sigma$ it follows that $H\sigma\gamma$ is correctly typed in its input positions. Then,

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

Then we can conclude that $max|A| > max|L_i\theta_i|$. ■

Let us now prove our general result.

Theorem 4.5 *Let P be a well-typed program, $|\ |$ be a typed level mapping for P and M be a complete model of $ground_\tau(P)$.*

- *If P is typed acceptable wrt. $|\ |$ and M then P is typed terminating.*

Proof. We decompose P into a hierarchy of $n \geq 1$ programs $P := P_1 \cup \ldots \cup P_n$ such that $P_n \sqsupset \ldots \sqsupset P_1$ and for every $i \in \{1, \ldots, n\}$ if the relation symbols p_i and q_i are both defined in P_i then neither $p_i \sqsupset q_i$ nor $q_i \sqsupset p_i$ (i.e., either they are mutually recursive or independent). Moreover, for each P_i, we consider the level mapping $|\ |_i$ defined in the following way: if A is defined in P_i then $|A|_i = |A|$ else $|A|_i = 0$. Notice that each $|\ |_i$ is a typed level mapping and each P_i is typed acceptable wrt. $|\ |_i$ and M.

We prove that for all well-typed queries Q, all LDNF-derivations of $P \cup \{Q\}$ are finite. By induction on n.

Base. Let $n = 1$. This case follows immediately by Theorem 4.4, by putting $P = P_1$ and R empty.

Induction step. Let $n > 1$. Also this case follows by Theorem 4.4, by putting $P = P_n$, and $R = P_1 \cup \ldots \cup P_{n-1}$. In fact,

- if the predicate symbols p_n and q_n are both defined in P_n then neither $p_n \sqsupset q_n$ nor $q_n \sqsupset p_n$;
- P_n is typed acceptable wrt. $|\ |_n$ and M;
- $(P_1 \cup \ldots \cup P_{n-1})$ is typed terminating, by the inductive hypothesis.

■

Example 4.6 *The well-typed program* `ROTATE` *in the modes and types of Example 3.6 is typed acceptable wrt.*

- *the typed level mapping of Example 4.2, and*
- *a complete model M of $ground_\tau$(`ROTATE`) such that*

$$M \models \mathtt{append}(s, [0], t) \textit{ iff } |s|_{\mathtt{length0}} = |t|_{\mathtt{length0}}.$$

It is worth noticing that the condition of typed acceptability offers an extremely powerful and simple method for proving typed termination of a well-typed program. Consider a program (for instance the program `MAP_COLOR` in [10]) composed by many definitions of independent recursive relations and a "main" procedure which correctly calls such relations. All what we have to do here for proving typed termination is to prove termination independently for each recursive definition on its correct calls.

5 Characterizing Typed Terminating Programs

In this section we prove the converse of Theorem 4.5. This provides us with an exact characterization of well-typed, typed terminating general programs.

Similarly to what has been done in [9] such a characterization is limited to non-floundering programs. We recall that an LDNF-derivation *flounders* if there occurs in it or in any of its subsidiary LDNF-trees a query with the first literal being non-ground and negative. An LDNF-tree is called non-floundering if none of its branches flounders.

To prove the converse of Theorem 4.5 we analyze the size of finite LDNF-trees.

We need the following lemma from [9], where for a program P and a query Q, $nodes_P(Q)$ denotes the total number of nodes in the LDNF-tree of $P \cup \{Q\}$ and in all its subsidiary LDNF-trees.

Lemma 5.1 *[9] Let P be a program and Q be a query such that the LDNF-tree of $P \cup \{Q\}$ is finite and non-floundering. Then*

(i) for all substitutions θ, the LDNF-tree of $P \cup \{Q\theta\}$ is finite and non-floundering and $nodes_P(Q\theta) \leq nodes_P(Q)$;

(ii) for all prefixes Q' of Q, the LDNF-tree of $P \cup \{Q'\}$ is finite and non-floundering and $nodes_P(Q') \leq nodes_P(Q)$;

(iii) for all non-root nodes Q' in the LDNF-tree of $P \cup \{Q\}$, $nodes_P(Q') < nodes_P(Q)$.

We will use the following notion.

Definition 5.2 (Non-Floundering on Well-Typed Atoms) *Let P be a typed program. We say that P is* non-floundering on well-typed atoms *if no LDNF-derivation starting in a well-typed atom flounders.*

Notice that if P is a well-typed program, the previous condition is satisfied whenever all positions of negative literals occurring in the clause bodies are input positions and have types which imply groundness.

The following result is proved in the Appendix.

Theorem 5.3 *Let P be a well-typed program such that P is typed terminating and non-floundering on well-typed atoms. Then*

$$\{A \in B_P \mid A \text{ is well-typed and there is a successful LDNF-derivation of } P \cup \{A\}\}$$

is a complete model of $ground_\tau(P)$.

We are now ready to prove the main result of this section.

Theorem 5.4 *Let P be a well-typed program, non-floundering on well-typed atoms.*

- *If P is typed terminating then there exists a typed level mapping $|\ |$ and a complete model M for $ground_\tau(P)$ such that P is typed acceptable wrt. $|\ |$ and M.*

Proof. Let us define a level mapping for P as follows: for all $A \in B_P$

$$\begin{array}{ll} |A| = nodes_P(A) & \text{if } A \text{ is well-typed} \\ |A| = 0 & \text{otherwise.} \end{array}$$

Assume that P is typed terminating. Then the level mapping $|\ |$ for P is well-defined. Moreover, it is a typed level mapping. Note that by definition, for $A \in B_P$, $nodes_P(\neg A) > nodes_P(A) = |A| = |\neg A|$, so $nodes_P(\neg A) > |\neg A|$.

Let M be the complete model for $ground_\tau(P)$ of Theorem 5.3.

We prove that P is typed acceptable wrt. $|\ |$ and M.

Take a clause $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of P and a ground instance $A\theta \leftarrow \mathbf{A}\theta, B\theta, \mathbf{B}\theta$ of it such that $A\theta$ is correctly typed in its input positions. We need to show that

$$\text{if } M \models \mathbf{A}\theta \text{ and } rel(A\theta) \approx rel(B\theta) \text{ then } |A\theta| > |B\theta|.$$

Let σ be an *mgu* of $A\theta$ and A, then $\theta = \sigma\delta$ for some δ. We have:

$$\begin{array}{rcll} |A\theta| & = & nodes_P(A\theta) & \text{(by definition of } |\ |) \\ & > & nodes_P(\mathbf{A}\sigma, B\sigma, \mathbf{B}\sigma) & \text{(by Lemma 5.1 } (iii) \text{ and the fact that} \\ & & & (\mathbf{A}\sigma, B\sigma, \mathbf{B}\sigma) \text{ is a resolvent of } P \cup \{A\theta\}) \\ & \geq & nodes_P(\mathbf{A}\theta, B\theta, \mathbf{B}\theta) & \text{(by Lemma 5.1 } (i)\text{, since } \theta = \sigma\delta) \\ & \geq & nodes_P(B\theta, \mathbf{B}\theta) & \text{(by Lemma 5.1 } (iii)\text{, since } M \models \mathbf{A}\theta) \\ & \geq & nodes_P(B\theta) & \text{(by Lemma 5.1 } (ii)) \\ & = & |B\theta| & \text{(by definition of } |\ |). \end{array}$$

■

6 Conclusions

In this paper we propose a new termination property for general logic programs: typed termination. A general program is typed terminating if it terminates for any well-typed query. We follow the style introduced by Apt and Pedreschi for left termination in [9], and give an algebraic characterization of well-typed, typed terminating programs. To this end we use the concepts of typed level mappings, namely level mappings for which any well-typed query is bounded, and typed acceptability. We also prove that, for well-typed programs, typed acceptability is a necessary and sufficient condition for typed termination.

Most of the programs we write are well-typed and typed termination seems to be a very natural termination property for them. Furthermore typed acceptability supplies a very simple way to prove termination since it requires only to compare the levels of "reachable" mutually recursive literals. Thus in the termination proofs very simple level mappings can be used by exploiting both the independence and the hierarchical dependence among predicate definitions.

Moreover the class of typed terminating programs is included neither into the class of left terminating programs nor into the class of well-terminating ones. In fact there are well-typed programs which terminate for all well-typed queries, but they do not terminate for all ground queries or for all well-moded ones.

The present characterization of typed termination is also a generalization of our previous work on well-termination [22]. In fact in [22] we consider only definite programs while for typed termination we consider general programs. Moreover, when we restrict our type system to the only type *Ground*, i.e., the set of ground terms, well-typed programs coincide with well-moded ones. A moded level mapping is also a typed level mapping, since all well-moded atomic queries are bounded wrt. a moded level mapping. But the reverse is not true, namely a typed level mapping is not a moded level mapping, hence our present requirement of a typed level mapping is less restrictive. In [22] it was not possible to prove that any well-moded well-terminating program is well-acceptable: this property was proved only for a subclass of well-moded programs, the *simply-moded* ones. By weakening the condition on the level mapping, now we obtain a full characterization for well-terminating programs.

Another approach which can capture typed termination is the one proposed by Pedreschi and Ruggeri in [24]. They give a general framework for proving partial and total correctness of general logic programs wrt. Pre/Post specifications. Clearly with Pre/Post specifications also moding and typing properties can be described and well-typing can be expressed. They basically consider *well-asserted programs*, as they are called in [8], which are a generalization of well-typed ones. On the other hand, for proving termination they adopt the classical notion of acceptability defined in [9], thus they require a level mapping for comparing all "reachable" literals in program clauses not only the recursive ones. This is a much stronger requirement than our, it produces in general more complicated level mappings and termination proofs, and in some cases it may make impossible to find a proof, even for programs which are typed terminating. This is due to the fact that they cannot give a full characterization of well-asserted programs terminating for well-asserted queries.

References

[1] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[3] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3&4):335–363, 1991.

[4] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[5] K. R. Apt and H. C. Doets. A new definition of SLDNF-resolution. *Journal of Logic Programming*, 18(2):177–190, 1994.

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

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

[8] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

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

[10] K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.

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

[12] A. Bossi, N. Cocco, S. Etalle, and S. Rossi. Termination in a hierarchy of general logic programs. Technical Report CS-2001-05, Dipartimento di Informatica, Università Ca' Foscari Di Venezia, Italy, March 2001.

[13] A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science*, 124:297–328, 1994.

[14] A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. *Theory and Practice of Logic Programming (TPLP), to appear*, 2000. Available on CoRR: http://arXiv.org/abs/cs/0101023.

[15] F. Bronsard, T. K. Lakshman, and U. S. Reddy. A framework of directionality for proving termination of logic programs. In K. R. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 321–335. MIT Press, 1992.

[16] L. Cavedon. Continuity, consistency and completeness properties for logic programs. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 571–584. The MIT press, 1989.

[17] K. L. Clark. Negation as failure rule. In H. Gallaire and G. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[18] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.

[19] D. De Schreye and K. Verschaetse. Deriving linear size relations for logic programs by abstract interpretation. *New Generation Computing*, 13(2):117–154, 1995.

[20] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. In ICOT Staff, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'92), Tokio*, pages 481–488. ICOT, 1992.

[21] S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms: a missing link in automatic termination analysis. In D. Miller, editor, *Proc. Tenth International Logic Programming Symposium*, number 526 in Lecture Notes in Computer Science, pages 420–436. Springer-Verlag, 1993.

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

[23] J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, 1987. Second edition.

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

[25] J.-G. Smaus. Proving termination of input-consuming logic programs. In D. De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming*, pages 335–349. MIT Press, 1999.

[26] Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Australian Computer Science Conference*, 1995. available at http://www.cs.mu.oz.au/mercury/papers.html.

[27] K. Verschaetse and D. De Schreye. Deriving termination proofs for logic programs using abstract procedures. In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 301–315. MIT Press, 1991.

A Appendix

In this appendix we report the proofs of the technical results used in the paper.

Let us first establish the following claim.

Claim 1 *Let P and Q be a well-typed program and a well-typed query, respectively. The following statements hold.*

(i) *If the LDNF-tree of $ground(P) \cup \{Q\}$ is finitely failed then also the LDNF-tree of $ground_\tau(P) \cup \{Q\}$ is finitely failed.*

(ii) *If there is a successful LDNF-derivation of $ground(P) \cup \{Q\}$ then there is a successful LDNF-derivation of $ground_\tau(P) \cup \{Q\}$.*

Proof. By simultaneous induction on $k = rank(\mathcal{T}, \vartheta)$ where

(1) in case (i), $\mathcal{T}$ is the finitely failed LDNF-tree of $ground(P) \cup \{Q\}$ and $\vartheta = \epsilon$,

(2) in case (ii), $\mathcal{T}$ is the LDNF-tree of $ground(P) \cup \{Q\}$ containing a successful LDNF-derivation and ϑ is its computed answer substitution.

For a formal definition of $rank(\mathcal{T}, \vartheta)$ the reader is referred to [5]. Intuitively, k denotes the number of subsidiary trees that the interpreter would explore during the construction of the finitely failed LDNF-tree of $ground(P) \cup \{Q\}$ in case (i), or the successful LDNF-derivation of $ground(P) \cup \{Q\}$ in case (ii).

Base. $k = 0$. In this case no subsidiary tree is explored during the construction of the LDNF-tree of $ground(P) \cup \{Q\}$.

(i) Let the LDNF-tree of $ground(P) \cup \{Q\}$ be finitely failed. Since $ground(P) \supseteq ground_\tau(P)$, the LDNF-tree of $ground_\tau(P) \cup \{Q\}$ is finitely failed too.

(ii) Let δ be a successful LDNF-derivation of $ground(P) \cup \{Q\}$. We prove that all clauses from $ground(P)$ used to resolve an atom in δ are correctly typed and thus belong to $ground_\tau(P)$. Indeed, let $c := H \leftarrow \mathbf{L}$ be a clause of $ground(P)$ and A be a selected atom in δ such that A and H unify. By properties of LDNF-resolution, since there exists a successful derivation of $ground(P) \cup \{A\}$, then there exists a successful derivation of $ground(P) \cup \{H\}$ too. Hence, by Lemma 3.10, H is correctly typed. Moreover, since $\mathbf{L}$ is an LDNF-resolvent of $ground(P) \cup \{A\}$, then there exists also a successful derivation of $ground(P) \cup \{\mathbf{L}\}$. Again, by Lemma 3.10, $\mathbf{L}$ is correctly typed. This proves that the clause c is correctly typed and thus belong to $ground_\tau(P)$.

Induction step. $k > 0$.

(i) Let the LDNF-tree of $ground(P) \cup \{Q\}$ be finitely failed. The proof follows by a secondary induction on the depth h of this tree. Let $Q := L_1, \ldots, L_n$.

`Base`. $h = 1$. We distinguish two cases.

(a) L_1 is a positive literal. In this case there is no clause in $ground(P)$ whose head unifies with L_1. Since $ground(P) \supseteq ground_\tau(P)$, then there is also no clause in $ground_\tau(P)$ whose head unifies with L_1, i.e., the LDNF-tree of $ground_\tau(P) \cup \{Q\}$ is finitely failed.

(b) L_1 is a negative literal. Let $L_1 = \neg A$. In this case, there exists a successful LDNF-derivation of $ground(P) \cup \{A\}$. By the principal induction on k, there exists also a successful LDNF-derivation of $ground_\tau(P) \cup \{A\}$. This proves that the LDNF-tree of $ground_\tau(P) \cup \{Q\}$ is finitely failed.

`Induction step.` $h > 1$. Again we distinguish two cases.

(a) L_1 is a positive literal. In this case, all direct LDNF-descendants of $ground(P) \cup \{Q\}$ have a finitely failed LDNF-tree in $ground(P)$. From the fact that $ground(P) \supseteq ground_\tau(P)$, it follows that the set of direct LDNF-descendants of $ground_\tau(P) \cup \{Q\}$ is contained in the set of direct LDNF-descendants of $ground(P) \cup \{Q\}$. The thesis follows by the secondary induction on h, since the depth of the subtrees is smaller than h.

(b) L_1 is a negative literal. Let $L_1 = \neg A$. Since $h > 1$, the LDNF-tree of $ground(P) \cup \{A\}$ is finitely failed. By the principal induction on k, the LDNF-tree of $ground_\tau(P) \cup \{A\}$ is finitely failed too. Hence $L_2, \ldots, L_n$ is the direct LDNF-descendant both of $ground(P) \cup \{Q\}$ and of $ground_\tau(P) \cup \{Q\}$. The thesis follows by the secondary induction on h.

(ii) Let δ be a successful LDNF-derivation of $ground(P) \cup \{Q\}$ and $k > 0$. By the principal induction on k, all the subsidiary trees explored during the construction of δ are finitely failed in $ground_\tau(P)$. Moreover, as in the base case for $k = 0$, for all positive literals selected in δ, we can prove that all input clauses are correctly typed and thus belong to $ground_\tau(P)$. This proves that there exists a successful LDNF-derivation of $ground_\tau(P) \cup \{Q\}$. ∎

We are now in position to prove Theorem 3.11.

Theorem 3.11 *Let P and Q be a well-typed program and a well-typed query, respectively, and M be a complete model of $ground_\tau(P)$. If there is a successful LDNF-derivation of $P \cup \{Q\}$ with computed answer substitution θ then $M \models Q\theta$.*

Proof Suppose that there is a successful LDNF-derivation of $P \cup \{Q\}$ with computed answer θ. For any ground instance Q' of $Q\theta$, there is a successful LDNF-derivation of $ground(P) \cup \{Q'\}$, by properties of LDNF-resolution. Thus, there is a successful LDNF-derivation of $ground_\tau(P) \cup \{Q'\}$, by Claim 1 ($ii$). Let M be a complete model of $ground_\tau(P)$. By soundness of LDNF-resolution wrt. completion [17], for any ground instance Q' of $Q\theta$, $M \models Q'$, i.e., $M \models Q\theta$. ∎

Let us recall the following theorem due to Apt, Blair and Walker [4] which provides a method for verifying whether an interpretation is a model of $comp(P)$. It uses the following definition.

Definition A.1 (Supported Model) *A model M of a program P is called supported if for all ground atoms A, $I \models A$ implies that $I \models \mathbf{L}$ for some general clause $A \leftarrow \mathbf{L} \in ground(P)$.*

Theorem A.2 *A Herbrand interpretation I is a model of $comp(P)$ iff it is a supported model of P.*

We can then prove the following theorem.

Theorem 5.3 *Let P be a well-typed program such that P is typed terminating and non-floundering on well-typed atoms. Then*

$$\{A \in B_P \mid A \textit{ is well-typed and there is a successful LDNF-derivation of } P \cup \{A\}\}$$

is a complete model of $ground_\tau(P)$.

Proof. Let M be the set

$$\{A \in B_P \mid A \text{ is well-typed and there is a successful LDNF-derivation of } P \cup \{A\}\}.$$

We show that M is a Herbrand model of $comp(ground_\tau(P))$. To this end, we use Theorem A.2 and show that M is a supported model of $ground_\tau(P)$.

To establish that M is a model of $ground_\tau(P)$, assume by contradiction that some clause $A \leftarrow \mathbf{L}$ from $ground_\tau(P)$ is false in M. Then $M \models \mathbf{L}$ and $M \not\models A$. Since P is typed terminating and non-floundering on well-typed atoms, $M \not\models A$ implies that the LDNF-tree for $P \cup \{A\}$ is finitely failed and non-floundering.

For some ground substitution γ, $A \leftarrow \mathbf{L} = (A' \leftarrow \mathbf{L}')\gamma$ where $c := A' \leftarrow \mathbf{L}'$ is a clause from P. Thus A and A' unify.

Let $\mathbf{L}'\sigma$ be the resolvent of $P \cup \{A\}$ with the input clause c. The LDNF-tree of $P \cup \{\mathbf{L}'\sigma\}$ is also finitely failed and non-floundering. As $\mathbf{L}$ is an instance of $\mathbf{L}'\sigma$, by Lemma 5.1 (i) we have that the LDNF-tree for $P \cup \{\mathbf{L}\}$ is non-floundering. Moreover, it is finitely failed, since the queries occurring in the LDNF-tree of $P \cup \{\mathbf{L}\}$ are all instances of the queries occurring in the LDNF-tree of $P \cup \{\mathbf{L}'\sigma\}$. But the fact that $\mathbf{L}$ is well-typed and the LDNF-tree of $P \cup \{\mathbf{L}\}$ is finitely failed and non-floundering contradicts the hypothesis that $M \models \mathbf{L}$.

To establish that M is a supported model of $ground_\tau(P)$, consider $A \in B_P$ such that $M \models A$, and let c be the first input clause used in a successful LDNF-derivation of $P \cup \{A\}$. Let $\mathbf{L}'\sigma$ be the resolvent of $P \cup \{A\}$ from the clause c. Clearly, a successful LDNF-derivation for $P \cup \{\mathbf{L}'\sigma\}$ with computed answer θ can be extracted from the successful LDNF-derivation of $P \cup \{A\}$. Let $\mathbf{L}$ be a ground instance of $\mathbf{L}'\sigma\theta$. By Lemma 3.10, both A and $\mathbf{L}$ are correctly typed. Hence $A \leftarrow \mathbf{L} \in ground_\tau(P)$. By properties of LDNF-resolution there exists a successful LDNF-derivation for $P \cup \{\mathbf{L}\}$, hence $M \models \mathbf{L}$. This establishes that M is a supported interpretation for $ground_\tau(P)$. ∎