TLP **5** (3): 355–390, 2005. © 2005 Cambridge University Press
doi:10.1017/S1471068404002248 Printed in the United Kingdom

On termination of meta-programs

ALEXANDER SEREBRENIK*
LaQuSo, TU/e, Den Dolech 2, 5600 MB Eindhoven, The Netherlands
(*e-mail:* a.serebrenik@tue.nl)

DANNY DE SCHREYE
Department of Computer Science, K.U. Leuven, Celestijnenlaan 200A, B-3001, Heverlee, Belgium
(*e-mail:* Danny.DeSchreye@cs.kuleuven.ac.be)

Abstract

The term *meta-programming* refers to the ability of writing programs that have other programs as data and exploit their semantics. The aim of this paper is presenting a methodology allowing us to perform a correct termination analysis for a broad class of practical meta-interpreters, including negation and performing different tasks during the execution. It is based on combining the power of general orderings, used in proving termination of term-rewrite systems and programs, and on the well-known acceptability condition, used in proving termination of logic programs. The methodology establishes a relationship between the ordering needed to prove termination of the interpreted program and the ordering needed to prove termination of the meta-interpreter together with this interpreted program. If such a relationship is established, termination of one of those implies termination of the other one, i.e. the meta-interpreter preserves termination. Among the meta-interpreters that are analysed correctly are a proof trees constructing meta-interpreter, different kinds of tracers and reasoners.

KEYWORDS: termination analysis, meta-programming, meta-interpreter

1 Introduction

The choice of logic programming as a basis for meta-programming offers a number of practical and theoretical advantages. One of them is the possibility of tackling critical foundation problems of meta-programming within a framework with a strong theoretical basis. Another is the surprising ease of programming. These advantages motivated intensive research on meta-programming inside the logic programming community (Apt and Turini 1995; Hill and Gallagher 1998; Levi and Ramundo 1993; Martens and De Schreye 1995b; Pedreschi and Ruggieri 1997). Meta-programming in logic languages is also a powerful technique for many

* This work was carried out during the author's stay at Department of Computer Science, K. U. Leuven, Belgium.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

different application areas such as aspect-oriented programming (De Volder and D'Hondt 1999; Brichau *et al.* 2002) and constraints solving (Lamma *et al.* 1996).

Moreover, termination analysis is one of the most intensive research areas in logic programming as well (Bossi *et al.* 2002; Bruynooghe *et al.* 2002; Decorte *et al.* 1999; Dershowitz *et al.* 2001; Genaim *et al.* 2002; Lee 2002; Mesnard and Ruggieri 2003; Verbaeten *et al.* 2001).

Traditionally, termination analysis of logic programs has been done either by the "transformational" approach or by the "direct" one. A transformational approach first transforms the logic program into an "equivalent" term-rewrite system (or, in some cases, into an equivalent functional program). Here, equivalence means that, at the very least, the termination of the term-rewrite system should imply the termination of the logic program, for some predefined collection of queries.[1] Direct approaches do not include such a transformation, but prove the termination directly on the basis of the logic program. These approaches are usually based on *level mappings*, functions that map atoms to natural numbers, and *norms* that map terms to natural numbers. De Schreye *et al.* (1992) proved that termination is equivalent to *acceptability*, i.e. to existence of a level mapping that decreases from the call to the head of a clause to the appropriately instantiated call to the recursive body subgoal. We have developed (De Schreye and Serebrenik 2002) an approach that provides the best of both worlds: a means to incorporate into "direct" acceptability-based approaches the generality of general term-orderings.

The aim of this paper is to present a methodology that allows us to perform a correct termination analysis for a broad class of meta-interpreters. This methodology is based on the "combined" approach to termination analysis mentioned above. It makes possible the reuse of termination proofs obtained for the interpreted program as a base for the termination proof of the meta-program. As Example 2 will illustrate, with the level mappings based techniques the reuse would be impossible, even if the simplest "vanilla" meta-interpreter, presented in the following example, was considered.

Example 1

Our research has been motivated by the famous "vanilla" meta-interpreter M_0, undoubtedly belonging to logic programming classics.

$$solve(true).$$
$$solve((Atom, Atoms)) \leftarrow solve(Atom), solve(Atoms).$$
$$solve(Head) \leftarrow clause(Head, Body), solve(Body).$$

Atoms of the form $clause(Head, Body)$ represent the interpreted program. Termination of the "vanilla" meta-interpreter has been studied by Pedreschi and Ruggieri. They have proved that termination of the query Q with respect to a program P implies termination of the query $solve(Q)$ with respect to M_0 and P (Pedreschi and Ruggieri 1997: Corollary 40). We are going to see that the two statements are

[1] The approach of Arts (1997) is exceptional in the sense that the termination of the logic program is concluded from a weaker property of *single-redex normalisation* of the term-rewrite system.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

equivalent, i.e., the query Q terminates with respect to a program P if and only if the query $solve(Q)$ LD-terminates with respect to M_0 and P. □

Even though the termination of an interpreted program might easily be proved with level mappings, the termination proof of the meta-interpreter extended by this program with respect to the set of the corresponding queries might be much more complex. As the following example demonstrates, in some cases no linear level mapping can prove termination of the meta-program, despite the fact that termination of an interpreted program can be shown with linear level mappings. Recall, that a level mapping $|\cdot|$ and a norm $\|\cdot\|$ are called *linear* if

$$|p(t_1,\ldots,t_n)| = c^p + \Sigma_{i=1}^n a_i^p \|t_i\|,$$
$$\|f(t_1,\ldots,t_n)\| = c^f + \Sigma_{i=1}^n a_i^f \|t_i\|,$$

and for all p, f and for all i, the coefficients are non-negative integers.

Example 2
Let P be the following program:

$l(X) \leftarrow p(X), r(X).$ $\quad p(X) \leftarrow q(X,Y), p(Y).$ $\quad r(f(X)) \leftarrow s(Y), r(X).$
$q(f(Z), Z).$ $\quad p(0).$ $\quad r(0).$
$s(0).$

This program clearly terminates for $l(t)$ for every ground term t. To show termination one may, for example, use a term-size norm $\|\cdot\|$, defined on a term t as a number of nodes in the tree-representation of t, and a level mapping based on $\|\cdot\|: |l(t)| = \|t\|$, $|p(t)| = \|t\|$, $|r(t)| = \|t\|$, $|s(t)| = 0$ and $|q(t_1, t_2)| = 0$. One can show that the program above satisfies the acceptability condition of De Schreye *et al.* (1992) with respect to this level mapping and, hence, queries of the form $l(t)$ with a ground argument t terminate with respect to the program.

Corollary 40 of Pedreschi and Ruggieri (1997) implies that $solve(l(t))$ terminates for every ground t as well. However, if a linear level mapping $|\cdot|$ and a linear norm $\|\cdot\|$ are used in the traditional way to prove termination of the meta-program, the following constraints are obtained among others:

$$|solve(p(X))| > |solve((q(X,Y), p(Y)))| \quad (1)$$
$$|solve(r(f(X)))| > |solve((s(Y), r(X)))| \quad (2)$$
$$|solve((s(Y), r(X)))| > |solve(r(X))|, \quad (3)$$

where $>$ denotes the traditional ordering on natural numbers. In general, the last constraint should take into consideration the intermediate body atom $s(Y)$ as well, but one can prove that it cannot affect $r(X)$. Observe that, unlike the interpreted program, the comma of "$(s(Y), r(X))$" in the meta-program is a functor to be considered during the termination analysis.

One can show that there is no linear level mapping that can satisfy (1)–(3). Indeed, constraints (1)–(3) can be reduced to the following (without loss of generality, $c^{solve} = 0$ and $a_1^{solve} = 1$). For functors having only one argument the subscript is

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

dropped.

$$c^p + a^p\|X\| > c' + a'_1c^q + a'_1a_1^q\|X\| + a'_1a_2^q\|Y\| + a'_2c^p + a'_2a^p\|Y\|$$
$$c^r + a^rc^f + a^ra^f\|X\| > c' + a'_1c^s + a'_1a^s\|Y\| + a'_2c^r + a'_2a^r\|X\|$$
$$c' + a'_1c^s + a'_1a^s\|Y\| + a'_2c^r + a'_2a^r\|X\| > c^r + a^r\|X\|$$

Further reduction gives, among others, the following inequalities:

$$c^p > c' + a'_1c^q + a'_2c^p \tag{4}$$
$$c^r + a^rc^f > c' + a'_1c^s + a'_2c^r \tag{5}$$
$$c' + a'_1c^s + a'_2c^r > c^r \tag{6}$$
$$a'_2a^r \geqslant a^r \tag{7}$$

Condition (4) implies that $a'_2 = 0$. Thus, by (7) $a^r = 0$ holds as well. However, (5) and (6) imply that $a^rc^f > 0$, which provides the desired contradiction. □

Note that by the well-known result of De Schreye *et al.* (1992), termination of the meta-program implies existence of a non-linear level mapping that would prove termination. However, such a level mapping might be difficult or even impossible to generate automatically.

One can consider a number of possible solutions to this problem. First, we may restrict attention to a specific class of interpreted programs, such that if their termination can be proved via linear norms and level mappings, so should be the termination of the meta-program obtained from it and M_0. As the following example illustrates even for this restricted class of programs no obvious relation can be established between a level mapping required to prove termination of the interpreted program and a level mapping required to prove termination of the meta-program. Observe, that results on modular termination proofs for logic programs (Apt and Pedreschi 1994; Bossi *et al.* 2002; Pedreschi and Ruggieri 1996; Verbaeten *et al.* 2001) further discussed in section 7 are not applicable here, since the level-mapping required to prove termination of calls to *clause* is trivial.

Example 3
Let P be the following program:

$$p([X, Y|T]) \leftarrow p([Y|T]), p(T).$$

Termination of the set of queries $\{p(t)|\ t$ is a list of finite length$\}$ can easily be proved, for example by using a level mapping $|p(X)| = \|X\|_l$, where $\|\cdot\|_l$ is a list-length norm defined as $\|[h|t]\|_l = 1 + \|t\|_l$ for lists and as $\|t\|_l = 0$ for terms other than lists. However, when this program is considered together with M_0 this level mapping and this norm cannot be extended in a way allowing us to prove termination, even though there exists a linear level mapping and a linear norm that provide a termination proof. In our case, the following linear level mapping is sufficient to prove termination: $|solve(A)| = \|A\|$, $\|(A, B)\| = 1 + \|A\| + \|B\|$, $\|p(X)\| = 1 + \|X\|$, $\|[H|T]\| = 1 + 3\|T\|$. □

Thus, even though modern termination analysis techniques, such as the constraint-based approach of Decorte *et al.* (1999), are able to derive the level mapping required,

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

they cannot reuse any information from a termination proof of the interpreted program to do so, and the process has to be restarted from scratch. Moreover, the constraints set up for such examples are fairly complex (n body atoms are interpreted as a ,/2-term of depth n and reasoning on them requires products of (at least) n parameters). Other approaches based on level mappings work on a basis of fixed norms (Dershowitz *et al.* 2001; Codish and Taboch 1999), like list-length and term-size, and therefore fail to prove termination of the example. Hence, we follow a different way and instead of considering level mappings and norms, we move to the general orderings based framework originally presented in De Schreye and Serebrenik (2002).

In order for meta-interpreters to be useful in applications they should be able to cope with a richer language than the one of the "vanilla" meta-interpreter, including, for example, negation. Moreover, typical applications of meta-interpreters, such as debuggers, also require the production of some additional output or the performance of some additional tasks during the execution, such as constructing proof trees or cutting "unlikely" branches for an uncertainty reasoner with cutoff. These extensions can and usually will influence termination properties of the meta-interpreter. To this end we first identify popular classes of meta-interpreters, including the important *extended meta-interpreters* (Martens and De Schreye 1995b). Next, we use the orderings-based framework to find conditions implying that termination is not violated or not improved. By combining these conditions one obtains the requirements for termination preservation.

The rest of this paper is organised as follows. After some preliminary remarks, we present the general orderings based framework of De Schreye and Serebrenik (2002). Next we introduce basic definitions and discuss the methodology developed, as it applies to the "vanilla" meta-interpreter M_0. Afterwards, we show how the same methodology can be applied to more advanced meta-interpreters.

2 Preliminaries

We follow the standard notation for terms and atoms. A *query* is a finite sequence of atoms. Given an atom A, $rel(A)$ denotes the predicate occurring in A. $Term_P$ and $Atom_P$ denote, respectively, the sets of all terms and atoms that can be constructed from the language underlying P. The extended Herbrand Universe U_P^E (the extended Herbrand base B_P^E) is a quotient set of $Term_P$ ($Atom_P$) modulo the variant relation. Recall that the *quotient set* of a set X with respect to an equivalence relation $\sim$ is the set consisting of all equivalence classes induced on X by $\sim$.

We refer to an SLD-tree constructed using the left-to-right selection rule of Prolog as an LD-tree. We will say that a query Q *LD-terminates* for a program P, if the LD-tree for $\{Q\} \cup P$ is finite.

The following definition is similar to Definition 6.30 Apt (1997).

Definition 1
Let P be a program and p, q be predicates occurring in it.

- We say that p *refers to* q *in* P if there is a clause in P that uses p in its head and q in its body.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

- We say that p *depends on* q *in* P and write $p \sqsupseteq q$, if (p,q) is in the transitive closure of the relation *refers t o*.
- We say that p *and* q *are mutually recursive* and write $p \simeq q$, if $p \sqsupseteq q$ and $q \sqsupseteq p$.

The only difference between this definition and the original definition of Apt is that we define $\sqsupseteq$ as a transitive closure of the *refers to* relation and not as a reflexive transitive closure of it. Thus, we can say that the predicate p is recursive if and only if $p \simeq p$, while if the definition of Apt is followed, $p \simeq p$ holds for all p. We also abbreviate $p \sqsupseteq q$, $q \not\sqsupseteq p$ by $p \sqsupset q$.

Results for termination of meta-interpreters presented in this paper are based on the notion of order-acceptability with respect to a set of queries, studied in De Schreye and Serebrenik (2002). This notion of order-acceptability generalises the notion of acceptability with respect to a set (Decorte and De Schreye 1998) in two ways: (1) it generalises it to general orderings, and (2) it generalises it to mutual recursion, using the standard notion of mutual recursion (Apt 1997) – the original definition of acceptability required decrease only for calls to the predicate that appears in the head of the clause. This restriction limited the approach to programs with direct recursion only.

We start by reviewing some properties of orderings. A *quasi-ordering* over a set S is a reflexive and transitive relation $\geqslant$ defined on elements of S. We define the associated equivalence relation $\leqslant\geqslant$ as $s \leqslant\geqslant t$ if and only if $s \geqslant t$ and $t \geqslant s$, and the associated *ordering* $>$ as $s > t$ if and only if $s \geqslant t$ but not $t \geqslant s$. If neither $s \geqslant t$, nor $t \geqslant s$ we write $s \|_{\geqslant} t$. If $s \geqslant t$ or $t \geqslant s$ hold for all s and t, the quasi-ordering is called *total*, otherwise it is called *partial*. Sometimes, in order to distinguish between different quasi-orderings and associated relations, we also use $\succeq$, $\succ$, $\preceq\succeq$ and $\|_{\succeq}$. An ordered set S is said to be *well-founded* if there are no infinite descending sequences $s_1 > s_2 > \ldots$ of elements of S. If the set S is clear from the context we will say that the ordering, defined on it, is well-founded.

Before introducing the order-acceptability we need the following notion. The *call set*, $Call(P,S)$, is the set of all atoms A, such that a variant of A is a selected atom in some branch of the LD-tree for $P \cup \{Q\}$, for some $Q \in S$. Techniques for inferring the supersets of the call set were suggested in Janssens and Bruynooghe (1992) and Janssens *et al.* (1994). For the sake of simplicity we write $Call(P,Q)$ instead of $Call(P,\{Q\})$.

Definition 2
Let S be a set of atomic queries and P a definite program. P is *order-acceptable with respect to* S if there exists a well-founded ordering $>$ over $Call(P,Q)$, such that

- for any $A \in Call(P,S)$
- for any clause $A' \leftarrow B_1,\ldots,B_n$ in P, such that $\mathrm{mgu}(A,A') = \theta$ exists,
- for any atom B_i, such that $rel(B_i) \simeq rel(A)$
- for any computed answer substitution σ for $\leftarrow (B_1,\ldots,B_{i-1})\theta$:

$$A > B_i\theta\sigma$$

In De Schreye and Serebrenik (2002) we prove the following theorem.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

Theorem 1
Let P be a program. P is order-acceptable with respect to a set of atomic queries S if and only if P is LD-terminating for all queries in S.

We discovered that order-acceptability is a powerful notion, allowing us a wide variety of programs, such as *normalisation* (Decorte *et al.* 1999), *derivative* (Dershowitz and Manna 1979), *bid* (Bueno *et al.* 1994), and credit evaluation expert system (Sterling and Shapiro 1994) to mention a few. In this paper we will see that order-acceptability plays a key role in analysing termination behaviour of meta-programs. We also remark that the declarative version of our termination proof method, so called *rigid acceptability* also presented in De Schreye and Serebrenik (2002), cannot be used to analyse termination behaviour of meta-programs as rigid acceptability implies termination but it is no longer equivalent to it.

3 Basic definitions

In this section, we present a number of basic definitions. We start by defining the kind of program we call a meta-program. Then we introduce two semantic notions that relate computed answers of the interpreted program to computed answers of the meta-program. Finally, we conclude by discussing an appropriate notion of termination for meta-interpreters.

We have seen already in Example 1 that the input program is represented as a set of atoms of the predicate *clause*. We call this representation a *clause-encoding* and define it formally as follows:

Definition 3
Let P be a program. The *clause-encoding* $ce(P)$ is a collection of facts of a new predicate *clause*, such that $clause(H, B) \in ce(P)$ if and only if $H \leftarrow B$ is a clause in P.

Example 4
Let P be the following program:

$$p(X) \leftarrow q(X). \quad q(b). \quad s \leftarrow r, t.$$

Then, the following program is $ce(P)$:

$$clause(p(X), q(X)). \quad clause(q(b), true). \quad clause(s, (r, t)). \qquad \square$$

An alternative representation of an interpreted program and the related meta-interpreter will be discussed in section 6 (Example 25).

A *meta-interpreter* for a language is an interpreter for the language written in the language itself. We follow Sterling and Shapiro (1994) by using a predicate *solve* for the meta-interpreter predicate. Sometimes a name *demo* is used (Pedreschi and Ruggieri 1997). One of the first meta-interpreters was introduced by Kowalski (1979).

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

Definition 4
The program P is called a *meta-program* if it can be represented as $M \cup I$, such that:

- I is a clause-encoding of some program P'.
- M defines a predicate *solve* that does not appear in P'.

M is called the *meta-interpreter*. P' is called the *interpreted program.*

We also assume that neither , /2 nor *clause*/2 appears in the language underlying the interpreted program. Observe that if this assumption is violated, a clear distinction between the meta-interpreter and the interpreted program is no longer possible. Note that this restriction implies that we cannot study higher-order meta-programs (i.e. programs with two or more meta-layers). Distinguishing between the meta-interpreter and the interpreted program is essential, for example, to ease the recognition of a program as a meta-program. Observe that meta-interpreters like "vanilla" mix an interpreted language and the meta-language because of the call to *clause*/2 in the definition of *solve*. An alternative class of meta-interpreters keeping these languages strictly separate is considered in section 6 (Example 25).

Now we are going to define the notions of *soundness* and *completeness* for meta-interpreters, that relate computed answers of the interpreted program to the computed answers of the meta-program. It should be noted that we define these notions for *solve* with arity $n \geqslant 1$. Second, third, etc. arguments are often required to support added functionality (see M_2 in Example 5). Often these arguments will store some information about the interpreted program obtained during the execution of the meta-interpreter. For example, M_2 in Example 5 returns the maximal depth of the LD-tree of the interpreted program and the interpreted query.

Definition 5
The meta-interpreter M defining *solve* with arity n is called *sound* if for every program P and every query $Q_0 \in B_P^E$ for every $s_1, \ldots, s_{n-1} \in U_M^E$ if $solve(t_0, t_1, \ldots, t_{n-1})$ is a computed answer for $\{solve(Q_0, s_1, \ldots, s_{n-1})\} \cup M \cup ce(P)$ then t_0 is a correct answer for $\{Q_0\} \cup P$.

The definition of soundness, as well as further definitions, requires some property to hold for *all* programs. These definitions do not depend on the considered class of programs. However, constructing meta-interpreters that will satisfy the properties required for all Prolog programs can be difficult. Thus, we start by restricting the class of programs considered to definite logic programs.

To simplify the presentation, we denote $Vars_n$ a (countably infinite) set of linear sequences of length n of free variables. Recall that a sequence of free variables is called *linear* if all the variables are different.

Definition 6
The meta-interpreter M defining *solve* with arity n is called *complete* if for every program P, every query $Q_0 \in B_P^E$ and every computed answer t_0 for $\{Q_0\} \cup P$ holds:

- if $n = 1$, then there exists $t \in U_{M \cup ce(P)}^E$ such that $solve(t)$ is a computed answer for $\{solve(Q_0)\} \cup M \cup ce(P)$ and t_0 is an instance of t.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

- if $n>1$, then there exist $(v_1,\ldots,v_{n-1})\in Vars_{n-1}$ and $t,t_1,\ldots,t_{n-1}\in U^E_{M\cup ce(P)}$ such that $solve(t,t_1,\ldots,t_{n-1})$ is a computed answer for $\{solve(Q_0,v_1,\ldots,v_{n-1})\}\cup(M\cup ce(P))$, and t_0 is an instance of t.

Example 5

The following meta-interpreter M_1 is sound, but is not complete: $solve(A)\leftarrow fail$. The meta-interpreter $solve(A,X).$ is complete, but it is not sound. The "vanilla" meta-interpreter M_0 (Example 1) is both sound and complete, as shown in Levi and Ramundo (1993).

The following meta-interpreter M_2 is also sound and complete. Given a program P and a query Q, computing $solve(Q,v)$, where v is a free variable, with respect to $M_2\cup ce(P)$ not only mimics the execution of Q with respect to P but also calculates the maximal depth of the LD-tree.

$$solve(true,0).$$
$$solve((A,B),K)\leftarrow solve(A,M),solve(B,N),max(M,N,K).$$
$$solve(A,s(N))\leftarrow clause(A,B),solve(B,N).$$

$$max(0,0,0).$$
$$max(s(X),0,s(X)).$$
$$max(0,s(X),s(X)).$$
$$max(s(X),s(Y),s(Z))\leftarrow max(X,Y,Z).$$

It is intuitively clear why this meta-interpreter is sound. In the latter part of the paper we investigate an important class of meta-interpreters, including M_2, and prove that all meta-interpreters in this class are sound (Lemma 2).

To see that M_2 is complete recall that the second argument of a *query* that is required by Definition 6 should be a variable. Indeed, in this case unifications of calls to *solve* with the corresponding clauses will depend only on the first argument of *solve* and calls to *max* cannot affect meta-variables, fail or introduce infinite execution. Formally, completeness of this meta-interpreter follows from Lemma 8 and from the fact that for any interpreted program P and any interpreted query Q, M_2 extended by P terminates for all calls to *max* obtained with respect to $solve(Q,v)$, where v is a free variable. □

Recall that our aim is to study termination of meta-interpreters, that is termination of queries of the form $solve(Q_0,H_1,\ldots,H_n)$, where Q_0 is a query with respect to the interpreted program. Thus, the crucial issue is to define an appropriate *notion of termination* for meta-interpreters. It should be observed that requiring termination of $solve(Q_0,H_1,\ldots,H_n)$ for all possible queries Q_0 and all possible interpreted programs P is undesirable. In fact, there are no interesting meta-interpreters satisfying this property. Therefore, instead of termination we consider termination preservation. For many applications, such as debuggers, this is the desired behaviour of a meta-interpreter. However, there are many meta-interpreters that may change termination behaviour of the interpreted program, either by improving or by violating it.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

Definition 7
(non-violating LD-termination)

1. Let M be a meta-interpreter defining *solve* with arity 1. M is called *non-violating LD-termination* if for every program P and every query $Q_0 \in B_P^E$ if the LD-tree of $\{Q_0\} \cup P$ is finite, then the LD-tree of $\{solve(Q_0)\} \cup (M \cup ce(P))$ is finite as well.
2. Let M be a meta-interpreter defining *solve* with arity $n+1$, $n>0$. M is called *non-violating LD-termination with respect to* $S \subseteq (U_M^E)^n$ if for every program P and every query $Q_0 \in B_P^E$ if the LD-tree of $\{Q_0\} \cup P$ is finite, then for every sequence $(H_1, \ldots, H_n) \in S$, the LD-tree of $\{solve(Q_0, H_1, \ldots, H_n)\} \cup (M \cup ce(P))$ is finite as well.

Observe that S formalises the intuitive notion of a set of sequences of "arguments other than the meta-argument". It should be noted that the two cases can not be collapsed to one, i.e. $n>0$ is essential. Indeed, assume that $n=0$ in the second case. Then, $(U_M^E)^n$ is empty and, therefore, S is empty as well. Hence, there exists no sequence $(H_1, \ldots, H_n) \in S$ and universally quantified term in the "then"-clause is always true. In other words, M would be called non-violating LD-termination if for every program P and every query Q_0 the LD-tree of $\{Q_0\} \cup P$ is finite, which is useless as a definition. Therefore, the case $n=0$ should be considered separately.

It should be noted that traditionally the feature introduced in Definition 7 is called *improving termination*. However, this term is not quite accurate, since by improving we do not mean that the meta-program terminates *more often* than the original one, but that it terminates *at least as often* as the original one. Thus, we chose to use more clear terminology.

It also follows from the definition of non-violation that every meta-interpreter defining *solve* with arity greater than 1 does not violate termination with respect to the empty set.

Example 6
Recall the meta-interpreters shown in Example 5. M_1 does not violate termination, and M_2 does not violate termination with respect to $(U_{M_2}^E)^1$, that is with respect to $U_{M_2}^E$. □

The dual notion is of termination non-improving.

Definition 8
(non-improving LD-termination)

1. Let M be a meta-interpreter defining *solve* with arity 1. M is called *non-improving LD-termination* if for every program P and every $solve(Q_0) \in B_{M \cup ce(P)}^E$, finiteness of the LD-tree of $\{solve(Q_0)\} \cup (M \cup ce(P))$ implies finiteness of the LD-tree of $\{Q_0\} \cup P$.
2. Let M be a meta-interpreter defining *solve* with arity $n+1$, $n>0$. M is called *non-improving LD-termination with respect to* $S \subseteq (U_M^E)^n$ if for every program P and every query $solve(Q_0, H_1, \ldots, H_n) \in B_{M \cup ce(P)}^E$, such that $(H_1, \ldots, H_n) \in S$, finiteness of the LD-tree of $\{solve(Q_0, H_1, \ldots, H_n)\} \cup (M \cup ce(P))$ implies finiteness of the LD-tree of $\{Q_0\} \cup P$.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

Example 7
The meta-interpreter M_2 does not improve termination with respect to *Vars*, where *Vars* is a countably infinite set of variables. □

Finally, we define *termination preservation.*

Definition 9
Let M be a meta-interpreter defining *solve* with arity $n+1$. We say that M is *preserving termination* (*preserving termination with respect to* $S \subseteq (U_M^E)^n$, if $n > 0$), if it is non-violating LD-termination (i.e. non-violating LD-termination with respect to S) and non-improving LD-termination (i.e. non-improving LD-termination with respect to S).

The "vanilla" meta-interpreter M_0 preserves termination and the meta-interpreter M_2 preserves termination with respect to *Vars*, that is if it is used to measure the depth of LD-refutation of a given query, and not to bound it. In the next sections we prove these statements.

4 Termination of the "vanilla" meta-interpreter

Termination of the "vanilla" meta-interpreter, presented in Example 1, has been studied by Pedreschi and Ruggieri. They have proved that "vanilla" does not violate termination (Pedreschi and Ruggieri, 1997: Corollary 40). However, we can claim more – this meta-interpreter preserves termination.

We base our proof on soundness and completeness of "vanilla", proved in Levi and Ramundo (1993). Observe that, in general, soundness and completeness are not sufficient for the call set to be preserved. Indeed, consider the following example, motivated by the ideas of unfolding (Bossi and Cocco 1994).

Example 8
The following meta-interpreter M_3 eliminates calls to undefined predicates.

$$
\begin{aligned}
&solve(true).\\
&solve((A,B)) \leftarrow solve(A), solve(B).\\
&solve(A) \leftarrow clause(A,B), check(B), solve(B).\\
&\\
&check((A,B)) \leftarrow check(A), check(B).\\
&check(A) \leftarrow clause(A,_).\\
&check(true).
\end{aligned}
$$

This meta-interpreter is sound and complete, i.e. preserves computed answers. However, it does not preserve termination. Indeed, let P be the following program:

$$
\begin{aligned}
&p \leftarrow q, r.\\
&q \leftarrow q.
\end{aligned}
$$

and let p be the query. Then, p with respect to P does not terminate, while $solve(p)$ with respect to $M_3 \cup ce(P)$ terminates (finitely fails). Thus, this meta-interpreter does not preserve LD-termination. Observe that unfolding may only improve

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

termination (Bossi and Cocco 1994). Thus, this meta-interpreter is non-violating LD-termination. □

Thus, the claim that the "vanilla" meta-interpreter preserves the calls set should be proven separately.

Lemma 1
Let P be an interpreted program, M_0 be the "vanilla" meta-interpreter and $Q \in B_P^E$, then:

- for every call $A \in Call(P, Q)$, there exists $solve(A') \in Call(M_0 \cup ce(P), solve(Q))$ such that A and A' are variants;
- for every call $solve(A) \in Call(M_0 \cup ce(P), solve(Q))$, such that $A \in B_P^E$, there exists $A' \in Call(P, Q)$, such that A and A' are variants.

Proof
The proof can be found in Serebrenik and De Schreye (2003). □

This lemma extends Theorem 9 (Pedreschi and Ruggieri 1997) by claiming not only that every call of the meta-program and of the meta-query "mimics" the original execution, but also that every call of the original program and query is "mimicked" by the meta-program.

Now we can complete the analysis of the "vanilla" meta-interpreter, namely, prove that it does not improve termination. The main idea is to construct a quasi-ordering relation $\succeq$ for atoms of the interpreted program based on the quasi-ordering $\geqslant$ such that the meta-program is order-acceptable via the quasi-ordering relation $\geqslant$. To complete the proof, we have to show that the interpreted program is order-acceptable via $\succeq$.

Theorem 2
Let P be a definite program, S a set of atomic queries, and M_0 the "vanilla" meta-interpreter, such that $M_0 \cup ce(P)$ is LD-terminating for all queries in $\{solve(Q) \mid Q \in S\}$. Then, P is LD-terminating for all queries in S.

Proof
By Theorem 1 $M_0 \cup ce(P)$ is order-acceptable with respect to a set $\{solve(Q) \mid Q \in S\}$. We are going to prove order-acceptability of P with respect to S. By Theorem 1 termination will be implied.

Since $M_0 \cup ce(P)$ is order-acceptable with respect to $\{solve(Q) \mid Q \in S\}$ there exists a well-founded quasi-ordering $\geqslant$, satisfying requirements of Definition 2. Let $\succeq$ be a new quasi-ordering on B_P^E defined as $A \succ B$ if $solve(A) > solve(B)$ and $A \preceq \succeq A$ for all A.

The ordering is defined on $\{A \mid A \in B_P^E \wedge \exists Q \in S$ such that $solve(A) \in Call(M_0 \cup ce(P), solve(Q))\}$. By Lemma 1 this set coincides with $Call(P, S)$. The ordering $\succ$ is well-defined and well-founded. These properties follow immediately from the corresponding properties of $>$.

Next, we prove that P is order-acceptable with respect to S via $\succeq$. Let $Q \in S$, $A \in Call(P, Q)$ and let $A' \leftarrow B_1, \ldots, B_n$ be a clause in P, such that $mgu(A, A') = \theta$

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

exists. Let B_i be such that $rel(B_i) \simeq rel(A)$ and let σ be a computed answer substitution for $(B_1, \ldots, B_{i-1})\theta$. We have to show that $A \succ B_i\theta\sigma$.

By Lemma 1 $solve(A) \in Call(M_0 \cup ce(P), solve(Q))$. The only clause that can be used in the resolution with it is $solve(Head) \leftarrow clause(Head, Body), solve(Body)$. Observe that $mgu(solve(A), solve(Head))$ affects neither A nor $Body$. Let τ be a computed answer substitution that unifies A and A', $Body$ and $(B_1, \ldots, B_n)$. By the choice of τ, τ maps $Body$ to $(B_1, \ldots, B_n)\theta$. Then, $solve(Body)\tau = solve((B_1, \ldots, B_n))\theta$ and $solve(A) > solve((B_1, \ldots, B_n)\theta)$ by the order-acceptability of $M_0 \cup ce(P)$ via $\geqslant$.

If $n = 1$ then $solve(A) > solve(B_1\theta)$. By definition of $\succ$, $A \succ B_1\theta$, completing the proof.

Assume that $n > 1$. In this case, $solve((B_1, \ldots, B_n)\theta)$ is another call in the call set. The clause $solve((Atom, Atoms)) \leftarrow solve(Atom), solve(Atoms)$ is the only one that can be used at the resolution step. Let δ be the most general unifier of $solve((B_1, \ldots, B_n)\theta)$ with the head of the clause above. The substitution δ does not affect the variables appearing in $(B1, \ldots, B_n)\theta$, since $Atom$ and $Atoms$ are variables. Thus, $B_j\theta\delta = B_j\theta$ for all j and δ is omitted in the lines to come.

Order-acceptability of the meta-program implies that $solve((B_1, \ldots, B_n)\theta) > solve(B_1\theta)$ and for any computed answer substitution σ_1 for $solve(B_1\theta)$, $solve((B_1, \ldots, B_n)\theta) > solve((B_2, \ldots, B_n)\theta\sigma_1)$. Proceeding in the same way, we obtain

$$solve((B_2, \ldots, B_n)\theta\sigma_1) > solve((B_3, \ldots, B_n)\theta\sigma_1\sigma_2)$$
$$\vdots$$
$$solve((B_{i-1}, \ldots, B_n)\theta\sigma_1 \ldots \sigma_{i-2}) > solve((B_i, \ldots, B_n)\theta\sigma_1 \ldots \sigma_{i-1}),$$

where σ_j is a computed answer substitution for $solve(B_j\theta\sigma_1 \ldots \sigma_{j-1})$.

Moreover, $solve((B_i, \ldots, B_n)\theta\sigma_1 \ldots \sigma_{i-1}) > solve(B_i\theta\sigma_1 \ldots \sigma_{i-1})$. Transitivity of the $>$-ordering implies $solve(A) > solve(B_i\theta\sigma_1 \ldots \sigma_{i-1})$. Computed answer substitutions are preserved by M_0 Levi and Ramundo (1993). Thus, for all j, σ_j is also a computed answer substitution for $B_j\theta\sigma_1 \ldots \sigma_{j-1}$. Therefore, σ_j's can be chosen such that $\sigma = \sigma_1 \ldots \sigma_{i-1}$. In other words, $solve(A) > solve(B_i\theta\sigma)$. Thus, $A \succ B_i\theta\sigma$ by definition of $\succ$, completing the proof. □

The other direction of the theorem has been proved by Pedreschi and Ruggieri (1997). It allows us to state the following corollary.

Corollary 1
The "vanilla" meta-interpreter M_0 preserves LD-termination.

The proof of Theorem 2 presented above suggests the following methodology for proving that a particular meta-interpreter does not improve LD-termination. First, define an ordering on the set of calls to the meta-interpreter, such that the ordering reflects its behaviour. Then, establish the relationship between a new ordering and the one that reflects order-acceptability with respect to a set of the interpreted program. Prove, using this relationship, that the newly defined ordering is well-defined, well-founded and reflects order-acceptability of the meta-program with respect to a corresponding set of calls. In order for the proofs to be correct, one may need to assume (or to prove as a prerequisite) that the meta-interpreter is

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

sound and that the set of calls of the interpreted program and of the meta-program correspond to each other. The opposite direction, i.e. that the meta-interpreter does not violate termination, can be proved using a similar methodology. Therefore, in the following section we will define an ordering for advanced meta-interpreters based on the existing ordering for M_0 and vice versa.

We illustrate the methodology proposed by considering M_0 and Example 2. Recall that we have seen in section 1 that level-mapping based approaches experience serious difficulties with analysing termination of this example.

Example 9
Let P be the program discussed in Example 2. Acceptability (and thus, termination) of $\{solve(l(t)) \mid t \text{ is a ground term}\}$ with respect to the corresponding meta-program can be established via the ordering $>$ that satisfies for all ground terms t_1 and t_2:

$$
\begin{aligned}
&solve(l(t_1)) > solve((p(t_1), r(t_1)))\\
&solve((p(t_1), r(t_1))) > solve(p(t_1))\\
&solve((p(t_1), r(t_1))) > solve(r(t_1))\\
&solve(p(t_1)) > solve((q(t_1, t_2), p(t_2)))\\
&solve((q(t_1, t_2), p(t_2))) > solve(q(t_1, t_2))\\
&solve((q(t_1, t_2), p(t_2))) > solve(p(t_2))\\
&solve(r(f(t_1))) > solve((s(t_2), r(t_1)))\\
&solve((s(t_1), r(t_2))) > solve(s(t_1))\\
&solve((s(t_1), r(t_2))) > solve(r(t_2))
\end{aligned}
$$

These inequalities imply:

$$solve(p(t_1)) > solve(p(t_2)) \text{ for terms such that } solve(q(t_1, t_2)) \text{ holds} \quad (8)$$

$$solve(r(f(t_1))) > solve(r(t_1)) \quad (9)$$

Recall the construction we applied in the proof of Theorem 2. A new ordering has been proposed for the atoms of B_P^E: $A \succ B$ if $solve(A) > solve(B)$. Hence, (8) and (9) lead, in our case, to the following definition of $\succ$: $p(t_1) \succ p(t_2)$, for terms such that $q(t_1, t_2)$ holds, and $r(f(t_1)) \succ r(t_1)$. One can easily see that $\succ$ is indeed well-founded and that P is order-acceptable with respect to $\{l(t) \mid t \text{ is a ground term}\}$ via this ordering. □

5 Double extended meta-interpreters

Typical applications of meta-interpreters require the production of some additional output or the performance of some additional tasks during the execution, such as constructing proof trees (essential for debugging and explanation applications) or cutting "unlikely" branches (required for uncertainty reasoners with cutoff). As we are going to see in the examples to come, these extensions can and usually will influence termination properties of the meta-interpreter.

In this section, we still consider definite meta-interpreters, but their clauses, which still follow the general outline of M_0, are enriched with extra subgoals,

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

providing additional functionality. This class of meta-interpreters expands the class of extended meta-interpreters studied by Martens and De Schreye (1995b). It includes many useful meta-interpreters, such as a meta-interpreter which constructs proof trees (Sterling and Shapiro 1994) and can be used as a basis for explanation facilities in expert system, as well as meta-interpreters which allow reasoning about theories and provability (Brogi *et al.* 1990; Martens and De Schreye 1995a) or meta-interpreters which implement reasoning with uncertainty (Sterling and Shapiro 1994). Moreover, this class also describes a depth tracking tracer for Prolog, a reasoner with threshold cutoff (Sterling and Shapiro 1994) and a pure four port box execution model tracer (Bowles and Wilk 1989). The methodology presented so far is expanded to analyse double extended meta-interpreters, and conditions implying termination non-violation and non-improvement are established.

Definition 10
A definite program of the following form

$$
\begin{aligned}
&solve(true, t_{11}, \ldots, t_{1n}) \leftarrow C_{11}, \ldots, C_{1m_1}.\\
&solve((A, B), t_{21}, \ldots, t_{2n}) \leftarrow\\
&\qquad D_{11}, \ldots, D_{1k_1}, solve(A, t_{31}, \ldots, t_{3n}),\\
&\qquad D_{21}, \ldots, D_{2k_2}, solve(B, t_{41}, \ldots, t_{4n})\\
&\qquad C_{21}, \ldots, C_{2m_2}.\\
&solve(A, t_{51}, \ldots, t_{5n}) \leftarrow\\
&\qquad D_{31}, \ldots, D_{3k_3}, clause(A, B, s_1, \ldots, s_k),\\
&\qquad D_{41}, \ldots, D_{4k_4}, solve(B, t_{61}, \ldots, t_{6n})\\
&\qquad C_{31}, \ldots, C_{3m_3}.
\end{aligned}
$$

together with clauses defining any other predicate occurring in the C_{kl} and D_{pq} (none of which contain *solve* or *clause*) is called a *double extended meta-interpreter*.

The name of this class of the meta-interpreters stems from the fact that they further generalise the class of extended meta-interpreters (Martens and De Schreye 1995a). Extended meta-interpreters are double extended meta-interpreters, such that for all p and q, D_{pq} is *true*. Note that despite the similarity between the definition and Example 8, Example 8 is not a double extended meta-interpreter due to the call to predicate *clause* in the definition of *check*. Thus, the results established in this section are not applicable to it.

Example 10
The following program (Bowles and Wilk 1989) shows the pure Prolog tracer for the four port box execution model of Byrd (1980) (in the original paper *interp* was used instead of *solve* to denote the meta-predicate; we renamed the predicate for the sake of uniformity).

$$
\begin{aligned}
&solve(true).\\
&solve((G1, G2)) \leftarrow solve(G1), solve(G2).\\
&solve(G) \leftarrow before(G), clause(G, B), solve(B), after(G). \qquad (10)
\end{aligned}
$$

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

$$before(G) \leftarrow write('call'), write(G), nl. \quad (11)$$
$$before(G) \leftarrow write('fail'), write(G), nl, fail. \quad (12)$$

$$after(G) \leftarrow write('succeed'), write(G), nl.$$
$$after(G) \leftarrow write('redo'), write(G), nl, fail.$$

This program is a double extended meta-interpreter, since it clearly has the form prescribed by Definition 10 and neither *before* nor *after* is depending on *solve* or *clause*. □

The next example of a double extended meta-interpreter is motivated by program 17.8 of Sterling and Shapiro (1994). Intuitively, a *proof tree* is a convenient way to represent the proof. The root of the proof tree for an atomic query is the query itself. If a clause $H \leftarrow B_1, \ldots, B_n$ has been used to resolve an atomic query A via an mgu θ, there is a directed edge (represented by $\leftarrow$ in the example to come) from a node representing $A\theta$ to a node corresponding to a query $(B_1, \ldots, B_n)\theta$. The proof tree for a conjunctive query is a collection of proof trees for the individual conjuncts.

Example 11
The following meta-interpreter constructs a proof tree, while solving a query. Proof trees are often used both for debugging (Naish 1997) and explanation (Hammond 1984; Arora *et al.* 1993) purposes.

$$solve(true, true).$$
$$solve((A, B), (ProofA, ProofB)) \leftarrow solve(A, ProofA), solve(B, ProofB).$$
$$solve(A, (A \leftarrow Proof)) \leftarrow clause(A, B), solve(B, Proof).$$

This meta-interpreter is a double extended meta-interpreter (for all k, l, p and q, $C_{kl} = true$ and $D_{pq} = true$). □

The study of double extended meta-interpreters might require different clause encoding than the encoding considered in the previous section. For example, reasoners with uncertainty might require that a *certainty factor* is integrated in the encoding. Therefore, we adjust the definition of the clause encoding as follows:

Definition 11
(cf. Definition 3) Let D be a double extended meta-interpreter, let P be a program. The *clause-encoding* $ce^D(P)$ is a collection of facts of a new predicate *clause*, such that for every clause $H \leftarrow B$ in P there exists a unique atom $clause(H, B, s_1, \ldots, s_k) \in ce^D(P)$ and for every atom $clause(H, B, s_1, \ldots, s_k) \in ce^D(P)$ there exists a clause $H \leftarrow B$ in P.

In the remainder of the section, we discuss non-violation of LD-termination and non-improving of LD-termination.

5.1 Non-violating LD-termination

We start with a discussion of non-violation of LD-termination. Before proving this formally, we reconsider Example 11 and apply to it the designed methodology. The

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

treatment is done on the intuitive level. A more formal discussion is postponed until Example 14.

Example 12
Example 11, continued. To prove that the meta-interpreter does not violate LD-termination, we have to show that for any definite program P, and for any query Q terminating with respect to P, $solve(Q,u)$ terminates with respect to $D \cup ce^D(P)$ for any term u, where D is the double extended meta-interpreter that constructs proof-trees.

Given that Q terminates with respect to P, $solve(Q)$ terminates with respect to $M_0 \cup ce(P)$ and there exists a quasi-ordering $\geqslant$ such that $M_0 \cup ce(P)$ is order-acceptable with respect to $solve(Q)$ via $\geqslant$. Let $\succeq$ be a quasi-ordering on $B^E_{D \cup ce^D(P)}$ defined as follows for any atom a and any terms s, t, u_1 and u_2:

- $solve(s,u_1) \succ solve(t,u_2)$ if $solve(s) > solve(t)$
- $solve(s,u_1) \succ a$ if $rel(a) \neq solve$
- $a \preceq\succeq a$

Next, we have to show that $D \cup ce^D(P)$ is order-acceptable with respect to $solve(Q,u)$ via $\succeq$. This claim is intuitively clear since $\succeq$ is defined to ignore the extra argument of *solve*, and if this argument is dropped from the clauses of D, M_0 is obtained. □

To formalise the intuition presented in Example 12 we need to prove a number of auxiliary statements. First of all, observe that double extended meta-interpreters are sound.

Lemma 2
Let D be a double extended meta-interpreter. Then, D is sound.

Proof
To prove the lemma, we use the *s*-semantics approach presented in Bossi *et al.* (1994). The semantics is recalled and the formal proof of the lemma is presented in Serebrenik and De Schreye (2003). □

As the following example demonstrates, unlike the "vanilla" meta-interpreter M_0, double extended meta-interpreters do not necessarily preserve the set of calls. However, one can show that there is a certain correspondence between the calls obtained with respect to a double extended meta-interpreter D and the "vanilla" meta-interpreter M_0. Recall that Lemma 1 established that M_0 preserves the calls set.

Example 13
Let D be the following double extended meta-interpreter and let P be the program of Example 4.

$$solve(true).$$
$$solve((A,B)) \leftarrow solve(A), solve(B).$$
$$solve(A) \leftarrow clause(A,B), B = q(f(Z)), solve(B).$$

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

The set of calls of $D \cup ce^D(P)$ and $solve(p(X))$ is

$$\{solve(p(X)), solve(q(f(Z)), clause(p(X), q(X))\}.$$

The set of calls for $M_0 \cup ce(P)$ and $solve(p(X))$ is

$$\{solve(p(X)), solve(q(X)), solve(true), clause(p(X), q(X)), clause(q(b), true)\}.$$

There is no call in $Call(M_0 \cup ce(P), solve(p(X)))$ that is a *variant* of $solve(q(f(Z))$. However, there is a call $solve(q(X))$ in $Call(M_0 \cup ce(P), solve(p(X)))$ such that $solve(q(f(Z))$ is its instance. □

In Example 13 a correspondence was established between the sets of calls obtained with respect to D and those obtained with respect to M_0. Lemma 3 proves that such a correspondence can always be established for double extended meta-interpreters. Observe that, in general, not every sound meta-interpreter has this property (Example 8).

Lemma 3
Let D be a double extended meta-interpreter. Let P be an interpreted program, Q_0 be an interpreted query and let $u_1, \ldots, u_n$ be a sequence of terms. Then, for every call $solve(Q, t_1, \ldots, t_n)$ in $Call(D \cup ce^D(P), solve(Q_0, u_1, \ldots, u_n))$ there exists a call $solve(G)$ in $Call(M_0 \cup ce(P), Q_0)$ such that Q is an instance of G.

Proof
The proof is similar to the proof of Lemma 1 and can be found in Serebrenik and De Schreye (2003). □

Now we are ready to prove that double extended meta-interpreters do not violate termination. Before presenting the result formally, we need to introduce the following auxiliary definition.

Definition 12
Let P be a definite program and let Q be an atomic query, such that P is order-acceptable with respect to $\{Q\}$ via a quasi-ordering $\geqslant$. The quasi-ordering $\geqslant$ is called *minimal* if there exists no quasi-ordering $\geqslant_1$ such that P is order-acceptable with respect to $\{Q\}$ via $\geqslant_1$ and $\geqslant_1 \subset \geqslant$.

First, we need to show the existence of a minimal ordering.

Lemma 4
Let P be a definite program and let Q be an atomic query. If P is order-acceptable with respect to $\{Q\}$, there exists a minimal quasi-ordering $\geqslant$ such that P is order-acceptable with respect to $\{Q\}$ via $\geqslant$.

Proof
Let $\mathcal{O}$ be the set of all quasi-orderings such that P is order-acceptable with respect to $\{Q\}$ via them. Since P is order-acceptable with respect to $\{Q\}$, $\mathcal{O}$ is not empty. Hence we define a new quasi-ordering on $Call(P, Q)$ as following:

$$A \preceq\succeq B \quad \text{if} \quad A \text{ is identical to } B$$
$$A \succ B \quad \text{if} \quad A > B \text{ for all } \geqslant \in \mathcal{O}$$

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

It is straightforward to check that $\succeq$ is again a quasi-ordering. We are going to show that P is order-acceptable with respect to $\{Q\}$ via $\succeq$, i.e. that $\succeq \in \mathcal{O}$. Indeed, let A be in $Call(P, Q)$, $A' \leftarrow B_1, \ldots, B_n$ be a clause such that $mgu(A, A') = \theta$ exists, B_i a body subgoal such that $rel(A) = rel(B_i)$ and σ be a computed answer substitution for $\leftarrow (B_1, \ldots, B_{i-1})\theta$. Then, order-acceptability of P with respect to $\{Q\}$ via the quasi-orderings in $\mathcal{O}$ implies that for any $\geqslant \in \mathcal{O}$, $A > B_i\theta\sigma$. By definition of $\succ$, it holds that $A \succ B_i\theta\sigma$. Hence, P is order-acceptable with respect to $\{Q\}$ via $\succeq$.

The construction of $\succeq$ above also implies immediately that this quasi-ordering is minimal. □

Intuitively, minimal quasi-orderings contain decreases that are essential for proving order-acceptability, and only these decreases. To prove the statement formally we need the following notions introduced by Verschaetse (1992).

Definition 13

- Let $Q_0, Q_1, Q_2, \ldots, \theta_1, \theta_2, \ldots$ be a derivation with selected atoms $A_0, A_1, A_2, \ldots$ and applied renamed clauses $H^i \leftarrow B^i_1, \ldots, B^i_{n_i}$ $(i = 1, 2, \ldots)$. We say that A_k is a *direct descendant* of A_i, if $k > i$ and A_k is the atom $B^{i+1}_j\theta_{i+1} \ldots \theta_k$, $(1 \leqslant j \leqslant n_{i+1})$.
- Let $Q_0, Q_1, Q_2, \ldots, \theta_1, \theta_2, \ldots$ be a derivation with selected atoms $A_0, A_1, A_2, \ldots$. A *subsequence* of derivation steps, $Q_{i(0)}, Q_{i(1)}, \ldots, \theta_{i(0)+1}, \ldots$ with selected atoms $A_{i(0)}, A_{i(1)}, A_{i(2)}, \ldots$ is *directed*, if for each k $(k > 1)$, $A_{i(k)}$ is a direct descendant of $A_{i(k-1)}$ in the given derivation. A derivation $Q_0, Q_1, Q_2, \ldots, \theta_1, \theta_2, \ldots$ is *directed* if it is its own directed subsequence.

Verschaetse (1992) also proved the following lemma:

Lemma 5
Let P be a definite program and A be an atomic query. If P and A have an infinite derivation, then they have an infinite directed derivation.

Now we are ready to prove the statement of Lemma 6 formally.

Lemma 6
Let P be a program and Q be a query. Let $\geqslant$ be a minimal quasi-ordering such that P is order-acceptable with respect to $\{Q\}$ via it. Then, for all A and B in $Call(P, Q)$, if $A > B$ then there exists a directed derivation $Q_0, \ldots, Q_n$ with selected atoms $A_0, \ldots, A_n$, such that $A_0 = A$, $A_n = B$ and for all $0 \leqslant i < n$, $A_i > A_{i+1}$.

Proof
For the sake of contradiction assume that there exist A and B such that $A > B$ and no directed derivation exists as required. We are going define a new quasi-ordering $\geq$ on $Call(P, Q)$ that will contradict the minimality of $\geqslant$.

Let $K \succ M$ be defined as a transitive closure of the following relation: "$K > M$ and M is a direct descendant of K". Let $K \preceq\succeq M$ if and only if K is identical to M.

By definition of the notion of a direct descendant (Definition 13) and order-acceptability of P with respect to $\{Q\}$ via $\geqslant$, it follows that P is order-acceptable with respect to $\{Q\}$ via $\succeq$. Moreover, it is clear that $\succeq \subseteq \geqslant$. To complete the proof we show that $A \not\succ B$.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

For the sake of contradiction assume that $A \succ B$. Then, since $\succ$ is defined as a transitive closure, there exists a sequence of atoms $A_0, \ldots, A_n$ such that $A_i > A_{i+1}$ and A_{i+1} is a direct descendant of A_i for all $0 \leqslant i < n$. Since $A_i \in Call(P, Q)$, A_i is a selected atom of some query Q_i. Thus, we have found a directed derivation $Q_0, \ldots, Q_n$ as described by the lemma for A and B. Therefore, our assumption was wrong and $A \not\succ B$. Hence, $\succeq \subset \geqslant$, that contradicts the minimality of $\geqslant$. □

Finally, we can prove the main result of this section, namely, that double extended meta-interpreters do not violate termination (under certain extra conditions). Since the proof is long and technical only the general out-line is presented. Technical details can be found in Serebrenik and De Schreye (2003).

Theorem 3
Let P be an interpreted program, D a double extended meta-interpreter, and also let $Q \in B^E_{D \cup ce^D(P)}$, such that Q is terminating with respect to P. Let $u_1, \ldots, u_n$ be a sequence of terms such that $\{A \mid A \in Call(D \cup ce^D(P), solve(Q, u_1, \ldots, u_n)), solve \neq rel(A)\}$ is terminating with respect to D. Then $solve(Q, u_1, \ldots, u_n)$ terminates with respect to $D \cup ce^D(P)$.

Proof (sketch)
Let M_0 be the "vanilla" meta-interpreter. One can show that $M_0 \cup ce(P)$ is order-acceptable with respect to $S = \{A\eta \mid A \in Call(M_0 \cup ce(P), solve(Q)), \eta$ is a substitution$\}$. Let $\geqslant_1$ be a minimal well-founded quasi-ordering, such that $M_0 \cup ce(P)$ is order-acceptable with respect to S via it. Similarly, let $\geqslant_2$ be a well-founded quasi-ordering such that D is order-acceptable with respect to $\{A \mid A \in Call(D \cup ce^D(P), solve(Q, u_1, \ldots, u_n)), solve \neq rel(A)\}$ via $\geqslant_2$.

We have to show that there exists a well-founded quasi-ordering $\succeq$ such that $D \cup ce^D(P)$ is order-acceptable with respect to $\{solve(Q, u_1, \ldots, u_n)\}$ via $\succeq$. By Theorem 1 this will imply termination.

Let $\succeq$ be defined on $B^E_{D \cup ce^D(P)}$ as follows for any terms $t_1, t_2, t_1^1, \ldots, t_n^1, t_1^2, \ldots, t_n^2$ and any atoms a_1, a_2:

1. $solve(t_1, t_1^1, \ldots, t_n^1) \succ solve(t_2, t_1^2, \ldots, t_n^2)$, if there is a term t, such that $solve(t_1) >_1 solve(t)$ and $t_2 = t\theta$ for some substitution θ;
2. $a_1 \succ a_2$, if $rel(a_1) \neq solve$, $rel(a_2) \neq solve$ and $a_1 >_2 a_2$;
3. $solve(t_1, t_1^1, \ldots, t_n^1) \succ a_1$, if $rel(a_1) \neq solve$;
4. $a_1 \preceq\succeq a_2$, if a_1 and a_2 are identical.

First we show that $\succ$ is an ordering and that this ordering is well-founded. To this end we make use of the minimality of $>_1$ and of the Lifting Theorem (Apt 1997: Theorem 3.22). Next we prove that $D \cup ce^D(P)$ is order-acceptable with respect to $solve(Q, u_1, \ldots, u_n)$. Let $A_0 \in Call(D \cup ce^D(P), solve(Q, u_1, \ldots, u_n))$. If $rel(A_0) \neq solve$ the desired decrease follows from the order-acceptability of D with respect to $\{A \mid A \in Call(D \cup ce^D(P), solve(Q, u_1, \ldots, u_n)), solve \neq rel(A)\}$ via $>_2$. If $rel(A_0) = solve$ each clause defining $solve$ has to be considered separately. In each one of the cases we show that if all arguments but the first one are dropped from A_0 the resulting atom belongs to S and hence order-acceptability via $\geqslant_1$ can be

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

used. Next we make use of Lemma 3 and complete the proof by reasoning on the substitutions involved. □

This theorem provides an important theoretical result, namely, that double extended meta-interpreters do not violate termination if the interpreter terminates with respect to a set of calls to predicates different from *solve* and generated by the meta-program and the meta-query.

Example 14
Meta-interpreters presented in Examples 10, 11 and 13 are double extended meta-interpreters. Termination of the calls to predicates different from *solve* is immediate because in all the examples these predicates can be completely unfolded, i.e. they do not depend (directly or indirectly) on recursive predicates. Thus, by Theorem 3 meta-interpreters of Examples 10, 11 and 13 do not violate termination.

The quasi-ordering derived by Theorem 3 for Example 11 has been discussed in Example 12. □

In general, to ensure that given a program P a double extended meta-interpreter D terminates with respect to $\{A \mid A \in Call(D \cup ce^D(P), solve(Q, u_1, \ldots, u_n)), solve \neq rel(A)\}$ one can require D to terminate for all calls to predicates different from *solve*. To verify the latter condition one can use the notion of order-acceptability (Theorem 1).

5.2 Non-improving LD-termination

In this section we are going to study the opposite direction of the implication, namely, given that a meta-program $D \cup ce^D(P)$ terminates with respect to $solve(Q, u_1, \ldots, u_n)$ we would like to prove that

- the meta-interpreter D terminates with respect to $\{A \mid A \in Call(D \cup ce^D(P), solve(Q, u_1, \ldots, u_n)), solve \neq rel(A)\}$
- the interpreted program P terminates with respect to the interpreted query Q

To prove the first statement, we first observe that $\{A \mid A \in Call(D \cup ce^D(P), solve(Q, u_1, \ldots, u_n)), solve \neq rel(A)\}$ is a subset of $Call(D \cup ce^D(P), solve(Q, u_1, \ldots, u_n))$. Since $D \cup ce^D(P)$ terminates with respect to the latter set, it terminates with respect to the former set as well. This implies that any subset of $D \cup ce^D(P)$ terminates with respect to the same set of queries.

For general double extended meta-interpreter the second statement we would like to prove, i.e. *"if the meta-program terminates then the interpreted program terminates as well"* does not necessarily hold. Indeed, one can find many double extended meta-interpreters that are designed to improve termination. However, we are interested in termination non-improvement and would like to establish conditions that ensure it. Given a non-terminating interpreted program, termination of a meta-program can result from one of the following problems:

- failure while unifying a call and a clause head.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

Example 15
Indeed, consider the following meta-interpreter D:

$$solve(true, a).$$
$$solve((A, B), a) \leftarrow solve(A, a), solve(B, a).$$
$$solve(A, a) \leftarrow clause(A, B), solve(B, a).$$

Let P be an interpreted program, such that Q does not terminate with respect to it. However, $solve(Q, b)$ terminates with respect to $D \cup ce^D(P)$. □

To eliminate this problem we require that, for every call, the unification success or failure with the head of the clause depends only on their first arguments. In general, predicting success of the unification during the execution is known to be an undecidable problem. However, sufficient conditions ensuring unification success can be proposed.

- failure of intermediate body subgoals.

Example 16
Indeed, consider the following meta-interpreter D:

$$solve(true).$$
$$solve((A, B)) \leftarrow solve(A), solve(B).$$
$$solve(A) \leftarrow fail, clause(A, B), solve(B).$$

Let P be an interpreted program, such that Q does not terminate with respect to it. However, $solve(Q)$ terminates with respect to $D \cup ce^D(P)$. □

To solve this problem, one has to guarantee non-failure of the intermediate body subgoals. The general problem of non-failure analysis is well-known to be undecidable (Debray *et al.* 1997). Fortunately, the problem is decidable for a restricted class of problems (Debray *et al.* 1997). For the specific meta-interpreters we consider, failure of the corresponding intermediate body subgoals turns out to be decidable.
It should also be noted that failure of the body atoms to the right of the last recursive call may influence termination as well.

Example 17

$$solve(true) \leftarrow fail.$$
$$solve((A, B)) \leftarrow solve(A), solve(B).$$
$$solve(A) \leftarrow clause(A, B), solve(B).$$

Let P be the following interpreted program:

$$r \leftarrow p, r.$$
$$p.$$

Clearly, the query r does not terminate with respect to P. However, $solve(r)$ terminates with respect to $D \cup ce^D(P)$. □

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

Therefore, non-failure is required also for these subgoals.

- different call sets. In principle, even if no failure occurs during the execution, D can change the call set, as the following example illustrates:

Example 18

$$solve(true).$$
$$solve((A, B)) \leftarrow A = p(0), solve(A), solve(B).$$
$$solve(A) \leftarrow clause(A, B), solve(B).$$

Let P be the following program:

$$q \leftarrow p(X), r.$$
$$p(f(X)) \leftarrow p(X).$$
$$p(0).$$
$$r.$$

The termination behaviour of q with respect to P differs from the termination behaviour of $solve(q)$ with respect to $D \cup ce(P)$. The former computation does not terminate, while the latter terminates. The reason for this is that D changes the set of calls, due to the unification $A = p(0)$ in the body of the second clause. Indeed, the call set of P with respect to $\{q\}$ contains $p(f(X))$, while the call set of $D \cup ce(P)$ with respect to $solve(q)$ does not contain the corresponding atom $solve(p(f(X)))$. □

Similar problem occurs in Example 13. To solve this problem one has to ensure that meta-variables are not affected by the intermediate body atoms.

We summarise the discussion above in the following definition.

Definition 14

Let D be the following double extended meta-interpreter:

$$solve(true, t_{11}, \ldots, t_{1n}) \leftarrow C_{11}, \ldots, C_{1m_1}.$$
$$solve((A, B), t_{21}, \ldots, t_{2n}) \leftarrow$$
$$\quad D_{11}, \ldots, D_{1k_1}, solve(A, t_{31}, \ldots, t_{3n}),$$
$$\quad D_{21}, \ldots, D_{2k_2}, solve(B, t_{41}, \ldots, t_{4n}),$$
$$\quad C_{21}, \ldots, C_{2m_2}.$$
$$solve(A, t_{51}, \ldots, t_{5n}) \leftarrow$$
$$\quad D_{31}, \ldots, D_{3k_3}, clause(A, B, s_1, \ldots, s_k),$$
$$\quad D_{41}, \ldots, D_{4k_4}, solve(B, t_{61}, \ldots, t_{6n}),$$
$$\quad C_{31}, \ldots, C_{3m_3}.$$

(together with the clauses defining C_{kl} and D_{pq}), such that

- for any computed answer of the preceding atoms,
  - — either the corresponding instances of $(t_{31}, \ldots, t_{3n})$, $(t_{41}, \ldots, t_{4n})$, $(t_{61}, \ldots, t_{6n})$, and $(s_1, \ldots, s_k)$ are linear sequences of free variables,

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

— or $(t_{11},\ldots,t_{1n})$, $(t_{21},\ldots,t_{2n})$, $(t_{51},\ldots,t_{5n})$, are linear sequences of free variables and for every program P and for every $clause(s,t,t_1,\ldots,t_k) \in ce^D(P)$, $t_1,\ldots,t_k$ is a linear sequence of free variables;

- $D_{11},\ldots,D_{1k_1}, D_{21},\ldots,D_{2k_2}, D_{31},\ldots,D_{3k_3}, D_{41},\ldots,D_{4k_4}, C_{11},\ldots,C_{1m_1}, C_{21},\ldots,C_{2m_2}$, and $C_{31},\ldots,C_{3m_3}$ do not fail for the corresponding calls (independently on the values of the meta-variables), i.e. for all P, for all Q and for all $t_1,\ldots,t_n$, calls to C_{kl} and D_{pq} in $Call(D \cup ce^D(P), solve(Q,t_1,\ldots,t_n)$ do not fail;
- for any instance $(D_{i1},\ldots,D_{ik_i})\theta$ of $D_{i1},\ldots,D_{ik_i}$ and for any computed answer substitution σ for $(D_{i1},\ldots,D_{ik_i})\theta$, $A\theta\sigma$ is identical to $A\theta$ and $B\theta\sigma$ is identical to $B\theta$.

Then D is called *restricted*.

It should be noted that Definition 14 is not syntactical. However, syntactical conditions implying it can be proposed. We postpone discussing them until after we formulate the termination non-violation theorem (Theorem 4).

We also introduce a notion of a *restricted query*, corresponding to Definition 14.

Definition 15
Given a restricted double extended meta-interpreter D and a query $solve(Q,v_1,\ldots,v_n)$, the query is called *restricted* if:

- either $(v_1,\ldots,v_n) \in Vars_n$,
- or $(t_{11},\ldots,t_{1n}),(t_{21},\ldots,t_{2n}),(t_{51},\ldots,t_{5n}) \in Vars_n$.

Example 19
Recall the meta-interpreter that constructs proof-trees we considered in Example 11. Note that this meta-interpreter is restricted. Observe that t_{11} is *true*, i.e. $t_{11} \notin Vars_n$. Thus, for a query $solve(Q,u)$ to be restricted, u should be in $Vars_1$, i.e. u should be a free variable. □

First, we are going to see that the condition imposed on the arguments of the head (or recursive body subgoals) ensures the requirement stated after Example 15.

Lemma 7
Let D be a restricted double extended meta-interpreter, P be an interpreted program, Q_0 be an interpreted query and $(v_1,\ldots,v_n)$ a sequence of terms such that $solve(Q_0,v_1,\ldots,v_n)$ is restricted. Then, for any call $solve(Q,u_1,\ldots,u_n) \in Call(D \cup ce^D(P), solve(Q_0,v_1,\ldots,v_n))$ and for any $solve(H,t_1,\ldots,t_n) \leftarrow B_1,\ldots,B_k$ in D, $u_1,\ldots,u_n$ is unifiable with $t_1,\ldots,t_n$.

Proof
We distinguish between two cases.

If $(t_{11},\ldots,t_{1n}) \notin Vars_n$, then, $(v_1,\ldots,v_n) \in Vars_n$. Hence, the unification step succeeds and if θ is the mgu, then $(t_{11},\ldots,t_{1n})\theta = (t_{11},\ldots,t_{1n})$. The first condition in Definition 14 implies that for any computed answer of the preceding atoms, the corresponding instances of $(t_{31},\ldots,t_{3n})$, $(t_{41},\ldots,t_{4n})$, $(t_{61},\ldots,t_{6n})$, and $(s_1,\ldots,s_k)$ are linear sequences of free variables. Thus, for any call $solve(Q,u_1,\ldots,u_n) \in Call(D \cup ce^D(P), solve(Q_0,v_1,\ldots,v_n))$, $u_1,\ldots,u_n$ is a linear sequence of free variables. Therefore,

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

for any sequence of terms $t_1,\ldots,t_n$, the sequence $u_1,\ldots,u_n$ can be unified with $t_1,\ldots,t_n$.

If $(t_{11},\ldots,t_{1n}) \in Vars_n$, then the second subcase of the first condition in Definition 14 holds, i.e., $(t_{21},\ldots,t_{2n}),(t_{51},\ldots,t_{5n}) \in Vars_n$ as well. Recall, that $solve(H, t_1,\ldots,t_n) \leftarrow B_1,\ldots,B_k$ a clause in D. Thus, $t_1,\ldots,t_n$ is one of the $(t_{11},\ldots,t_{1n}),(t_{21},\ldots,t_{2n}),(t_{51},\ldots,t_{5n})$. In other words, $(t_1,\ldots,t_n)$ is a linear sequence of free variables. Therefore, for any sequence of atoms $(u_1,\ldots,u_n)$, $(t_1,\ldots,t_n)$ is unifiable with it. □

The rest of the section is dedicated to the proof that restricted double extended meta-interpreters do not improve termination. To provide some intuition how the actual proof will proceed we recall Example 11 and show that it does not improve termination. The treatment is done on the intuitive level. More precise discussion of this example can be found in Example 22.

Example 20

Example 11, continued. In order to prove that this meta-interpreter does not improve LD-termination, we have to show that for any definite program P, and for any query Q if a restricted query $solve(Q,u)$ terminates with respect to $D \cup ce^D(P)$, then Q terminates with respect to P. Observe that the requirement that $solve(Q,u)$ should be restricted means that u should be a free variable (Example 19).

Given that $solve(Q,u)$ terminates with respect to $D \cup ce^D(P)$ there exists a quasi-ordering $\geqslant$ such that $D \cup ce^D(P)$ is order-acceptable with respect to $solve(Q,u)$ via $\geqslant$. Let $\succeq$ be a quasi-ordering on $B^E_{M_0 \cup ce(P)}$ defined as follows: for any atom a, it holds that $a \preceq\succeq a$, and for any terms s,t,u_1 and u_2, if $solve(s,u_1) > solve(t,u_2)$ then $solve(s) \succ solve(t)$. Next, one has to show that $\succeq$ is a well-founded quasi-ordering and that $M_0 \cup ce(P)$ is order-acceptable with respect to $solve(Q)$. Since D is restricted, termination cannot be enforced by the information contained in the second argument of *solve*. Hence, both well-foundedness and order-acceptability follow from the corresponding properties of $\geqslant$. □

To prove that restricted double extended meta-interpreters do not improve termination, we would like to show the completeness result for this class of meta-interpreters. However, as the following example illustrates, completeness does not necessarily hold.

Example 21

Consider the following restricted double extended meta-interpreter.

$$solve(true, X).$$
$$solve((A,B),X) \leftarrow solve(A,Y), solve(B,Z).$$
$$solve(A,X) \leftarrow p, clause(A,B), solve(B,Y).$$
$$p \leftarrow p.$$

This meta-interpreter is not complete due to its non-termination. □

Therefore, we prove a more restricted result that will be sufficient for showing termination non-improvement.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

Lemma 8
Let D be a restricted double extended meta-interpreter, P be an interpreted program, Q be an interpreted query and $(v_1,\ldots,v_n)$ be terms, such that $solve(Q,v_1,\ldots,v_n)$ is restricted, and $D \cup ce^D(P)$ terminates with respect to $\{A \mid A \in Call(D \cup ce^D(P), solve(Q,v_1,\ldots,v_n)), solve \neq rel(A)\}$. Then, for every computed answer t for Q obtained with respect to P, there exists a computed answer $solve(t',t_1,\ldots,t_n)$ for $solve(Q,v_1,\ldots,v_n)$ with respect to $D \cup ce^D(P)$, such that t is a variant of t'.

Proof
The proof, analogous to Lemma 2, can be found in Serebrenik and De Schreye (2003). □

Lemma 8 implies that there is a one-to-one correspondence between the calls to *solve* obtained with respect to D and those obtained with respect to M_0. More formally, there is a call $solve(Q)$ in $Call(M_0 \cup ce(P), solve(Q_0))$ if and only if, there is a call $solve(Q',u_1,\ldots,u_n)$ in the set $Call(D \cup ce^D(P), solve(Q_0,v_1,\ldots,v_n))$, such that Q and Q' are variants. Indeed, if there are no intermediate calls to *solve*, the claim follows from Definition 14. Otherwise, it follows from the preceding Lemma 8.

Now we are ready to prove that restricted meta-interpreters preserve termination. Similarly to Theorem 3 only a proof sketch is included, technical details can be found in Serebrenik and De Schreye (2003).

Theorem 4
Let D be a restricted double extended meta-interpreter. Let P be an interpreted program and let Q be an interpreted query, such that $D \cup ce^D(P)$ LD-terminates for $solve(Q,v_1,\ldots,v_n)$, where $(v_1,\ldots,v_n)$ are terms such that $solve(Q,v_1,\ldots,v_n)$ is restricted. Then, P LD-terminates with respect to Q.

Proof (sketch)
In order to show that P LD-terminates for Q it is sufficient to prove that $M_0 \cup ce(P)$ LD-terminates with respect to $solve(Q)$. Then, by Theorem 2 P LD-terminates with respect to Q. Thus, we aim to establish order-acceptability of $M_0 \cup ce(P)$ with respect to $solve(Q)$.

First of all, we define a relationship on $B^E_{M_0 \cup ce(P)}$. Since $D \cup ce^D(P)$ LD-terminates for $solve(Q,v_1,\ldots,v_n)$, $D \cup ce^D(P)$ is order-acceptable with respect to $solve(Q,v_1,\ldots,v_n)$ via a quasi-ordering. Let a minimal quasi-ordering such that $D \cup ce^D(P)$ is order-acceptable with respect to $solve(Q,v_1,\ldots,v_n)$ via it, be denoted $\geqslant$. Then, we define $solve(s) \succ solve(t)$ if there exist $solve(s,s_1,\ldots,s_n), solve(t,t_1,\ldots,t_n) \in Call(D \cup ce^D(P), solve(Q,v_1,\ldots,v_n))$ such that $solve(s,s_1,\ldots,s_n) > solve(t,t_1,\ldots,t_n)$ and $solve(s) \preceq\succeq solve(t)$ if $solve(s)$ and $solve(t)$ are identical.

One can show that $\succ$ is indeed a well-founded ordering. To show that $M_0 \cup ce(P)$ is order-acceptable with respect to $solve(Q)$ via $\succeq$ we make use of Lemmata 7 and 8, and of Definition 14. □

Example 22
Example 10, continued. The meta-interpreter is restricted and Theorem 4 ensures that it preserves termination. However, if the clause (11) would have been removed,

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

the meta-interpreter would no longer be restricted. Indeed, the call to *before* would fail, violating the second requirement of Definition 14. Thus, Theorem 4 would not have been applicable. It should be noted that indeed, this meta-interpreter improves termination. □

The following example illustrates that the fact that D is restricted is *sufficient but not necessary* for LD-termination of a restricted query $solve(Q, v_1, \ldots, v_n)$ with respect to $D \cup ce^D(P)$ to imply LD-termination of Q with respect to P.

Example 23
Let D be the following meta-interpreter.

$$solve(true).$$
$$solve((A, B)) \leftarrow fail, solve(A), solve(B).$$
$$solve(A) \leftarrow loop, clause(A, B), solve(B).$$
$$loop \leftarrow loop.$$

This meta-interpreter is double extended, but it is not restricted since one of the intermediate body atoms fails. Observe that D does not improve termination. Indeed, for any program P and for any query Q the restricted query $solve(Q)$ does not terminate with respect to $D \cup ce^D(P)$. Thus, the following implication is trivially *true*: "if $solve(Q)$ LD-terminates with respect to $D \cup ce^D(P)$ then Q LD-terminates with respect to P". □

To conclude this section we discuss syntactical conditions that can be used to ensure that a double extended meta-interpreter is restricted. The first condition of Definition 14 requires certain sequences of arguments to be linear sequences of variables for any computed answer of the preceding atoms. Recall that we made a distinction between two possibilities:

- either the corresponding instances of $(t_{31}, \ldots, t_{3n})$, $(t_{41}, \ldots, t_{4n})$, $(t_{61}, \ldots, t_{6n})$, and $(s_1, \ldots, s_k)$ are linear sequences of free variables. To ensure this we require that $(t_{31}, \ldots, t_{3n})$, $(t_{41}, \ldots, t_{4n})$, $(t_{61}, \ldots, t_{6n})$, and $(s_1, \ldots, s_k)$ are linear sequences of free variables *and* that none of these variables appear in the preceding subgoals.
- or $(t_{11}, \ldots, t_{1n})$, $(t_{21}, \ldots, t_{2n})$, $(t_{51}, \ldots, t_{5n})$, are linear sequences of free variables and for every program P and for every $clause(s, t, t_1, \ldots, t_k) \in ce^D(P)$, $t_1, \ldots, t_k$ is a linear sequence of free variables. Observe that the sequences of arguments appear in heads of the clauses in this case. Thus, there are *no preceding atoms*. In other words, if the condition holds on the syntactic level, then it trivially holds for any computed answer of the preceding atoms.

To ensure the remaining conditions we require every atom a among C_{kl} and D_{pq} to satisfy one of the following (A and B denote the meta-variables of the clauses):

1. a is *true*
2. a is $u = f(u_1, \ldots, u_n)$ and u is a fresh variable and $A, B \notin Var(f(u_1, \ldots, u_n))$;
3. a is a call to a built-in predicate p and p is either *write* or *nl*;
4. a is $p(u_1, \ldots, u_n)$ for a user-defined predicate p, p cannot fail.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

The latter condition can be safely approximated by compiler by means of determinism analysis (cf. Henderson *et al.* 1996).

These requirements may seem to be very restrictive. However, they are satisfied by the majority of the meta-interpreters considered including a meta-interpreter that constructs proof trees presented in Example 11, as well as a reasoner with uncertainty (Sterling and Shapiro 1994), and meta-interpreters allowing to reason about theories and provability (Brogi *et al.* 1990; Martens and De Schreye 1995a). However, as the following example illustrates, not every restricted meta-interpreter satisfies these conditions.

Example 24
Consider the following variant of the meta-interpreter that constructs proof trees (cf. 11).

$$solve(true, true).$$
$$solve((A, B), (ProofA, ProofB)) \leftarrow solve(A, ProofA), solve(B, ProofB).$$
$$solve(A, (A \leftarrow Proof)) \leftarrow clause(A, B), foo(Proof), solve(B, Proof).$$
$$foo(_).$$

This meta-interpreter is a restricted double extended meta-interpreter. However, the syntactic conditions specified above do not hold, since *Proof* appears in the preceding subgoal of *foo*. □

6 Extending the language of the interpreted programs

So far we have considered only definite programs. However, in order to make our approach practical, the language of the underlying interpreted programs should be extended to include negation, frequently appearing in applications of the meta-interpreters.

As earlier, to prove that meta-interpreters with negation preserve termination, we use among others a termination analysis framework based on order-acceptability. By using this result and applying the same methodology as above one can prove that the following meta-interpreter M_4, being an immediate extension of the "vanilla" meta-interpreter to normal programs (Hill and Gallagher 1998), preserves LDNF-termination. By LDNF-termination we understand finiteness of the LDNF-forest. Soundness and completeness of M_4 are proved in Hill and Lloyd (1989: Theorem 2.3.3).

$$solve(true).$$
$$solve((Atom, Atoms)) \leftarrow solve(Atom), solve(Atoms).$$
$$solve(\neg Atom) \leftarrow \neg solve(Atom).$$
$$solve(Head) \leftarrow clause(Head, Body), solve(Body).$$

Theorem 5
Let P be a normal program, S be a set of queries. Then P LDNF-terminates with respect to S if and only if $M_4 \cup ce(P)$ LDNF-terminates with respect to $\{solve(Q) \mid Q \in S\}$.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

Proof
Mimicking the proof of Theorem 2 and the result of Pedreschi and Ruggieri (1997). First, for each one of the LDNF-trees, calls and semantics are preserved by the corresponding results for definite programs. Second, given the definition of an LDNF-forest, if $\neg A$ is discovered, the new tree with A as a root is started. Observe that $\neg solve(A)$ is ground if and only if A is ground. Thus, the derivation obtained with respect to $M_4 \cup ce(P)$ flounders if and only if the derivation obtained with respect to P does. □

Theorem 5 allows us to consider termination of different kinds of meta-interpreters, namely, those using the *ground* representation of interpreted programs (Bowen and Kowalski 1982; Hill and Lloyd 1994; Hill and Gallagher 1998). This idea can be traced back to Gödel, who suggested a one-to-one mapping, called *Gödel numbering*, of expressions in a first order language to natural numbers (Gödel 1931). The idea of numbering is a key idea of ground representation. Intuitively, we number predicates $p(0), p(1), p(2), \ldots$, then functors $f(0), f(1), f(2), \ldots$, constants $c(0), c(1), c(2), \ldots$ and, finally, variables used in the program $v(0), v(1), v(2), \ldots$ Each one of the sets is finite since the program itself is finite. Atoms are represented as $atom(P, L)$, where P is the encoding of the predicate and L is the list of the encodings of the arguments. Terms are represented in a similar way as $term(F, L)$, where F is the encoding of the main functor and L is the list of the encodings of the arguments. To represent a clause, we use *and*, denoting a conjunction, and *if*, standing for an implication.

For example, instead of representing a clause

$$permute(L, [El|T]) \leftarrow delete(El, L, L1), permute(L1, T),$$

as $clause(permute(L, [El|T]), (delete(El, L, L1), permute(L1, T)))$ as we used to do, it is represented as

$$if(atom(p(0), [v(0), term(f(0), [v(1), term(f(0), [v(2), c(0)])])]),$$
$$and(atom(p(1), [v(1), v(0), v(3)]), atom(p(0), [v(3), v(2)]))),$$

where the following correspondence holds:

predicates	$p(0)$	*permute*
	$p(1)$	*delete*
functors	$f(0)$	./2 also known as *cons*
constants	$c(0)$	[]
variables	$v(0)$	L
	$v(1)$	El
	$v(2)$	T
	$v(3)$	$L1$.

Meta-interpreters using the ground representation can be considered "more pure" than other meta-interpreters we studied, as the meta language and the language of the interpreted program are strictly separated. However, a number of primitive operations, such as unification, provided for the non-ground case by the underlying Prolog system, have to be defined explicitly. Moreover, while the preceding meta-interpreters can be recognised as such by looking for the built-in predicate *clause*,

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

recognising a meta-interpreter based on the ground representation can not be done easily, unless some extra information, such as type declarations, is provided explicitly.

The following meta-interpreter *idemo*, inspired by Kowalski (1990) has been borrowed from Hill and Gallagher (1998). Given the ground representation of a normal program and the ground representation of a query the meta-interpreter returns ground representations of the computed answers corresponding to the query.

Example 25

Given the ground representation of a program and the ground representation of a query, predicate *idemo* proceeds in two steps. First it computes the non-ground version of a query by calling predicate *instance_of*, i.e. replaces $v(i)$'s with variables, while recording bindings. Secondly, it calls an alternative version of the meta-interpreter (*idemo1*) to resolve the non-ground version of a query with the ground representation of a given program. Observe how a non-ground instance of clause is computed in the last clause of *idemo1*.

```
idemo(P, X, Y) ←
        instance_of(X, Y),
        idemo1(P, Y).

idemo1(_, true).
idemo1(P, and(X, Y)) ←
        idemo1(P, X),
        idemo1(P, Y).
idemo1(P, not(X)) ←
        ¬idemo1(P, X).
idemo1(P, atom(Q, Xs)) ←
        member(Z, P),
        instance_of(Z,
            if(atom(Q, Xs), B)),
        idemo1(P, B).

inst_term(v(N), X, [], [bind(N, X)]).
inst_term(v(N), X, [bind(N, X)|S],
    [bind(N, X)|S]).
inst_term(v(N), X, [bind(M, Y)|S],
    [bind(M, Y)|S1]) ←
        N ≠ M,
        inst_term(v(N), X, S, S1).
inst_term(term(F, Xs),
    term(F, Ys), S, S1) ←
        inst_args(Xs, Ys, S, S1).

instance_of(X, Y) ←
        inst_formula(X, Y, [], _).

inst_formula(atom(Q, Xs),
    atom(Q, Ys), S, S1) ←
        inst_args(Xs, Ys, S, S1).
inst_formula(and(X, Y),
    and(Z, W), S, S2) ←
        inst_formula(X, Z, S, S1),
        inst_formula(Y, W, S1, S2).
inst_formula(if(X, Y),
    if(Z, W), S, S2) ←
        inst_formula(X, Z, S, S1),
        inst_formula(Y, W, S1, S2).
inst_formula(not(X), not(Z), S, S1) ←
        inst_formula(X, Z, S, S1).
inst_formula(true, true, S, S).

inst_args([], [], S, S).
inst_args([X|Xs], [Y|Ys], S, S2) ←
        inst_term(X, Y, S, S1),
        inst_args(Xs, Ys, S1, S2).
```

We are interested in proving termination of *idemo*(p, q, v), where p is the ground representation of a program, q is a ground representation of a query and v is a free variable that will be bound to the ground representations of computed answers corresponding to the query.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

Existing termination techniques, such as Dershowitz *et al.* (2001), are powerful enough to prove termination of *idemo1* calls to *instance_of*(t, v), where t is a term, being a ground representation of a term, atom or clause, and v is a variable that will be bounded to the non-ground representation of the same object. However, they are not powerful enough to analyse correctly this example, both due to imprecise representation of all possible ground terms (in particular all possible ground representations of programs, by the same abstraction) and due to the nature of *idemo1* as a meta-interpreter.

It should be noted that the "troublesome" part of this example is a definition of *idemo1*. However, *idemo1* is very similar to the meta-interpreter M_4 discussed. The only differences are that the clauses of the interpreted program are stored in the first argument and that a *non*-ground instance of a a clause has to be computed before resolving a query with it. Despite of these differences a theorem analogous to Theorem 5 holds. Hence, termination of the meta-program is equivalent to termination of the interpreted program. □

7 Conclusion

We have presented a methodology for proving termination properties of meta-programs. It is well-known that termination verification plays a crucial role in meta-programming (Pfenning and Schuermann 1998). Our main contribution is in providing a technique linking termination behaviour of an interpreted program with a termination behaviour of the meta-program. We have shown that for a wide variety of meta-interpreters, a relatively simple relation can be defined between the ordering that satisfies the requirements of order-acceptability for an interpreted program and the ordering that satisfies the requirements of order-acceptability for the meta-interpreter extended by this interpreted program and a corresponding set of queries. This category of meta-interpreters includes many important ones, such as extended meta-interpreters studied by Martens and De Schreye (1995b), meta-interpreter, that constructs proof trees (Sterling and Shapiro 1994), reasoners about theories and provability (Brogi *et al.* 1990; Martens and De Schreye 1995a), and reasoners with uncertainty (Sterling and Shapiro 1994). Moreover, it also describes a depth tracking tracer for Prolog, a reasoner with threshold cutoff (Sterling and Shapiro 1994), a pure four port box execution model tracer (Bowles and Wilk 1989) and the *idemo* meta-interpreter of Hill and Gallagher (1998). The relationship established between the orderings allows termination proofs to be reused, i.e. a termination proof obtained for an interpreted program can be used for showing termination of the meta-program and vice versa. Example 2 demonstrated such a simple relation cannot be established if linear level mappings were considered instead of general orderings.

Ease of meta-programming is often considered to be one of the advantages of logic programming. From the early days meta-interpreters were developed to implement different control strategies for Prolog (Gallaire and Lasserre 1982; Beckstein *et al.* 1996). Furthermore, meta-programming finds a wide variety of applications in such areas as artificial intelligence, compilation, constraints solving, debugging, and

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

program analysis (Codish and Taboch 1999; Hill and Gallagher 1998; Lamma *et al.* 1996; Martens and De Schreye 1995b; Sterling and Shapiro 1994). Meta-interpreters have also been successfully applied to aspect-oriented programming (De Volder and D'Hondt 1999; Brichau *et al.* 2002). Recently, Sheard presented a number of challenges in meta-programming (Sheard 2001).

Despite the intensive research on meta-programming inside the logic programming community (Apt and Ben-Eliyahu 1996; Apt and Turini 1995; Levi and Ramundo 1993; Martens and De Schreye 1995b), termination behaviour of meta-programs has attracted relatively little attention, with Pedreschi and Ruggieri being the only known notable exception. In their work (Pedreschi and Ruggieri 1997), a generic verification method is used, based on specifying preconditions and postconditions. Unfortunately, their termination results are restricted only to the "vanilla" meta-interpreter M_0. It is not immediately obvious how their results can be extended to alternative meta-interpreters, nor if a relationship between termination characterisation of the interpreted program and the meta-program can be established.

Researchers working on modular termination aim to discover how level mappings required to prove termination of separate modules can be combined to obtain a termination proof for the entire program (Apt and Pedreschi 1994; Bossi *et al.* 2002; Pedreschi and Ruggieri 1996; Verbaeten *et al.* 2001). Since meta-program can be viewed as a union of a meta-interpreter and of the clause-encoding of an interpreted program, these results might seem applicable. However, clause-encoding represents a program as a set of facts. Therefore, for any program P, termination of $clause(H, B)$ with respect to $ce(P)$ is trivial and any level-mapping is sufficient to show termination. Hence, no useful information on termination of P is provided by the level-mapping and termination of the meta-program cannot be established.

Our methodology gains its power from the use of the integrated approach presented in De Schreye and Serebrenik (2002), which extends the traditional notion of acceptability by adding a wide class of orderings that have been studied in the context of the term-rewriting systems. Theoretically, this approach has exactly the same power as the classical level mappings based results, but in practice, quite often a simple ordering is sufficient to prove termination in an example that would otherwise require the application of a complex level mapping. Meta-programs provide typical examples of this kind.

The study of termination preservation for general meta-interpreters is an extremely difficult task. We do not believe that termination preservation conditions can be formulated without assuming any additional information on the meta-interpreter or on the interpreted programs. Therefore, we have identified a number of important classes of meta-interpreters and proposed conditions implying termination preservation for each one of the classes. Some authors (Bruynooghe *et al.* 1992; Cheng *et al.* 1989; van Harmelen 1989) have studied a meta-interpreter that uses a list of goals instead of a traditional conjunction of goals. Study of termination preservation properties of this meta-interpreter is considered as a future work.

The paper by Pedreschi and Ruggieri (1997) is, to the best of our knowledge, the only one to study additional verification properties of the meta-interpreters such as absence of errors and partial correctness. Their results hint at further research directions in the context of verification of meta-interpreters.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

References

Apt, K. R. 1997. *From Logic Programming to Prolog*. Prentice-Hall International Series in Computer Science. Prentice Hall.

Apt, K. R. and Ben-Eliyahu, R. 1996. Meta-variables in logic programming, or in praise of ambivalent syntax. *Fundamenta Informaticae 28,* 1–2, 23–36.

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

Apt, K. R. and Turini, F., Eds. 1995. *Meta-Logics and Logic Programming*. Logic Programming. The MIT Press, Cambridge, MA, USA.

Arora, T., Ramakrishnan, R., Roth, W. G., Seshadri, P. and Srivastava, D. 1993. Explaining program execution in deductive systems. In *Deductive and Object-Oriented Databases, Third International Conference Proceedings, DOOD'93*, S. Ceri, K. Tanaka and S. Tsur, Eds. Lecture Notes in Computer Science, vol. 760. Springer Verlag, 101–119.

Arts, T. 1997. Automatically proving termination and innermost normalisation of term rewriting systems. PhD thesis, Universiteit Utrecht.

Beckstein, C., Stolle, R. and Tobermann, G. 1996. Meta-programming for generalized Horn clause logic. In *Proceedings of the 5th International Workshop on Metaprogramming and Metareasoning in Logic (META-96)*, Bonn, Germany, 27–42.

Bossi, A. and Cocco, N. 1994. Preserving universal temination through unfold/fold. In *Algebraic and Logic Programming*, G. Levi and M. Rodríguez-Artalejo, Eds. Lecture Notes in Computer Science, vol. 850. Springer Verlag, 269–286.

Bossi, A., Cocco, N., Etalle, S. and Rossi, S. 2002. On modular termination proofs of general logic programs. *Theory and Practice of Logic Programming 2,* 3, 263–291.

Bossi, A., Gabbrielli, M., Levi, G. and Martelli, M. 1994. The *s*-semantics approach: theory and applications. *Journal of Logic Programming 19/20*, 149–197.

Bowen, K. A. and Kowalski, R. A. 1982. Amalgamating language and meta-language in logic programming. In *Logic Programming*, K. Clark and S.-Å. Tärnlund, Eds. Academic Press, 153–172.

Bowles, A. and Wilk, P. 1989. Tracing requirements for multi-layered meta-programming. In *Meta-Programming in Logic Programming*, H. Abramson and M. H. Rogers, Eds. MIT Press, Cambridge, MA, USA, 205–216.

Brichau, J., Mens, K., and De Volder, K. 2002. Building composable aspect-specific languages with logic metaprogramming. In *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference*, D. S. Batory, C. Consel, and W. Taha, Eds. Lecture Notes in Computer Science, vol. 2487. Springer Verlag, 110–127.

Brogi, A., Mancarella, P., Pedreschi, D. and Turini, F. 1990. Composition operators for logic theories. In *Proceedings Esprit Symp. on Comp. Logic*, J. W. Lloyd, Ed. Springer Verlag, 117–134.

Bruynooghe, M., Codish, M., Genaim, S. and Vanhoof, W. 2002. Reuse of results in termination analysis of typed logic programs. In *Static Analysis, 9th International Symposium*, M. V. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer Verlag, 477–492.

Bruynooghe, M., De Schreye, D. and Martens, B. 1992. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing 11,* 1, 47–79.

Bueno, F., García de la Banda, M. J. and Hermenegildo, M. V. 1994. Effectiveness of global analysis in strict independence-based automatic parallelization. In *Logic Programming, Proceedings of the 1994 International Symposium*, M. Bruynooghe, Ed. MIT Press, 320–336.

Byrd, L. 1980. Understanding the control flow of Prolog programs. In *Proceedings of the Logic Programming Workshop*, S.-Å. Tärnlund, Ed. 127–138.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

Cheng, M. H., van Emden, M. H. and Strooper, P. A. 1989. Complete sets of frontiers in logic-based program transformation. In *Meta-Programming in Logic Programming*, H. Abramson and M. H. Rogers, Eds. The MIT Press, Cambridge, MA, USA, 283–297.

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

De Schreye, D. and Serebrenik, A. 2002. Acceptability with general orderings. In *Computational Logic. Logic Programming and Beyond. Essays in Honour of Robert A. Kowalski, Part I*, A. C. Kakas and F. Sadri, Eds. LNCS, vol. 2407. Springer Verlag, 187–210.

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 *Proceedings of the International Conference on Fifth Generation Computer Systems*., I. Staff, Ed. IOS Press, 481–488.

De Volder, K. and D'Hondt, T. 1999. Aspect-orientated logic meta programming. In *Meta-Level Architectures and Reflection, Second International Conference*, P. Cointe, Ed. Lecture Notes in Computer Science, vol. 1616. Springer Verlag, 250–272.

Debray, S. K., López-García, P. and Hermenegildo, M. V. 1997. Non-failure analysis for logic programs. In *Proceedings of the Fourteenth International Conference on Logic Programming*, L. Naish, Ed. MIT Press, 63–77.

Decorte, S. and De Schreye, D. 1998. Termination analysis: some practical properties of the norm and level mapping space. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, J. Jaffar, Ed. MIT Press, 235–249.

Decorte, S., De Schreye, D. and Vandecasteele, H. 1999. Constraint-based termination analysis of logic programs. *ACM TOPLAS 21,* 6 (November), 1137–1195.

Dershowitz, N., Lindenstrauss, N., Sagiv, Y. and Serebrenik, A. 2001. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing 12,* 1–2, 117–156.

Dershowitz, N. and Manna, Z. 1979. Proving termination with multiset orderings. *Communications of the ACM (CACM) 22,* 8 (August), 465–476.

Gallaire, H. and Lasserre, C. 1982. Metalevel control for logic programs. In *Logic Programming*, K. L. Clark and S.-Å. Tärnlund, Eds. Academic Press, 153–172.

Genaim, S., Codish, M., Gallagher, J. and Lagoon, V. 2002. Combining norms to prove termination. In *Third International Workshop on Verification, Model Checking and Abstract Interpretation*, A. Cortesi, Ed. Lecture Notes in Computer Science, vol. 2294. Springer Verlag, 126–138.

Gödel, K. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik 38*, 173–198.

Hammond, P. 1984. micro-PROLOG for Expert Systems. In *micro-PROLOG: Programming in Logic*, K. L. Clark and F. G. McCabe, Eds. Prentice-Hall International Series in Computer Science. Prentice-Hall, 294–319.

Henderson, F., Somogyi, Z. and Conway, T. 1996. Determinism analysis in the Mercury compiler. In *Proceedings of the Australian Computer Science Conference*. 337–346.

Hill, P. and Gallagher, J. P. 1998. Meta-programming in logic programming. In *Handbook of logic in Artificial Intelligence and Logic Programming*, D. M. Gabbay, C. Hogger and J. Robinson, Eds. Clarendon press, 421–498. volume 5. Logic Programming.

Hill, P. and Lloyd, J. 1994. *The Gödel Programming Language*. The MIT Press, Cambridge, MA, USA.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

Hill, P. and Lloyd, J. W. 1989. Analysis of meta-programs. In *Meta-Programming in Logic Programming*, H. Abramson and M. H. Rogers, Eds. The MIT Press, Cambridge, MA, USA, 23–52.

Janssens, 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 (July), 205–258.

Janssens, G., Bruynooghe, M. and Englebert, V. 1994. Abstracting numerical values in CLP(H, N). In *Programming Language Implementation and Logic Programming, 6th International Symposiumsium, PLILP'94*, M. V. Hermenegildo and J. Penjam, Eds. Lecture Notes in Computer Science, vol. 844. Springer Verlag, 400–414.

Kowalski, R. A. 1979. *Logic for Problem Solving*. North-Holland. Available at `http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/rak.html`.

Kowalski, R. A. 1990. Problems and promises of computational logic. In *Computational Logic*, J. W. Lloyd, Ed. Springer Verlag, 1–36.

Lamma, E., Milano, M. and Mello, P. 1996. Combining solvers in a meta constraint logic programming architecture. In *Frontiers of Combining Systems*, F. Baader and K. U. Schulz, Eds. Applied Logic Series, vol. 3. Kluwer Academic, 267–283.

Lee, C. S. 2002. Program termination analysis in polynomial time. In *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002*, D. Batory, C. Consel, and W. Taha, Eds. Lecture Notes in Computer Science, vol. 2487. ACM, Springer Verlag, 218–235.

Levi, G. and Ramundo, D. 1993. A formalization of metaprogramming for real. In *Logic Programming, Proceedings of the 10th International Conferencerence on Logic Programming*, D. S. Warren, Ed. MIT Press, 354–373.

Martens, B. and De Schreye, D. 1995a. Two semantics for definite meta-programs, using the non-ground representation. In *Meta-Logics and Logic Programming*, K. R. Apt and F. Turini, Eds. The MIT Press, Cambridge, MA, USA, 57–81.

Martens, B. and De Schreye, D. 1995b. Why untyped nonground metaprogramming is not (much of) a problem. *Journal of Logic Programming 22,* 1 (January), 47–99.

Mesnard, F. and Ruggieri, S. 2003. On proving left termination of constraint logic programs. *ACM Transaction on Computational Logic 4,* 2, 207–259.

Naish, L. 1997. A three-valued declarative debugging scheme. In *Workshop on Logic Programming Environments*, A. J. Kusalik, M. Ducassé, and U. Neumerkel, Eds. 1–12.

Pedreschi, D. and Ruggieri, S. 1996. Modular verification of logic programs. Tech. Rep. TR-96-24, Dipartimento di Informatica, Università di Pisa.

Pedreschi, D. and Ruggieri, S. 1997. Verification of meta-interpreters. *Journal of Logic and Computation 7,* 2 (November), 267–303.

Pfenning, F. and Schuermann, C. 1998. Twelf user's guide. Tech. Rep. CMU-CS-98-173, Department of Computer Science, Carnegie Mellon University. November.

Serebrenik, A. and De Schreye, D. 2003. On termination of meta-programs. Tech. Rep. cs.PL/0110035, The Computing Research Repository (CoRR). Available at `http://arxiv.org/abs/cs.PL/0110035`.

Sheard, T. 2001. Accomplishments and research challenges in meta-programming. In *Semantics, Applications, and Implementation of Program Generation, Second International Workshop, SAIG 2001, Florence, Italy, September 6, 2001, Proceedings*, W. Taha, Ed. Lecture Notes in Computer Science, vol. 2196. Springer Verlag, 2–44.

Sterling, L. and Shapiro, E. 1994. *The Art of Prolog*. The MIT Press, Cambridge, MA, USA.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press

van Harmelen, F. 1989. A classification of meta-level architectures. In *Meta-Programming in Logic Programming*, H. Abramson and M. H. Rogers, Eds. The MIT Press, Cambridge, MA, USA, 103–122.

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

Verschaetse, K. 1992. Static termination analysis for definite horn clause programs. PhD thesis, Department of Computer Science, K. U. Leuven, Leuven, Belgium.

https://doi.org/10.1017/S1471068404002248 Published online by Cambridge University Press