TLP **2** (3): 369–418, May 2002. © 2002 Cambridge University Press
DOI: 10.1017/S1471068402001400 Printed in the United Kingdom

Classes of terminating logic programs

DINO PEDRESCHI, SALVATORE RUGGIERI
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy,
(*e-mail:* {pedre,ruggieri}@di.unipi.it)

JAN–GEORG SMAUS*
Institut für Informatik, Universität Freiburg,
Georges-Köhler-Allee 52, 79110 Freiburg im Breisgau, Germany
(*e-mail:* smaus@informatik.uni-freiburg.de)

Abstract

Termination of logic programs depends critically on the selection rule, i.e. the rule that determines which atom is selected in each resolution step. In this article, we classify programs (and queries) according to the selection rules for which they terminate. This is a survey and unified view on different approaches in the literature. For each class, we present a sufficient, for most classes even necessary, criterion for determining that a program is in that class. We study six classes: a program *strongly* terminates if it terminates for *all* selection rules; a program *input terminates* if it terminates for selection rules which only select atoms that are sufficiently instantiated in their input positions, so that these arguments do not get instantiated any further by the unification; a program *local delay* terminates if it terminates for local selection rules which only select atoms that are bounded w.r.t. an appropriate level mapping; a program *left-terminates* if it terminates for the usual left-to-right selection rule; a program ∃-*terminates* if there exists a selection rule for which it terminates; finally, a program has *bounded nondeterminism* if it only has finitely many refutations. We propose a semantics-preserving transformation from programs with bounded nondeterminism into strongly terminating programs. Moreover, by unifying different formalisms and making appropriate assumptions, we are able to establish a formal hierarchy between the different classes.

KEYWORDS: universal termination, logic program, selection rule, norm, level mapping, dynamic scheduling, left-termination, control

1 Introduction

The paradigm of logic programming originates from the discovery that a fragment of first order logic can be given an elegant computational interpretation. Kowalski (1979) advocates the separation of the *logic* and *control* aspects of a logic program and has coined the famous formula

Algorithm = Logic + Control.

* Supported by the ERCIM fellowship programme.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

The programmer should be responsible for the logic part, and hence a logic program should be a (first order logic) specification. The control should be taken care of by the logic programming system. One aspect of control in logic programs is the *selection rule*. This is a rule stating which atom in a query is selected in each derivation step. It is well-known that soundness and completeness of SLD-resolution is independent of the selection rule (Apt, 1997). However, a stronger property is usually required for a selection rule to be useful in programming, namely termination.

Definition 1.1
A *terminating control* for a program P and a query Q is a selection rule s such that every SLD-derivation of P and Q via s is finite.

In reality, logic programming is far from the ideal that the logic and control aspects are separated. Without the programmer being aware of the control and writing programs accordingly, logic programs would usually be hopelessly inefficient or even non-terminating.

The usual selection rule of early systems is the *LD* selection rule: in each derivation step, the leftmost atom in a query is selected for resolution. This selection rule is based on the assumption that programs are written in such a way that the data flow within a query or clause body is from left to right. Under this assumption, this selection rule is usually a terminating control. For most applications, this selection rule is appropriate in that it allows for an efficient implementation.

Second generation logic languages adopt more flexible control primitives, which allow for addressing logic and control separately. Program clauses have their usual logical reading. In addition, programs are augmented by *delay declarations* or *annotations* that specify restrictions on the admissible selection rules. These languages include NU-Prolog (Thom and Zobel, 1988), Gödel (Hill and Lloyd, 1994) and Mercury (Somogyi *et al*., 1996).

In this survey, we classify programs and queries according to the selection rules under which they terminate, hence investigating the influence of the selection rule on termination. As most approaches to the termination problem, we are interested in *universal* termination of logic programs and queries, that is, showing that *all* derivations for a program and query (via a certain selection rule) are finite. This is in contrast to *existential* termination (Baudinet, 1992; De Schreye and Decorte, 1994; Marchiori, 1996b). Also, we consider *definite* logic programs, as opposed to logic programs that also contain negated literals in clause bodies.

Figure 1 gives an overview of the classes we consider. Arrows drawn with solid lines stand for set inclusion ('$\rightarrow$ corresponds to $\subset$'). The numbers in the figure correspond to propositions in section 9.

A program P and query Q *strongly terminate* if they terminate for *all* selection rules. This class of programs has been studied mainly by Bezem (1993). Naturally, this class is the smallest we consider. A program P and query Q *left-terminate* if they terminate for the *LD* selection rule. The vast majority of the literature is concerned with this class; see De Schreye and Decorte (1994) for an overview. A program P and query Q $\exists$-*terminate* if there *exists* a selection rule for which they terminate. This notion of termination has been introduced by Ruggieri (2001; 1999).

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

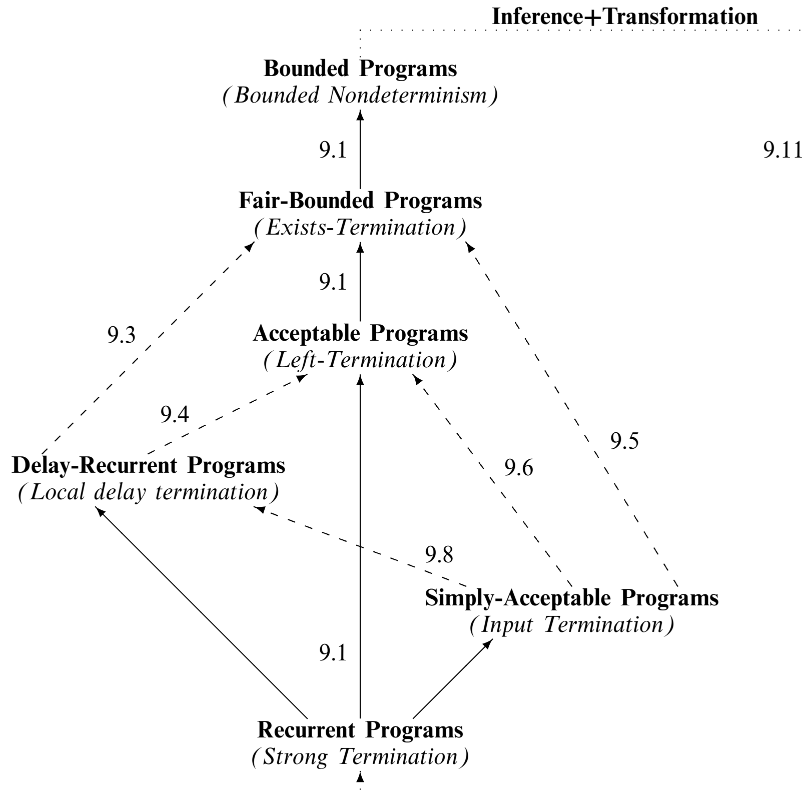

Fig. 1. An overview of the classes.

Surprisingly, this is still not the largest class we consider. Namely, there is the class of programs for which there are only finitely many *successful* derivations (although there could also be infinite derivations). We say that these programs have *bounded nondeterminism*, a notion studied by Pedreschi and Ruggieri (1999a). Such programs can be transformed into equivalent programs which strongly terminate, as indicated in the figure and stated in Theorem 9.11.

To explain the two remaining classes shown in the figure, and their relationship with left-terminating programs, we have to introduce the concept of *modes*. A mode is a labelling of each argument position of a predicate as either input or output. It indicates the intended data flow in a query or clause body.

An *input-consuming* derivation is a derivation where an atom can be selected only when its input arguments are instantiated to a sufficient degree, so that unification with the head of the clause does not instantiate them further. A program and a query *input terminate* if all input-consuming derivations for this program and query are finite. This class of programs has been studied by Smaus (1999b) and Bossi *et al.* (1999; 2000; 2001).

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

A *local* selection rule is a selection rule specifying that an atom can only be selected if there is no other atom which was introduced (by resolution) more recently. Marchiori and Teusink (1999) have studied termination for selection rules that are both local and *delay-safe*, i.e. they respect the *delay declarations*. We will call termination w.r.t. such selection rules *local delay* termination. *A priori*, the LD selection rule, input-consuming selection rules and local delay-safe selection rules are not formally comparable. Under reasonable assumptions however, one can say that assuming input-consuming selection rules is weaker than assuming local and delay-safe selection rules, which is again weaker than assuming the LD selection rule. This is indicated in the figure by arrows drawn with dashed lines. Again, the numbers in the figure correspond to propositions in section 9.

In this survey, we present declarative characterisations of the classes of programs and queries that terminate with respect to each of the mentioned notions of termination. The characterisations make use of level mappings and Herbrand models in order to provide proof obligations on program clauses and queries. All characterisations are sound. Except for the case of local delay termination, they are also complete (in the case of input termination, this holds only under certain restrictions).

This survey is organised as follows. The next section introduces some basic concepts and fixes the notation. Then we have six sections corresponding to the six classes in figure 1, defined by increasingly strong assumptions about the selection rule. In each section, we introduce a notion of termination and provide a declarative characterisation for the corresponding class of terminating programs and queries. In section 9, we establish relations between the classes, formally showing the implications of figure 1. Section 10 discusses the related work, and section 11 concludes.

2 Background and notation

We use the notation of Apt (1997), when not otherwise specified. In particular, throughout this article we consider a fixed language L in which programs and queries are written. All the results are *parametric* with respect to L, provided that L is rich enough to contain the symbols of the programs and queries under consideration. We denote with U_L and B_L the Herbrand universe and the Herbrand base on L. $Term_L$ and $Atom_L$ denote the set of terms and atoms on L. We use typewriter font for logical variables, e.g. `X`, `Ys`, upper case letters for arbitrary terms, e.g. Xs, and lower case letters for ground terms, e.g. t, x, xs. We denote by $inst_L(P)$ ($ground_L(P)$) the set of (ground) instances of all clauses in P that are in language L. The notation $ground_L(Q)$ for a query Q is defined analogously.

The domain (resp., set of variables in the range) of a substitution θ is denoted as $Dom(\theta)$ (resp., $Ran(\theta)$).

2.1 Modes

For a predicate p/n, a *mode* is an atom $p(m_1, \dots, m_n)$, where $m_i \in \{I, O\}$ for $i \in [1, n]$. Positions with I are called *input positions*, and positions with O are called *output positions* of p. To simplify the notation, an atom written as $p(\mathbf{s}, \mathbf{t})$ means: $\mathbf{s}$ is the

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

vector of terms filling in the input positions, and $\mathbf{t}$ is the vector of terms filling in the output positions. An atom $p(\mathbf{s}, \mathbf{t})$ is *input-linear* if $\mathbf{s}$ is linear, i.e. each variable occurs at most once in $\mathbf{s}$. The atom is *output-linear* if $\mathbf{t}$ is linear.

In the literature, several correctness criteria concerning the modes have been proposed, the most important ones being nicely-modedness and well-modedness (Apt, 1997). In this article, we need *simply* moded programs (Apt and Etalle, 1993), which are a special case of nicely moded programs, as well as *well moded* programs.

Definition 2.1
A clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is *simply moded* if $\mathbf{t}_1, \dots, \mathbf{t}_n$ is a linear vector of variables and for all $i \in [1, n]$

$$Var(\mathbf{t}_i) \cap Var(\mathbf{t}_0) = \emptyset \quad \text{and} \quad Var(\mathbf{t}_i) \cap \bigcup_{j=1}^{i} Var(\mathbf{s}_j) = \emptyset.$$

A query $\mathbf{B}$ is *simply moded* if the clause $q \leftarrow \mathbf{B}$ is simply moded, where q is any variable-free atom. A program is simply moded if all of its clauses are.

A query (clause, program) is *permutation simply moded* if it is simply moded modulo reordering of the atoms of the query (each clause body).

Thus, a clause is simply moded if the output positions of body atoms are filled in by distinct variables, and every variable occurring in an output position of a body atom does not occur in an earlier input position. In particular, every unit clause is simply moded.

Definition 2.2
A query $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is *well moded* if for all $i \in [1, n]$ and $K = 1$

$$Vars(\mathbf{s}_i) \subseteq \bigcup_{j=K}^{i-1} Vars(\mathbf{t}_j) \tag{1}$$

The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is *well moded* if (1) holds for all $i \in [1, n+1]$ and $K = 0$. A program is *well moded* if all of its clauses are well moded.

A query (clause, program) is *permutation well moded* if it is well moded modulo reordering of the atoms of the query (each clause body).

Almost all programs we consider in this article are permutation well and simply moded with respect to the same set of modes. The program in figure 7 is an exception due to the fact that our notion of modes cannot capture that sub-arguments of a term can have different modes. We do not always give the modes explicitly, but they are usually easy to guess.

2.2 Selection rules

Let *INIT* be the set of initial fragments of SLD-derivations in which the last query is non-empty. The standard definition of *selection rule* is as follows: a selection rule is a function that, when applied to an element in *INIT*, yields an occurrence of an atom in its last query (Apt, 1997). In this article, we assume an extended definition:

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

we also allow that a selection rule may select no atom (a situation called *deadlock*), and we allow that it not only returns the selected atom, but also specifies the set of program clauses that may be used to resolve the atom. Whenever we want to emphasise that a selection rule always selects exactly one atom together with the entire set of clauses for that atom's predicate, we speak of a *standard selection rule*. Note that for the extended definition, completeness of SLD-resolution is lost in general. Selection rules are denoted by s.

We now define the selection rules used in this article, except for *delay-safe* selection rules, since these rely on notions introduced only later.

Input-consuming selection rules are defined w.r.t. a given mode. A selection rule s is *input-consuming* for a program P if either

- s selects an atom $p(\mathbf{s}, \mathbf{t})$ and a non-empty set of clauses of P such that $p(\mathbf{s}, \mathbf{t})$ and each head of a clause in the set are unifiable with an mgu σ, and $Dom(\sigma) \cap Vars(\mathbf{s}) = \emptyset$, or
- s selects an atom $p(\mathbf{s}, \mathbf{t})$ that unifies with no clause head from P, together with all clauses in P (this models *failure*), or
- if the previous cases are impossible, s selects no atom (i.e. we have *deadlock*).

Consider a query, containing atoms A and B, in an initial fragment ξ of a derivation. Then A is *introduced more recently* than B if the derivation step introducing A comes after the step introducing B, in ξ. A *local selection rule* is a selection rule that specifies that an atom in a query can be selected only if there is no more recently introduced atom in the query.

The usual *LD* selection rule (also called *leftmost* selection rule) always selects the leftmost atom in the last query of an element in $INIT$. The *RD* selection rule (also called *rightmost*) always selects the rightmost atom.

A standard selection rule s is *fair* if for every SLD-derivation ξ via s either ξ is finite or for every atom A in ξ, (some further instantiated version of) A is eventually selected.

2.3 Universal termination

In general terms, the problem of universal termination of a program P and a query Q w.r.t. a set of admissible selection rules consists of showing that every rule in the set is a terminating control for P and Q.

Definition 2.3
A program P and a query Q *universally terminate* w.r.t. a set of selection rules $\mathscr{S}$ if every SLD-derivation of P and Q via any selection rule from $\mathscr{S}$ is finite.

Note that, since SLD-trees are finitely branching, by König's Lemma, "every SLD-derivation for P and Q via a selection rule s is finite" is equivalent to stating that the SLD-tree of P and Q via s is finite.

We say that a class of programs and queries is a *sound* characterisation of universal termination w.r.t. $\mathscr{S}$ if every program and query in the class universally terminate w.r.t. $\mathscr{S}$. Conversely, it is *complete* if every program and query that universally terminate w.r.t. $\mathscr{S}$ are in the class.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

2.4 Norms and level mappings

All the characterisations of terminating programs we propose make use of the notions of norm and level mapping (Cavedon, 1989). Depending on the approach, such notions are defined on ground or arbitrary objects.

In the following definition, $Term_L/\sim$ denotes the set of equivalence classes of terms modulo variance. Similarly, we define $Atom_L/\sim$.

Definition 2.4

A *norm* is a function $|.| : U_L \to \mathbb{N}$. A *level mapping* is a function $|.| : B_L \to \mathbb{N}$. For a ground atom A, $|A|$ is called the level of A.

An atom A is *bounded* w.r.t. the level mapping $|.|$ if there exists $k \in \mathbb{N}$ such that for every $A' \in ground_L(A)$, we have $k > |A'|$.

A *generalised norm* is a function $|.| : Term_L/\sim \; \to \mathbb{N}$. A *generalised level mapping* is a function $|.| : Atom_L/\sim \; \to \mathbb{N}$. Abusing notation, we write $|T|$ ($|A|$) to denote the value of $|.|$ on the equivalence class of the term T (the atom A).

(Generalised) level mappings are used to measure the 'size' of a query and show that this size decreases along a derivation, hence showing termination. They are usually defined based on (generalised) norms. Therefore, we often use the same notation $|.|$ for a norm and a level mapping based on it.

Of course, a generalised norm or level mapping can be interpreted as an ordinary norm or level mapping by restricting its domain to ground objects. Therefore, we now give some examples of *generalised* norms and level mappings.

One commonly used generalised norm is the term size norm, defined as

$$\begin{array}{rcll} size(f(T_1,\ldots,T_n)) & = & 1 + size(T_1) + \ldots + size(T_n) & \text{if } n > 0 \\ size(T) & = & 0 & \text{if } T \text{ constant/variable.} \end{array}$$

Intuitively, the size of a term T is the number of function symbols occurring in T, excluding constants. Another widely used norm is the list-length function, defined as

$$\begin{array}{rcll} |[T|Ts]| & = & 1 + |Ts| & \\ |f(\ldots)| & = & 0 & \text{if } f \neq [\,.\,|\,.\,]. \end{array}$$

In particular, for a nil-terminated list $[T_1,\ldots,T_n]$, the list-length is n.

We will see later that usually, level mappings measure the *input* arguments of a query, even though this is often just an intuitive understanding and not explicit. Moreover, the choice of a particular selection rule often reflects a particular mode of the program. In this sense, the choice of the level mapping must depend on the selection rule, via the modes. This will be seen in our examples.

However, apart form the dependency just mentioned, the choice of level mapping is an aspect of termination which is rather independent from the choice of the selection rule. In particular, one does not find any interesting relationship between the underlying *norms* and the selection rule. This is why the detailed study of various norms and level mappings is beyond the scope of this article, although it is an important aspect of automated proofs of termination (Decorte *et al.*, 1993; Bossi *et al.*, 1994).

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

We now define level mappings where the dependency on the modes is made explicit (Etalle *et al.*, 1999).

Definition 2.5
A *moded (generalised) level mapping* $|.|$ is a (generalised) level mapping such that for any (not necessarily) ground $\mathbf{s}$, $\mathbf{t}$ and $\mathbf{u}$, $|p(\mathbf{s},\mathbf{t})| = |p(\mathbf{s},\mathbf{u})|$.

The condition $|p(\mathbf{s},\mathbf{t})| = |p(\mathbf{s},\mathbf{u})|$ states that the *level* of an atom is independent from the terms in its output positions.

2.5 Models

Several of the criteria for termination we consider rely on information supplied by a model of the program under consideration. We provide the definition of Herbrand interpretations and models (Apt, 1997).

A *Herbrand interpretation* I is a set of ground atoms. A ground atom A is *true in* I, written $I \models A$, if $A \in I$. This notation is extended to ground queries in the obvious way. I is a Herbrand *model* of program P if for each $A \leftarrow B_1, \dots, B_n \in ground_L(P)$, we have that $I \models B_1, \dots, B_n$ implies $I \models A$.

When speaking of the *least* Herbrand model of P, we mean least w.r.t. set inclusion. In termination analysis, it is usually not necessary to consider the least Herbrand model, which may be difficult or impossible to determine. Instead, one uses models that capture some *argument size relationship* between the arguments of each predicate (De Schreye and Decorte, 1994). For example, a model for the usual append predicate is

$$\{\mathtt{append}(xs, ys, zs) \mid |zs| = |xs| + |ys|\}$$

where $|.|$ is the list-length function.

3 Strong termination

3.1 Operational definition

Early approaches to the termination problem treated universal termination w.r.t. *all* selection rules, called *strong* termination. Generally speaking, strongly terminating programs and queries are either very trivial or especially written for theoretical considerations.

Definition 3.1
A program P and query Q *strongly terminate* if they universally terminate w.r.t. the set of all selection rules.

3.2 Declarative characterisation

In the following, we recall the approach of Bezem (1993), who defined the class of recurrent programs and queries. Intuitively, a program is recurrent if for every ground instance of a clause, the level of the body atoms is smaller than the level of the head.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Definition 3.2
Let $|.|$ be a level mapping.

A program P is *recurrent by* $|.|$ if for every $A \leftarrow B_1, \dots, B_n$ in $ground_L(P)$:

$$\text{for } i \in [1,n] \quad |A| > |B_i|.$$

A query Q is *recurrent by* $|.|$ if there exists $k \in \mathbb{N}$ such that for every $A_1, \dots, A_n \in ground_L(Q)$:

$$\text{for } i \in [1,n] \quad k > |A_i|.$$

In the above definition, the proof obligations for a query Q are derived from those for the program $\{\mathtt{p} \leftarrow Q\}$, where $\mathtt{p}$ is a fresh predicate symbol. Intuitively, this is justified by the fact that the termination behaviour of the query Q and a program P is the same as for the query $\mathtt{p}$ and the program $P \cup \{\mathtt{p} \leftarrow Q\}$. So k plays the role of the level of the atom $\mathtt{p}$. In the original work (Bezem, 1993), the query was called *bounded*. Throughout the paper, we prefer to maintain a uniform naming convention both for programs and queries.

In section 9.1, we will compare recurrence to other characterisations.

Termination properties of recurrent programs are summarised in the following theorem.

Theorem 3.3 (Bezem, 1993)
Let P be a program and Q a query.

If P and Q are both recurrent by a level mapping $|.|$, then they strongly terminate.

Conversely, if P and every *ground query* strongly terminate, then P is recurrent by some level mapping $|.|$. If in addition P and Q strongly terminate, then P and Q are both recurrent by some level mapping $|.|$.

Proof
The result is shown in Bezem (1993) for standard selection rules. It easily extends to our generalisation of selection rules by noting that P and Q strongly terminate iff they universally terminate w.r.t. the set of standard selection rules. The only-if part is immediate. The if-part follows by noting that a derivation via an arbitrary selection rule is a (prefix of a) derivation via a *standard* selection rule. □

3.3 Examples

Example 3.4
The program `SAT` in figure 2 decides propositional satisfiability. The program is readily checked to be recurrent by $|.|$, where we define

$$|\mathtt{sat}(t)| = |\mathtt{inval}(t)| = size(t).$$

Note that Definition 3.2 imposes no proof obligations for unit clauses. The query `sat(`X`)` is recurrent iff there exists a natural k such that for every ground instance x of X, we have that $size(x)$ is bounded by k. Obviously, this is the case iff X is already a ground term. For instance, the query `sat(not(true) ∧ false)` is recurrent, while the query `sat(false ∧ X)` is not.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

```
% sat(Formula) ←
%    there is a true instance of Formula

  sat(true).
  sat(X ∧ Y) ←
     sat(X), sat(Y).
  sat(not X) ← inval(X).

  inval(false).
  inval(X ∧ Y) ← inval(X).
  inval(X ∧ Y) ← inval(Y).
  inval(not X) ← sat(X).
```

Fig. 2. SAT.

```
% append(Xs,Ys,Zs) ←
%    Zs is the result of concatenating the lists Xs and Ys.

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

Fig. 3. APPEND.

Note that the choice of an appropriate level mapping depends upon the intended mode of the program and query. Even though this is usually not explicit, level mappings measure the size of the *input* arguments of an atom (Etalle *et al*., 1999).

Example 3.5

Figure 3 shows the APPEND program. It is easy to check that APPEND is recurrent by the level mapping $|\text{append}(xs, ys, zs)| = |xs|$ and also by $|\text{append}(xs, ys, zs)| = |zs|$ (recall that $|.|$ is the list-length function). A query append(Xs,Ys,Zs) is recurrent by the first level mapping iff Xs is a list, and by the second iff Zs is a list. The level mapping

$$|\text{append}(xs, ys, zs)| = min\{|xs|, |zs|\}$$

combines the advantages of both level mappings. APPEND is easily seen to be recurrent by it, and if Xs *or* Zs is a list, append(Xs, Ys, Zs) is recurrent by it.

3.4 On completeness of the characterisation

Note that completeness is not stated in full general terms, i.e. recurrence is not a complete proof method for strong termination. Informally speaking, incompleteness is due to the use of level mappings, which are functions that must specify a value for every ground atom. Therefore, if P strongly terminates for a certain ground query Q but not for all ground queries, we cannot conclude that P is recurrent. We provide a general completeness result in section 6 for a class of programs containing recurrent programs.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

```
% even(X) ←
%     X is an even natural number.

  even(s(s(X))) ← even(X).
  even(0).
```

```
% lte(X,Y) ←
%     X,Y are natural numbers
%     s.t. X is smaller or equal than Y.

  lte(s(X),s(Y)) ← lte(X,Y).
  lte(0,Y).
```

Fig. 4. EVEN.

4 Input termination

We have said above that the class of strongly terminating programs and queries is very limited. Even if a program is recurrent, it may not strongly terminate for a query of interest since the query is not recurrent.

Example 4.1
The program EVEN in figure 4 is recurrent by defining

$$\begin{aligned} |\texttt{even}(x)| &= \mathit{size}(x) \\ |\texttt{lte}(x,y)| &= \mathit{size}(y). \end{aligned}$$

Now consider the query $Q = \texttt{even}(\texttt{X}),\ \texttt{lte}(\texttt{X}, \texttt{s}^{100}(\texttt{0}))$, which is supposed to compute the even numbers not exceeding 100. By always selecting the leftmost atom, one can easily obtain an infinite derivation for EVEN and Q. As a consequence of Theorem 3.3, Q is not recurrent.

4.1 Operational definition

We now define termination for input-consuming derivations (Bossi *et al.*, 2001), i.e. derivations via an input-consuming selection rule.

Definition 4.2
A program P and query Q *input terminate* if they universally terminate w.r.t. the set consisting of the input-consuming selection rules.

The requirement of input-consuming derivations merely reflects the very meaning of *input*: an atom must only consume its own input, not produce it. In existing implementations, input-consuming derivations can be ensured using control constructs such as delay-declarations (Hill and Lloyd, 1994; SICStus, 1998; Somogyi *et al.*, 1996; Thom and Zobel, 1988).

In the above example, the obvious mode is $\texttt{even}(I)$, $\texttt{lte}(O,I)$. With this mode, we will show that EVEN and Q input terminate. If we assume a selection rule that is input-consuming while always selecting the leftmost atom if possible, then the above example is a contrived instance of the *generate-and-test* paradigm. This paradigm involves two procedures, one which generates a set of candidates, and another which tests whether these candidates are solutions to the problem. The test occurs to the left of the generator so that tests take place as soon as possible, i.e. as soon as sufficient input has been generated for the derivation to be input-consuming.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Proofs of input termination differ from proofs of strong termination in an important respect. For the latter, we require that the initial query is recurrent, and as a consequence we have that all queries in any derivation from it are recurrent (we say that recurrence is *persistent* under resolution). This means that, at the time an atom is selected, the depth of its SLD tree is bounded. In contrast, input termination does not need such a strong requirement on each selected atom.

Example 4.3
Consider the EVEN program and the following input-consuming derivation, where we underline the selected atom in each step

$$\begin{array}{ll} \texttt{even(X)},\ \underline{\texttt{lte(X,}\,\texttt{s}^{100}\texttt{(0))}} \longrightarrow & \texttt{even(s(X'))},\ \underline{\texttt{lte(X',}\,\texttt{s}^{99}\texttt{(0))}} \longrightarrow \\ \underline{\texttt{even(s(s(X'')))}},\ \texttt{lte(X'',}\,\texttt{s}^{98}\texttt{(0))} \longrightarrow & \texttt{even(X'')},\ \underline{\texttt{lte(X'',}\,\texttt{s}^{98}\texttt{(0))}} \ldots \end{array}$$

At the time when even(s(s(X″))) is selected, the depth of its SLD-tree is not bounded (without knowing the eventual instantiation of X″).

4.2 Information on data flow: simply-local substitutions and models

Since the depth of the SLD-tree of the selected atom depends on further instantiation of the atom, it is important that programs are well-behaved w.r.t. the modes. This is illustrated in the following example.

Example 4.4
Consider the APPEND program in mode append(I, I, O) and the query

$$\texttt{append([1|As], [], Bs), append(Bs, [], As)}.$$

Then we have the following infinite input-consuming derivation:

$$\begin{array}{l} \underline{\texttt{append([1|As], [], Bs)}},\ \texttt{append(Bs, [], As)} \longrightarrow \\ \texttt{append(As, [], Bs')},\ \underline{\texttt{append([1|Bs'], [], As)}} \longrightarrow \\ \underline{\texttt{append([1|As'], [], Bs')}},\ \texttt{append(Bs', [], As')} \longrightarrow \ldots \end{array}$$

This well-known termination problem of programs with coroutining has been identified as *circular modes* by Naish (1992).

To avoid the above situation, we require programs to be simply moded (see section 2.1).

We now define *simply-local* substitutions, which reflect the way simply moded clauses become instantiated in input-consuming derivations. Given a clause $c = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ used in an input-consuming derivation, first $\mathbf{t}_0$ becomes instantiated, and the range of that substitution contains only variables from outside of c. Then, by resolving $p_1(\mathbf{s}_1, \mathbf{t}_1)$, the vector $\mathbf{t}_1$ becomes instantiated, and the range of that substitution contains variables from outside of c in addition to variables from $\mathbf{s}_1$. Continuing in the same way, finally, by resolving $p_n(\mathbf{s}_n, \mathbf{t}_n)$, the vector $\mathbf{t}_n$ becomes instantiated, and the range of that substitution contains variables from outside of c in addition to variables from $\mathbf{s}_1 \ldots \mathbf{s}_n$. A substitution is *simply-local* if it is composed from substitutions as sketched above. The formal definition is as follows.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Definition 4.5
A substitution θ is *simply-local* w.r.t. the clause $c = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ if there exist substitutions $\sigma_0, \sigma_1 \ldots, \sigma_n$ and disjoint sets of fresh (w.r.t. c) variables $v_0, v_1, \ldots, v_n$ such that $\theta = \sigma_0\sigma_1 \cdots \sigma_n$ where for $i \in \{0, \ldots, n\}$,

- $Dom(\sigma_i) \subseteq Vars(\mathbf{t}_i)$,
- $Ran(\sigma_i) \subseteq Vars(\mathbf{s}_i\sigma_0\sigma_1 \cdots \sigma_{i-1}) \cup v_i$.[1]

θ is *simply-local* w.r.t. a query $\mathbf{B}$ if θ is simply-local w.r.t. the clause $q \leftarrow \mathbf{B}$ where q is any variable-free atom.

Note that in the case of a simply-local substitution w.r.t. a query, σ_0 is the empty substitution, since $Dom(\sigma_0) \subseteq Var(q)$ where q is an (imaginary) variable-free atom. Note also that if $\mathbf{A}, B, \mathbf{C} \longrightarrow (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ is an input-consuming derivation step using clause $c = H \leftarrow \mathbf{B}$, then $\theta_{|H}$ is simply-local w.r.t. the clause $H \leftarrow$ and $\theta_{|B}$ is simply-local w.r.t. the atom B (Bossi *et al.*, 2001).

Example 4.6
Consider `APPEND` in mode $\mathtt{append}(I, I, O)$, and its recursive clause

$$c = \mathtt{append}([\mathtt{H}|\mathtt{Xs}], \mathtt{Ys}, [\mathtt{H}|\mathtt{Zs}]) \leftarrow \mathtt{append}(\mathtt{Xs}, \mathtt{Ys}, \mathtt{Zs}).$$

The substitution $\theta = \{\mathtt{H}/\mathtt{V}, \mathtt{Xs}/[], \mathtt{Ys}/[\mathtt{W}], \mathtt{Zs}/[\mathtt{W}]\}$ is simply-local w.r.t. c: let $\sigma_0 = \{\mathtt{H}/\mathtt{V}, \mathtt{Xs}/[], \mathtt{Ys}/[\mathtt{W}]\}$ and $\sigma_1 = \{\mathtt{Zs}/[\mathtt{W}]\}$; then $Dom(\sigma_0) \subseteq \{\mathtt{H}, \mathtt{Xs}, \mathtt{Ys}\}$, and $Ran(\sigma_0) \subseteq v_0$ where $v_0 = \{\mathtt{V}, \mathtt{W}\}$, and $Dom(\sigma_1) \subseteq \{\mathtt{Zs}\}$, and $Ran(\sigma_1) \subseteq Vars((\mathtt{Xs}, \mathtt{Ys})\sigma_0)$.

Based on simply-local substitutions, we now define a restricted notion of model.

Definition 4.7
Let $I \subseteq Atom_L$. We say that I is a *simply-local model* of $c = H \leftarrow B_1, \ldots, B_n$ if for every substitution θ simply-local w.r.t. c,

$$\text{if } B_1\theta, \ldots, B_n\theta \in I \text{ then } H\theta \in I. \tag{2}$$

I is a *simply-local model* of a program P if it is a simply-local model of each clause of it.

Note that a simply-local model is not necessarily a model in the classical sense, since I is not necessarily a set of ground atoms, and the substitution in (2) is required to be simply-local. For example, given the program $\{\mathtt{q}(1),\ \mathtt{p}(\mathtt{X}) \leftarrow \mathtt{q}(\mathtt{X})\}$ with modes $\mathtt{q}(I)$, $\mathtt{p}(O)$, a model must contain the atom $\mathtt{p}(1)$, whereas a simply-local model does not necessarily contain $\mathtt{p}(1)$, since $\{\mathtt{X}/1\}$ is not simply-local w.r.t. $\mathtt{p}(\mathtt{X}) \leftarrow \mathtt{q}(\mathtt{X})$. The next subsection will further clarify the role of simply-local models.

Let SM_P be the set of all simply moded atoms in $Atom_L$. It has been shown that the least simply-local model of P containing SM_P exists and can be computed by a variant of the well-known T_P-operator (Bossi *et al.*, 2001). We denote the least simply-local model of P containing SM_P by PM_P^{SL}, for *partial model*.

[1] Note that $\mathbf{s}_0$ is undefined. By abuse of notation, $Vars(\mathbf{s}_0 \ldots) = \emptyset$.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Example 4.8
Consider APPEND. To compute $PM^{SL}_{\texttt{APPEND}}$, we must iterate the above mentioned variant of the T_P-operator starting from the fact clause 'append([],Ys,Ys).' and any simply moded atom. It turns out that

$$PM^{SL}_{\texttt{APPEND}} = \bigcup_{n=0}^{\infty} (\{\texttt{append}([T_1,\dots,T_n],T,[T_1,\dots,T_n|T])\} \cup \{\texttt{append}([T_1,\dots,T_n|S],T,[T_1,\dots,T_n|\texttt{X}]) \mid \texttt{X} \text{ is fresh}\}).$$

We refer to Bossi *et al.* (2001) for the details of this calculation.

4.3 Declarative characterisation

We now define *simply-acceptability*, which is the notion of decrease used for proving input termination.

We write $p \simeq q$ if p and q are mutually recursive predicates (Apt, 1997). Abusing notation, we also use $\simeq$ for *atoms*, where $p(\mathbf{s},\mathbf{t}) \simeq q(\mathbf{u},\mathbf{v})$ stands for $p \simeq q$.

Definition 4.9
Let P be a program, $|.|$ a moded generalised[2] level mapping and I a simply-local model of P containing SM_P. A clause $A \leftarrow B_1,\dots,B_n$ is *simply-acceptable by* $|.|$ *and* I if for every substitution θ simply-local w.r.t. it,

$$\text{for all } i \in [1,n], \quad (B_1,\dots,B_{i-1})\theta \in I \text{ and } A \simeq B_i \quad \text{implies} \quad |A\theta| > |B_i\theta|.$$

The program P is *simply-acceptable by* $|.|$ *and* I if each clause of P is simply-acceptable by $|.|$ and I.

Admittedly, the proof obligations may be difficult to verify, especially in the cases where a small (precise) simply-local model is required. However, as our examples show, often it is not necessary at all to consider the model, as one can show the decrease for arbitrary instantiations of the clause.

Unlike all other characterisations in this article, simply-acceptability is not based on ground instances of clauses, but rather on instances obtained by applying simply-local substitutions, which arise in input-consuming derivations of simply moded programs. This is also why we use *generalised* level mappings and a special kind of models.

Also note that in contrast to recurrence and other decreasing notions to be defined later, simply-acceptability has no proof obligation on queries (apart from the requirement that queries must be simply moded). Intuitively, such a proof obligation is made redundant by the mode conditions (simply-acceptability and moded level mapping) and the fact that derivations must be input-consuming. We also refer to section 9.1.

We can now show that this concept allows to characterise the class of input terminating programs.

[2] In (Bossi *et al.*, 2001), the word "generalised" is dropped, but here we prefer to emphasise that non-ground atoms are included in the domain.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

```
% permute(Xs,Ys) ←
%    Ys is a permutation of the list Xs.

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

```
% insert(Xs,X,Zs) ←
%    Zs is obtained by inserting X into Xs.

  insert([],X,[X]).
  insert([U|Xs],X,[U|Zs]) ←
     insert(Xs,X,Zs).
```

Fig. 5. PERMUTE.

Theorem 4.10 (Bossi et al.*, 2001)*
Let P and Q be a simply moded program and query.

If P is simply-acceptable by some $|.|$ and I, then P and Q input terminate.

Conversely, if P and every simply moded query input terminate, then P is simply-acceptable by some $|.|$ and PM_P^{SL}.

Note that the formulation of the theorem differs slightly from the original for reasons of consistency, but one can easily see that the formulations are equivalent.

The definition of input-consuming derivations is independent from the textual order of atoms in a query, and so the textual order is irrelevant for termination. This means of course that if we can prove input termination for a program and query, we have also proven termination for a program obtained by permuting the body atoms of each clause and the query in an arbitrary way. This will be seen in the next example. It would have been possible to state this explicitly in the above theorem, but that would have complicated the definition of simply-local substitution and subsequent definitions. Generally, the question of whether or not it is necessary to make the permutations of body atoms explicit was discussed by Smaus (1999a).

4.4 Examples

Example 4.11
The program EVEN in figure 4 is simply-acceptable with modes even(I), lte(O,I) by using the level mapping in Example 4.1, interpreted as moded *generalised* level mapping in the obvious way, and using any simply-local model. Moreover, the query even(X), lte(X, $\mathtt{s}^{100}$(0)) is permutation simply moded. Hence EVEN and this query input terminate.

Example 4.12
Figure 5 shows the program PERMUTE. Note that permute $\not\simeq$ insert. Assume the modes permute(I,O), insert(I,I,O). The program is readily checked to be simply-acceptable, using the moded generalised level mapping

$$|\mathtt{permute}(Xs, Ys)| = |\mathtt{insert}(Xs, Ys, Zs)| = size(Xs)$$

and any simply-local model. Thus the program and any simply moded query input terminate. It can also easily be shown that the program is not recurrent.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

```
%       quicksort(Xs, Ys) ← Ys is an ordered permutation of  Xs.

        quicksort(Xs,Ys) ← quicksort_dl(Xs,Ys,[]).

        quicksort_dl([X|Xs],Ys,Zs) ←
           partition(Xs,X,Littles,Bigs),
           quicksort_dl(Bigs,Ys1,Zs).
           quicksort_dl(Littles,Ys,[X|Ys1]),
        quicksort_dl([],Xs,Xs).

        partition([X|Xs],Y,[X|Ls],Bs) ← X =< Y, partition(Xs,Y,Ls,Bs).
        partition([X|Xs],Y,Ls,[X|Bs]) ← X > Y, partition(Xs,Y,Ls,Bs).
        partition([],Y,[],[]).
```

Fig. 6. QUICKSORT.

Example 4.13
Figure 6 shows program 15.3 from (Sterling and Shapiro, 1986): QUICKSORT using a form of difference lists (we permuted two body atoms for the sake of clarity). This program is simply moded with the modes quicksort(I, O), quicksort_dl(I, O, I), partition(I, I, O, O), =<(I, I), >(I, I).

Let $|.|$ be the list-length function (see subsection 2.4). We use the following moded generalised level mapping (positions with _ are irrelevant)

$$|\texttt{quicksort\_dl}(Xs, \_, \_)| \quad = \quad |Xs|,$$
$$|\texttt{partition}(Xs, \_, \_, \_)| \quad = \quad |Xs|.$$

The level mapping of all other atoms can be set to 0. Concerning the model, the simplest solution is to use the model that expresses the dependency between the list lengths of the arguments of partition, i.e. I should contain all atoms of the form partition(S_1, X, S_2, S_3) where $|S_1| \geqslant |S_2|$ and $|S_1| \geqslant |S_3|$ ($|.|$ being the list-length function). Note that this includes all simply-moded atoms using partition, and that this model is a fortiori simply-local since (2) in Definition 4.7 is true even for arbitrary θ.

The program is then simply-acceptable by $|.|$ and I and hence input terminates for every simply moded query.

5 Local delay termination

The class of programs and queries that terminate for all input-consuming derivations is considerable, but there are still many interesting programs not contained in it.

Example 5.1
Consider again the PERMUTE program (figure 5), but this time assume the mode permute(O, I), insert(O, O, I). Consider also the query permute(X, [1]). It is easy to check that there is an infinite *input-consuming* derivation for this query obtained by selecting always the leftmost atom that can be selected. In fact, PERMUTE in this mode cannot be simply-acceptable, not even after reordering of atoms in clause bodies. To see this, we first reorder the body atoms of the recursive clause to obtain

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

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

so that the program is simply moded and thus our method showing input termination is applicable in principle. Now $PM^{SL}_{\texttt{PERMUTE}}$ contains every atom of the form insert(Us, U, *Vs*), i.e., every simply moded atom whose predicate is insert. Therefore in particular insert(Us, U, Vs) $\in PM^{SL}_{\texttt{PERMUTE}}$ (note that Vs is a variable). The substitution $\theta = \{\texttt{Ys/Vs}, \texttt{Zs/Us}, \texttt{X/U}\}$ is simply-local w.r.t. the clause. Therefore, for the clause to be simply-acceptable, there would have to be a moded generalised level mapping such that |permute([U|Xs], Vs)| > |permute(Xs, Us)|. This is a contradiction since a *moded* generalised level mapping is necessarily defined as a generalised norm of the second argument of permute, and Vs and Us are equivalent modulo variance.

However, all derivations for this query are finite w.r.t. the RD selection rule, which for this example happens to be an instance of the selection rules considered in this section.

5.1 Operational definition

Marchiori and Teusink (1999) have considered local selection rules controlled by delay declarations. They define a *safe delay declaration* so that an atom can be selected only when it is bounded w.r.t. a level mapping. In order to avoid even having to define delay declarations, we take a shortcut, by defining the following.

Definition 5.2
A selection rule is *delay-safe (w.r.t.* |.|*)* if it specifies that an atom A can be selected only when A is bounded w.r.t. |.|.

Note that delay-safe selection rules imply that the depth of the SLD-tree of the selected atom does not depend on further instantiation as in the previous section.

Definition 5.3
A program P and query Q *local delay terminate* (w.r.t. |.|) if they universally terminate w.r.t. the set of selection rules that are both local and delay-safe (w.r.t. |.|).

Unlike in the previous section, modes are not used explicitly in the definition of delay-safe selection rules. Therefore it is possible to invent an example of a program and a query that input terminate but do not local delay terminate. Such an example is of course contrived, in that the level mapping is chosen in an inappropriate way.

Example 5.4
The APPEND program and the query append([],[],X), append(X,[],Y) input terminate for the mode append(I,I,O). However, they do not local delay terminate w.r.t. a level mapping |.| such that $|A| = 0$ for every A (e.g., consider the RD selection rule).

However, in section 9 we will see that under natural assumptions (in particular, the level mapping must be moded) delay-safe selection rules are also input-consuming. Then, input termination implies local delay termination, and as is witnessed by Example 5.1, this implication is strict.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

5.2 Information on data flow: covers

Delay-safe selection rules ensure that selected atoms are bounded. To ensure that the level mapping *decreases* during a derivation, we exploit additional information provided by a model of the program. Given an atom B in a query, we are interested in other atoms that share variables with B, so that instantiating these variables makes B bounded. A set of such atoms is called a *direct cover*. The only way of making B bounded is by resolving away one of its direct covers. The formal definition is as follows.

Definition 5.5
Let $|.|$ be a level mapping, $A \leftarrow Q$ a clause containing a body atom B, and $\tilde{C}$ a subset[3] of Q such that $B \notin \tilde{C}$. We say that $\tilde{C}$ is a *direct cover for B (w.r.t. $A \leftarrow Q$ and $|.|$)* if there exists a substitution θ such that $B\theta$ is bounded w.r.t. $|.|$ and $Dom(\theta) \subseteq Vars(A, \tilde{C})$.

A direct cover is *minimal* if no proper subset is a direct cover.

Note that the above concept is similar to well-modedness, assuming a moded level mapping. In this case, for each atom, the atoms to the left of it are a direct cover. This generalises in the obvious way to *permutation* well moded queries.

Considering an atom B, we have said that the only way of making B bounded is by resolving away one of B's direct covers. However, for an atom in a direct cover, say atom A, to be selected, A must be bounded, and the only way of making A bounded is by resolving away one of A's direct covers. Iterating this reasoning gives rise to a kind of closure of the notion of direct cover.

Definition 5.6
Let $|.|$ be a level mapping and $A \leftarrow Q$ a clause. Consider the least set $\mathscr{C}$, subset of $\mathscr{P}(Q \times \mathscr{P}(Q))$, such that

1. $\langle B, \emptyset \rangle \in \mathscr{C}$ whenever B has $\emptyset$ as minimal direct cover for B in $A \leftarrow Q$;
2. $\langle B, \tilde{C} \rangle \in \mathscr{C}$ whenever $B \notin \tilde{C}$, and $\tilde{C} = \{C_1, \ldots, C_k\} \cup \tilde{D}_1 \cup \ldots \tilde{D}_k$, where $\{C_1, \ldots, C_k\}$ is a minimal direct cover of B in $A \leftarrow Q$, and for $i \in [1, k]$, $\langle C_i, \tilde{D}_i \rangle \in \mathscr{C}$.

The set $Covers(A \leftarrow Q) \subseteq Q \times \mathscr{P}(Q)$ is defined as the set obtained by deleting from $\mathscr{C}$ each element of the form $\langle B, \tilde{C} \rangle$ if there exists another element of $\mathscr{C}$ of the form $\langle B, \tilde{C}' \rangle$ such that $\tilde{C}' \subset \tilde{C}$.

We say that $\tilde{C}$ is a *cover for B (w.r.t. $A \leftarrow Q$ and $|.|$)* if $\langle B, \tilde{C} \rangle$ is an element of $Covers(A \leftarrow Q)$.

5.3 Declarative characterisation

The following concept is used to show that programs terminate for local and delay-safe selection rules. We present a definition slightly different from the original one (Marchiori and Teusink, 1999), albeit equivalent.

[3] By abuse of terminology, here we identify a query with the set of atoms it contains.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Definition 5.7
Let $|.|$ be a level mapping and I a Herbrand interpretation. A program P is *delay-recurrent by* $|.|$ *and* I if I is a model of P, and for every clause $c = A \leftarrow B_1, \ldots, B_n$ of P, for every $i \in [1, n]$, for every cover $\tilde{C}$ for B_i, for every substitution θ such that $c\theta$ is ground,

$$\text{if } I \models \tilde{C}\theta \text{ then } |A\theta| > |B_i\theta|.$$

We believe that this notion should have better been called *delay-acceptable*, since the convention is to call decreasing notions that involve models *(…)-acceptable*, and the ones that do not involve models *(…)-recurrent*.

The most essential differences between delay-recurrence and simply-acceptability are that the former is based on models, whereas the latter is based on simply-local models, and that the former requires decreasing for all body atoms, whereas the latter only for mutually recursive calls.

Just as simply-acceptability, delay-recurrence imposes no proof obligation on queries. Such a proof obligation is made redundant by the fact that selected atoms must be bounded. Note that if no most recently introduced atom in a query is bounded, we obtain termination by deadlock. We also refer to section 9.1.

For delay-recurrence to ensure termination, it is crucial that when an atom is selected, its cover is resolved away completely (this allows to use the premise $I \models \tilde{C}\theta$ in Definition 5.7). To this end, local selection rules must be adopted. We can now state the result of this section.

Theorem 5.8 (Marchiori and Teusink, 1999)
Let P be a program. If P is delay-recurrent by a level mapping $|.|$ and a Herbrand interpretation I, then for every query Q, P and Q local delay terminate.

5.4 Example

Example 5.9
Consider again PERMUTE (figure 5), with the level mapping and model

$$\begin{aligned} |\texttt{permute}(xs, ys)| &= |ys| + 1 \\ |\texttt{insert}(xs, ys, zs)| &= |zs| \\ I &= \{\texttt{permute}(xs, ys) \mid |xs| = |ys|\} \cup \\ &\quad\ \{\texttt{insert}(xs, y, zs) \mid |zs| = |xs| + 1\}. \end{aligned}$$

The program is delay-recurrent by $|.|$ and I. We check the recursive clause for permute. Consider an arbitrary ground instance

$$\texttt{permute}([x|xs], ys) \leftarrow \texttt{permute}(xs, zs),\ \texttt{insert}(zs, x, ys). \tag{3}$$

First, we observe that I is a model of this instance. In fact, if its body is true in I, then $|ys| = |zs| + 1$ and $|xs| = |zs|$. This implies $|ys| = |xs| + 1$, and hence $\texttt{permute}([x|xs], ys)$ is true in I.

Let us now show the decrease from the head to the permute body atom. There is only one cover insert(Zs, X, Ys), so we must show that

$$|ys| = |zs| + 1 \quad \text{implies} \quad |ys| + 1 > |zs| + 1,$$

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

which is clearly true. Now consider the second body atom. It has an empty cover. This time, for every instance of the clause such that the head is ground, we have that $|ys| + 1 > |ys|$. Hence we have shown that the clause is delay-recurrent.

It is interesting to compare this to Example 5.1, where we were not able to show a decrease.

5.5 On completeness of the characterisation

Note that delay-recurrence is a sufficient but not necessary condition for local delay termination. The limitation lies in the notion of cover: to make an atom bounded, one has to resolve one of its covers; but conversely, it is not true that resolving any cover will make the atom bounded.

Example 5.10
Consider the following simple program

```
z ← p(X), q(X), r(X).
p(0).
q(s(X)) ← q(X).
r(X).
```

The program and any query Q local delay terminate w.r.t. the level mapping:

$$|\mathtt{z}| = |\mathtt{p}(t)| = |\mathtt{r}(t)| = 0$$
$$|\mathtt{q}(t)| = size(t)$$

In fact, the only source of non-termination for a query might be an atom q(X). However, for any such atom selected by a delay-safe selection rule, X is a ground term. Hence the recursive clause in the program cannot generate an infinite derivation. On the other hand, it is not the case that the program is delay-recurrent. Consider, in fact, the first clause. Since `r(X)` is a cover for `q(X)`, we would have to show for some $|.|'$ that for every t:

$$|\mathtt{z}|' > |\mathtt{q(t)}|'.$$

This is impossible, since delay-recurrence on the third clause implies $|\mathtt{q}(\mathtt{s}^k(0))|' \geqslant k$ for any natural k.

6 Left-termination

In analogy to previous sections, we should start this section with an example illustrating that the assumption of local delay-safe selection rules is sometimes too weak to ensure termination, and thereby motivate the 'stronger' assumption of the LD selection rule. Such an example can easily be given.

Example 6.1
Consider the program

```
p ← q, p.
```

with query `p`, where $|\mathtt{p}| = |\mathtt{q}| = 0$. It left-terminates but does not local delay terminate.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

```
% trans(x,y,e) ← x ⇝_e y for a DAG e

trans(X,Y,E) ←                              member(X,[X|Xs]).
   member(arc(X,Y),E).
                                            member(X,[Y|Xs]) ←
trans(X,Y,E) ←                                 member(X,Xs).
   member(arc(X,Z),E), trans(Z,Y,E).
```

Fig. 7. TRANSP.

However, the example is somewhat artificial, and in fact, we believe that assuming the LD selection rule is only slightly stronger than assuming an arbitrary local delay-safe selection rule, as far as termination is concerned. Nevertheless, there are several reasons for studying this selection rule in its own right. First, the conditions for termination are easier to formulate than for local delay termination. Secondly, the vast majority of works consider this rule, being the standard selection rule of Prolog. Finally, for the class of programs and queries that terminate w.r.t. the LD selection rule we are able to provide a sound and complete characterisation.

6.1 Operational definition

Definition 6.2
A program P and query Q *left-terminate* if they universally terminate w.r.t. the set consisting of only the LD selection rule.

Formally comparing this class to the two previous ones is difficult. In particular, left-termination is not necessarily stronger than input or local delay termination.

Example 6.3
We have shown in Example 5.9 that PERMUTE and every query local delay terminate w.r.t. the level mapping given there. Moreover, no derivation deadlocks. However, PERMUTE and the query permute(X, [1]) do not left terminate. Similarly to Example 5.1, this example is contrived since the program is intended for the RD selection rule.

One could easily construct a similar example comparing left termination with input termination.

Also, local delay termination may not imply left-termination because of the deadlock problem.

6.2 On completeness of the characterisation

Left-termination was addressed by Apt and Pedreschi (1993), who introduced the class of acceptable logic programs. However, their characterisation encountered a completeness problem similar to the one highlighted for Theorem 3.3.

Example 6.4
Figure 7 shows TRANSP, a program that terminates on a strict subset of ground queries only. In the intended meaning of the program, trans(x, y, e) succeeds iff

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

$x \rightsquigarrow_e y$, i.e. if $\mathtt{arc}(x, y)$ is in the transitive closure of a direct acyclic graph (DAG) e, which is represented as a list of arcs. It is readily checked that if e is a graph that contains a cycle, infinite derivations may occur.

In the approach by Apt and Pedreschi, TRANSP cannot be reasoned about, since the same incompleteness problem as for recurrent programs occurs, namely that they characterise a class of programs that (left-)terminate for every ground query.

The cause of the restricted form of completeness of Theorem 3.3 lies in the use of level mappings, which must specify a natural number for every ground atom – hence termination is forced for every ground query. A more subtle problem with using level mappings is that one must specify values also for *uninteresting atoms*, such as $\mathtt{trans}(x, y, e)$ when e is not a DAG. The solution to both problems is to consider *extended* level mappings (Ruggieri, 1997, 1999).

Definition 6.5
An *extended level mapping* is a function $|.| : B_L \rightarrow \mathbb{N}^\infty$ of ground atoms to $\mathbb{N}^\infty$, where $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$.

The inclusion of ∞ in the codomain is intended to model non-termination and uninteresting instances of program clauses. First, we extend the $>$ order on $\mathbb{N}$ to a relation $\rhd$ on $\mathbb{N}^\infty$.

Definition 6.6
We define $n \rhd m$ for $n, m \in \mathbb{N}^\infty$ iff $n = \infty$ or $n > m$. We write $n \unrhd m$ iff $n \rhd m$ or $n = m$.

6.3 Declarative characterisation

Therefore, $\infty \rhd m$ for every $m \in \mathbb{N}^\infty$. With this additional notation we are now ready to introduce (a revised definition of) acceptable programs and queries. A program P is acceptable if for every ground instance of a clause from P, the level of the head is greater than the level of each atom in the body such that the body atoms to its left are true in a Herbrand model of the program.

Definition 6.7
Let $|.|$ be an extended level mapping, and I a Herbrand interpretation. A program P is *acceptable by* $|.|$ *and* I if I is a model of P, and for every $A \leftarrow B_1, \ldots, B_n$ in $ground_L(P)$:

$$\text{for all } i \in [1, n], \quad I \models B_1, \ldots, B_{i-1} \quad \text{implies} \quad |A| \rhd |B_i|.$$

A query Q is *acceptable by* $|.|$ *and* I if there exists $k \in \mathbb{N}$ such that for every $A_1, \ldots, A_n \in ground_L(Q)$:

$$\text{for all } i \in [1, n], \quad I \models A_1, \ldots, A_{i-1} \quad \text{implies} \quad k \rhd |A_i|.$$

Let us compare this definition with the definition of delay-recurrence (Definition 5.7). In the case of local and delay-safe selection rules, an atom cannot be selected before one of its covers is completely resolved. In the case of the LD selection rule, an atom cannot be selected before the atoms to its left are completely

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

resolved. Because of the correctness of LD resolution (Apt, 1997), this explains why, in both cases, a decrease is only required if the instance of the cover, resp. the instance of the atoms to the left, are in some model of the program. We also refer to section 9.1.

Acceptable programs and queries precisely characterise left-termination.

Theorem 6.8 (Apt and Pedreschi, 1993; Ruggieri, 1997)
Let P be a program and Q a query. If P and Q are both acceptable by an extended level mapping $|.|$ and a Herbrand interpretation I, then P and Q left-terminate.

Conversely, if P and Q left-terminate, then there exist an extended level mapping $|.|$ and a Herbrand interpretation I such that P and Q are both acceptable by $|.|$ and I.

6.4 Example

Example 6.9
We will show that TRANSP is acceptable. We have pointed out that in the intended use of the program, e is supposed to be a DAG. We define:

$$\begin{aligned} |\mathtt{trans}(x,y,e)| &= \begin{cases} |e| + 1 + Card\{v \mid x \rightsquigarrow_e v\} & \text{if } e \text{ is a DAG} \\ \infty & \text{otherwise} \end{cases} \\ |\mathtt{member}(x,e)| &= |e| \\ I &= \{\mathtt{trans}(x,y,e) \mid x,y,e \in U_L\} \cup \\ &\quad\ \{\mathtt{member}(x,e) \mid x \text{ is in the list } e\}. \end{aligned}$$

where $Card$ is the set cardinality operator. It is easy to check that TRANSP is acceptable by $|.|$ and I. In particular, consider a ground instance of the second clause:

$$\mathtt{trans}(x,y,e) \leftarrow \mathtt{member}(\mathtt{arc}(x,z),e), \mathtt{trans}(z,y,e).$$

It is immediate to see that I is a model of it. In addition, we have the proof obligations:

$$\begin{array}{lll} \text{(i)} & & |\mathtt{trans}(x,y,e)| \rhd |\mathtt{member}(\mathtt{arc}(x,z),e)| \\ \text{(ii)} & \mathtt{arc}(x,z) \text{ is in } e \;\Rightarrow & |\mathtt{trans}(x,y,e)| \rhd |\mathtt{trans}(z,y,e)|. \end{array}$$

The first one is easy to show since $|\mathtt{trans}(x,y,e)| \rhd |e|$. Considering the second one, we distinguish two cases. If e is not a DAG, the conclusion is immediate. Otherwise, $\mathtt{arc}(x,z)$ in e implies that $Card\{v \mid x \rightsquigarrow_e v\} > Card\{v \mid z \rightsquigarrow_e v\}$, and so:

$$\begin{aligned} |\mathtt{trans}(x,y,e)| &= |e| + 1 + Card\{v \mid x \rightsquigarrow_e v\} \\ &\rhd |e| + 1 + Card\{v \mid z \rightsquigarrow_e v\} = |\mathtt{trans}(z,y,e)|. \end{aligned}$$

Finally, observe that for a DAG e, the queries $\mathtt{trans}(x,\mathtt{Y},e)$ and $\mathtt{trans}(\mathtt{X},\mathtt{Y},e)$ are acceptable by $|.|$ and I. The first one is intended to compute all nodes y such that $x \rightsquigarrow_e y$, while the second one computes the binary relation $\rightsquigarrow_e$. Therefore, the TRANSP program and those queries left-terminate.

Note that this is of course also an example of a program and a query which left-terminate, but do not strongly terminate (e.g. consider the RD selection rule).

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

```
(s)   system(N) ←                          (c)  cons([D|Bs],s(N)) ←
         prod(Bs), cons(Bs,N).                     cons(Bs,N), wait(D).
                                                cons([], 0).
(p1)  prod([s(0)|Bs])) ←
         prod(Bs).                         (w)  wait(s(D)) ←
(p2)  prod([s(s(0))|Bs])) ←                        wait(D).
         prod(Bs).                              wait(0).
      prod([]).
```

Fig. 8. PRODCONS.

7 ∃-termination

So far we have considered four classes of terminating programs, making increasingly strong assumptions about the selection rule, or in other words, considering in each section a smaller set of selection rules. In the previous section we have arrived at a singleton set containing the LD selection rule. Therefore we can clearly not strengthen our assumptions, in the same sense as before, any further.

We will now consider an assumption about the selection rule which is equally abstract as assuming *all* selection rules (section 3). We introduce ∃-*termination* of logic programs, claiming that it is an essential concept for separating the *logic* and *control* aspects of a program.

However, we first motivate the limitations of left-termination.

Example 7.1
The program PRODCONS in figure 8 abstracts a (concurrent) system composed of a producer and a consumer. For notational convenience, we identify the term $\mathtt{s}^n\mathtt{(0)}$ with the natural number n. Intuitively, prod is the producer of a non-deterministic sequence of 1's and 2's, and cons the consumer of the sequence. The shared variable Bs in clause *(s)* acts as an unbounded buffer. The overall system is started by the query system(n). Note that the program is well moded with the obvious mode $\{\mathtt{prod}(O), \mathtt{cons}(I,I), \mathtt{wait}(I)\}$, but assuming LD (and hence, input-consuming) derivations does not ensure termination. The crux is that prod can produce a message sequence of arbitrary length. Now cons can only consume a message sequence of length n, but for this to ensure termination, atoms using cons must be eventually selected. We will see that a selection rule exists for which this program and the query system(n) terminate.

7.1 Operational definition

We introduce next the notion of ∃-termination.

Definition 7.2
A program P and a query Q ∃-terminate if there exists a non-empty set $\mathscr{S}$ of standard selection rules such that P and Q universally terminate w.r.t. $\mathscr{S}$.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

If P and Q do not ∃-terminate, then no standard selection rule can be terminating. For extensions of the standard definition of selection rule, such as input-consuming and delay-safe rules, this is not always true.

Example 7.3
The simple program

```
p(s(X)) ← p(X).
p(X).
```

with mode $\mathtt{p}(I)$ and query p(X) input terminates, but does not ∃-terminate. The same program and query local delay terminate (w.r.t. $|\mathtt{p}(t)| = size(t)$).

In section 9, we will show that *permutation well-modedness* is a sufficient condition to ensure that if P and Q input terminate then they ∃-terminate.

Here, we observe that ∃-termination coincides with universal termination w.r.t. the set of fair selection rules. Therefore, any fair selection rule is a terminating control for any program and query for which a terminating control exists.

Theorem 7.4 (Ruggieri, 2001; Ruggieri, 1999)
A program P and a query Q ∃-terminate iff they universally terminate w.r.t. the set of fair selection rules.

Concerning Example 7.1, it can be said that viewed as a concurrent system, the program inherently relies on fairness for termination.

7.2 Declarative characterisation

Ruggieri (2001; 1999) offers a characterisation of ∃-termination using the notion of *fair-bounded* programs and queries. Just as Definition 6.7, it is based on *extended* level mappings.

Definition 7.5
Let $|.|$ be an extended level mapping, and I a Herbrand interpretation. A program P is *fair-bounded by* $|.|$ *and* I if I is a model of P such that for every $A \leftarrow B_1, \ldots, B_n$ in $ground_L(P)$:

(a) $I \models B_1, \ldots, B_n$ implies that for every $i \in [1, n]$, $|A| \rhd |B_i|$, and
(b) $I \not\models B_1, \ldots, B_n$ implies that there exists $i \in [1, n]$ with $I \not\models B_i \wedge |A| \rhd |B_i|$.

A query Q is *fair-bounded by* $|.|$ *and* I if there exists $k \in \mathbb{N}$ such that for every $A_1, \ldots, A_n \in ground_L(Q)$:

(a) $I \models A_1, \ldots, A_n$ implies that for every $i \in [1, n]$, $k \rhd |A_i|$, and
(b) $I \not\models A_1, \ldots, A_n$ implies that there exists $i \in [1, n]$ with $I \not\models A_i \wedge k \rhd |A_i|$.

Note that the hypotheses of conditions (a) and (b) are *mutually exclusive*.

Let us discuss in more detail the meaning of proof obligations (a) and (b) in Definition 7.5. Consider a ground instance $A \leftarrow B_1, \ldots, B_n$ of a clause.

If the body $B_1, \ldots, B_n$ is true in the model I, then there might exist a SLD-refutation for it. Condition (a) is then intended to bound the length of the refutation.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

If the body is not true in the model I, then it cannot have a refutation. In this case, termination actually means that there is an atom in the body that has a finitely failed SLD-tree. Condition (b) is then intended to bound the depth of the finitely failed SLD-tree. As a consequence of this, the complement of I is necessarily included in the finite failure set of the program.

Compared to acceptability, the model and the extended level mapping in the proof of fair-boundedness have to be chosen more carefully, due to more binding proof obligations. As we will see in section 9, however, the simpler proof obligations of recurrence and acceptability are sufficient conditions for proving fair-boundedness. Note also that, as in the case of acceptable programs, the inclusion of ∞ in the codomain of extended level mapping allows for excluding *unintended atoms* and *non-terminating atoms* from the termination analysis. In fact, if $|A| = \infty$ then (a, b) in Definition 7.5 are trivially satisfied.

Fair-bounded programs and queries precisely characterise $\exists$-termination, i.e. the class of logic programs and queries for which a terminating control exists.

Theorem 7.6 (Ruggieri, 2001; Ruggieri, 1999)
Let P be a program and Q a query.

If P and Q are both fair-bounded by an extended level mapping $|.|$ and a Herbrand interpretation I, then P and Q $\exists$-terminate.

Conversely, if P and Q $\exists$-terminate, then there exist an extended level mapping $|.|$ and a Herbrand interpretation I such that P and Q are both fair-bounded by $|.|$ and I.

7.3 Example

Example 7.7
The PRODCONS program is fair-bounded. First, we introduce the *list-max* norm:

$$\begin{array}{rcll} lmax(f(x_1,\ldots,x_n)) & = & 0 & \text{if } f \neq [\,.\,|\,.\,] \\ lmax([x|xs]) & = & max\{lmax(xs), size(x)\} & \text{otherwise.} \end{array}$$

Note that for a ground list xs, $lmax(xs)$ equals the maximum size of an element in xs. Then we define:

$$\begin{array}{rcl} |\mathtt{system}(n)| & = & size(n)+3 \\ |\mathtt{prod}(bs)| & = & |bs| \\ |\mathtt{cons}(bs,n)| & = & \left\{ \begin{array}{ll} size(n)+lmax(bs) & \text{if } I \models \mathtt{cons}(bs,n) \\ size(n) & \text{if } I \not\models \mathtt{cons}(bs,n) \end{array} \right. \\ |\mathtt{wait}(t)| & = & size(t) \\ I & = & \{\mathtt{system}(n) \mid n \in U_L\} \cup \{\mathtt{prod}(bs) \mid lmax(bs) \leqslant 2\} \cup \\ & & \{\mathtt{cons}(bs,n) \mid |bs| = size(n)\} \cup \{\mathtt{wait}(x) \mid x \in U_L\}. \end{array}$$

Let us show the proof obligations of Definition 7.5. Those for unit clauses are trivial. Consider now the recursive clauses *(w), (c), (p1), (p2),* and *(s).*

(w). I is obviously a model of *(w)*. In addition, $|\mathtt{wait}(\mathtt{s}(d))| = size(d)+1 \rhd size(d) = |\mathtt{wait}(d)|$. This implies (a, b).

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

(c). Consider a ground instance $\texttt{cons}([d|bs], \texttt{s}(n)) \leftarrow \texttt{cons}(bs, n),\ \texttt{wait}(d)$ of *(c)*. If $I \models \texttt{cons}(bs, n),\ \texttt{wait}(d)$, then $|bs| = size(n)$, and so

$$|[d|bs]| = |bs| + 1 = size(n) + 1 = size(\texttt{s}(n)),$$

i.e. $I \models \texttt{cons}([d|bs], \texttt{s}(n))$. Therefore, I is a model of *(c)*. Let us show proof obligations (a, b) of Definition 7.5.

(a) Suppose that $I \models \texttt{cons}(bs, n),\ \texttt{wait}(d)$. We have already shown that $I \models \texttt{cons}([d|bs], \texttt{s}(n))$. We calculate:

$$\begin{array}{rcl} |\texttt{cons}([d|bs], \texttt{s}(n))| & = & size(n) + 1 + max\{lmax(bs), size(d)\} \\ & \rhd & size(n) + lmax(bs) = |\texttt{cons}(bs, n)| \\ |\texttt{cons}([d|bs], \texttt{s}(n))| & = & size(n) + 1 + max\{lmax(bs), size(d)\} \\ & \rhd & size(d) = |\texttt{wait}(d)|. \end{array}$$

These two inequalities show that (a) holds.

(b) If $I \not\models \texttt{cons}(bs, n),\ \texttt{wait}(d)$, then necessarily $I \not\models \texttt{cons}(bs, n)$. Therefore

$$\begin{array}{rcl} |\texttt{cons}([d|bs], \texttt{s}(n))| & \trianglerighteq & size(n) + 1 \\ & \rhd & size(n) = |\texttt{cons}(bs, n)|, \end{array}$$

and so we have (b). Recall that (b) states that the depth of the finitely failed SLD-tree must be bounded. In fact, it is the decrease of the 'counter', the second argument of cons, which in this case bounds the depth of the SLD-tree.

(p1,p2). I is obviously a model of *(p1)*. Moreover we have

$$|\texttt{prod}([\texttt{s}(0)|bs])| = |bs| + 1 \ \rhd\ |bs| \ = \ |\texttt{prod}(bs)|,$$

which implies (a) and (b). The reasoning for *(p2)* is analogous.

(s). Consider a ground instance $\texttt{system}(n) \leftarrow \texttt{prod}(bs),\ \texttt{cons}(bs, n)$ of *(s)*. Obviously I is a model of *(s)*. Let us show *(a,b)*.

(a) Suppose that $I \models \texttt{prod}(bs),\ \texttt{cons}(bs, n)$. This implies $lmax(bs) \leqslant 2$ and $|bs| = size(n)$. These imply:

$$\begin{array}{rcl} |\texttt{system}(n)| = size(n) + 3 & \rhd & |bs| = |\texttt{prod}(bs)| \\ |\texttt{system}(n)| = size(n) + 3 & \rhd & size(n) + lmax(bs) = |\texttt{cons}(bs, n)|. \end{array}$$

These two inequalities show (a).

(b) Suppose that $I \not\models \texttt{prod}(bs),\ \texttt{cons}(bs, n)$. Intuitively, this means that $\texttt{prod}(bs),\ \texttt{cons}(bs, n)$ has no refutation. We distinguish two cases. If $I \not\models \texttt{cons}(bs, n)$ ($\texttt{cons}(bs, n)$ has no refutation) then:

$$|\texttt{system}(n)| \ = \ size(n) + 3 \ \rhd\ size(n) \ = \ |\texttt{cons}(bs, n)|,$$

i.e. the depth of the SLD-tree of $\texttt{cons}(bs, n)$ is bounded (hence, the SLD-tree is

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

```
% even(X) ←
%     X is an even natural number.

  even(s(X)) ← odd(X).
  even(0).

% odd(X) ←
%     X is an odd natural number.

  odd(s(X)) ← even(X).
```

Fig. 9. ODDEVEN.

finitely failed). If $I \models$ cons(bs, n) and $I \not\models$ prod(bs) (prod(bs) has no refutation) then $|bs| = size(n)$, which implies:

$$|\texttt{system}(n)| = size(n) + 3 \rhd |bs| = |\texttt{prod}(bs)|,$$

i.e. the depth of the SLD-tree of prod(bs) is bounded.

We conclude this example by noting that for every $n \in \mathbb{N}$ the query system(n) is fair-bounded by $|.|$ and I, and so every fair SLD-derivation of PRODCONS and system(n) is finite.

8 Bounded nondeterminism

In the previous section, we have made the strongest possible assumption about the selection rule, in that we considered programs and queries for which there *exists* a terminating control. In general, a terminating control may not exist. Even in this case however, all is not lost. If we can establish that a program and query have only finitely many successful derivations, then we can transform the program so that it terminates.

Example 8.1
The program ODDEVEN in figure 9 defines the even and odd predicates, with the usual intuitive meaning. The query even(X), odd(X) is intended to check whether there is a number that is both even and odd. It is readily checked that ODDEVEN and the query do not ∃-terminate. However, ODDEVEN and the query have only finitely many, namely 0, successful derivations.

8.1 Operational definition

Pedreschi and Ruggieri (1999a) propose the notion of *bounded nondeterminism* to model programs and queries with finitely many refutations.

Definition 8.2
A program P and query Q have *bounded nondeterminism* if for every standard selection rule s there are finitely many SLD-refutations of P and Q via s.

By the Switching Lemma (Apt, 1997), each refutation via some standard selection rule is isomorphic to some refutation via any other standard selection rule. Therefore, bounded nondeterminism could have been defined by requiring finitely many SLD-refutations of P and Q via *some* standard selection rule. Also, note that, while

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

bounded nondeterminism implies that there are finitely many refutations also for non-standard selection rules, the converse implication does not hold, in general (see Example 7.3).

Bounded nondeterminism, although not being a notion of termination in the strict sense, is closely related to termination. In fact, if P and Q ∃-terminate, then P and Q have bounded nondeterminism. Conversely, if P and Q have bounded nondeterminism then there exists an upper bound for the length of the SLD-refutations of P and Q. If the upper bound is known, then we can syntactically transform P and Q into an equivalent program and query that strongly terminate, i.e. any selection rule will be a terminating control for them. Note that this transformation is even interesting for programs and queries that ∃-terminate, since few existing systems adopt fair selection rules. In addition, even if we adopt a selection rule that ensures termination, we may apply the transformation to prune the SLD-tree from unsuccessful branches.

8.2 Declarative characterisation

In the following, we present a declarative characterisation of programs and queries that have bounded nondeterminism, by introducing the class of *bounded* programs and queries. Just as Definitions 6.7 and 7.5, it is based on *extended* level mappings.

Definition 8.3
Let $|.|$ be an extended level mapping, and I a Herbrand interpretation. A program P is *bounded by* $|.|$ *and* I if I is a model of P such that for every $A \leftarrow B_1, \dots, B_n$ in $ground_L(P)$:

$$I \models B_1, \dots, B_n \text{ implies that for every } i \in [1, n],\ |A| \rhd |B_i|.$$

A query Q is *bounded by* $|.|$ *and* I if there exists $k \in \mathbb{N}$ such that for every $A_1, \dots, A_n \in ground_L(Q)$:

$$I \models A_1, \dots, A_n \text{ implies that for every } i \in [1, n],\ k \rhd |A_i|.$$

It is straightforward to check that the definition of bounded programs is a simplification of Definition 7.5 of fair-bounded programs, where proof obligation (b) is discarded. Intuitively, the definition of boundedness only requires the decreasing of the extended level mapping when the body atoms are true in some model of the program, i.e. they might have a refutation.

Bounded programs and queries precisely characterise the notion of bounded nondeterminism.

Theorem 8.4 (Pedreschi and Ruggieri, 1999a; Ruggieri, 1999)
Let P be a program and Q a query.

If P and Q are both bounded by an extended level mapping $|.|$ and a Herbrand interpretation I, then P and Q have bounded nondeterminism.

Conversely, if P and Q have bounded nondeterminism, then there exist an extended level mapping $|.|$ and a Herbrand interpretation I such that P and Q are both bounded by $|.|$ and I.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

8.3 Examples

Example 8.5

Consider again the ODDEVEN program. It is readily checked that it is bounded by defining:

$$|\mathtt{even}(x)| = |\mathtt{odd}(x)| = size(x)$$
$$I = \{\mathtt{even}(\mathtt{s}^{2\cdot i}(0)),\ \mathtt{odd}(\mathtt{s}^{2\cdot i+1}(0)) \mid i \geqslant 0\}.$$

The query even(X), odd(X) is bounded by $|.|$ and I. In fact, since no instance of it is true in I, Definition 8.3 imposes no requirement. Therefore, ODDEVEN and the query above have bounded nondeterminism.

Generally, for a query that has no instance in a model of the program (it is *unsolvable*), the k in Definition 8.3 can be chosen as 0. An automatic method to check whether a query (at a node of a SLD-tree) is unsolvable has been proposed by Bruynooghe *et al.* (1998). Of course, the example is somewhat a limit case, since one does not even need to run a query if it has been shown to be unsolvable. However, we have already mentioned that the benefits of characterising bounded nondeterminism also apply to programs and queries belonging to the previously introduced classes. In addition, it is still possible to devise an example program and a *satisfiable* query that do not ∃-terminate but have bounded nondeterminism.

Example 8.6

We now define the predicate all such that the query all(n_0, n_1, Xs) collects in Xs the answers of a query q(m, A) for values m ranging from n_0 to n_1.

```
all(N,N,[A]) ← q(N,A).
all(N,N1,[A|As]) ← q(N,A), all(s(N),N1,As).
q(Y, Y). %just as an example
```

The program and the query all(0,s(s(0)),As) do not ∃-terminate, but they have only one computed answer, namely As = [0,s(0),s(s(0))]. The program and the query are bounded (and thus have bounded nondeterminism) by defining:

$$|\mathtt{all}(n,m,x)| = max\{size(m) - size(n), 0\} + 1$$
$$|\mathtt{q}(x,y)| = 0$$
$$I = \{\mathtt{all}(n,m,x) \mid size(n) \leqslant size(m)\} \cup$$
$$= \{\mathtt{q}(x,y)\}.$$

9 Relations between classes

We have introduced six classes of programs and queries, which provide declarative characterisations of operational notions of universal termination and bounded nondeterminism. In this section we summarise the relationships between these classes.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Table 1. *Comparison of characterisations.*

	only ground?	only recursive?	uses model?	query oblig.?	∞ in codomain?	neg. model info.?
boundedness	yes	no	yes	yes	yes	no
fair-boundedness	yes	no	yes	yes	yes	yes
acceptability	yes	no	yes	yes	yes	no
delay-recurrence	yes	no	yes	no	no	no
simply-acceptability	no	yes	yes	no	no	no
recurrence	yes	no	no	yes	no	n.a.

9.1 Comparison of characterisations

We now try to provide an intuitive understanding of the significance of the technical differences between the characterisations of termination we have proposed. These are summarised in table 1.

The first difference concerns the question of whether a decrease is defined for all ground instances of a clause, or rather for instances specified in some other way. All characterisations, except simply-acceptability, require a decrease for all ground instances of a clause. One cannot clearly say that this difference lies in the nature of the termination classes themselves: the first characterisation of input-termination by Smaus (1999b) also required a decrease for the ground instances of a clause, just as there are characterisations of left-termination (Bossi *et al*., 1994; De Schreye *et al*., 1992) based on generalised level mappings and hence non-ground instances of clauses. However, one can say that our characterisation of input-termination inherently relies on measuring the level of non-ground atoms, which may change via further instantiation. Nevertheless, this instantiation is not arbitrary: it is controlled by the fact that derivations are input-consuming and the programs are simply moded. This is reflected in the condition that a decrease holds for all simply-local instantiations of a clause.

The second difference concerns the question of whether a decrease is required for recursive body atoms only, or whether recursion plays no role. Simply-acceptability is the only characterisation that requires a decrease for recursive body atoms only. We attribute this difference essentially to the explicit use of modes. Broadly speaking, modes restrict the data flow of a program in a way that allows for termination proofs that are inherently *modular*. Therefore one does not explicitly require a decrease for non-recursive calls, but rather one requires that for the predicate of the non-recursive call, termination has already been shown (independently). To support this explanation, we refer to Etalle *et al*. (1999): there left-termination for *well moded* programs is shown, using *well-acceptability*. Well-acceptability requires a decrease only for recursive body atoms.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

The third difference concerns the question of whether the method relies on (some kind of) models or not. It is not surprising that a method for showing strong termination cannot rely on models: one cannot make any assumptions about certain atoms being resolved before an atom is selected. However, the original methods of showing termination for input-consuming derivations were also not based on models (Smaus, 1999b; Bossi *et al*., 1999), and it was noted that the principle underlying the use of models in proofs of left-termination cannot be easily transferred to input termination. By restricting to simply moded programs and defining a special notion of model, this was nevertheless achieved. For a clause $H \leftarrow A_1, \ldots, A_n$, assuming that A_i is the selected atom, we exploited that provided that programs and queries are simply moded, we know that even though $A_1, \ldots, A_{i-1}$ may not be resolved completely, $A_1, \ldots, A_{i-1}\theta$ will be in any 'partial model' of the program.

The fourth difference concerns the question of whether proof obligations are imposed on queries. Delay-recurrence and simply-acceptability are the characterisations that impose no proof obligations for queries (except that in the latter case, the query must be simply moded). The reason is that the restrictions on the selectability of an atom, which depends on the degree of instantiation, take the role of such a proof obligation.

The fifth difference concerns the question of whether ∞ is in the codomain of level mappings. This is the case for acceptability, fair-boundedness and boundedness. In all three cases, this allows for excluding *unintended atoms* and *non-terminating atoms* from the termination analysis. For an atom A with $|A| = \infty$ the proof obligations are trivially satisfied. Also, the use of ∞ allows to achieve full completeness of the characterisation.

A final difference concerns the way information on data flow (modes, simply-local models, covers, Herbrand models) is used in the declarative characterisations. For recurrence this is not applicable. Apart from that, in all except fair-boundedness, such information is used only in a 'positive' way, i.e. 'if . . . *is* in the model then . . . '. In fair-boundedness, it is also used in a 'negative' way, namely 'if . . . *is not* in the model then . . . '. Intuitively, in all characterisation, except fair-boundedness, the relevant part of the information concerns a characterisation of atoms that are logical consequences of the program. In fair-boundedness, it is also relevant the characterisation of atoms that are not logical consequences, since for those atoms we must ensure finite failure.

9.2 From strong termination to bounded nondeterminism

In this subsection, we show inclusions between the introduced classes, i.e. we justify each arrow in figure 1. We first leave aside input termination and local delay termination, since for these classes, the comparison is much less clearcut.

Looking at the four remaining classes from an operational point of view, we note that strong termination of a program and a query implies left-termination, which in turn implies $\exists$-termination, which in turn implies bounded nondeterminism. Examples 6.9, 7.1 and 8.1 show that these implications are strict.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Since the declarative characterisations of those notions are sound and complete, the same strict inclusions hold among recurrence, acceptability, fair-boundedness and boundedness. This allows for reusing or simplifying termination proofs.

Theorem 9.1
Let P be a program and Q a query, $|.|$ an extended level mapping and I a Herbrand model of P. Each of the following statements strictly implies the statements below it:

(i) P and Q are recurrent by $|.|$,
(ii) P and Q are acceptable by $|.|$ and I,
(iii) P and Q are fair-bounded by $|.|$ and I,
(iv) P and Q are bounded by $|.|$ and I.

In the following example, we show how the above theorem allows for reuse of termination proofs.

Example 9.2
In Example 6.9 we showed that the `TRANSP` program is acceptable by a level mapping $|.|$ and a model I. The proof obligations of acceptability had to be shown for every clause of the program.

However, we note that the clauses defining the predicate `member` are a sub-program which is readily checked to be recurrent by the same $|.|$. By Theorem 9.1, we conclude that the proof obligations for clauses defining `member` are satisfied for every Herbrand model of `TRANSP` and thus in particular for I.

We refer the reader to Apt and Pedreschi (1994) for a collection of results on reuse of proofs of recurrence to show acceptability, and on proving acceptability of $P \cup P'$ by reusing separate proofs for P and P'.

Consider now local delay termination. Obviously, it is implied by strong termination. However, we have observed with the programs and queries of Examples 6.3 and 7.3 that local delay termination does not imply left-termination or ∃-termination, in general. These results can be obtained under reasonable assumptions, which, in particular, rule out deadlock.

The following proposition relates local delay termination with ∃-termination.

Proposition 9.3
Let P and Q be a permutation well moded program and query, and $|.|$ a moded level mapping.

If P and Q local delay terminate (w.r.t. $|.|$) then they ∃-terminate.

If P is delay-recurrent by $|.|$ and some Herbrand interpretation then P and Q are fair-bounded by some extended level mapping and Herbrand interpretation.

Proof
Since P and Q are permutation well moded, every query Q' in a derivation of P and Q is permutation well moded (Smaus, 1999a), i.e., there exists a permutation $\tilde{Q}'$ of Q' which is well moded. By Definition 2.2, the leftmost atom in $\tilde{Q}'$ is ground in its input positions and hence bounded w.r.t. $|.|$. Consider the selection rule which

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

always selects this 'leftmost' (modulo the permutation) atom. This selection rule is local and delay-safe, and it is a standard selection rule (since there is always a selected atom). Therefore, local delay termination implies ∃-termination.

Concerning the second claim, since fair-boundedness is a complete characterisation of ∃-termination, we have the conclusion. □

The next proposition relates local delay termination with left-termination. In this case, programs must be well moded, not just *permutation* well moded. The proof is similar to the previous one but simpler.

Proposition 9.4

Let P and Q be a well moded program and query, and $|.|$ a moded level mapping.

If P and Q local delay terminate (w.r.t. $|.|$) then they left-terminate.

If P is delay-recurrent by $|.|$ and some Herbrand interpretation then P and Q are acceptable by some extended level mapping and Herbrand interpretation.

Marchiori and Teusink (1999) propose a program transformation such that the original program is delay-recurrent iff the transformed program is acceptable. This transformation allows us to use automated proof methods originally designed for acceptability for the purpose of showing delay-recurrence.

Consider now input termination. As before, it is implied by strong termination. However, as observed in Examples 5.4, 6.3 and 7.3, input termination does not imply local delay termination, left-termination, or ∃-termination, in general. Again, these results can be obtained under reasonable assumptions.

The following proposition relates input termination to ∃-termination.

Proposition 9.5

Let P and Q be a permutation well moded program and query. If P and Q input terminate then they ∃-terminate.

Let P and Q be a permutation well and simply moded program and query. If P is simply-acceptable by some $|.|$ and I then P and Q are fair-bounded by some extended level mapping and Herbrand interpretation.

Proof

Since P and Q are permutation well moded, every query Q' in a derivation of P and Q is permutation well moded (Smaus, 1999a), and so Q' contains an atom that is ground in its input position. The selection rule s that always selects this atom together with all program clauses is an input-consuming selection rule, and also a standard selection rule. Therefore, input termination implies universal termination w.r.t. $\{s\}$ and hence ∃-termination.

Concerning the second claim, by Theorem 4.10, P and Q input terminate. As shown above, this implies that they ∃-terminate. Since fair-boundedness is a complete characterisation of ∃-termination, we have the conclusion. □

The next proposition gives a direct comparison between input and left-termination. The proof is similar to the previous one.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Proposition 9.6

Let P and Q be a well moded program and query. If P and Q input terminate then they left-terminate.

Let P and Q be a well and simply moded program and query. If P is simply-acceptable by some $|.|$ and I then P and Q are acceptable by some extended level mapping and Herbrand interpretation.

To relate input termination to local delay termination, we introduce a notion that relates delay-safe derivations with input-consuming derivations, based on an a similar concept from Apt and Luitjes (1995).

Definition 9.7

Let P be a program and $|.|$ a moded level mapping.

We say that $|.|$ *implies matching* (w.r.t. $|.|$) if for every atom $A = p(\mathbf{s}, \mathbf{t})$ bounded w.r.t. $|.|$ and for every $B = p(\mathbf{v}, \mathbf{u})$ head of a renaming of a clause from P which is variable-disjoint with A, if A and B unify, then $\mathbf{s}$ is an instance of $\mathbf{v}$.

Note that, in particular, $|.|$ implies matching if every atom bounded by $|.|$ is ground in its input positions.

Proposition 9.8

Let P and Q be a permutation simply moded program and query, and $|.|$ a moded level mapping that implies matching.

If P and Q input terminate then they local delay terminate (w.r.t. $|.|$).

Proof

The conclusion follows by showing that any derivation of P and any permutation simply moded query Q' via a local delay-safe selection rule (w.r.t. $|.|$) is also a derivation via an input-consuming selection rule. So, let s be a local delay-safe selection rule and Q' a permutation simply-well moded query such that s selects atom $A = p(\mathbf{s}, \mathbf{t})$. Then by Definition 9.7, for each $B = p(\mathbf{v}, \mathbf{u})$, head of a renaming of a clause from P, if A and B unify, then $\mathbf{s}$ is an instance of $\mathbf{v}$, i.e. $\mathbf{s} = \mathbf{v}\theta$ for some substitution θ such that $dom(\theta) \subseteq Vars(\mathbf{v})$. By (Apt and Luitjes, 1995, Corollary 31), this implies that the resolvent of Q' and any clause in P is again permutation simply moded. Moreover, by applying the unification algorithm (Apt, 1997), it is readily checked that, if A and B unify, then $\sigma = \theta \cup \{\mathbf{t}/\mathbf{u}\theta\}$ is an mgu. Permutation simply-modedness implies that $\mathbf{s}$ and $\mathbf{t}$ are variable-disjoint. Moreover, $\mathbf{s}$ and $\mathbf{v}$ are variable-disjoint. This implies that $Dom(\sigma) \cap Vars(\mathbf{s}) = \emptyset$, and so the derivation step is input-consuming.

By repeatedly applying this argument to all queries in the SLD-derivation of P and Q via s, it follows that the derivation is via some input-consuming selection rule. □

It remains an open question whether simply-acceptability implies delay-recurrence under some general hypotheses. The problem with showing such a result lies in the fact that delay-recurrence is a sufficient but not necessary condition for local delay termination.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Example 9.9
Consider again the program and the level mapping $|.|$ of Example 5.10. We have already observed that the program and any query local delay terminate.

In addition, given the mode $\{\mathtt{p}(O),\ \mathtt{q}(I),\ \mathtt{r}(I)\}$, it is readily checked that the program is simply moded, and that the level mapping is moded and implies matching. Also, note that the program is simply-acceptable by $|.|$ and any simply-local model.

However, this is not sufficient to show that the program is delay-recurrent, as proved in Example 5.10. Intuitively, the problem with showing delay-recurrence lies in the fact that the notion of cover does not appropriately describe the data flow in this program given by the modes.

9.3 From bounded nondeterminism to strong termination

Consider now a program P and a query Q which either do not universally terminate for a set of selection rules in question, or simply for which we (or our compiler) fail to *prove* termination. We have already mentioned that, if P and Q have bounded nondeterminism then there exists an upper bound for the length of the SLD-refutations of P and Q. If the upper bound is known, then we can syntactically transform P and Q into an equivalent program and query that strongly terminate. As shown by Pedreschi and Ruggieri (1999a), such an upper bound is related to the natural number k of Definition 8.3 of bounded queries. As in our notation for moded atoms, we use boldface letters to denote vectors of (possibly non-ground) terms.

Definition 9.10
Let P be a program and Q a query both bounded by $|.|$ and I, and let $k \in \mathbb{N}$. We define $Ter(P)$ as the program such that:

- for every clause $p_0(\mathbf{t}_0) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n)$ in P, with $n > 0$, the clause

$$p_0(\mathbf{t}_0, \mathtt{s}(\mathtt{D})) \leftarrow p_1(\mathbf{t}_1, \mathtt{D}), \ldots, p_n(\mathbf{t}_n, \mathtt{D})$$

  is in $Ter(P)$, where $\mathtt{D}$ is a fresh variable,
- and, for every clause $p_0(\mathbf{t}_0)$ in P, the clause

$$p_0(\mathbf{t}_0, \_) \leftarrow$$

  is in $Ter(P)$.

Also, for the query $Q = p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n)$, we define $Ter(Q, k)$ as the query

$$p_1(\mathbf{t}_1, \mathtt{s}^k(0)), \ldots, p_n(\mathbf{t}_n, \mathtt{s}^k(0))$$

The transformed program relates to the original one as shown in the following theorem.

Theorem 9.11 (Pedreschi and Ruggieri, 1999a; Ruggieri, 1999)
Let P be a program and Q a query both bounded by $|.|$ and I, and let k be a given natural number satisfying Definition 8.3.

Then, for every $n \in \mathbb{N}$, $Ter(P)$ and $Ter(Q, n)$ strongly terminate.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Moreover, there is a bijection between SLD-refutations of P and Q via a selection rule s and SLD-refutations of $Ter(P)$ and $Ter(Q, k-1)$ via s.

The intuitive reading of this result is that the transformed program and query maintain the same *success semantics* of the original program and query. Note that no assumption is made on the selection rule s, i.e. any selection rule is a terminating control for the transformed program and query.

Example 9.12
Reconsider the program `ODDEVEN` and the query $Q =$ `even(X), odd(X)` of Example 8.1. The transformed program Ter(`ODDEVEN`) is:

```
even(s(X),s(D)) ← odd(X,D).
even(0,_).

odd(s(X),s(D)) ← even(X,D).
```

and the transformed query $Ter(Q, k-1)$ for $k = 3$ is `even(X,s`^{2}`(0)),odd(X,s`^{2}`(0))`. By Theorem 9.11, the transformed program and query terminate for *any* selection rule, and the semantics w.r.t. the original program is preserved modulo the extra argument added to each predicate.

The transformations $Ter(P)$ and $Ter(Q, k)$ are of purely theoretical interest. In practice, one would implement these counters directly into the compiler/interpreter. Also, the compiler/interpreter should include a module that infers an upper bound k automatically. Approaches to the automatic inference of level mappings and models are briefly recalled in the next section. Pedreschi and Ruggieri (1999a) give an example showing how the approach of Decorte *et al.* (1999) could be rephrased to infer boundedness.

10 Related work

The survey on termination of logic programs by De Schreye and Decorte (1994) covers most work in this area until 1994. The authors distinguish three types of approaches: the ones that express necessary and sufficient conditions for termination, the ones that provide decidable *sufficient* conditions, and the ones that prove decidability or undecidability for subclasses of programs and queries. Under this classification, our survey falls in the first type. In the following, we mainly mention works published since 1994. We group the works according to the main focus or angle they take.

10.1 Other characterisations of left-termination

Apt and Pedreschi (1994) refined acceptability to make the method *modular*. Here, modularity means that the termination proof for a program $P \cup P'$ can be obtained from separate termination proofs for P and P'. Also, in Apt *et al.* (1994), acceptability is extended to reason on first-order built-in's of Prolog.

Etalle *et al.* (1999) propose a refinement of acceptability (called *well-acceptability*)

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

for well moded programs and queries. The requirement of well-modedness simplifies proofs of acceptability. On the one hand, no proof obligation is imposed on the *queries*. On the other hand, the decrease of the level mapping is now required only from the head to the mutually recursive clause body atoms. It is interesting to observe that the definition of well-acceptability is then very close to simply-acceptability. Actually, well-modedness of a program and a query implies that atoms selected by the LD selection rule are ground in their input positions, hence a derivation via the LD selection rule is input-consuming.

Serebrenik and De Schreye (2001) show that, when restricting to well moded programs and queries and moded level mappings (they call them *output-independent*), acceptability can be generalised by having any well-founded ordering, not necessarily $\mathbb{N}$, as co-domain of level mappings. This simplifies the proof of programs where complex level mappings may be required.

Also, a characterisation of acceptability in the context of metric spaces was provided by Hitzler and Seda (1999).

Alternative characterisations of left-termination consider proof obligations on generalised level mappings and thus on possibly non-ground instances of clauses and queries. Bossi *et al.* (1994) provide sufficient and necessary conditions that involve: (1) generalised level mappings (with an arbitrary well-founded ordering as the codomain) that do not increase w.r.t. substitutions; (2) a specification (Pre, $Post$), with $Pre, Post \subseteq Atom_L$, which is intended to characterise call patterns (Pre) and correct instances ($Post$) of atomic queries. Call patterns provide information on the structure of selected atoms, while correct instances provide information on data flow. The method has the advantage of reasoning both on termination and on partial correctness within the same framework. However, proof obligations are not well suited for *paper & pencil* proofs, since they require to reason on the strongly connected components of a graph abstracting the flow of control of the program under consideration. An adaption of acceptability to total correctness is presented in Pedreschi and Ruggieri (1999c). Also, we mention the works of Bronsard *et al.* (1992) and Deransart and Małuszyński (1993), which rely on partial correctness or typing information to characterise call patterns. Deransart and Małuszyński generalise the proof obligations on the left-to-right order of the LD selection rule to any acyclic ordering of body atoms. Another characterisation of left-termination particularly suited for automation is due to De Schreye *et al.* (1992; 1999). Their notion is similar to the one of Bossi *et al.*, but it uses: (1) generalised level mappings that are constant w.r.t. substitution (called *rigid* level mappings); (2) a pair (Pre, $Post$), with $Pre, Post \subseteq Atom_L$, where $Post$ is a model of the program and Pre is a characterisation of call patterns computed using abstract interpretation.

A generalisation of the definition of left-termination considers a program together with a *set* of queries (De Schreye *et al.*, 1992; Bossi *et al.*, 1994), while we considered a program and a single query. We say that a program P and a set of queries $\mathcal{Q}$ left-terminate if every derivation for P and any $Q \in \mathcal{Q}$ via the leftmost selection rule is finite. The benefit of such a definition consists of having just one single proof of termination for a set of queries rather than a set of proofs, one for each query in the set. However, we observe that in our examples on acceptability, proofs can

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

easily be generalised to a set of queries. For instance, for a level mapping such that $|\mathtt{p}(t)| = |t|$, it is immediate to conclude that all queries $\mathtt{p}(T)$, where T is a list, are acceptable. Conversely, is it the case that if P and $\mathcal{Q}$ left-terminate then P and any $Q \in \mathcal{Q}$ are acceptable by a same $|.|$ and I? The answer is affirmative. In fact, from the proof of the Completeness Theorem 6.8 (Ruggieri, 1999, Theorem 2.3.20), if P and Q left-terminate then they are acceptable by a level mapping $|.|_P$ and a Herbrand model I_P that *only* depend on P. This implies that every $Q \in \mathcal{Q}$ is acceptable by $|.|_P$ and I_P. In conclusion, acceptability by $|.|_P$ and I_P precisely characterises the maximal set $\mathcal{Q}$ such that P and $\mathcal{Q}$ left-terminate.

Finally, instead of considering left-termination of a program P and a query Q, one may be interested in proving left-termination of some permutation P' and Q' of them. A permutation of P (Q) is any program (query) obtained by reordering clause body atoms in P (Q). This notion is called σ-termination in Hoarau and Mesnard (2001), where a system for automatic inference is presented. σ-termination is strictly weaker than left-termination, and strictly stronger than $\exists$-termination (e.g. program PRODCONS in figure 8 and $\mathtt{system}(n)$, with $n \in \mathbb{N}$, $\exists$-terminate but do not σ-terminate).

10.2 Writing left-terminating programs

There are also works that are not directly concerned with proving an existing program left-terminating, but rather with heuristics and transformations that help write left-terminating programs.

Hoarau and Mesnard (1998) studied inferring and compiling termination for (constraint) logic programs. *Inferring* termination means inferring a set of queries for which a program 'potentially' terminates, that is to say, it terminates after possible reordering of atoms. This phase uses abstract interpretation and the Boolean μ-calculus. *Compiling* termination means reordering the body atoms so that the program terminates. The method is implemented.

Neumerkel and Mesnard (1999) studied the problem of localising and explaining reasons for nontermination in a logic programs. The work aims at assisting programmers in writing terminating programs and helping them to *understand* why their program does not terminate. The method has been implemented and is intended as a debugging tool, in particular for beginners (it has been used for teaching purposes). The idea is to localise a fragment of a program that is in itself already non-terminating, and hence constitutes an explanation for non-termination of the whole program.

10.3 Transformational approaches

It is possible to investigate termination of logic programs by transforming them to some other formal system. If the transformation preserves termination, one can resort to the compendium of techniques of those formal systems for the purpose of proving termination of the original logic program.

Baudinet (1992) considered transforming logic programs into functional programs. Termination of the transformed programs can then be studied by structural induc-

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

tion. Her approach covers general logic programs, existential termination and the effects of Prolog cut.

There is a considerable amount of literature on transforming logic programs to term rewriting systems (TRSs), which are perhaps the generic formalism for studying termination as such. It is very common in these transformational approaches to use modes. The intuitive idea is usually that the input of an atom has to rewrite into the output of that atom. Most of those works assume the left-to-right selection rule. One valuable exception is due to Krishna Rao *et al.* (1998), where termination is considered w.r.t. selection rules that respect a producer-consumer relation among variables in clauses. Such a producer-consumer relation is formalised with an extension of the notion of well-modedness. The approach improves over the original proposal of the authors (Krishna Rao *et al.*, 1992), where the LD selection rule was assumed.

The approach by Aguzzi and Modigliani (1993) takes into account that logic programs can be used in several modes, even within the same run of a program. Moreover, the approach is able to handle *local* variables, i.e. variables occurring only in a clause body but not in the head. Such variables model what is sometimes called *sideways information passing*. One remarkable property of the transformation is that it provides a characterisation of termination, albeit only for the limited class of *input driven* logic programs (Apt and Etalle, 1993). So for this limited class, a program terminates if and only if the corresponding TRS terminates.

Ganzinger and Waldmann (1992) proposed a transformation of logic programs into *conditional* TRSs. In such TRSs, the rules have the form $u_1 \to v_1, \ldots, u_n \to v_n \Rightarrow s \to t$, which is to be read as "if each u_i rewrites to v_i, then s rewrites to t". Well moded logic program clauses are transformed into such rules, where there is a correspondence between each u_i and the input of the ith body atom, each v_i and the output of the ith body atom, s and the input of the head, and t and the output of the head. The method improves over that by Krishna Rao *et al.* (1992) in applicability and simplicity.

Marchiori (1994) improves over the transformations of Aguzzi and Modigliani (1993) and Ganzinger and Waldmann (1992) by adopting enhanced methods to detect unification-freeness, i.e. situations where unification (used by SLD-resolution) boils down to matching (used by TRS operational semantics). Another contribution lies in the fact that the transformation proposed is modular, i.e. it considers each clause in isolation.

More recently, Arts (1997) investigated a new termination method for TRSs called *innermost normalisation* and applied it also to TRSs obtained by transforming well moded logic programs. The technique improves that of Krishna Rao *et al.* (1992).

10.4 Dynamic selection rules

By *dynamic* selection rules we mean those rules where selection depends on the degree of instantiation of atoms at run-time. Second generation logic languages adopt dynamic selection rules as control primitives. We mention here delay declarations, input-consuming derivations and guarded clauses.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Apt and Luitjes (1995) consider deterministic programs, i.e. programs where for each selectable atom (according to the delay declarations), there is at most one clause head unifiable with it. For such programs, the existence of one successful derivation implies that all derivations are finite. Apt and Luitjes also give conditions for the termination of `append`, but these are ad-hoc and do not address the general problem.

Lüttringhaus-Kappel (1993) proposes a method for generating control (delay declarations) automatically. The method finds *acceptable* delay declarations, ensuring that the most general selectable atoms have finite SLD-trees. What is required however are *safe* delay declarations, ensuring that *instances* of most general selectable atoms have finite SLD-trees. A *safe* program is a program for which every acceptable delay declaration is safe. Lüttringhaus-Kappel states that all programs he has considered are safe, but gives no hint as to how this might be shown in general. This work is hence not about *proving* termination. In some cases, the delay declarations that are generated require an argument of an atom to be a list before that atom can be selected. This is similar to requiring the atom to be bounded, i.e. to the approach of Marchiori and Teusink (1999) and Martin and King (1997), and of section 5.

Naish (1992) considers delay declarations that test for partial instantiation of certain predicate arguments. Such delay declarations implicitly ensure input-consuming derivations. He gives good intuitive explanations about possible causes of loops, essentially *circular modes* and *speculative output bindings*. The first cause (see Example 4.4) can be eliminated by requiring programs to be *permutation nicely*[4] *moded*. Speculative output bindings are indeed a good explanation for the fact that `permute`(O, I) (see Example 5.1) does not input terminate. Naish then makes the additional assumption that the selection rule always selects the leftmost selectable atom, and proposes to put recursive calls last in clause bodies. Effectively, this guarantees that the recursive calls are *ground* in their input positions, which goes beyond assuming input-consuming derivations.

Naish's proposal has been formalised and refined by Smaus *et al.* (2001). They consider atoms that may loop when called with insufficient input, or in other words, atoms for which assuming input-consuming derivations is insufficient to guarantee termination. It is proposed to place such atoms sufficiently late; all producers of input for such atoms must occur textually earlier. Effectively, this is an assumption about the selection rule that lies between input-consuming derivations and local delay-safe derivations.

Our characterisation of input termination only requires (permutation) simply moded programs and queries. The first sound but incomplete characterisation of Smaus (1999b) assumed well and nicely moded programs. It was then found that the condition of well-modedness could easily be lifted (Bossi *et al.*, 1999). It was only by restricting to *simply* moded programs that one could give a characterisation that is also complete. This means of course that the method of Bossi *et al.* (1999) does not subsume the method of (Bossi *et al.*, 2001) surveyed here, but nevertheless, we

[4] A slightly more general notion than permutation *simply*-modedness.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

believe that the fact that the characterisation is complete is more important. Input-consuming derivations can be ensured in existing systems using *delay declarations* such as provided by Gödel (Hill and Lloyd, 1994) or SICStus (SICStus, 1998). This is shown in Bossi *et al.* (2000, 2001) and Smaus (1999a).

The definition of input-consuming derivations has a certain resemblance with derivations in the parallel logic language of *(Flat) Guarded Horn Clauses* (Ueda, 1988). In (F)GHC, an atom and clause may be resolved only if the atom is an instance of the clause head, and a test (*guard*) on clause selectability is satisfied. Termination of GHC programs was studied by Krishna Rao *et al.* (1997) by transforming them into TRSs.

Pedreschi and Ruggieri (1999b) characterised a class of programs with guards and queries that have no failed derivation. For those programs, termination for one selection rule implies termination (with success) for all selection rules. This situation has been previously described as saying that a program does not make speculative bindings (Smaus *et al.*, 2001). The approach by Pedreschi and Ruggieri is an improvement w.r.t. the latter one, since what might be called 'shallow' failure does not count as failure. For example, the program `QUICKSORT` is considered failure-free in the approach of Pedreschi and Ruggieri.

10.5 ∃-termination and bounded nondeterminism

Concerning termination w.r.t. fair selection rules, i.e. ∃-termination, we are aware only of the works of Gori (2000) and McPhee (2000). Gori proposes an automatic system based on abstract interpretation analysis that infers ∃-termination. McPhee proposes the notion of *prioritised fair selection rules*, where atoms that are known to terminate are selected first, with the aim of improving efficiency of fair selection rules. He adopts the automatic test of Lindenstrauss and Sagiv (1997) to infer (left-)termination, but, in principle, the idea applies to any automatic termination inference system.

Concerning bounded nondeterminism, Martin and King (1997) define a transformation for Gödel programs, which shares with the transformation of Definition 9.10 the idea of not following derivations longer than a certain length. However, they rely on sufficient conditions for inferring the length of refutations, namely termination via a class of selection rules called *semilocal*. Their transformation adds run-time overhead, since the maximum length is computed at run-time. On the other hand, a run-time analysis is potentially able to generate more precise upper bounds than our static transformation, and thus to cut more unsuccessful branches. Also, the idea of pruning SLD-derivations at run-time is common to the research area of loop checking (Bol *et al.*, 1991).

Sufficient (semi-)automatic methods to approximate the number of computed instances by means of lower and upper bounds have been studied in the context of cost analysis of logic programs (Debray and Lin, 1993) and of cardinality analysis of Prolog programs (Braem *et al.*, 1994). Of course, if ∞ is a lower bound to the number of computed instances of P and Q then they do not have bounded nondeterminism. Dually, if $n \in N$ is an upper bound then P and Q have bounded nondeterminism.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

In this case, however, we are still left with the problem of determining a depth of the SLD-tree that includes all the refutations.

10.6 Automatic termination inference

On a theoretical level, the problem of deciding whether a program belongs to one of the classes studied in this article is undecidable. This was formally shown by Bezem (1993) for recurrence, and by Ruggieri (1999) for acceptability, fair-boundedness and boundedness. On a practical level, however, many methods have been proposed to infer (usually: left-) termination automatically. This research stream is currently very active, and some efficient tools are already integrated in compilers.

A challenging topic of the research in automatic termination inference consists in finding standard forms of level mappings and models, so that the solution of the resulting proof obligations can be reduced to known problems for which efficient algorithms exist (Bossi *et al.*, 1994; Benoy and King, 1997; Decorte *et al.*, 1993; Plümer, 1990; van Gelder, 1991).

As an example, we mention the detailed account of automatic termination analysis by Decorte *et al.* (1999). The main idea is as follows. Termination analysis is parametrised by several factors, such as the choice of modes and level mappings. In practice, these are usually inferred using abstract interpretation techniques. This is often not very efficient. Therefore Decorte *et al.* propose to encode all those parameters and the conditions that have to hold for them as constraints. So for example, there are constraint variables for each weighting parameter used in the definition of (semi-) linear norms and level mappings. To show termination of the analysed program, one has to find a solution to the constraint system.

Lindenstrauss and Sagiv (1997) developed the system *TermiLog* for checking termination. They use linear norms, (monotonicity and equality) constraint inference and the termination test of Sagiv (1991), originally designed for Datalog programs. The implementation of the static termination analysis algorithm of the Mercury system (Speirs *et al.*, 1997) exploits mode and type information provided by the programmer. Speirs *et al.* claim a better performance than the *TermiLog* system in the average case. The implementation of fair selection rules has been announced for future releases of Mercury. Codish and Taboch (1999) proposed a formal semantics basis that facilitates abstract interpretation for inferring left-termination.

Recently, Mesnard *et al.* (2000) developed the cTI system for *bottom-up* left-termination inference of logic programs. Bottom-up refers to the use of abstract interpretation based fixpoint computations whose output is a set of queries for which the system infers termination. The results show that, on several benchmark programs, the sets of queries inferred by cTI strictly include the set of queries for which the top-down methods of Decorte *et al.* (1999), Lindenstrauss and Sagiv (1997) and Speirs *et al.* (1997) can show termination.

Finally, we recall the approach by Stärk (1998) to prove both termination and partial correctness together. His system, called LPTP, is implemented in Prolog and consists of an interactive theorem prover able to prove termination and correctness of Prolog programs with negation, arithmetic built-in's and meta-predicates such as

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

`call`. The formal theory underlying LPTP is an inductive extension of pure Prolog programs that allows to express modes and types of predicates.

10.7 Extensions of pure logic programming

In this paper, we have assumed the standard definition of SLD-derivations for definite logic programs. We now briefly discuss termination of alternative or generalised execution models of logic programs.

A declarative characterisation of strong termination for *general* logic programs and queries (i.e, with negation) was proposed by Apt and Bezem (1991). The execution model assumed is *SLDNF* resolution with a *safe* (not to be confused with *delay-safe* (Marchiori and Teusink, 1999)) selection rule, meaning that negative literals can be selected only if they are ground. Also, we mention the bottom-up approach of Balbiani (1992), where an operator T_P is provided such that its ordinal closure coincides with those ground atoms A such that P and A strongly terminate.

Apt and Pedreschi (1993) have generalised acceptability to reason on programs with negation under SLDNF resolution. The characterisation is sound. Also, it is complete for safe selection rules. Marchiori (1996a) proposes a modification of acceptability to reason on programs with Chan's constructive negation resolution.

Termination of *abductive* logic programs has been studied by Verbaeten (1999). The execution model of abductive logic programs, called *SLDNFA* resolution, extends SLDNF resolution. Just as for Apt and Bezem (1991), the selection rule is an arbitrary safe one, but the definition of safe is weaker in this context. Essentially, SLDNFA resolution behaves worse than SLDNF resolution w.r.t. termination, which is why the conditions given by Apt and Bezem (1991) have to be strengthened. Finally, we point out that the conditions given are sufficient but not necessary.

Tabled logic programming is particularly interesting in the context of termination analysis since tabling improves the termination behaviour of a logic program, compared to ordinary execution. The works we discuss in the following take advantage of this, i.e. they can show termination in interesting cases where ordinary execution does not terminate. They assume tabled execution based on the left-to-right selection rule.

A declarative characterisation of tabled termination has been given by Decorte *et al.* (1998). To automate termination proofs of tabled logic programs, this work has been combined by Verbaeten and De Schreye (2001) with the constraint-based approach to proving left-termination automatically, discussed above (Decorte *et al.*, 1999). Verbaeten *et al.* (2001) have studied termination of programs using a mix of tabled and ordinary execution.

Concerning *constraint* logic programming (CLP), Colussi *et al.* (1995) first proposed a necessary and sufficient condition for left-termination, inspired by the method of Floyd for termination of flowchart programs. Their method consists of assigning a data flow graph to a program, and then to state conditions to prevent the program to enter an infinite loop in the graph.

Also, Ruggieri (1997) proposed an extension of acceptability that is sound and complete for *ideal* CLP languages. A CLP language is ideal if its constraint solver,

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

the procedure used to test consistency of constraints, returns *true* on a consistent constraint and *false* on an inconsistent one. In contrast, a non-ideal constraint solver may return *unknown* if it is unable to determine (in)consistency. An example of non-ideal CLP language is the CLP($\mathscr{R}$) system, for which Ruggieri proposes additional proof obligations (based on a notion of modes) to acceptability in order to obtain a sound characterisation of left-termination.

Mesnard (1996) provides sufficient termination conditions based on approximation techniques and boolean μ-calculus, with the aim of *inferring* a class of left-terminating CLP queries. The approach has been refined and implemented by Hoarau and Mesnard (1998).

11 Conclusion

In this paper, we have surveyed six different classes of terminating logic programs and queries. For each class, we have provided a sound declarative characterisation of termination. Except for local delay termination, this characterisation was also complete. We have offered a unified view of those classes allowing for non-trivial formal comparisons.

In section 9.1, we have compared the different characterisations w.r.t. certain technical details with the aim of understanding the role each technical detail plays.

In section 9.2, we have compared the classes themselves. The inclusion relations among them are summarised in the hierarchy of figure 1. Intuitively, as the assumptions about the selection rule become stronger, the proof obligations about programs become weaker.

One may ask: in how far is such a hierarchy ad-hoc, and could other classes be considered? We believe that the interest in strong termination, ∃-termination and bounded nondeterminism is evident because they are cornerstones of the whole spectrum of classes. The interest in left-termination is motivated by the fact that the standard selection rule of Prolog is assumed.

The interest in input termination and local delay termination is more arguable. We cannot claim that there are no other interesting classes in the surroundings of those two classes. Nevertheless, we believe that the distinction between input-consuming and local delay-safe selection rules captures an important difference among dynamic selection rules: requiring derivations to be input-consuming can be considered a reasonable minimum requirement to ensure termination, as we have argued that only very simple or contrived programs strongly terminate. In particular, the selection rule does not allow for methods showing termination that rely on boundedness of the selected atom. At the time of the selection, the depth of the SLD tree of an atom is not determined (by the atom itself). In contrast, local delay-safe selection rules require that the selected atom is bounded, and thus the depth of the SLD tree of an atom is determined.

We thus hope that we have captured much of the essence of the effect different choices of selection rules have on termination. This should be a step towards a possible automatic choice of selection rule and thus towards realising Kowalski's ideal.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

References

Aguzzi, G. and Modigliani, U. (1993) Proving termination of logic program by transforming them into equivalent term rewriting systems. In: Shyamasundar, R. K. (ed.), *Proc. of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science: Lecture Notes in Computer Science 761*, pp. 114–124. Springer-Verlag.

Apt, K. R. (1997) *From logic programming to Prolog*. Prentice Hall.

Apt, K. R. and Bezem, M. (1991) Acyclic programs. *New Generation Computing*, **29**(3), 335–363.

Apt, K. R. and Etalle, S. (1993) On the unification free Prolog programs. In: Borzyszkowski, A. and Sokolowski, S. (eds.), *Proc. of the 18th International Symposium on Mathematical Foundations of Computer Science: Lecture Notes in Computer Science 711*, pp. 1–19. Springer-Verlag.

Apt, K. R. and Luitjes, I. (1995) Verification of logic programs with delay declarations. In: Alagar, V. S. and Nivat, M. (eds.), *Proc. of the 4th International Conference on Algebraic Methodology and Software Technology: Lecture Notes in Computer Science 936*, pp. 66–90. Springer-Verlag.

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

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

Apt, K. R., Marchiori, E. and Palamidessi, C. (1994) A declarative approach for first-order built-in's of Prolog. *Applicable Algebra in Engineering, Communication and Computation*, **5**(3/4), 159–191.

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

Balbiani, P. (1992) The finiteness of logic programming derivations. In: Levi, G. (ed.), *Proc. of the 3rd Conference on Algebraic and Logic Programming: Lecture Notes in Computer Science 632*, pp. 403–419. Springer-Verlag.

Baudinet, M. (1992) Proving termination properties of Prolog programs: a semantic approach. *J. Logic Programming*, **14**, 1–29.

Benoy, F. and King, A. (1997) Inferring argument size relationships with CLP(R). In: Gallagher, J. P. (ed.), *Proc. of the 6th International Workshop on Logic Programming Synthesis and Transformation: Lecture Notes in Computer Science 1207*, pp. 204–223. Springer-Verlag.

Bezem, M. A. (1993) Strong termination of logic programs. *J. Logic Programming*, **15**(1/2), 79–98.

Bol, R. N., Apt, K. R. and Klop, J. W. (1991) An analysis of loop checking mechanism for logic programs. *Theoretical Computer Science*, **86**(1), 35–79.

Bossi, A., Cocco, N. and Fabris, M. (1994) Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science*, **124**(2), 297–328.

Bossi, A., Etalle, S. and Rossi, S. (1999) Properties of input-consuming derivations. In: Etalle, S. and Smaus, J.-G. (eds.), *Proc. of the ICLP Workshop on Verification of Logic Programs: Electronic Notes in Theoretical Computer Science*, 30(1).

Bossi, A., Etalle, S. and Rossi, S. (2000) Semantics of input-consuming logic programs. In: Lloyd, J. W. *et al.* (eds.), *Proc. of the 1st International Conference on Computational Logic: Lecture Notes in Computer Science 1861*, pp. 194–208. Springer-Verlag.

Bossi, A., Etalle, S., Rossi, S. and Smaus, J.-G. (2001) Semantics and termination of logic programs with dynamic scheduling. In: Sands, D. (ed.), *Proc. of the 10th European Symposium on Programming: Lecture Notes in Computer Science 2028*, pp. 402–416. Springer-Verlag.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Braem, C., Charlier, B. Le, Modart, S. and Van Hentenryck, P. (1994) Cardinality analysis of Prolog. In: Bruynooghe, M. (ed.), *Proc. of the International Logic Programming Symposium*, pp. 457—471. MIT Press.

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

Bruynooghe, M., Vandecasteele, H., de Waal, D. A. and Denecker, M. (1998) Detecting unsolvable queries for definite logic programs. In: Palamidessi, C. *et al.* (eds.), *Proc. of PLILP/ALP '98: Lecture Notes in Computer Science 1490*, pp. 118–133. Springer-Verlag.

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

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

Colussi, L., Marchiori, E. and Marchiori, M. (1995) On termination of constraint logic programs. In: Bruynooghe, M. and Penjam, J. (eds.), *Proc. of the 1st International Conference of Principles and Practice of Constraint Programming: Lecture Notes in Computer Science 976*, pp. 431–448. Springer-Verlag.

De Schreye, D. and Decorte, S. (1994) Termination of logic programs: the never-ending story. *J. Logic Programming*, **19-20**, 199–260.

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: *Proc. of the International Conference on Fifth Generation Computer Systems*, pp. 481–488. Institute for New Generation Computer Technology.

Debray, S. K. and Lin, N. W. (1993) Cost analysis of logic programs. *ACM Trans. on Programming Languages and Systems*, **15**(5), 826–875.

Decorte, S., De Schreye, D. and Fabris, M. (1993) Automatic inference of norms: A missing link in automatic termination analysis. In: Miller, D. (ed.), *Proc. of the International Logic Programming Symposium*, pp. 420–436. The MIT Press.

Decorte, S., De Schreye, D., Leuschel, M., Martens, B. and Sagonas, K. (1998) Termination analysis for tabled logic programming. In: Fuchs, N. E. (ed.), *Proc. of the 7th International Workshop on Logic Programming Synthesis and Transformation: Lecture Notes in Computer Science 1463*, pp. 111–127. Springer-Verlag.

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

Deransart, P. and Małuszyński, J. (1993) *A Grammatical View of Logic Programming*. The MIT Press.

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

Ganzinger, H. and Waldmann, U. (1992) Termination proofs of well-moded logic programs via conditional rewrite systems. In: Rusinowitch, M. and Rémy, J. L. (eds.), *Proc. of the 3rd International Workshop on Conditional Term Rewriting Systems: Lecture Notes in Computer Science 656*, pp. 430–437. Springer-Verlag.

Gori, R. (2000) An abstract interpretation approach to termination of logic programs. In: Parigot, M. and Voronkov, A. (eds.), *Proc. of the 7th International Conference on Logic for Programming and Automated Reasoning: Lecture Notes in Computer Science 1955*, pp. 362–380. Springer-Verlag.

Hill, P. M. and Lloyd, J. W. (1994) *The Gödel Programming Language*. The MIT Press.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Hitzler, P. and Seda, A. K. (1999) Acceptable programs revisited. In: Etalle, S. and Smaus, J.-G. (eds.), *Proc. of ICLP Workshop on Verification of Logic Programs: Electronic Notes in Theoretical Computer Science*, 30(1).

Hoarau, S. and Mesnard, F. (1998) Inferring and compiling termination for constraint logic programs. In: Flener, P. (ed.), *Proc. of the 8th International Workshop on Logic Programming Synthesis and Transformation: Lecture Notes in Computer Science 1559*, pp. 240–254. Springer-Verlag.

Hoarau, S. and Mesnard, F. (2001) σ-termination: from a characterisation to an automated sufficient condition. *Theory and Practice of Logic Programming*. To appear.

Kowalski, R. A. (1979) Algorithm = Logic + Control. *Comm. of the ACM*, **22**(7), 424–436.

Krishna Rao, M .R. K., Kapur, D. and Shyamasundar, R. K. (1992) A transformational methodology for proving termination of logic programs. In: Börger, E., Jäger, G., Kleine Büning, H. and Richter, M. M. (eds.), *Proc. of the 5th Workshop on Computer Science Logic: Lecture Notes in Computer Science 626*, pp. 213–226. Springer-Verlag.

Krishna Rao, M. R. K., Kapur, D. and Shyamasundar, R. K. (1997) Proving termination of GHC programs. *New Generation Computing*, **15**(3), 293–338.

Krishna Rao, M. R. K., Kapur, D. and Shyamasundar, R. K. (1998) Transformational methodology for proving termination of logic programs. *J. Logic Programming*, **34**(1), 1–41.

Lindenstrauss, N. and Sagiv, Y. (1997) Automatic termination analysis of logic programs. In: Naish, L. (ed.), *Proc. of the International Conference on Logic Programming*, pp. 63–77. The MIT Press.

Lüttringhaus-Kappel, S. (1993) Control generation for logic programs. In: Warren, D. S. (ed.), *Proceedings of the 10th International Conference on Logic Programming*, pp. 478–495. MIT Press.

Marchiori, E. (1996a) On termination of general logic programs w.r.t. constructive negation. *J. Logic Programming*, **26**(1), 69–89.

Marchiori, E. and Teusink, F. (1999) On termination of logic programs with delay declarations. *J. Logic Programming*, **39**(1-3), 95–124.

Marchiori, M. (1994) Logic programs as term rewriting systems. In: Levi, G. and Rodríguez-Artalejo, M. (eds.), *Proc. of the 4th International Conference on Algebraic and Logic Programming: Lecture Notes in Computer Science 850*, pp. 223–241. Springer-Verlag.

Marchiori, M. (1996b) Proving existential termination of normal logic programs. In: Wirsing, M. and Nivat, M. (eds.), *Proc. of the 5th International Conference on Algebraic Methodology and Software Technology: Lecture Notes in Computer Science 1101*, pp. 375–390. Springer-Verlag.

Martin, J. and King, A. (1997) Generating efficient, terminating logic programs. In: Bidoit, M. and Dauchet, M. (eds.), *Proc. of the 7th International Conference on Theory and Practice of Software Development: Lecture Notes in Computer Science 1214*, pp. 273–284. Springer-Verlag.

McPhee, R. (2000) *Compositional logic programming*. PhD thesis, Oxford University Computing Laboratory.

Mesnard, F. (1996) Inferring left-terminating classes of queries for constraint logic programs. In: Maher, M. (ed.), *Proc. of the Joint International Conference and Symposium on Logic Programming*, pp. 7–21. The MIT Press.

Mesnard, F., Neumerkel, U. and Hoarau, S. (2000) Implementing cTI : a constrained-based left-termination inference tool for LP. In: de Castro Dutra, I., Costa, V. S., Silva, F., Pontelli, E. and Carro, M. (eds.), *Proc. of the CL Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pp. 152–166.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

Naish, L. (1992) *Coroutining and the construction of terminating logic programs*. Technical report 92/5, Department of Computer Science, University of Melbourne.

Neumerkel, U. and Mesnard, F. (1999) Localizing and explaining reasons for nonterminating logic programs with failure slices. In: Nadathur, G. (ed.), *Proc. of the International Conference on Principles and Practice of Declarative Programming: Lecture Notes in Computer Science 1702*, pp. 328–341. Springer-Verlag.

Pedreschi, D. and Ruggieri, S. (1999a) Bounded nondeterminism of logic programs. In: De Schreye, D. (ed.), *Proc. of the International Conference on Logic Programming*, pp. 350–364. The MIT Press.

Pedreschi, D. and Ruggieri, S. (1999b) On logic programs that do not fail. In: Etalle, S. and Smaus, J.-G. (eds.), *Proc. of ICLP Workshop on Verification of Logic Programs: Electronic Notes in Theoretical Computer Science*, 30(1).

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

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

Ruggieri, S. (1997) Termination of constraint logic programs. In: Degano, P., Gorrieri, R. and Marchetti-Spaccamela, A. (eds.), *Proc. of the 24th International Colloquium on Automata, Languages and Programming (ICALP '97): Lecture Notes in Computer Science 1256*, pp. 838–848. Springer-Verlag.

Ruggieri, S. (1999) *Verification and validation of logic programs*. PhD thesis, Dipartimento di Informatica, Università di Pisa.

Ruggieri, S. (2001) ∃-universal termination of logic programs. *Theoretical Computer Science*, **254**(1-2), 273–296.

Sagiv, Y. (1991) A termination test for logic programs. In: Saraswat, V. A. and Ueda, K. (eds.), *Proc. of the International Logic Programming Symposium*, pp. 518–532. The MIT Press.

Serebrenik, A. and Schreye, D. De. (2001) Non-transformational termination analysis of logic programs, based on general term-orderings. In: Lau, K. (ed.), *Proc. of the 10th International Workshop on Logic Programming Synthesis and Transformation, selected papers: Lecture Notes in Computer Science 2042*, pp. 69–85. Springer-Verlag.

SICStus (1998) *SICStus Prolog user's manual*. Intelligent Systems Laboratory, Swedish Institute of Computer Science, PO Box 1263, S-164 29 Kista, Sweden. `http://www.sics.se/sicstus/docs/3.7.1/html/sicstus_toc.html`.

Smaus, J.-G. (1999a) *Modes and types in logic programming*. PhD thesis, University of Kent at Canterbury.

Smaus, J.-G. (1999b) Proving termination of input-consuming logic programs. In: De Schreye, D. (ed.), *Proc. of the International Conference on Logic Programming*, pp. 335–349. MIT Press.

Smaus, J.-G., Hill, P. M. and King, A. M. (2001) Verifying termination and error-freedom of logic programs with `block` declarations. *Theory and Practice of Logic Programming*, **1**(4), 447–486.

Somogyi, Z., Henderson, F. and Conway, T. (1996) The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Logic Programming*, **29**(1–3), 17–64.

Speirs, C., Somogyi, Z. and Søndergaard, H. (1997) Termination analysis for Mercury. In: Van Hentenryck, P. (ed.), *Proc. of the 4th International Static Analysis Symposium: Lecture Notes in Computer Science 1302*, pp. 160–171. Springer-Verlag.

Stärk, R. F. (1998) The theoretical foundations of LPTP (A Logic Program Theorem Prover). *J. Logic Programming*, **36**(3), 241–269.

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press

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

Thom, J. and Zobel, J. (1988) *NU-Prolog reference manual, version 1.3*. Technical report, Department of Computer Science, University of Melbourne, Australia.

Ueda, K. (1988) Guarded Horn Clauses, a parallel logic programming language with the concept of a guard. In: Nivat, M. and Fuchi, K. (eds.), *Programming of Future Generation Computers*, pp. 441–456. North Holland, Amsterdam.

van Gelder, A. (1991) Deriving constraints among argument sizes in logic programs. *Annals of Mathematics and Artificial Intelligence*, **3**, 361–392.

Verbaeten, S. (1999) Termination analysis for abductive general logic programs. In: De Schreye, D. (ed.), *Proc. of the International Conference on Logic Programming*, pp. 365–379. The MIT Press.

Verbaeten, S. and De Schreye, D. (2001) Termination of simply-moded well-typed logic programs under a tabled execution mechanism. *Applicable Algebra in Engineering, Communication and Computing*, **12**(1/2), 157–196.

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

https://doi.org/10.1017/S1471068402001400 Published online by Cambridge University Press