Characterisations of Termination in Logic Programming

Dino Pedreschi[1], Salvatore Ruggieri[1], and Jan–Georg Smaus[2]

[1] Dipartimento di Informatica, Università di Pisa, Via F. Buonarroti 2, 56125 Pisa, Italy, {pedre,ruggieri}@di.unipi.it
[2] Institut für Informatik, Universität Freiburg, Georges-Köhler-Allee 52, 79110 Freiburg im Breisgau, Germany, smaus@informatik.uni-freiburg.de

Abstract. The procedural interpretation of logic programs and queries is parametric to the selection rule, i.e. the rule that determines which atom is selected in each resolution step. Termination of logic programs and queries depends critically on the selection rule. In this survey, we present a unified view and comparison of seven notions of universal termination considered in the literature, and the corresponding classes of programs. For each class, we focus on a sufficient, and in most cases even necessary, declarative characterisation for determining that a program is in that class. By unifying different formalisms and making appropriate assumptions, we are able to establish a formal hierarchy between the different classes and their respective declarative characterisations.

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 [40] advocates the separation of the *logic* and *control* aspects of a logic program and has coined the famous formula

Algorithm = Logic + Control.

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 [2]. 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 programming languages allow for *dynamic scheduling*, i.e. they have primitives 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 [74] and Gödel [38].

In this survey, we classify programs and queries according to the selection rules for which they terminate, hence investigating the influence of the selection rule on termination. Like 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 [10, 23, 48]. 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 $\subseteq$"). The numbers in the figure correspond to statements and examples related to the pair of classes in question.

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 [11]. 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 [23] 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 [62, 63]. 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 & Ruggieri [58]. Such programs can be transformed into equivalent programs which strongly terminate, as indicated in the figure and stated in Theorem 10.11.

The three remaining classes shown in the figure are related to *dynamic scheduling*, i.e. selection rules where the selection of an atom depends on its degree of instantiation at runtime. To explain these classes 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.

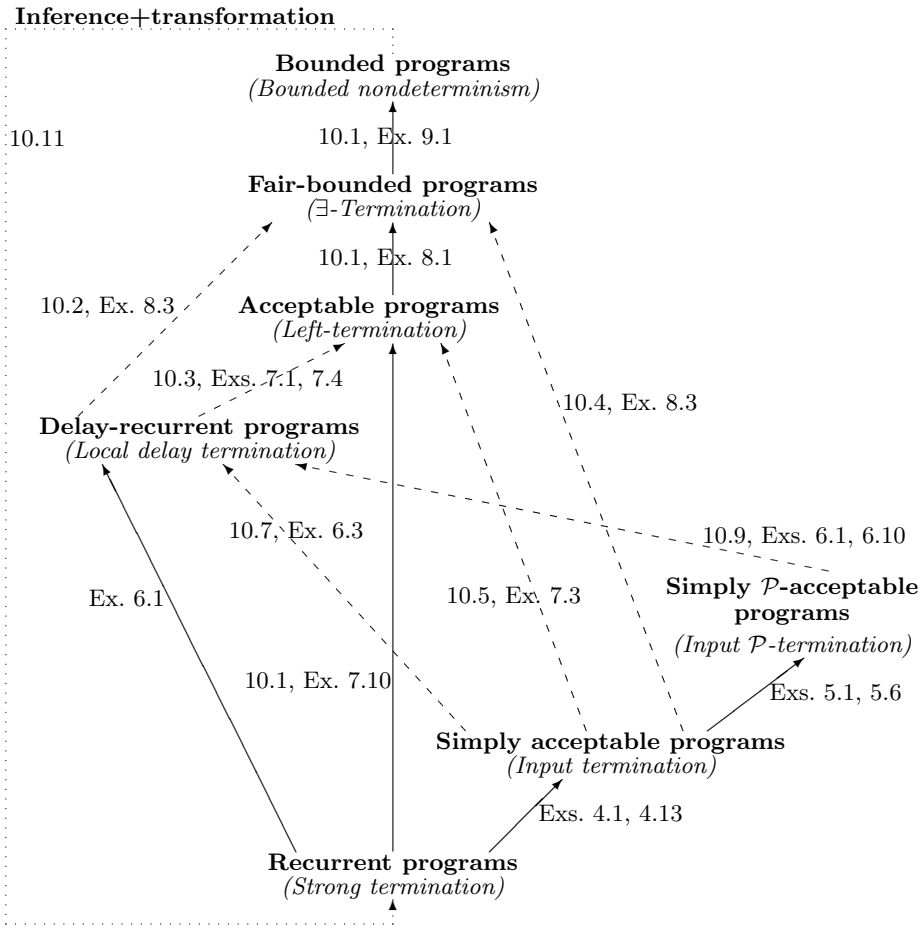

Fig. 1. An overview of the classes

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 [67] and Bossi *et al.* [15–17].

Input-consuming derivations can be restricted by imposing some additional instantiation property $\mathcal{P}$ that each selected atom must have. For example, $\mathcal{P}$ might be the set of all atoms that are bounded w.r.t. a given level mapping. A program and a query *input $\mathcal{P}$-terminate* if all input-consuming derivations for this program and query, restricted by $\mathcal{P}$, are finite. This class of programs has been studied by Smaus in very recent work [68].

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 & Teusink [47] 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 (possibly restricted by a property $\mathcal{P}$) 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. While assuming input-consuming selection rules is trivially (though not necessarily strictly) weaker than assuming input-consuming selection rules with an additional property $\mathcal{P}$, there is little sense in making general comparisons between selection rules restricted by some $\mathcal{P}$ and the other classes — it depends on the $\mathcal{P}$. However, we can choose $\mathcal{P}$ so that it exactly captures delay-safe selection rules, and then it follows of course that assuming $\mathcal{P}$-selection rules is weaker than assuming local and delay-safe selection rules. All these inclusions that depend on additional assumptions are indicated in the figure by dashed lines. Again, the numbers in the figure correspond to statements and examples.

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 cases of local delay termination and input $\mathcal{P}$-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 seven sections corresponding to the seven classes in Fig. 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 Sec. 10, we establish relations between the classes, formally showing the implications of Fig. 1. Section 11 discusses the related work, and Sec. 12 concludes.

2 Background and Notation

We use the notation of Apt [2], 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 by U_L (resp., B_L) the Herbrand universe (resp., base) on L. We denote by $Term_L$ (resp., $Atom_L$) the set of terms (resp., 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., $Ran(\theta)$).

The set $\{1, \ldots, n\}$ is denoted by $[1, n]$.

2.1 Modes

For a predicate p/n, a *mode* is an atom $p(m_1, \ldots, m_n)$, 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 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. A mode for a program consists of a mode for each of its predicates.

In the literature, several correctness criteria concerning the modes have been proposed, most importantly nicely-modedness and well-modedness [2]. In this article, we need *simply* moded programs [4] and *well moded* programs. The former are a special case of nicely moded programs. Note that the use of the letters $\mathbf{s}$ and $\mathbf{t}$ is reversed for clause heads. We believe that this notation naturally reflects the data flow within a clause.

Definition 2.1. A clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is *simply moded* if $\mathbf{t}_1, \ldots, \mathbf{t}_n$ 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 $\mathtt{p} \leftarrow \mathbf{B}$ is simply moded, where $\mathtt{p}/0$ is a fresh predicate symbol. 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 the 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 $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ 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.

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 Fig. 9 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.

Conceptually, we assume that whenever modes are used in this article, the mode of a predicate is unique. To realise the use of one predicate in several modes, one can introduce multiple (renamed) versions of the predicate [4, 5, 32, 55]. But it is also possible to realise multiple modes without any actual code duplication. Then, a mode should be associated with each *occurrence* of a predicate in a program [66, 69].

2.2 Norms and Level Mappings

All the characterisations of terminating programs we propose make use of the notions of norm and level mapping [20]. 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 2.3. 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.

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

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

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

$$\begin{aligned} length([T|Ts]) &= 1 + length(Ts) \\ length(T) &= 0 && \text{if } T \neq [\ldots|\ldots]. \end{aligned}$$

In particular, for a nil-terminated list $[T_1, \ldots, T_n]$, the list-length is n. We call a term of the form $[T_1, \ldots, T_n|\mathtt{Ts}]$, where $n \geq 0$, an *open list*. In particular, any variable is an open list.

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 [14, 27].

We now define level mappings where the dependency on the modes is made explicit [32].

Definition 2.4. 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.

2.3 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 [2]. In this article, we assume an extended definition: 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.

In practice, selection rules should always be *computable* functions, but we are not concerned with this issue here.

We now define the various notions of selection rules used in this article.

A $\mathcal{P}$-selection rule is a selection rule where each selected atom is in some set of atoms $\mathcal{P}$, closed under instantiation. Note that this notion is very abstract, but this does not mean that every selection rule can be defined as a $\mathcal{P}$-selection rule.

Definition 2.5. 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*).

A selection rule is *delay-safe w.r.t. a level mapping* $|.|$ if it specifies that an atom A can be selected only when A is bounded w.r.t. $|.|$.[3]

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.

2.4 Universal Termination

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

Definition 2.6. A program P and a query Q *universally terminate* w.r.t. a set of selection rules $\mathcal{S}$ if every SLD-derivation of P and Q via any selection rule from $\mathcal{S}$ 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. $\mathcal{S}$ if every program and query in the class universally terminate w.r.t. $\mathcal{S}$. Conversely, it is *complete* if every program and query that universally terminate w.r.t. $\mathcal{S}$ are in the class.

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 [2].

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

[3] The reader may be surprised that delay-safe selection rules make no reference to *delay declarations*. This is a terminological shortcut.

queries in the obvious way. I is a Herbrand *model* of program P if for each $A \leftarrow B_1, \ldots, B_n \in \mathit{ground}_L(P)$, we have that $I \models B_1, \ldots, B_n$ 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 [23]. For example, a model for the usual `append` predicate is

$$\{\texttt{append}(xs, ys, zs) \mid \mathit{length}(zs) = \mathit{length}(xs) + \mathit{length}(ys)\}.$$

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 [11], 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.

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

A program P is *recurrent by* $|.|$ if for every $A \leftarrow B_1, \ldots, B_n \in \mathit{ground}_L(P)$:

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

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

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

In the above definition, the proof obligations for a query Q are derived from those for the program $\{\mathtt{p} \leftarrow Q\}$, where $\mathtt{p}/0$ 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 [11], the query was called *bounded*. Throughout the paper, we prefer to maintain a uniform naming convention both for programs and queries.

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

```
% 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

Theorem 3.3 ([11]). Let P be a program and Q a query.

If P and Q are both recurrent by some $|.|$, 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 [11] 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 Fig. 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 Def. 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.

Note that the choice of an appropriate level mapping depends on 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 [32].

Example 3.5. Figure 3 shows the APPEND program. It is easy to check that APPEND is recurrent by the level mapping $|\mathtt{append}(xs, ys, zs)| = length(xs)$ and also by $|\mathtt{append}(xs, ys, zs)| = length(zs)$. A query append($Xs$, Ys, Zs) is recurrent by the first level mapping iff Xs is anything other than an open list, and by the second iff Zs is anything other than an open list. The level mapping

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

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

```
% reverse(Xs,Ys) ←
%     Xs is the reverse of list Ys.

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

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

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

Fig. 3. APPEND and NAIVE_REVERSE

```
% 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

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 Sec. 7 for a class of programs containing recurrent programs.

4 Input Termination

In this section, we consider input-consuming selection rules [17].

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 Fig. 4 is recurrent by defining

$$|\mathtt{even}(x)| = size(x)$$
$$|\mathtt{lte}(x,y)| = size(y).$$

Now consider the query $Q = \mathtt{even(X)}, \mathtt{lte(X,s^{100}(0))}$, 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

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 [38, 70, 73, 74].

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.

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 have such a strong requirement on each selected atom.

Example 4.3. Consider the EVEN program in Fig. 4 and the following input-consuming derivation, where we underline the selected atom in each step:

$$\texttt{even(X)},\ \underline{\texttt{lte(X, s}^{100}\texttt{(0))}} \longrightarrow \texttt{even(s(X'))},\ \underline{\texttt{lte(X', s}^{99}\texttt{(0))}} \longrightarrow$$
$$\underline{\texttt{even(s(s(X")))}},\ \texttt{lte(X", s}^{98}\texttt{(0))} \longrightarrow \texttt{even(X")},\ \underline{\texttt{lte(X", s}^{98}\texttt{(0))}} \ldots$$

At the time when $\texttt{even(s(s(X")))}$ is selected, the depth of its SLD-tree is not bounded. In fact, this depth depends on the eventual instantiation of $\texttt{X"}$.

The method for showing input termination inherently relies on a notion of *level* for atoms such as $\texttt{even(s(s(X")))}$, although this level is not bounded. This is the key to showing termination for derivations with coroutining (interleaving subderivations). In contrast, most approaches to termination assume that the level of the selected atom is bounded. We refer to Subsec. 11.7 and [66, Sec. 11.1].

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 (Fig. 3) in mode $\texttt{append}(I, I, O)$ and the query

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

Then we have the following infinite input-consuming derivation:

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

This well-known termination problem of programs with coroutining has been identified as *circular modes* by Naish [55].

To avoid the above situation, we require programs and queries to be simply moded (see Subsec. 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. We now give the formal definition [17].

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 $V_0, V_1, \ldots, V_n$ consisting of of fresh (w.r.t. c) variables such that $\theta = \sigma_0\sigma_1 \cdots \sigma_n$ where for $i \in \{0, \ldots, n\}$,

- $Dom(\sigma_i) \subseteq Vars(\mathbf{t}_i)$,
- $Ran(\sigma_i) \subseteq Vars(\mathbf{s}_i\sigma_0\sigma_1 \cdots \sigma_{i-1}) \cup V_i$.[4]

θ is *simply-local* w.r.t. a query $\mathbf{B}$ if θ is simply-local w.r.t. the clause $\mathtt{p} \leftarrow \mathbf{B}$ where $\mathtt{p}/0$ is a fresh predicate symbol.

Note that in the case of a simply-local substitution w.r.t. a query, σ_0 is the empty substitution, since $Dom(\sigma_0) \subseteq Var(\mathtt{p})$ where $\mathtt{p}$ is a fresh predicate symbol. 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 [17].

Example 4.6. Consider the PERMUTE_BACK program in Fig. 5 (the name has been chosen to distinguish it from PERMUTE to be introduced later). Assume mode $\texttt{permute}(O, I), \texttt{insert}(O, O, I)$. We examine the recursive clause for insert.

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

```
% permute(Xs,Ys) ←                    % insert(Xs,X,Zs) ←
%     Ys is a permutation of the list Xs. %     Zs is obtained by inserting X into Xs.

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

Fig. 5. PERMUTE_BACK

The substitution $\sigma = \{\mathtt{Y/V}, \mathtt{Zs/[W]}, \mathtt{Xs/[]}, \mathtt{X/W}\}$ is simply-local w.r.t. it: let $\sigma_0 = \{\mathtt{Y/V}, \mathtt{Zs/[W]}\}$, $\sigma_1 = \{\mathtt{X/W}, \mathtt{Xs/[]}\}$; then $Dom(\sigma_0) \subseteq \{\mathtt{Y}, \mathtt{Zs}\}$, $Ran(\sigma_0) \subseteq V_0$ where $V_0 = \{\mathtt{V}, \mathtt{W}\}$, $Dom(\sigma_1) \subseteq \{\mathtt{Xs}, \mathtt{X}\}$, and $Ran(\sigma_1) \subseteq Vars(\mathtt{Zs}\,\sigma_0)$.

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 $\{\mathtt{q(1)},\ \mathtt{p(X)}\leftarrow\mathtt{q(X)}\}$ with mode $\mathtt{q}(I),\ \mathtt{p}(O)$, 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. $\mathtt{p(X)} \leftarrow \mathtt{q(X)}$. The next subsection will further clarify the role of simply-local models.

Let SM_P be the set of all simply moded atoms[5] 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 [17]. We denote the least simply-local model of P containing SM_P by PM_P^{SL}, for *partial model*.

Example 4.8. Recall Ex. 4.6. SM_P consists of all atoms $\mathtt{insert(Us, U}, Vs)$ where $\mathtt{Us, U} \notin Vars(Vs)$. To construct PM_P^{SL}, we iterate T_P^{SL} starting from any atom in SM_P (the resulting atoms are written on the l.h.s. below) and the fact clause (r.h.s.). Each line below corresponds to one iteration of T_P^{SL}. We have $PM_P^{SL} =$

$$\begin{array}{ll}
\{\ \mathtt{insert(Us, U}, Vs), & \\
\quad \mathtt{insert}([Y_1|\mathtt{Us}], \mathtt{U}, [Y_1|Vs]), & \mathtt{insert}(Xs_1, X_1, [X_1|Xs_1]), \\
\quad \mathtt{insert}([Y_2, Y_1|\mathtt{Us}], \mathtt{U}, [Y_2, Y_1|Vs]), & \mathtt{insert}([Y_1|Xs_1], X_1, [Y_1, X_1|Xs_1]), \\
\quad \ldots & \ldots \\
\quad \mid Vs, Xs_1, X_1, Y_1, Y_2, \ldots \text{arbitrary} & \text{where } \mathtt{Us, U} \notin Vars(Vs)\}.
\end{array} \tag{3}$$

[5] We sometimes say "atom" for "query containing only one atom".

Observe the variable occurrences of `U, Us` in the atoms on the l.h.s. In Ex. 5.5, we will see the importance of such variable occurrences.

4.3 Declarative Characterisation

Bossi *et al.* [17] 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 [2]. 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 simply moded program, $|.|$ a moded generalised[6] level mapping and I a simply-local model of P containing SM_P. A clause $A \leftarrow B_1, \ldots, B_n$ is *simply acceptable by* $|.|$ *and* I if for every substitution θ simply-local w.r.t. it,

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

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.

Simply-acceptability, and $\mathcal{P}$-simply-acceptability to be introduced in the next section, are 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 in contrast to all other characterisations in this article, and explains 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 Subsec. 10.1.

Simply-acceptability characterises the class of input terminating programs.

Theorem 4.10 ([17]). 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 moded generalised level mapping $|.|$ and PM_P^{SL}.

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

[6] In [17], the word "generalised" is dropped, but here we prefer to emphasise that non-ground atoms are included in the domain.

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

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

Fig. 6. PERMUTE

Remark 4.11. 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. Therefore, 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.

It would have been possible to state this remark explicitly in the above theorem, but that would have complicated the definition of simply-local substitution and subsequent definitions. Generally, the question of when it is necessary to make the permutations of body atoms explicit is discussed in [66, Sec. 5.3].

4.4 Examples

Example 4.12. The program `EVEN` in Fig. 4 is simply acceptable with mode $\texttt{even}(I)$, $\texttt{lte}(O, I)$ by using the level mapping in Ex. 4.1, interpreted as moded *generalised* level mapping in the obvious way, and using any simply-local model. Moreover, the query $\texttt{even}(\texttt{X})$, $\texttt{lte}(\texttt{X}, \texttt{s}^{100}(\texttt{0}))$ is permutation simply moded (see Remark 4.11). Hence `EVEN` and this query input terminate.

Example 4.13. The program `PERMUTE` is shown in Fig. 6. Assume the mode $\texttt{permute}(I, O)$, $\texttt{insert}(I, I, O)$. Note that compared to Fig. 5, two body atoms have been reordered to make the program simply moded in this mode. Note also that $\texttt{permute} \not\approx \texttt{insert}$. The program is readily checked to be simply acceptable, using the moded generalised level mapping

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

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.

Example 4.14. Figure 7 shows program 15.3 from [72]: `QUICKSORT` using a form of difference lists (we permuted two body atoms for the sake of clarity). This program is simply moded with mode $\texttt{quicksort}(I, O)$, $\texttt{quicksort\_dl}(I, O, I)$, $\texttt{partition}(I, I, O, O)$, $\texttt{=<}(I, I)$, $\texttt{>}(I, I)$.

We use the following moded generalised level mapping (positions with _ are irrelevant)

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

```
%       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. 7. QUICKSORT

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 $\texttt{partition}(S_1, X, S_2, S_3)$ where $|S_1| \geq |S_2|$ and $|S_1| \geq |S_3|$. Note that this includes all simply moded atoms using `partitition`, and that this model is a fortiori simply-local since (2) in Def. 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.

In essence, looking at the clause before any instantiation, there is a decrease between the input of the clause head and the recursive body atoms (`[X|Xs]` is bigger than both `Bigs` and `Littles`). Moreover, by the model information about the atom $\texttt{partition}(\texttt{Xs}, \texttt{X}, \texttt{Littles}, \texttt{Bigs})$ we know that this decrease is preserved as the clause becomes instantiated.

5 Input $\mathcal{P}$-Termination

In this section, we consider input-consuming selection rules that are additionally parametrised by some instantiation property $\mathcal{P}$ that each selected atom must have. In particular, delay-safe derivations can be modelled this way. This section is based on very recent work [68].

We first give an example of a program that is not input terminating.

Example 5.1. Consider again the `PERMUTE_BACK` program in Fig. 5. So the mode is $\texttt{permute}(O, I), \texttt{insert}(O, O, I)$. It is immediate to check that the program is not input terminating: by repeatedly selecting the rightmost atom that may be selected, the query $\texttt{permute}(\texttt{Xs}, [1])$ generates an infinite input-consuming derivation.

One can understand this by explaining why the program cannot be simply acceptable. Recall Ex. 4.8. $PM^{SL}_{\texttt{PERMUTE\_BACK}}$ contains every atom of the form $\texttt{insert}(\texttt{Us}, \texttt{U}, Vs)$, i.e. every simply moded atom whose predicate is `insert`.

Therefore in particular $\texttt{insert}(\texttt{Us}, \texttt{U}, \texttt{Vs}) \in PM^{SL}_{\texttt{PERMUTE\_BACK}}$ (note that `Vs` is a variable). Consider the recursive clause for `permute`. The substitution $\theta = \{\texttt{Ys}/\texttt{Vs}, \texttt{Zs}/\texttt{Us}, \texttt{X}/\texttt{U}\}$ 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 $|\texttt{permute}([\texttt{U}|\texttt{Xs}], \texttt{Vs})| > |\texttt{permute}(\texttt{Xs}, \texttt{Us})|$. 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 $\texttt{permute}(\texttt{Xs}, [\texttt{1}])$ are finite if we require input-consuming derivations where each atom must be bounded w.r.t. an appropriate level mapping.

The attentive reader may have noticed that `PERMUTE_BACK` falls out of the class of input terminating programs for a very simple reason: Due to the variable `Ys` in the input position of the clause head, it follows that an atom using `permute` can *always* be selected.

Now it is tempting to think that the program misses the property of input termination "just narrowly", and that there is a simple fix to obtain input termination: replace `Ys` by $[\texttt{Y}|\texttt{Ys}]$ in the above clause. This is a fallacy. The resulting program is still not input terminating. This is related to *speculative output bindings* and has first been observed by Naish [55].

Programs that "just narrowly" miss the property of input termination may also be analysed using the methods of this section. We refer to [68].

5.1 Operational Definition

We now define termination for input-consuming $\mathcal{P}$-derivations, i.e. derivations via an input-consuming $\mathcal{P}$-selection rule.

Definition 5.2. A program P and query Q *input $\mathcal{P}$-terminate* if they universally terminate w.r.t. the set consisting of the input-consuming $\mathcal{P}$-selection rules.

Of course, input termination is just a special case of input $\mathcal{P}$-termination for a trivial $\mathcal{P}$ containing all atoms. However, in contrast to the previous section, it is unknown if the characterisation given here is complete. This justifies having the previous section on its own. Also, the previous section surveys well-established work while the work reported here is very recent.

5.2 Declarative Characterisation

Definition 5.3. Let P be a simply moded program, $|.|$ a moded generalised level mapping and I a simply-local model of P containing SM_P. A clause $A \leftarrow B_1, \ldots, B_n$ is *simply $\mathcal{P}$-acceptable by $|.|$ and I* if for every substitution σ simply-local w.r.t. it, for all $i \in [1, n]$,

$$B_1\sigma, \ldots, B_{i-1}\sigma \in I \text{ and } A \simeq B_i \text{ and } B_i\sigma \in \mathcal{P} \quad \text{imply} \quad |A\sigma| > |B_i\sigma|. \qquad (4)$$

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

The only difference to *simply acceptable* clauses is the condition $B_i\sigma \in \mathcal{P}$. Simply-local models capture all input-consuming derivations of a simply moded query, including the ones where we impose an additional condition $\mathcal{P}$. Hence this small modification gives us a sufficient criterion for input $\mathcal{P}$-termination.

Theorem 5.4 ([68]). Let P and Q be a simply moded program and query. If P is simply $\mathcal{P}$-acceptable by some $|.|$ and I, then P and Q input $\mathcal{P}$-terminate.

5.3 Examples

We give two examples where $\mathcal{P}$ is used exactly to model *delay-safe* selection rules. These programs need delay-safe selection rules to overcome the problem of speculative output bindings [55].

Example 5.5. Consider `PERMUTE_BACK` (Fig. 5) assuming mode $\texttt{permute}(O, I)$, $\texttt{insert}(O, O, I)$. Recall Ex. 4.8. We define the level mapping as

$$|\texttt{permute}(Xs, Ys)| = \mathit{length}(Ys)$$
$$|\texttt{insert}(Zs, X, Ys)| = \mathit{length}(Ys).$$

Now for all atoms $\texttt{insert}(Zs, X, Ys) \in PM_P^{SL}$, we have $|Ys| \geq |Zs|$; for the ones on the r.h.s. in (3) even $|Ys| > |Zs|$. Let $\mathcal{P}$ be the set of bounded atoms w.r.t. $|.|$.

Now let us look at the recursive clause for `permute`. We verify that the second body atom fulfils the requirement of Def. 5.3, where I is PM_P^{SL}. So we have to consider all simply-local substitutions σ such that $\texttt{insert}(\texttt{Zs}, \texttt{X}, \texttt{Ys})\sigma \in PM_P^{SL}$. For the atoms on the l.h.s. in (3), this means that

$$\sigma \supseteq \{\texttt{Ys}/[Y_n, \ldots, Y_1 | Vs], \texttt{Zs}/[Y_n, \ldots, Y_1 | \texttt{Us}], \texttt{X}/\texttt{U}\} \qquad (n \geq 0).$$

Clearly, $\texttt{permute}(\texttt{Xs}, \texttt{Zs})\sigma \notin \mathcal{P}$, and hence no proof obligation arises. For the atoms on the r.h.s. in (3), this means that

$$\sigma \supseteq \{\texttt{Ys}/[Y_n, \ldots, Y_1, X_1 | Xs_1], \texttt{Zs}/[Y_n, \ldots, Y_1 | Xs_1], \texttt{X}/X_1\} \qquad (n \geq 0).$$

But then $|\texttt{permute}([\texttt{X}|\texttt{Xs}], \texttt{Ys})\sigma| > |\texttt{permute}(\texttt{Xs}, \texttt{Zs})\sigma|$.

The other clauses are trivial to check, and so `PERMUTE_BACK` is simply $\mathcal{P}$-acceptable.

Example 5.6. The program `NAIVE_REVERSE` (Fig. 3) in mode $\texttt{reverse}(O, I)$, $\texttt{append}(O, O, I)$ is not input terminating, but it is input $\mathcal{P}$-terminating for $\mathcal{P}$ chosen in perfect analogy to Ex. 5.5.

In our opinion, the difference between delay-safe selection rules and (just) input-consuming selection rules is a fundamental one. Looking at the literature, the termination problem for the latter has been considered a much harder problem than for the former [45, 47, 49, 55]. We also refer to Subsec. 11.7.

```
r(X) ← p(X,Y), r(Y).          p(X,s(X)) ← fail.
r(0).                         p(s(X),X).
```

Fig. 8. A program for which locality is crucial

5.4 On Completeness of the Characterisation

Our investigations so far suggest that the criterion of simply $\mathcal{P}$-acceptability is *not* a necessary criterion, but that modifications are needed. More specifically, it seems that the condition (4) in Def. 5.3 must be "weakened" to something like

$$B_1\sigma, \ldots, B_{i-1}\sigma \in I \text{ and } A \simeq B_i \text{ and } B_i\sigma \in \mathcal{P} \text{ and } A\sigma \in \mathcal{P} \text{ imply } |A\sigma| > |B_i\sigma|,$$

but it is not clear if this is *strictly* weaker. Therefore, we cannot provide a counterexample showing that $\mathcal{P}$-acceptability is not a necessary criterion.

So while the completeness issue is still work in progress, we hope that a modified criterion will eventually be found. It should then probably be take over the name *simply $\mathcal{P}$-acceptability*, replacing our current definition.

Another interesting topic for future work would consist of investigating the automatic inference of properties $\mathcal{P}$ for which termination of a given program can be established.

6 Local Delay Termination

In this section, we consider selection rules that are both local and delay-safe. We first give an example of a program that is not input $\mathcal{P}$-terminating, for a $\mathcal{P}$ that ensures delay-safe selection rules. We shall see that the program terminates for all selection rules that are local in addition to being delay-safe (see Ex. 6.9).

Example 6.1. Let P be the program in Fig. 8 in mode $\mathtt{r}(I), \mathtt{p}(I, O)$. Setting $\mathcal{P} = \{\mathtt{p}(x, Y) \mid x \text{ ground}, Y \text{ arbitrary}\} \cup \{r(x) \mid x \text{ ground}\}$, we have the following infinite input-consuming $\mathcal{P}$-derivation:

$$\underline{\mathtt{r(0)}} \longrightarrow \underline{\mathtt{p(0,Y_1)}},\ \mathtt{r(Y_1)} \longrightarrow \mathtt{fail},\ \underline{\mathtt{r(s(0))}} \longrightarrow$$
$$\mathtt{fail},\ \underline{\mathtt{p(s(0),Y_1)}},\ \mathtt{r(Y_1)} \longrightarrow \mathtt{fail},\ \mathtt{fail},\ \underline{\mathtt{r(s(s(0)))}} \longrightarrow \ldots$$

We give an intuitive explanation why P cannot be $\mathcal{P}$-simply acceptable. Since $\mathtt{fail}$ is a simply moded atom, it turns out that for any X, we have $\mathtt{p}(X, \mathtt{s}(X)) \in PM_P^{SL}$. So for the recursive clause to be $\mathcal{P}$-simply acceptable, we would need $|X| > |\mathtt{s}(X)|$ for all X, which is impossible since there are no infinite descending chains in $\mathbb{N}$.

This example also demonstrates that the class of local delay terminating programs strictly includes the class of strongly terminating programs.

The example is artificial. We will come back to this point in the conclusion. In any case, the assumption of local selection rules is crucial for the method for showing termination of this section.

6.1 Operational Definition

Marchiori and Teusink [47] 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 took a shortcut by assuming *delay-safe* selection rules. This seems legitimate given that Marchiori and Teusink do not give the exact syntax of delay declarations either.

Definition 6.2. 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 two sections, modes are not used explicitly in the definition of delay-safe selection rules. Therefore it is possible to contrive an example of a program and a query that input terminate (and hence a fortiori input $\mathcal{P}$-terminate) but do not local delay terminate. The example is obtained by deliberately choosing a level mapping that does not reflect the mode of the query at hand.

Example 6.3. 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 Subsec. 10.2 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. As is witnessed by Ex. 6.1, a program which local delay terminates but does not even input $\mathcal{P}$-terminate, this implication is strict.

6.2 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 6.4. Let $|.|$ be a level mapping, $A \leftarrow Q$ a clause containing a body atom B, and $\tilde{C}$ a subset[7] 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.

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

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. In the following definition, Pow stands for the powerset.

Definition 6.5. Let $|.|$ be a level mapping and $A \leftarrow Q$ a clause. Consider the least set $\mathcal{C}$, subset of $Pow(Q \times Pow(Q))$, such that

1. $\langle B, \emptyset \rangle \in \mathcal{C}$ whenever $\emptyset$ is a minimal direct cover for B in $A \leftarrow Q$;
2. $\langle B, \tilde{C} \rangle \in \mathcal{C}$ 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 $i \in [1, k]$, $\langle C_i, \tilde{D}_i \rangle \in \mathcal{C}$.

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

6.3 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 [47], albeit equivalent.

Definition 6.6. 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*.

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.

In order 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 Def. 6.6). This is the reason why the selection rule is assumed to be local. We can now state the result of this section.

Theorem 6.7 ([47]). Let P be a program. If P is delay-recurrent by some $|.|$ and I, then for every query Q, P and Q local delay terminate.

Remark 6.8. Remark 4.11 applies to local delay termination as well.

6.4 Examples

Example 6.9. Consider again the program in Fig. 8, with the level mapping and model

$$\begin{aligned} |\mathtt{p}(x,y)| &= size(x) \\ |\mathtt{r}(x)| &= size(x)+1 \\ I &= \{\mathtt{p}(\mathtt{s}(z),z) \mid z \text{ arbitrary}\} \cup \{\mathtt{r}(\mathtt{s}^n(0)) \mid n \geq 0\}. \end{aligned}$$

The program is delay-recurrent by $|.|$ and I. We check the recursive clause for $\mathtt{r}$. Consider an arbitrary ground instance

$$\mathtt{r}(x) \leftarrow \mathtt{p}(x,y),\ \mathtt{r}(y). \tag{5}$$

First, we observe that I is a model of this instance. In fact, if its body is true in I, then $x = \mathtt{s}^{n+1}(0)$ and $y = \mathtt{s}^n(0)$ for some $n \geq 0$, and so $\mathtt{r}(x)$ is true in I.

Consider the first body atom. It has an empty cover. Since $size(x)+1 > size(x)$, we have a decrease as required.

Consider now the second body atom. There is only one cover $\mathtt{p}(\mathtt{X},\mathtt{Y})$, so we must show that

$$x = \mathtt{s}^{n+1}(0) \text{ and } y = \mathtt{s}^n(0) \quad \text{imply} \quad size(x)+1 > size(y)+1,$$

which is evident. Hence we have shown that the clause is delay-recurrent.

Note that for the $\mathcal{P}$ given in Ex. 6.1, any input-consuming $\mathcal{P}$-derivation is delay-safe. So it is the locality property that makes the difference to that example.

We now give another example, which seems even more contrived than Ex. 6.1, but turns out to be interesting because of the similarity to Ex. 7.1.

Example 6.10. Consider

```
p(X) ← q(Y), p(Y).
q(0) ← fail.
```

with $|\mathtt{p}(0)| = 0$.

For the sake of comparison, assume the mode $\mathtt{p}(I), \mathtt{q}(O)$ and let $\mathcal{P} = \{\mathtt{p}(0)\} \cup \{\mathtt{q}(X) \mid X \text{ arbitrary}\}$.

Then the program local delay terminates but does not input $\mathcal{P}$-terminate for the query $\mathtt{p}(0)$. We will discuss this example further in the conclusion.

In an article that was the predecessor of this one [60], we gave `PERMUTE_BACK` (Fig. 5) as an example of a delay-recurrent program, but since then, it has been shown that this program does not require locality for termination (Ex. 5.5).

6.5 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, resolving a cover will not necessarily make the atom bounded.

Example 6.11. 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 `z` 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 the first clause. Since `r(X)` is a cover for `q(X)` and since every model of the program contains $\mathtt{r}(t)$ for every t, 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 $|\mathtt{q}(\mathtt{s}^k(0))|' \geq k$ for any natural k.

7 Left-Termination

In this section, we consider the LD selection rule. We first give an example of a program that is not local delay terminating.

Example 7.1. Consider the program

```
p ← q, p.
```

with query `p`, where $|\mathtt{p}| = 1$ and $|\mathtt{q}| = 0$. It terminates for the LD selection rule but does not local delay terminate.

The example is artificial, and hence not a convincing motivation for studying the LD selection rule. We discuss this further in the conclusion, but in any case, there are several reasons for studying the LD 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.

7.1 Operational Definition

Definition 7.2. 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 three previous ones is difficult. In particular, left-termination is not necessarily stronger than input or local delay termination, e.g. when applied to programs written with the RD selection rule in mind.

Example 7.3. Consider the program `PERMUTE_BACK` in Fig. 5, but this time in mode $\texttt{permute}(I, O)$, $\texttt{insert}(I, I, O)$. This program input terminates but does not left-terminate (see Ex. 4.13 and note Remark 4.11).

Example 7.4. Consider the program in Fig. 8, where we permute two body atoms in the first clause to obtain

```
r(X) ← r(Y), p(X,Y).
```

By Remark 6.8 and Ex. 6.9, the program and every query local delay terminate w.r.t. the level mapping given there. Moreover, no derivation deadlocks. However, the program and the query `r(0)` do not left-terminate.

Also, local delay termination may not imply left-termination because of the deadlock problem. We will comment on this in the conclusion.

7.2 Extended Level Mappings

Left-termination was addressed by Apt & Pedreschi [7], 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 7.5. Figure 9 shows `TRANSP`, a program that terminates on a strict subset of ground queries only. In the intended meaning of the program, $\texttt{trans}(x, y, e)$ succeeds iff $x \rightsquigarrow_e y$, i.e. if $\texttt{arc}(x, y)$ 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 of [7], `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 $\texttt{trans}(x, y, e)$ when e is not a DAG. The solution to both problems is to consider *extended* level mappings [61, 62].

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

trans(X,Y,E) ←
   member(arc(X,Y),E).

trans(X,Y,E) ←
   member(arc(X,Z),E), trans(Z,Y,E).

member(X,[X|Xs]).

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

Fig. 9. TRANSP

Definition 7.6. An *extended level mapping* is a function $|.| : B_L \to \mathbb{N}^\infty$ of ground atoms to $\mathbb{N}^\infty$, where $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$.

In particular, we define $n \rhd m$ for $n, m \in \mathbb{N}^\infty$ iff $n = \infty$ or $n > m$. We write $n \trianglerighteq m$ iff $n \rhd m$ or $n = m$.

We have $\infty \rhd m$ for every $m \in \mathbb{N}^\infty$. In particular $\infty \rhd \infty$. For (only) this reason, $\rhd$ is not well-founded. However, this makes sense since the inclusion of ∞ in the codomain is intended to model non-termination and uninteresting instances of program clauses.

7.3 Declarative Characterisation

With the above notation we are now ready to introduce (a modified 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.

The modification w.r.t. [7] lies in the fact that the definition of an acceptable clause may involve clause *instances* where both the head and a body atom have level ∞. Intuitively, a non-terminating derivation would start in a query of level ∞ and always use clause instances where head and recursive body atoms have level ∞, while an acceptable (and terminating) query must have a level in $\mathbb{N}$.

Definition 7.7. 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 $A \leftarrow B_1, \ldots, B_n \in \mathit{ground}_L(P)$,

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

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

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

Let us compare this definition to the definition of delay-recurrence (Def. 6.6). 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

resolved. Because of the correctness of LD resolution [2], 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 Subsec. 10.1.

Acceptable programs and queries precisely characterise left-termination.

Theorem 7.8 ([7, 61]). Let P be a program and Q a query.

If P and Q are both acceptable by some $|.|$ and 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.

7.4 Examples

Example 7.9. The program in Ex. 7.1 is trivially acceptable. Let $|\mathtt{p}| = \mathtt{1}$ and $|\mathtt{q}| = \mathtt{0}$ and $I = \emptyset$. Then $|\mathtt{p}| \rhd |\mathtt{q}|$, and since I is a model of the program, $I \models \mathtt{q}$ implies $|\mathtt{p}| \rhd |\mathtt{p}|$.

We now give an example that highlights the use of extended level mappings in termination proofs. Note that we do not intend this example to be contrasted with the three preceding termination classes.

Example 7.10. 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:

$$|\mathtt{trans}(x,y,e)| = \begin{cases} \mathit{length}(e) + 1 + \mathit{Card}\{v \mid x \rightsquigarrow_e v\} & \text{if } e \text{ is a DAG} \\ \infty & \text{otherwise} \end{cases}$$

$$|\mathtt{member}(x,e)| = \mathit{length}(e)$$

$$I = \{\mathtt{trans}(x,y,e) \mid x,y,e \in U_L\} \cup \{\mathtt{member}(x,e) \mid x \text{ is in the list } e\}.$$

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:

(i)	$\lvert\mathtt{trans}(x,y,e)\rvert \rhd \lvert\mathtt{member}(\mathtt{arc}(x,z),e)\rvert$
(ii) $\mathtt{arc}(x,z)$ is in e implies	$\lvert\mathtt{trans}(x,y,e)\rvert \rhd \lvert\mathtt{trans}(z,y,e)\rvert$.

The first one is easy to show since $|\mathtt{trans}(x,y,e)| \rhd \mathit{length}(e)$. 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 $\mathit{Card}\{v \mid x \rightsquigarrow_e v\} > \mathit{Card}\{v \mid z \rightsquigarrow_e v\}$, and so:

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

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

Fig. 10. PRODCONS

Finally, observe that for a DAG e, the queries $\texttt{trans}(x, \texttt{Y}, e)$ and $\texttt{trans}(\texttt{X}, \texttt{Y}, e)$ 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).

8 ∃-Termination

So far we have considered five 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 the dual to assuming *all* selection rules (Sec. 3). We introduce ∃-*termination* of logic programs [63], claiming that it is an essential concept for separating the *logic* and *control* aspects of a program.

Before, however, we motivate the limitations of left-termination.

Example 8.1. The program PRODCONS in Fig. 10 abstracts a (concurrent) system composed of a producer and a consumer. For notational convenience, we identify the term $\texttt{s}^n(0)$ 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 $\texttt{system}(n)$. Note that the program is well moded with the obvious mode $\texttt{prod}(O), \texttt{cons}(I, I), \texttt{wait}(I)$, 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 $\texttt{system}(n)$ terminate.

8.1 Operational Definition

Definition 8.2. A program P and a query Q $\exists$-*terminate* if there exists a non-empty set $\mathcal{S}$ of standard selection rules such that P and Q universally terminate w.r.t. $\mathcal{S}$.

If P and Q do not $\exists$-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 8.3. The simple program

```
p(s(X)) ← p(X).
p(X).
```

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

We will come back to the issue of deadlock in Subsec. 10.2.

We observe that $\exists$-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 8.4 ([62, 63]). A program P and a query Q $\exists$-terminate iff they universally terminate w.r.t. the set of fair selection rules.

Concerning Ex. 8.1, it can be said that viewed as a concurrent system, the program inherently relies on fairness for termination.

8.2 Declarative Characterisation

Ruggieri [62, 63] offers a characterisation of $\exists$-termination using the notion of *fair-bounded* programs and queries. Just as Def. 7.7, it is based on *extended* level mappings.

Definition 8.5. 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 for some $i \in [1, n]$ 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 for some $i \in [1, n]$ 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 Def. 8.5. 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.

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 Subsec. 10.2, 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 Def. 8.5 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 8.6 ([62, 63]). Let P be a program and Q a query.

If P and Q are both fair-bounded by some $|.|$ and 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.

8.3 Example

Example 8.7. The `PRODCONS` program is fair-bounded. First, we introduce the *list-max* norm:

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

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

$$\begin{aligned} |\texttt{system}(n)| &= size(n) + 3 \\ |\texttt{prod}(bs)| &= length(bs) \\ |\texttt{cons}(bs, n)| &= \begin{cases} size(n) + lmax(bs) & \text{if } I \models \texttt{cons}(bs, n) \\ size(n) & \text{if } I \not\models \texttt{cons}(bs, n) \end{cases} \\ |\texttt{wait}(t)| &= size(t) \end{aligned}$$

$$\begin{aligned} I = \{&\texttt{system}(n) \mid n \in U_L\} \cup \{\texttt{prod}(bs) \mid lmax(bs) \leq 2\} \cup \\ &\{\texttt{cons}(bs, n) \mid length(bs) = size(n)\} \cup \{\texttt{wait}(x) \mid x \in U_L\}. \end{aligned}$$

Let us show the proof obligations of Def. 8.5. 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).*

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

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

i.e. $I \models \mathtt{cons}([d|bs], \mathtt{s}(n))$. Therefore, I is a model of *(c).* Let us show proof obligations *(a, b)* of Def. 8.5.

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

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

These two inequalities show that *(a)* holds.

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

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

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

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

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

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

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

$$\begin{aligned} &|\mathtt{system}(n)| = size(n) + 3 \ \rhd \ length(bs) = |\mathtt{prod}(bs)| \\ &|\mathtt{system}(n)| = size(n) + 3 \ \rhd \ size(n) + lmax(bs) = |\mathtt{cons}(bs, n)|. \end{aligned}$$

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

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

Fig. 11. ODDEVEN

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

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

If $I \models \texttt{cons}(bs, n)$ and $I \not\models \texttt{prod}(bs)$ ($\texttt{prod}(bs)$ has no refutation) then $length(bs) = size(n)$, which implies:

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

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

9 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 9.1. The program ODDEVEN in Fig. 11 defines the even and odd predicates, with the usual intuitive meaning. The query $\texttt{even}(\texttt{X}),\ \texttt{odd}(\texttt{X})$ 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 $\exists$-terminate. However, ODDEVEN and the query have only finitely many, namely zero, successful derivations.

9.1 Operational Definition

Pedreschi & Ruggieri [58] propose the notion of *bounded nondeterminism* to model programs and queries with finitely many refutations.

Definition 9.2. 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 [2], 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 bounded nondeterminism implies that there are finitely many refutations also for non-standard selection rules, the converse implication does not hold, in general (see Ex. 8.3).

Bounded nondeterminism, although not being a notion of termination in the strict sense, is closely related to termination. In fact, if P and Q $\exists$-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 $\exists$-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.

9.2 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 Defs. 7.7 and 8.5, it is based on *extended* level mappings.

Definition 9.3. 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 $A \leftarrow B_1, \ldots, B_n \in \mathit{ground}_L(P)$:

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

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

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

It is straightforward to check that the definition of bounded programs is a simplification of Def. 8.5 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 9.4 ([58, 62]). Let P be a program and Q a query.

If P and Q are both bounded by some $|.|$ and 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.

9.3 Examples

Example 9.5. Consider again the `ODDEVEN` program. It is readily checked that it is bounded by defining:

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

The query `even(X), odd(X)` is bounded by $|.|$ and I. In fact, since no instance of it is true in I, Def. 9.3 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 Def. 9.3 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 [19]. 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 $\exists$-terminate but have bounded nondeterminism.

Example 9.6. We define the predicate `all` such that the query $\texttt{all}(n_0, n_1, \texttt{Xs})$ collects in `Xs` the answers of a query $\texttt{q}(m, A)$ 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 $\texttt{all}(\texttt{0}, \texttt{s}(\texttt{s}(\texttt{0})), \texttt{As})$ do not $\exists$-terminate, but they have only one computed answer, namely $\texttt{As} = [\texttt{0}, \texttt{s}(\texttt{0}), \texttt{s}(\texttt{s}(\texttt{0}))]$. The program and the query are bounded (and thus have bounded nondeterminism) by defining:

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

10 Relations between Classes

We have defined seven 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.

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
$\mathcal{P}$-simply-acceptability	no	yes	yes	no	no	no
recurrence	yes	no	no	yes	no	n.a.

10.1 Comparison of Characterisations

We now try to provide an intuitive understanding of the technical differences between the characterisations of termination we have proposed. These are summarised in Table 1. Note that simply-acceptability is a special case of $\mathcal{P}$-simply-acceptability that does not need to be distinguished in this context.

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 $\mathcal{P}$-simply-acceptability require a decrease for all ground instances of a clause. One cannot attribute this difference to the termination classes themselves: the first criterion for input-termination by Smaus [67] also required a decrease for the ground instances of a clause, just as there are characterisations of left-termination [14, 25] based on generalised level mappings and hence non-ground instances of clauses. However, one can say that our characterisation of input $\mathcal{P}$-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. $\mathcal{P}$-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 [32], where left-termination for *well moded* programs is shown, using *well-acceptability*. Well-acceptability requires a decrease only for recursive body atoms.

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 first methods for showing termination of input-consuming derivations were also not based on models [16, 67], and it was remarked 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 $\mathcal{P}$-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, which is crucial for achieving full completeness of the characterisation. For an atom A with $|A| = \infty$ the proof obligations are trivially satisfied. However, we do not see any reason why some of the other characterisations could not also be generalised by allowing ∞ in the codomain of level mappings.

A final difference concerns the way information on data flow (modes, models, covers) 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.

10.2 From Strong Termination to Bounded Nondeterminism

In this subsection, we show inclusions between the introduced classes, i.e. we justify each arrow in Fig. 1. Note that in that figure, we have not only given the numbers of the statements, but also the numbers of two kinds of examples: examples that demonstrate that an inclusion is strict, and "counterexamples" that demonstrate that an inclusion does not hold without making additional assumptions.

We first leave aside the classes involving dynamic scheduling, i.e. input ($\mathcal{P}$-)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 7.10, 8.1 and 9.1 show that these implications are strict.

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 10.1. 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:

- P and Q are recurrent by $|.|$,
- P and Q are acceptable by $|.|$ and I,
- P and Q are fair-bounded by $|.|$ and I,
- P and Q are bounded by $|.|$ and I.

Consider now local delay termination. Obviously, it is implied by strong termination, and this implication is strict (Ex. 6.1). However, we have observed with the programs and queries of Exs. 7.4 and 8.3 that local delay termination does not imply left-termination or $\exists$-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 $\exists$-termination.

Proposition 10.2. 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 $\exists$-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 [66], and so by Def. 2.2, Q' contains an atom that is ground in its input positions and hence bounded w.r.t. $|.|$. Consider the selection rule that always selects this atom together with all program clauses. 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 $\exists$-termination.

Concerning the second claim, since fair-boundedness is a complete characterisation of $\exists$-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 10.3. 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 & Teusink [47] 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, and this implication is strict (Exs. 4.1 and 4.13). However, as observed in Exs. 6.3, 7.3 and 8.3, input termination does not imply local delay termination, left-termination, or $\exists$-termination, in general. Again, these results can be obtained under reasonable assumptions.

The following proposition relates input termination to $\exists$-termination.

Proposition 10.4. Let P and Q be a permutation well moded program and query. If P and Q input terminate then they $\exists$-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. The selection rule s constructed as in the proof of Prop. 10.2 is an input-consuming selection rule, and also a standard selection rule. Therefore, input termination implies universal termination w.r.t. $\{s\}$ and hence $\exists$-termination.

Concerning the second claim, by Theorem 4.10, P and Q input terminate. As shown above, this implies that they $\exists$-terminate. Since fair-boundedness is a complete characterisation of $\exists$-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.

Proposition 10.5. 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 [5].

Definition 10.6. Let P be a program and $|.|$ a moded generalised 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 10.7. Let P and Q be a permutation simply moded program and query, and $|.|$ a moded generalised 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 moded query such that s selects atom $A = p(\mathbf{s}, \mathbf{t})$. Then by Def. 10.6, 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 [5, (Apt & 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 [2], 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.

Definition 10.6 seems to express the natural condition for level mappings that ensure input-consuming derivations. Note that the proposition is not straightforward to generalise to, say, nicely moded programs, since in this case one cannot in general construct an mgu by matching as in the above proof.

It remains an open question if 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.

Example 10.8. Consider again the program and the level mapping $|.|$ of Ex. 6.11. 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 Ex. 6.11. 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.

Finally, we consider input $\mathcal{P}$-termination. Obviously, if a program and query input terminate, then they input $\mathcal{P}$-terminate. Whether or not this inclusion is strict depends on whether $\mathcal{P}$ is a trivial property or not. Examples 5.1 and 5.6 demonstrate situations where it is strict.

There is little sense in making general comparisons between $\mathcal{P}$-selection rules and the other classes — everything depends on $\mathcal{P}$. However, the following generalisation of Prop. 10.7 is particularly interesting.

Proposition 10.9. Let P and Q be a permutation simply moded program and query, and $|.|$ a moded generalised level mapping that implies matching. Let $\mathcal{P}$ be the set of atoms atoms that are bounded w.r.t. $|.|$.

If P and Q input $\mathcal{P}$-terminate then they local delay terminate (w.r.t. $|.|$).

Proof. By the same proof as the one of Prop. 10.7, 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. Moreover, by the definition of $\mathcal{P}$, such a derivation is also a $\mathcal{P}$-derivation.

10.3 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 & Ruggieri [58], such an upper bound is related to the natural number k of Def. 9.3 of bounded queries. As in our notation for moded atoms, we use boldface letters to denote vectors of (possibly non-ground) terms.

Definition 10.10. 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 10.11 ([58, 62]). Let P be a program and Q a query both bounded by $|.|$ and I, and let k be a given natural number satisfying Def. 9.3.

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

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 *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 10.12. Reconsider the program `ODDEVEN` and $Q =$ `even(X)`, `odd(X)` of Ex. 9.1. 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

$$\texttt{even(X,s}^2\texttt{(0)),odd(X,s}^2\texttt{(0))}.$$

By Theorem 10.11, 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 & Ruggieri [58] give an example showing how the approach of Decorte *et al.* [29] could be rephrased to infer boundedness.

11 Related Work

Termination in logic programming (and its extensions) has been the subject of intense research over the last fifteen years. The survey of De Schreye & Decorte [23], dated 1994, distinguishes 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, this survey paper has been mainly concerned with the first type. While we do not even try to survey the large amount of literature on automatic or semi-automatic approaches [14, 21, 29, 23, 44, 52, 53, 71], it must be observed that existing tools typically implement conditions for checking proof obligations of the characterisations we

surveyed. As an example, 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. Note that 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 [11] for recurrence, and by Ruggieri [62] for acceptability, fair-boundedness and boundedness. Therefore, the conditions implemented by automatic tools are, inevitably, *sufficient* conditions.

In the following, we recall other characterisations of the various notions of termination and relate them to those presented in this survey.

11.1 Acceptability: the Modularity Issue

A termination characterisation is modular if the proof obligations for the program $P = P_1 \cup \ldots \cup P_n$ can be obtained from separate proof obligations of programs $P_1, \ldots, P_n$. The modularity property is essential both in paper & pencil proofs and in automatic tools, since it allows for reasoning on termination of a large program by breaking it down to several small modules.

Since non-termination can only arise from recursion, the decomposition $P_1, \ldots, P_n$ should partition P in such a way that all clauses defining two mutually recursive predicates appear in a same module P_i. Therefore, a termination characterisation is modular if the proof obligations for a clause defining a predicate p depend only on predicates mutually recursive with p.

Apt & Pedreschi [8] refined acceptability to provide a partially modular method. The resulting notion, called *semi-acceptability*, requires that: for every $A \leftarrow B_1, \ldots, B_n \in \mathit{ground}_L(P)$,

$$\text{for all } i \in [1,n]: \ I \models B_1, \ldots, B_{i-1} \ \text{ implies } \begin{cases} |A| > |B_i| \text{ if } \ \mathit{rel}(A) \simeq \mathit{rel}(B) \\ |A| \geq |B_i| \text{ otherwise.} \end{cases}$$

Compared to acceptability, a strict decrease is now required for mutually recursive predicates only. Even if this simplifies proofs, it is a restricted notion of modularity, since changes in the level mapping of atoms defined in one module may make the proof obligations in higher modules invalid.

Etalle *et al.* [32] proposed a refinement of acceptability (*well-acceptability*) for well moded programs and queries. The requirement of well-modedness simplifies proofs of acceptability. On the one hand, the decrease of the level mapping is now required only for mutually recursive calls, i.e. for every $A \leftarrow B_1, \ldots, B_n \in \mathit{ground}_L(P)$,

$$\text{for all } i \in [1,n], \ I \models B_1, \ldots, B_{i-1} \ \text{ and } \mathit{rel}(A) \simeq \mathit{rel}(B) \text{ imply } |A| > |B_i|.$$

On the other hand, level mappings are assumed to be moded, and this leads to no proof obligation on *queries* (or better, queries are bounded as an immediate consequence). Also, it is interesting to observe that the definition of well-acceptability is then very close to simply-acceptability (Def. 4.9). 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.

De Schreye & Serebrenik [24] generalised well-acceptability to *order-acceptability*, by having any well-founded ordering, not necessarily $\mathbb{N}$, as codomain of level mappings. This allows us to show the same termination results and to simplify termination proofs when complex level mappings may be needed.

11.2 Non-ground Characterisations of Left-termination

Alternative characterisations of left-termination consider proof obligations on generalised level mappings and thus on possibly non-ground instances of clauses and queries. Let us recall the well-known approach of Decorte *et al.* [25, 29].

First, they use a non-ground notion of model.

Definition 11.1. A *generalised model*[8] of a program P is a set $I \subseteq Atom_L$ such that for every $A \leftarrow B_1, \ldots, B_n \in inst_L(P)$,

$$B_1, \ldots, B_n \in I \text{ implies } A \in I.$$

Second, they require (generalised) level mappings to be invariant under instantiation for atoms that may appear in a derivation starting from a set of intended queries. This is the counterpart of acceptability of a(n atomic) query.

Definition 11.2. For a program P and a set of queries $\mathcal{Q}$, let $Call(P, \mathcal{Q})$ be the set of atoms selected along a SLD-derivation of P and any $Q \in \mathcal{Q}$ via the LD selection rule.

A generalised level mapping $|.|$ is *rigid* if for every $A \in Call(P, \mathcal{Q})$ and every substitution θ, we have $|A| = |A\theta|$.

Usually, abstract interpretation techniques allow us to compute a superset of $Call(P, \mathcal{Q})$ given P and $\mathcal{Q}$, while for a broad class of norms, rigidity can be verified syntactically [14].

The proof method, called *rigid acceptability w.r.t. a set* $\mathcal{Q}$, requires that for a rigid level mapping $|.|$ and a generalised model I: for every $A \leftarrow B_1, \ldots, B_n \in inst_L(P)$,

$$\text{for all } i \in [1, n],\ I \models B_1, \ldots, B_{i-1} \text{ and } rel(A) \simeq rel(B) \text{ imply } |A| > |B_i|.$$

If those proof obligations are satisfied, then P and every $A \in \mathcal{Q}$ left-terminate.

This characterisation is fully modular, i.e. it does not require P to be well-moded as in the case of well-acceptability. However, the characterisation is not complete. The main problem is due to the notion of rigidity.

Example 11.1. The query `p(X)` and the simple program P below left-terminate.

[8] A generalised model coincides with a set of *valid interargument relations* in the terminology of [25, 29].

```
p(a) ← p(b).
p(b).
```

Consider now $\mathcal{Q} = \{\texttt{p(X)}\}$. We have $Call(P, \mathcal{Q}) = \{\texttt{p(X)},\texttt{p(a)},\texttt{p(b)}\}$. However, for any generalised level mapping $|.|$, proof obligations require $|\texttt{p(a)}| > |\texttt{p(b)}|$, which implies that $|.|$ cannot be rigid on $Call(P, \mathcal{Q})$.

The source of the problem lies in the requirement $|A| = |A\theta|$ of Def. 11.2. By assuming $|A| \geq |A\theta|$, the example program and query above can be reasoned about.

De Schreye and Serebrenik [24] have adapted this approach, i.e. the use of call sets, to *general orderings*, as opposed to level mappings. However, the aspect of incompleteness is pretty much the same as in the approach of Decorte *et al.* (see [24, Example 6]).

A general solution is provided by Bossi *et al.* [14] consisting of: (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. However, the 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.

11.3 Left-termination with Respect to a Set of Queries

Acceptability w.r.t. a set allows us to reason on a program and a *set* of queries, while acceptability seems to concentrate on a program and a *single* query at once. The benefit of acceptability w.r.t. a set 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 easily be generalised to a set of queries. If this was not the case, the practical use of termination analysis would be very limited. For instance, given a level mapping such that $|\texttt{p}(t)| = \mathit{length}(t)$, it is immediate to conclude that all queries $\texttt{p(}T\texttt{)}$, where T is a list, are acceptable.

Conversely, is it the case that if P and all queries in a set $\mathcal{Q}$ left-terminate then P and every $Q \in \mathcal{Q}$ are acceptable by a same $|.|$ and I?

The answer is affirmative. In fact, from the proof of the Completeness Theorem 7.8 [62, 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 Q left-terminate for each $Q \in \mathcal{Q}$.

11.4 Permutation Terminating Programs

A permutation of a program P (resp., query Q) is any program (query) obtained by reordering clause body atoms in P (atoms in Q). We say that P and Q *permutation terminate* if for some permutation P' of P and Q' of Q, P' and Q' left-terminate. Observe that permutation termination is strictly weaker than left-termination, and strictly stronger than $\exists$-termination (e.g. program `PRODCONS` in Fig. 10 and `system(`n`)`, with $n \in \mathbb{N}$, $\exists$-terminate but do not permutation terminate).

We have not included permutation termination in our formal hierarchy since it is trivial from a theoretical point of view to relate it to left-termination: simply analyse all possible permutations of the program and query for left-termination. Permutation termination is mainly an issue for automatic tools, since one would like to compute this permutation efficiently.

Deransart & Małuszyński [30] presented the proof obligations of their method by considering a generic permutation of body atoms. However, the choice of the permutation is left to the user.

The inference of an appropriate permutation has been proposed by Speirs *et al.* [71] and by Hoarau & Mesnard [39]. In [71], mode and type information provided by the programmer are used to reorder the body atoms. The resulting static termination algorithm is part of the Mercury system [70]. In contrast, the approach of [39] aims at *inferring* an as large as possible set of queries for which a program permutation terminate without involving the programmer in additional specifications.

11.5 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 [10] considered transforming logic programs into functional programs. Termination of the transformed programs can then be studied by structural induction. Her approach covers general logic programs, existential termination and the effects of the Prolog cut. Also, there is a considerable body of literature on transforming logic programs to term rewriting systems (TRSs), where a large set of well-founded orderings is available for reasoning about termination . 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 LD selection rule [9, 35, 41, 56]. One notable exception is due to Krishna Rao *et al.* [43], 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.

While the transformation must be sound (if the transformed program terminates then the original one terminates as well), the converse (if the original

program terminates then the transformed one terminates as well) is not well studied. One remarkable exception is the approach by Aguzzi & Modigliani [1], whose transformation is complete, albeit only for the limited class of *input driven* logic programs [4]. So for this limited class, a program terminates if and only if the corresponding TRS terminates.

11.6 Integer and Floating-Point Computations

For efficiency reasons, integers and integer predicates are implemented in Prolog (and other logic programming languages) by means of special terms and predicates, built-in's of the system. As an example, `3 < (2+2)` is an atom containing the less-than predicate `<` and the ground arithmetic expression terms `3` and `(2+2)`. As one could expect, the resolution of the atom above leads to success.

Integer arithmetic does not require special treatment when termination does not depend on integer computation, such as in the definition of the `partition` predicate in Ex. 7. In contrast, in presence of integer computations, the definition of the level mapping might take into account integer arguments of atoms. The approach of Dershowitz *et al.* [31] deduces automatically from a given program a finite abstract domain for representing ranges of integer expressions involved in program clauses. The abstract domain serves as a basis for checking the decreasing of level mappings over recursive calls.

Serebrenik [64] shows that the definition of a level mapping when integer arguments are critical for termination may be not as simple as expected, e.g. it may be non-linear. He proposed and implemented a sufficient condition for partitioning integers into intervals (called *adornments*) such that a linear level mapping can be defined on each of them. Even further, Serebrenik & De Schreye [65] extended the approach to reason on floating-point computations, i.e. in presence of rounding errors.

Also, Apt *et al.* [6] proposed a variant of acceptability for reasoning on built-in predicates, including arithmetic ones, `var`() and `ground`(). Their key concept is a specialised semantics (called Θ-semantics) and a notion of model w.r.t. such semantics to be used instead of Herbrand models in the definition of acceptability.

11.7 Dynamic Scheduling

The term *dynamic scheduling* refers to selection rules where the selection of an atom depends on its degree of instantiation at runtime. Dynamic scheduling can be implemented using delay declarations as provided by Gödel [38] or SICStus [73], or using *guards* (see Subsec. 11.12).

We believe that *modes* are important for understanding dynamic scheduling, even though some authors have not used them explicitly [45, 47, 49, 55]. Modes are the basis for defining input-consuming derivations, which are a formalism for describing dynamic scheduling while abstracting from the technical details of delay declarations. We also believe that within dynamic scheduling, there is an important qualitative distinction between what we call (here) *weak* and

strong selection rules. *Weak* selection rules are achieved by delay declarations that test for arguments being at least non-variable, and ideally correspond to input-consuming selection rules. *Strong* selection rules ensure that the depth of the SLD-tree of an atom is bounded at the time of selection, and more or less correspond to delay-safe selection rules.

Naish [55] considers delay declarations that would fall under weak selection rules. Naish has given two intuitive causes for loops: *circular modes* and *speculative output bindings*. The first cause (see Ex. 4.4) can be eliminated by requiring programs to be *permutation nicely moded*[9]. Speculative output bindings are indeed a good explanation for the fact that `permute`(O, I) (see Ex. 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 would fall under strong selection rules.

Lüttringhaus-Kappel [45] proposed a method for generating 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. Sometimes the generated delay declarations would fall under weak selection rules, but in some cases, the delay declarations require an argument of an atom to be a list before that atom can be selected, which would fall under strong selection rules.

Apt & Luitjes [5] made a first attempt to show termination for dynamic scheduling. They considered 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. Such a class of programs, however, is of limited interest. Apt & Luitjes also give conditions for the termination of `APPEND`, but these are ad-hoc and do not address the general problem.

The work by Marchiori & Teusink [47], which we surveyed in Sec. 6, not only assumes strong selection rules, but in addition selection rules must be *local*. A limitation of their method lies in the fact that the notion of cover is just an approximation of the data flow in a program (see Ex. 6.11). No implementation of local selection rules is mentioned by the authors. We refer to the conclusion for further discussion.

Martin & King [49] ensure termination by imposing a depth bound on the SLD-tree. This is realised by a program transformation introducing additional argument positions for each predicate, which are counters for the depth of the computation. Of course, this falls under strong selection rules.

Naish's proposal [55] has been formalised and refined by Smaus *et al.* [69]. The authors consider atoms that may loop when called with insufficient input.

[9] A generalisation of "permutation *simply* moded".

It is proposed to place such atoms sufficiently late; all producers of input for such atoms must occur textually earlier. Effectively, this is a hybrid selection rule where strong assumptions are made only for certain atoms.

Concerning input termination, the first sound but incomplete characterisation assumed well and nicely moded programs [67]. It was then found that the condition of well-modedness could easily be lifted [16]. By restricting to *simply* moded programs, it was possible to give a characterisation that is also complete [17], which is the work we survey in Sec. 4. It has been shown that under natural conditions, input-consuming derivations can be implemented using delay declarations [15, 17, 66].

The recent work of [68] considers input-consuming selection rules with additional assumptions. In one dimension, a selection rule can be parametrised by a property $\mathcal{P}$ that the selected atoms must have. This can be used to formalise delay-safe selection rules as we did in Sec. 5. However, the notion of $\mathcal{P}$-derivation abstracts from the distinction between weak and strong selection rules, since $\mathcal{P}$ could be any instantiation property. In another dimension, a selection rule can be local or not (necessarily) local. These dimensions can freely be combined.

11.8 ∃-Termination

Concerning termination w.r.t. fair selection rules, i.e. ∃-termination, we are aware only of the works of Gori [36] and McPhee [50]. Gori proposed an automatic system based on abstract interpretation analysis that infers ∃-termination. McPhee proposed 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 & Sagiv [44] to infer (left-)termination, but, in principle, the idea applies to any automatic termination inference system.

11.9 Bounded Nondeterminism

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 [26] and of cardinality analysis of Prolog programs [18]. As an example, cost analysis is exploited in the Ciao-Prolog system [37]. 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. In this case, however, we are still left with the problem of determining a depth of the SLD-tree that includes all the refutations.

The idea of cutting unsuccessful SLD-derivations is common to the research area of *loop checking* (see e.g. [12]). While a run-time analysis is potentially able to cut more unsuccessful branches, the evaluation of a pruning condition at run-time, such as for loop checks, involves a considerably higher computational overhead than statically checking the boundedness proof obligations.

11.10 General Programs

General programs admit negative literals in clause bodies and in queries. In presence of negation, there are several execution models proposed in the literature.

The most widely known is *SLDNF-resolution*, where negation is interpreted by the *negation-as-failure* rule. A declarative characterisation of strong termination for general logic programs and queries was proposed by Apt & Bezem [3]. They assume *safe* (not to be confused with *delay-safe* [47]) selection rules, meaning that negative literals can be selected only if they are ground. Apt & Pedreschi [7] have generalised acceptability to reason on programs with negation under SLDNF resolution. The characterisation is sound. Also, it is complete for safe selection rules.

When turning on other execution models, the class of (left-)terminating programs and queries may differ. A declarative characterisation of left-termination was provided by Marchiori [46] in the context of *constructive negation* by extending acceptability. Also, an elaborated notion extension of recurrence has been proposed in the context of *SLDNFA-resolution* by Verbaeten [76], and in the context of the EK-proof procedure by Mancarella *et al.* [57].

Finally, the modularity issue for general programs is discussed by Bossi *et al.* [13].

11.11 Extensions of LP: Constraint Logic Programs

The first work on characterisations of (left-)termination in *constraint* logic programming (CLP) is due to Colussi *et al.* [22], who proposed a necessary and sufficient condition inspired by the method of Floyd for termination of flowchart programs [33]. Their method consists of assigning a data flow graph to a program, where each node is labelled with the set of constraint stores of calls that may reach the associated program point. The decreasing of a function on every cycle of the data flow graph is then a necessary and sufficient condition for left-termination. A drawback of the method is that the set of constraints associated to nodes must be specified (the approach is not automated), which means reasoning operationally (as opposed to declaratively in terms of level mappings) on the program.

Ruggieri [61] proposed an extension of acceptability that is sound and complete for *ideal* CLP languages. A CLP language is ideal if its constraint solver, 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($\mathcal{R}$) system, for which Ruggieri proposes proof obligations (based on a notion of modes) in addition to acceptability in order to obtain a sound characterisation of left-termination.

Mesnard [51] provided sufficient termination conditions based on approximation techniques and Boolean μ-calculus, with the aim of *inferring* a class of left-terminating CLP queries. Recently, the approaches of Mesnard and Ruggieri

have been merged into a unified framework [54], for which an implementation is described in [52].

Finally, Frühwirth [34] adapted the notion of recurrent logic programs to show termination of *constraint handling rules*, a language closely related to concurrent constraint programming and especially designed for writing constraint solvers.

11.12 Extensions of LP: Programs with Guards

The definition of input-consuming derivations has a certain resemblance with derivations in the parallel logic language of *(Flat) Guarded Horn Clauses* [75]. 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.* [42] by transforming them into TRSs.

Pedreschi & Ruggieri [59] characterised a class of programs (with guards and delay declarations) 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 [69]. The approach by Pedreschi & 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 [59].

11.13 Extensions of LP: Tabled Programs

Tabled logic programming is particularly interesting since tabling improves the termination behaviour of a logic program, compared to ordinary execution.

A declarative characterisation of tabled left-termination has been given by Decorte *et al.* [28]. The method can show termination in interesting cases where ordinary execution does not terminate. The approach has been extended and automated by Verbaeten *et al.* [77], where a mix of tabled and ordinary SLD-resolution is also studied. The characterisation provided is in general sound, and complete under some conditions on tabled predicates.

12 Conclusion

In this article, we have surveyed seven different classes of terminating logic programs and queries. For each of them, we have provided a sound declarative characterisation of termination, which, in five cases, was also complete. We have offered a unified view of those classes allowing for non-trivial formal comparisons. In particular, we have shown strict inclusions among the classes, establishing the hierarchy shown in Fig. 1. We conclude by discussing two questions: Why, in some cases, did we need additional assumptions to obtain a unified view? How significant are the classes of the hierarchy?

To make the first question more specific: why do the inclusions between termination for dynamic selection rules on the one hand and left-termination and $\exists$-termination on the other hand not simply hold without additional assumptions? We have two kinds of counterexamples.

We have counterexamples where the textual order of atoms in the clause bodies of a program makes the program unsuitable for the LD selection rule (Exs. 7.3 and 7.4). It is not pathological for a program to be written for, say, the RD selection rule, but we should not be surprised about pathological (i.e. non-termination) behaviour when we run the program using the LD selection rule.

Moreover, we have counterexamples where a program input terminates, or local delay terminates, thanks to deadlock (Ex. 8.3). Is a program that relies on deadlock for termination pathological? Generally, deadlock is considered an undesirable situation, but it is still preferable to non-termination. Also, it should be noted that deadlock cannot necessarily be blamed on the program. The `APPEND` program and the query $\texttt{append}([\texttt{1}|\texttt{Xs}], \texttt{Ys}, \texttt{Zs})$ do not $\exists$-terminate, but they input terminate (for the mode $\texttt{input}(I, I, O)$), and in this sense, one could argue that selection rules allowing for deadlock are a *stronger* assumption for termination than any standard selection rule. This is in contrast to Props. 10.2, 10.3, 10.4 and 10.5 (where the hypotheses imply absence of deadlock).

Concerning the second question, there is of course a general answer: this is a survey article, and so we surveyed those works that are commonly recognised as most relevant in the field of termination for various selection rules, even if sometimes the significance of a result is diminished by a later result. However, we also have a few more specific answers.

The interest in strong termination, $\exists$-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. With the three classes related to dynamic scheduling, we have captured the important distinction between *weak* selection rules, *strong* selection rules, and *strong* and *local* selection rules, as explained in Subsec. 11.7.

The question can also be phrased differently: for each inclusion between classes, how significant is it that the inclusion is strict? If $A \subset B$ but $B \setminus A$ contains only some very obscure and contrived programs, then is it worthwhile to study B in detail?

The strict inclusion between input termination and input $\mathcal{P}$-termination, for an appropriate $\mathcal{P}$, is witnessed by Exs. 5.1 and 5.6. These programs are not contrived, in fact they are famous in this context [55], but they are small programs, and it remains to be seen if other examples can be found.

In our opinion, the strict inclusion between local delay termination and left-termination demonstrated by Ex. 7.1 is insignificant. The example is artificial. Most of the time, the LD selection rule turns out to be simple implementation of a local delay-safe selection rule — no more and no less.

Example 6.10 is very similar to Ex. 7.1 and suggests that the strict inclusion between input $\mathcal{P}$-termination and local delay termination is also insignificant,

or put differently, that the difference made by assuming local selection rules is insignificant. Actually, we are not aware of a realistic program where locality matters for termination. However, Ex. 6.1 exhibits a certain pattern that suggests that there could be a realistic example: consider the clause $\mathtt{r(X)} \leftarrow \mathtt{p(X,Y)}, \mathtt{r(Y)}$. There are two derivations for $\mathtt{p(X,Y)}$, one that generates a `Y` bigger (say, by the term size norm) than `X` but is bound to fail, and one that generates a `Y` smaller than `X` and succeeds. Locality is crucial so that this failure occurs before the recursive call $\mathtt{r(Y)}$.

Marchiori & Teusink justify the restriction of local derivations saying that "the termination behaviour of 'delay until nonvar'[10] is poorly understood", and that "the class of local selection rules [...] supports simple tools for proving termination" [47]. In the meantime, as discussed in Sections 4 and 5, both termination for input-consuming derivations and termination for delay-safe (but not necessarily local) derivations are well understood.

Can we conclude from the above that the strict inclusion between input $\mathcal{P}$-termination and left-termination is insignificant, and so all the research effort currently devoted to left-termination should be redirected towards input $\mathcal{P}$-termination? Not quite. Left-termination is the most important notion of termination in practice and has been studied under every conceivable aspect. One cannot expect that all this work will readily translate to input $\mathcal{P}$-termination.

References

1. G. Aguzzi and U. Modigliani. Proving termination of logic program by transforming them into equivalent term rewriting systems. In R. K. Shyamasundar, editor, *Proc. of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, pages 114–124. Springer-Verlag, 1993.
2. K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.
3. K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 29(3):335–363, 1991.
4. K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proc. of the 18th International Symposium on Mathematical Foundations of Computer Science*, volume 711 of *LNCS*, pages 1–19. Springer-Verlag, 1993.
5. K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In V. S. Alagar and M. Nivat, editors, *Proc. of the 4th International Conference on Algebraic Methodology and Software Technology*, volume 936 of *LNCS*, pages 66–90. Springer-Verlag, 1995.
6. K. R. Apt, E. Marchiori, and C. Palamidessi. A declarative approach for first-order built-in's of Prolog. *Applicable Algebra in Engineering, Communication and Computation*, 5(3/4):159–191, 1994.
7. K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

[10] This amounts to input-consuming derivations.

8. K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.
9. T. Arts. *Automatically proving termination and innermost normalisation of term rewriting systems*. PhD thesis, Universiteit Utrecht, 1997.
10. M. Baudinet. Proving termination properties of Prolog programs: a semantic approach. *Journal of Logic Programming*, 14:1–29, 1992.
11. M. A. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1 & 2):79–98, 1993.
12. R. N. Bol, K. R. Apt, and J. W. Klop. An analysis of loop checking mechanism for logic programs. *Theoretical Computer Science*, 86(1):35–79, 1991.
13. A. Bossi, N. Cocco, S. Etalle, and S. Rossi. On modular termination proofs of general logic programs. *Theory and Practice of Logic Programming*, 2(3):263–291, 2002.
14. A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science*, 124(2):297–328, 1994.
15. A. Bossi, S. Etalle, and S. Rossi. Semantics of input-consuming logic programs. In J. W. Lloyd et al., editor, *Proc. of the 1st International Conference on Computational Logic*, volume 1861 of *LNCS*, pages 194–208. Springer-Verlag, 2000.
16. A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. *Theory and Practice of Logic Programming*, 2(2):125–154, 2002.
17. A. Bossi, S. Etalle, S. Rossi, and J.-G. Smaus. Semantics and termination of simply moded logic programs with dynamic scheduling. *Transactions on Computational Logic*, 2004. To appear in summer 2004.
18. C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of Prolog. In M. Bruynooghe, editor, *Proc. of the International Logic Programming Symposium*, pages 457—471. MIT Press, 1994.
19. M. Bruynooghe, H. Vandecasteele, D. A. de Waal, and M. Denecker. Detecting unsolvable queries for definite logic programs. In C. Palamidessi et al., editor, *Proc. of PLILP/ALP '98*, volume 1490 of *LNCS*, pages 118–133. Springer-Verlag, 1998.
20. L. Cavedon. Continuity, consistency, and completeness properties for logic programs. In G. Levi and M. Martelli, editors, *Proceedings of the 6th International Conference on Logic Programming*, pages 571–584. The MIT Press, 1989.
21. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
22. L. Colussi, E. Marchiori, and M. Marchiori. On termination of constraint logic programs. In M. Bruynooghe and J. Penjam, editors, *Proc. of the 1st International Conference of Principles and Practice of Constraint Programming*, volume 976 of *LNCS*, pages 431–448. Springer-Verlag, 1995.
23. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.
24. D. De Schreye and A. Serebrenik. Acceptability with general orderings. In F. Sadri and A. Kakas, editors, *Computational Logic: Logic Programming and Beyond, Part I*, volume 2407 of *LNCS*, pages 187–210. Springer-Verlag, 2002.
25. D. De Schreye, K. Verschaetse, and M. Bruynooghe. 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*, pages 481–488. Institute for New Generation Computer Technology, 1992.

26. S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.
27. S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms: A missing link in automatic termination analysis. In D. Miller, editor, *Proc. of the International Logic Programming Symposium*, pages 420–436. The MIT Press, 1993.
28. S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K. Sagonas. Termination analysis for tabled logic programming. In N. E. Fuchs, editor, *Proc. of the 7th International Workshop on Logic Programming Synthesis and Transformation*, volume 1463 of *LNCS*, pages 111–127. Springer-Verlag, 1998.
29. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 21(6):1137–1195, 1999.
30. P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming*. The MIT Press, 1993.
31. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 2001(1/2):117–156, 2001.
32. S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *Journal of Logic Programming*, 38(2):243–257, 1999.
33. R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proc. Symposium in Applied Mathematics, vol. 19 of Mathematical Aspects in Computer Science*, pages 19–32. AMS, 1967.
34. T. Frühwirth. Proving termination of constraint solver programs. In K. R. Apt et al., editor, *New Trends in Constraints*, volume 1865 of *LNAI*, 2000.
35. H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. In M. Rusinowitch and J. L. Rémy, editors, *Proc. of the 3rd International Workshop on Conditional Term Rewriting Systems*, volume 656 of *LNCS*, pages 430–437. Springer-Verlag, 1992.
36. R. Gori. An abstract interpretation approach to termination of logic programs. In M. Parigot and A. Voronkov, editors, *Proc. of the 7th International Conference on Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS*, pages 362–380. Springer-Verlag, 2000.
37. M. V. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program analysis, debugging, and optimization using the Ciao system preprocessor. In D. De Schreye, editor, *Proc. of the International Conference on Logic Programming*, pages 52–66. MIT Press, 1999.
38. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.
39. S. Hoarau and F. Mesnard. Inferring and compiling termination for constraint logic programs. In P. Flener, editor, *Proc. of the 8th International Workshop on Logic Programming Synthesis and Transformation*, volume 1559 of *LNCS*, pages 240–254. Springer-Verlag, 1998.
40. R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.
41. M .R. K. Krishna Rao, D. Kapur, and R. K. Shyamasundar. A transformational methodology for proving termination of logic programs. In E. Börger, G. Jäger, H. Kleine Büning, and M. M. Richter, editors, *Proc. of the 5th Workshop on Computer Science Logic*, volume 626 of *LNCS*, pages 213–226. Springer-Verlag, 1992.
42. M. R. K. Krishna Rao, D. Kapur, and R. K. Shyamasundar. Proving termination of GHC programs. *New Generation Computing*, 15(3):293–338, 1997.

43. M. R. K. Krishna Rao, D. Kapur, and R. K. Shyamasundar. Transformational methodology for proving termination of logic programs. *Journal of Logic Programming*, 34(1):1–41, 1998.
44. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In L. Naish, editor, *Proc. of the 14th International Conference on Logic Programming*, pages 63–77. The MIT Press, 1997.
45. S. Lüttringhaus-Kappel. Control generation for logic programs. In D. S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 478–495. MIT Press, 1993.
46. E. Marchiori. On termination of general logic programs w.r.t. constructive negation. *Journal of Logic Programming*, 26(1):69–89, 1996.
47. E. Marchiori and F. Teusink. On termination of logic programs with delay declarations. *Journal of Logic Programming*, 39(1-3):95–124, 1999.
48. M. Marchiori. Proving existential termination of normal logic programs. In M. Wirsing and M. Nivat, editors, *Proc. of the 5th International Conference on Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 375–390. Springer-Verlag, 1996.
49. J. Martin and A. King. Generating efficient, terminating logic programs. In M. Bidoit and M. Dauchet, editors, *Proc. of the 7th International Conference on Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 273–284. Springer-Verlag, 1997.
50. R. McPhee. *Compositional Logic Programming.* PhD thesis, Oxford University Computing Laboratory, 2000.
51. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In M. Maher, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 7–21. The MIT Press, 1996.
52. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In P. Cousot, editor, *Proc. of the 8th Static Analysis Symposium*, volume 2126 of *LNCS*, pages 93–110. Springer-Verlag, 2001.
53. F. Mesnard, É. Payet, and U. Neumerkel. Detecting optimal termination conditions of logic programs. In M. V. Hermenegildo and G. Puebla, editors, *Proc. of the 9th Static Analysis Symposium*, volume 2477 of *LNCS*, pages 509–526. Springer-Verlag, 2002.
54. F. Mesnard and S. Ruggieri. On proving left-termination of constraint logic programs. *ACM Transactions on Computational Logic*, 4(2):207–259, 2003.
55. L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, Department of Computer Science, University of Melbourne, 1992.
56. E. Ohlebusch, C. Claves, and C. Marché. TALP: A tool for the termination analysis of logic programs. In Leo Bachmair, editor, *Proc. of the 11th International Conference on Rewriting Techniques and Applications*, volume 1833 of *Lecture Notes in Computer Science*, pages 270–273. Springer-Verlag, 2000.
57. D. Pedreschi P. Mancarella and S. Ruggieri. Negation as failure through abduction: Reasoning about termination. In F. Sadri and A. Kakas, editors, *Computational Logic: Logic Programming and Beyond, Part I*, volume 2407 of *LNCS*, pages 240–272. Springer-Verlag, 2002.
58. D. Pedreschi and S. Ruggieri. Bounded nondeterminism of logic programs. In D. De Schreye, editor, *Proc. of the International Conference on Logic Programming*, pages 350–364. The MIT Press, 1999. Extended version to appear in *Annals of Mathematics and Artificial Intelligence.*

59. D. Pedreschi and S. Ruggieri. On logic programs that always succeed. *Science of Computer Programming*, 48(2-3):163–196, 2003. Extended version of the paper "On logic programs that do not fail", Proc. of ICLP 1999 Workshop on Verification of Logic Programs, ENTCS 30(1) 1999.
60. D. Pedreschi, S. Ruggieri, and J.-G. Smaus. Classes of terminating logic programs. *Theory and Practice of Logic Programming*, 2(3):369–418, 2002.
61. S. Ruggieri. Termination of constraint logic programs. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proc. of the 24th International Colloquium on Automata, Languages and Programming (ICALP '97)*, volume 1256 of *LNCS*, pages 838–848. Springer-Verlag, 1997.
62. S. Ruggieri. *Verification and Validation of Logic Programs*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999.
63. S. Ruggieri. $\exists$-universal termination of logic programs. *Theoretical Computer Science*, 254(1-2):273–296, 2001.
64. A. Serebrenik. *Termination Analysis of Logic Programs*. PhD thesis, Katholieke Universiteit, Leuven, 2003.
65. A. Serebrenik and D. De Schreye. On termination of logic programs with floating point computations. In M. V. Hermenegildo and G. Puebla, editors, *Proc. of the 9th Static Analysis Symposium*, volume 2477 of *LNCS*, pages 151–164. Springer-Verlag, 2002.
66. J.-G. Smaus. *Modes and Types in Logic Programming*. PhD thesis, University of Kent at Canterbury, 1999.
67. J.-G. Smaus. Proving termination of input-consuming logic programs. In D. De Schreye, editor, *Proc. of the International Conference on Logic Programming*, pages 335–349. MIT Press, 1999.
68. J.-G. Smaus. Termination of logic programs for various dynamic selection rules. Technical Report 191, Insitut für Informatik, Universität Freiburg, 2003.
69. J.-G. Smaus, P. M. Hill, and A. M. King. Verifying termination and error-freedom of logic programs with `block` declarations. *Theory and Practice of Logic Programming*, 1(4):447–486, 2001.
70. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.
71. C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In P. Van Hentenryck, editor, *Proc. of the 4th International Static Analysis Symposium*, volume 1302 of *LNCS*, pages 160–171. Springer-Verlag, 1997.
72. L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1986.
73. Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 2003. `http://www.sics.se/isl/sicstuswww/site/documentation.html`.
74. J. Thom and J. Zobel. NU-Prolog reference manual, version 1.3. Technical report, Department of Computer Science, University of Melbourne, Australia, 1988.
75. K. Ueda. Guarded Horn Clauses, a parallel logic programming language with the concept of a guard. In M. Nivat and K. Fuchi, editors, *Programming of Future Generation Computers*, pages 441–456. North Holland, Amsterdam, 1988.
76. S. Verbaeten. Termination analysis for abductive general logic programs. In D. De Schreye, editor, *Proc. of the International Conference on Logic Programming*, pages 365–379. The MIT Press, 1999.
77. S. Verbaeten, K. Sagonas, and D. De Schreye. Termination proofs for logic programs with tabling. *ACM Transactions on Computational Logic*, 2(1):57–92, 2001.