Proving Termination of Input-Consuming Logic Programs*

Jan–Georg Smaus†
Computing Laboratory
University of Kent at Canterbury
Canterbury, Kent, CT2 7NF, United Kingdom

September 1999

Abstract

A class of predicates is identified for which termination does not depend on left-to-right execution. The only assumption about the selection rule is that derivations are *input-consuming*, that is, in each derivation step, the input arguments of the selected atom do not become instantiated. This assumption is a natural abstraction of previous work on programs with delay declarations. The method for showing that a predicate is in that class is based on level mappings, closely following the traditional approach for LD-derivations. Programs are assumed to be well and nicely moded, which are two widely used concepts for verification. Many predicates terminate under such weak assumptions. Knowing these predicates is useful even for programs where not *all* predicates have this property.

1 Introduction

Termination of logic programs has been widely studied for LD-derivations, that is derivations where the leftmost atom in a query is always selected [1, 3, 7, 8, 9, 10, 12]. All of these works are based on the following idea: at the time when an atom a in a query is selected, it is possible to *pin down the size*[1] of a. This size cannot change via further instantiation. It is then shown that for the atoms introduced in this derivation step, it is again possible to pin down

*This Technical Report 10-99 is the long version of a paper with the same title presented at ICLP'99 [22].

†The author is now working at INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France.

[1]The technical meaning of "pinning down the size" differs among different methods. This will be discussed in Sect. 7.

their size when eventually they are selected, and these atoms are smaller than a.

This idea has also been applied to arbitrary derivations [6]. Since no restriction is imposed on when an atom can be selected, it is required that in each query in a derivation, the size of each atom is always bounded. Programs that fulfill this requirement are called *strongly terminating*. The class of strongly terminating programs is very limited.

For most logic programs, it is necessary for termination to require a certain degree of instantiation of an atom before it can be selected. This can be achieved using delay declarations [2, 17, 18, 19, 20, 23, 24]. The problem is that, depending on what kind of delay declarations and selection rule are used, it is often not possible to pin down the size of the selected atom, since this size may depend on the resolution of other atoms in the query that are not yet resolved. Nevertheless, the approaches by Marchiori and Teusink [18] and Martin and King [19], and to a limited extent Lüttringhaus-Kappel [17] are based on the idea described above. Other approaches avoid any explicit mention of "size" and instead try to reduce the problem to showing termination for LD-derivations [20, 23, 24].

Our approach falls between the two extremes of making no assumptions about the selection rule on the one hand and making very specific assumptions on the other. We believe that a reasonable minimal requirement for termination can be formulated in terms of *modes*:

> In each derivation step, the input arguments of the selected atom cannot become instantiated.

In other words, an atom in a query can only be selected when it is sufficiently instantiated so that the most general unifier (MGU) with the clause head does not bind the input arguments of the atom. We call derivations which meet this requirement *input-consuming*.

This paper is about identifying predicates for which all input-consuming derivations are finite. Other works in this area have usually made specific assumptions about the selection rule and the delay declarations, for example *local* selection rules [18], delay declarations that test arguments for groundness or rigidness [17, 19], or the default left-to-right selection rule of most Prolog implementations [20, 23, 24]. In contrast, we show how previous results about LD-derivations can be generalised, the only assumption about the selection rule being that derivations are input-consuming.

We exploit that under certain conditions, it is enough to rely on a *relative* decrease in the size of the selected atom, even though this size cannot be pinned down.

Example 1.1 Consider the usual `append` program, where the first two argument positions are input positions. The following is an input-consuming derivation. The selected atom is always underlined. On the right hand side, we indicate some of the variable bindings made in this derivation.

$$
\begin{array}{ll}
\underline{\texttt{append}([1],[],\texttt{As})},\texttt{append}(\texttt{As},[],\texttt{Bs}) \leadsto & (\texttt{As} = [1|\texttt{As}']) \\
\texttt{append}([],[],\texttt{As}'),\underline{\texttt{append}([1|\texttt{As}'],[],\texttt{Bs})} \leadsto & (\texttt{Bs} = [1|\texttt{Bs}']) \\
\underline{\texttt{append}([],[],\texttt{As}')},\texttt{append}(\texttt{As}',[],\texttt{Bs}') \leadsto & (\texttt{As}' = []) \\
\underline{\texttt{append}([],[],\texttt{Bs}')} \leadsto \square & (\texttt{Bs}' = [])
\end{array}
$$

When $\texttt{append}([1|\texttt{As}'],[],\texttt{Bs})$ is selected, it is not possible to pin down its size in any meaningful way. In fact, nothing can be said about the length of the (input-consuming) derivation associated with $\texttt{append}([1|\texttt{As}'],[],\texttt{Bs})$ without knowing about other atoms which might instantiate $\texttt{As}'$. However, the derivation could be infinite only if some derivation associated with $\texttt{append}([],[],\texttt{As}')$ was infinite. Our method is based on such a dependency between the atoms of a query.

As we will discuss in Sect. 7, previous approaches [6, 17, 18, 19] cannot formally show termination of derivations with coroutining such as the one above.

Even though the class of programs for which all input-consuming derivations are finite is obviously larger than the class of strongly terminating programs, it is still quite limited. The following example illustrates this.

Example 1.2 Consider the following program, where for both predicates, the first position is the only input position.

```
permute([], []).                delete([X|Z], X, Z).
permute(Y, [U | X]) :-          delete([U|Y], X, [U|Z]) :-
  delete(Y, U, Z),                delete(Y, X, Z).
  permute(Z, X).
```

Then we have the following infinite input-consuming derivation:

$$
\begin{array}{ll}
\underline{\texttt{permute}([1],\texttt{W})} \leadsto & (\texttt{W} = [\texttt{U}'|\texttt{X}']) \\
\underline{\texttt{delete}([1],\texttt{U}',\texttt{Z}')},\texttt{permute}(\texttt{Z}',\texttt{X}') \leadsto & (\texttt{Z}' = [1|\texttt{Z}'']) \\
\texttt{delete}([],\texttt{U}',\texttt{Z}''),\underline{\texttt{permute}([1|\texttt{Z}''],\texttt{X}')} \leadsto & (\texttt{X}' = [\texttt{U}''|\texttt{X}'']) \\
\texttt{delete}([],\texttt{U}',\texttt{Z}''),\underline{\texttt{delete}([1|\texttt{Z}''],\texttt{U}'',\texttt{Z}''')},\texttt{permute}(\texttt{Z}''',\texttt{X}'') \leadsto & \\
\texttt{delete}([],\texttt{U}',\texttt{Z}''),\texttt{delete}(\texttt{Z}'',\texttt{U}'',\texttt{Z}''''),\underline{\texttt{permute}([1|\texttt{Z}''''],\texttt{X}'')} \leadsto \ldots &
\end{array}
$$

To ensure termination even for programs like the one above, most authors have made stronger assumptions about the selection rule, thereby neglecting the important class for which assuming input-consuming derivations is sufficient. We have attempted to formulate our results as generally as possible to make them widely applicable.

The rest of this paper is organised as follows. The next section fixes the notation. Section 3 introduces well and nicely moded programs and Section 4 shows that for these, it is sufficient to prove termination for one-atom queries. Section 5 then deals with how one-atom queries can be proven to terminate. In Sect. 6 we sketch how the method presented here could be applied. Section 7 discusses the results and the related work.

2 Preliminaries

Our notation follows Apt [1] and Etalle et al. [12]. For the examples we use Prolog syntax. We recall some important notions. The set of variables in a syntactic object o is denoted as $vars(o)$. A syntactic object is **linear** if every variable occurs in it at most once. The **domain** of a substitution σ is $dom(\sigma) = \{x \mid x\sigma \neq x\}$.

For a predicate p/n, a **mode** is an atom $p(m_1, \ldots, m_n)$, where $m_i \in \{I, O\}$ for $i \in \{1, \ldots, n\}$. Positions with I are called **input positions**, and positions with O are called **output positions** of p. We assume that a fixed mode is associated with each predicate in a program. To simplify the notation, an atom written as $p(\mathbf{s}, \mathbf{t})$ means: $\mathbf{s}$ is the vector of terms filling the input positions, and $\mathbf{t}$ is the vector of terms filling the output positions. An atom $p(\mathbf{s}, \mathbf{t})$ is **input-linear** if $\mathbf{s}$ is linear, **output-linear** if $\mathbf{t}$ is linear.

A **query** is a finite sequence of atoms. Atoms are denoted by a, b, h, queries by B, F, H, Q, R. We write $a \in B$ if a is an atom in B. A **derivation step** for a program P is a pair $\langle Q, \theta\rangle; \langle R, \theta\sigma\rangle$, where $Q = Q_1, p(\mathbf{s}, \mathbf{t}), Q_2$ and $R = Q_1, B, Q_2$ are queries; θ is a substitution; $p(\mathbf{v}, \mathbf{u}) \leftarrow B$ a renamed variant of a clause in P and σ an MGU of $p(\mathbf{s}, \mathbf{t})\theta$ and $p(\mathbf{v}, \mathbf{u})$. We call $p(\mathbf{s}, \mathbf{t})\theta$ the **selected atom** and $R\theta\sigma$ the **resolvent** of $Q\theta$ and $h \leftarrow B$. A derivation step is **input-consuming** if $dom(\sigma) \cap vars(\mathbf{s}\theta) = \emptyset$.[2]

A **derivation** ξ **for** a program P is a sequence $\langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle; \ldots$ where each pair $\langle Q_i, \theta_i\rangle; \langle Q_{i+1}, \theta_{i+1}\rangle$ in ξ is a derivation step. Alternatively, we also say that ξ is a **derivation of** $P \cup \{Q_0\theta_0\}$. We sometimes denote a derivation as $Q_0\theta_0; Q_1\theta_1; \ldots$. An **LD**-derivation is a derivation where the selected atom is always the leftmost atom in a query. An **input-consuming** derivation is a derivation consisting of input-consuming derivation steps.

If (F, a, H); $(F, B, H)\theta$ is a step in a derivation, then each atom in $B\theta$ is a **direct descendant** of a, and $b\theta$ is a **direct descendant** of b for all $b \in F, H$. We say b is a **descendant of** a if (b, a) is in the reflexive, transitive closure of the relation *is a direct descendant*. The descendants of a *set* of atoms are defined in the obvious way. Consider a derivation $Q_0; \ldots; Q_i; \ldots; Q_j; Q_{j+1}; \ldots$. We call $Q_j; Q_{j+1}$ an a**-step** if a is an atom in Q_i and the selected atom in $Q_j; Q_{j+1}$ is a descendant of a.

[2]Since the MGU is unique up to variable renaming, we may assume that whenever possible, an MGU σ is used such that $dom(\sigma) \cap vars(\mathbf{s}\theta) = \emptyset$.

3 Modes

In this section we introduce well moded and nicely moded programs, which are standard concepts used for verification of logic programs [2, 5, 11, 12, 13].

Well-modedness has been introduced by Dembinski and Małuszyński [11] and widely used since. In Mercury it is even mandatory that programs are well moded (possibly after reordering of atoms by the compiler), which is one of the reasons for its remarkable performance [25].

Definition 3.1 [well moded] 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 $i \in \{1, \ldots, n\}$ and $L = 1$

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

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

Note that a one-atom query $p(\mathbf{s}, \mathbf{t})$ is well moded if and only if $\mathbf{s}$ is ground.

Another widely used concept is the following.

Definition 3.2 [nicely moded] A query $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is **nicely moded** if $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a linear vector of terms and for all $i \in \{1, \ldots, n\}$

$$vars(\mathbf{s}_i) \cap \bigcup_{j=i}^{n} vars(\mathbf{t}_j) = \emptyset. \tag{2}$$

The clause $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is **nicely moded** if Q is nicely moded and

$$vars(\mathbf{t}_0) \cap \bigcup_{j=1}^{n} vars(\mathbf{t}_j) = \emptyset. \tag{3}$$

A program is **nicely moded** if all of its clauses are nicely moded.

Note that a one-atom query $p(\mathbf{s}, \mathbf{t})$ is nicely moded if and only if $vars(\mathbf{s}) \cap vars(\mathbf{t}) = \emptyset$ and $\mathbf{t}$ is linear. We can thus state the following proposition which follows from the definitions.

Proposition 3.1 A one-atom query $p(\mathbf{s}, \mathbf{t})$ is well and nicely moded if and only if $\mathbf{s}$ is ground and $\mathbf{t}$ is linear.

Example 3.1 The program in Ex. 1.2 is well and nicely moded in mode $\{\texttt{permute}(I, O), \texttt{delete}(I, O, O)\}$. It is neither well moded nor nicely moded in mode $\{\texttt{permute}(O, I), \texttt{delete}(O, I, I)\}$, however it can easily be made well and nicely moded by interchanging the two body atoms in the second clause.

The example shows that multiple modes of a predicate can be obtained by maintaining multiple (renamed) versions of a predicate, which differ in the order of atoms in the clause bodies. This is why some authors find it justified to assume that each predicate has a fixed mode [12, 20, 25]. However, in those works, assuming a fixed mode is, from a formal point of view, a real restriction, even if one may find that this code duplication is not a problem in practice.

In this paper, assuming a fixed mode for each predicate is *not at all* a restriction. We consider derivations where the textual position of an atom within a query is irrelevant for its selection. Therefore it immediately follows that if we show termination for a program, we have also shown termination for the same program where the atoms in each clause body are permuted in an arbitrary way. In this sense, we can assume that the program of Ex. 1.2 is well moded and nicely moded in *both* modes (Ex. 3.1). It is merely for notational convenience that we assume, in all formal statements, a "left-to-right" data flow in the above definitions.

Of course, using a program in different modes at each run requires that the selection rule somehow "knows" what mode is assumed in a particular run, since otherwise it would not be defined what an input-consuming derivation is. This can be realised for example by using delay declarations [23, 24], but in this paper, we do not worry about how this is achieved.

If one considers derivations where the textual position of an atom *is* relevant for its selection, one needs more general definitions than the ones above that involve permutations of the atoms [23]. The relationship of the textual order and the direction of data flow is discussed in detail in [21, Sect. 5.3].

The following lemmas state persistence properties of well-modedness and nicely-modedness.

Lemma 3.2 Every resolvent of a well moded query Q and a well moded clause C, where $vars(C) \cap vars(Q) = \emptyset$, is well moded [2, Lemma 16].

Lemma 3.3 Every resolvent of a nicely moded query Q and a nicely moded clause C, where $vars(C) \cap vars(Q) = \emptyset$ and the head of C is input-linear, is nicely moded [2, Lemma 11].

For input-consuming derivations, the requirement that the clause head is input-linear can be dropped. It is assumed that the selected atom is sufficiently instantiated, so that a multiple occurrence of the same variable in the input arguments of the clause head cannot cause any bindings to the query. Note that requiring input-linear clause heads is quite a severe restriction in that it rules out input arguments of the selected atom being tested for equality.

Lemma 3.4 Every resolvent of a nicely moded query Q and a nicely moded clause C, where the derivation step is input-consuming and $vars(C) \cap vars(Q) = \emptyset$, is nicely moded.

PROOF. Let $Q = a_1, \ldots, a_n$, $C = p(\mathbf{v}, \mathbf{u}) \leftarrow b_1, \ldots, b_m$, and suppose for some $k \in \{1, \ldots, n\}$, $a_k = p(\mathbf{s}, \mathbf{t})$ and $p(\mathbf{v}, \mathbf{u})$ are unifiable with MGU θ, and $dom(\theta) \cap vars(\mathbf{s}) = \emptyset$.

Now let $C' = p(\mathbf{v}', \mathbf{u}) \leftarrow b_1, \ldots, b_m$ be an input-linear clause such that

1. $vars(\mathbf{v}) \subseteq vars(\mathbf{v}')$ and $vars(\mathbf{v}') \cap vars(Q) = \emptyset$,
2. there exists a substitution σ such that $C'\sigma = C$ and $dom(\sigma) = vars(\mathbf{v}') \setminus vars(\mathbf{v})$.

Intuitively, $\mathbf{v}'$ is obtained from $\mathbf{v}$ by renaming, for each variable occurring several times, all but one occurrences apart using fresh variables.

Since $dom(\theta) \cap vars(\mathbf{s}) = \emptyset$, it follows that $\theta = \theta_1\theta_2$, where θ_1 is an MGU of $\mathbf{v}$ and $\mathbf{s}$, and $\mathbf{v}\theta_1 = \mathbf{s}$, and θ_2 is an MGU of $\mathbf{u}\theta_1$ and $\mathbf{t}\theta_1$.

By (2) and since $\mathbf{v}\theta_1 = \mathbf{s}$, we have $\mathbf{v}'\sigma\theta_1 = \mathbf{s}$. Moreover by (1), (2) and since $dom(\theta_1) \subseteq vars(\mathbf{v})$, we have $dom(\sigma\theta_1) \subseteq vars(\mathbf{v}')$, and hence $\sigma\theta_1$ is an MGU of $\mathbf{v}'$ and $\mathbf{s}$.

By (2), $\mathbf{u}\sigma = \mathbf{u}$ and $\mathbf{t}\sigma = \mathbf{t}$. Therefore θ_2 is an MGU of $\mathbf{u}\sigma\theta_1$ and $\mathbf{t}\sigma\theta_1$.

So we have that $\sigma\theta_1$ is an MGU of $\mathbf{v}'$ and $\mathbf{s}$, and θ_2 is an MGU of $\mathbf{u}\sigma\theta_1$ and $\mathbf{t}\sigma\theta_1$. Therefore $\sigma\theta_1\theta_2 = \sigma\theta$ is an MGU of $p(\mathbf{v}', \mathbf{u})$ and $p(\mathbf{s}, \mathbf{t})$ [1, Lemma 2.24]. Hence by Lemma 3.3, $(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\sigma\theta$ is a nicely moded resolvent of C' and Q. However, by (1) and (2),

$$\begin{aligned}&(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\theta = \\ &(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\sigma\theta,\end{aligned}$$

and so $(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\theta$ is nicely moded. □

For a nicely moded program and query, it is guaranteed that every input-consuming derivation step only instantiates other atoms in the query that occur to the right of the selected atom.

Lemma 3.5 Let P be a nicely moded program, $Q = Q_1, p(\mathbf{s}, \mathbf{t}), Q_2$ a nicely moded query, and $\langle Q, \emptyset \rangle; \langle Q_1, B, Q_2,\ \sigma \rangle$ an input-consuming derivation step. Then $dom(\sigma) \cap vars(Q_1) = \emptyset$.

PROOF. Since the derivation step is input-consuming, $dom(\sigma) \cap vars(Q) \subseteq vars(\mathbf{t})$. Thus since Q is nicely moded, $dom(\sigma) \cap vars(Q_1) = \emptyset$. □

This section mainly served the purpose of recalling some well-known mode concepts. However, Lemma 3.4 is an original result.

4 Controlled Coroutining

In this section we define *atom-terminating* predicates. A predicate p is atom-terminating if (under certain conditions) all input-consuming derivations of a query $p(\mathbf{s}, \mathbf{t})$ are finite. Like Etalle et al. [12], we then show that termination for one-atom queries implies termination for arbitrary queries.

For LD-derivations, it is almost obvious that it is sufficient to show termination for one-atom queries, and it only requires that programs and queries are well moded [12, Lemma 4.2]. Given an LD-derivation ξ for a query $a_1, \ldots, a_n$, the sub-derivations for each a_i do not interleave, and therefore ξ can be regarded as a derivation for a_1 followed by a derivation for a_2 and so forth. The following example illustrates that in the context of interleaving sub-derivations (coroutining), this is by no means obvious.

Example 4.1 Consider the usual append program

```
append([],Y,Y).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

in mode $\mathtt{append}(I, I, O)$ and the query

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

This query is well moded but not nicely moded. Then we have the following infinite input-consuming derivation:

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

This well-known termination problem of programs with coroutining has been identified as *circular modes* [20].

To avoid the problem, we require programs and queries to be nicely moded. Recall that by Prop. 3.1, a one-atom query $p(\mathbf{s}, \mathbf{t})$ is well and nicely moded if and only if $\mathbf{s}$ is ground and $\mathbf{t}$ is linear.

Definition 4.1 [atom-terminating predicate/atom] Let P be a well and nicely moded program. A predicate p in P is **atom-terminating** if for each well and nicely moded query $p(\mathbf{s}, \mathbf{t})$, all input-consuming derivations of $P \cup \{p(\mathbf{s}, \mathbf{t})\}$ are finite. An atom is **atom-terminating** if its predicate is atom-terminating.

The following lemma says that an atom-terminating atom cannot proceed indefinitely unless it is repeatedly fed by some other atom. The lemma is similar to [23, Lemma 4.2]. Note however that here, we do not require that clause heads are input-linear. There is a lemma [21, Lemma 6.2] which subsumes both [23, Lemma 4.2] and Lemma 4.1, but using this lemma would complicate this paper considerably.

Lemma 4.1 Let P be a well and nicely moded program and F, b, H a well and nicely moded query where b is an atom-terminating atom. An input-consuming derivation of $P \cup \{F, b, H\}$ can have infinitely many b-steps only if it has infinitely many a-steps, for some $a \in F$.

PROOF. In this proof, by an **F-step** we mean an a-step, for some $a \in F$; likewise we define an **H-step**. By Lemma 3.5, no H-step can instantiate any descendant of b. Thus the H-steps can be disregarded, and without loss of generality, we assume H is empty. Suppose ξ is an input-consuming derivation for $P \cup \{F, b\}$ containing finitely many F-steps. Let

$$\xi = \langle F,b,\ \emptyset\rangle; \ldots; \langle Q_0, \theta_0\rangle; \tilde{\xi}$$

such that $\langle Q_0, \theta_0\rangle; \tilde{\xi}$ contains no F-steps. Since by Lemma 3.5, no b-step can instantiate any descendant of F, there exists an input-consuming derivation

$$\xi_2 = \langle F,b,\ \emptyset\rangle; \ldots; \langle R, \rho\rangle; \ldots; \langle Q_0, \theta_0\rangle; \tilde{\xi}$$

such that $\langle F,b,\ \emptyset\rangle; \ldots; \langle R, \rho\rangle$ contains only F-steps and $\langle R, \rho\rangle; \ldots; \langle Q_0, \theta_0\rangle; \tilde{\xi}$ contains only b-steps (that is, the F-steps are moved forward using the Switching Lemma [16]). Since $R = R', b$ for some R', there exists an input-consuming derivation

$$\xi_3 = \langle b, \rho\rangle; \ldots; \langle I_0, \theta_0\rangle; \tilde{\xi}_3$$

obtained from $\langle R, \rho\rangle; \ldots; \langle Q_0, \theta_0\rangle; \tilde{\xi}$ by removing the prefix R' in each query.

By Lemmas 3.2 and 3.4, $R\rho$ is well and nicely moded. Let V be the set of variables in the output positions of $R'\rho$ and σ a substitution such that $dom(\sigma) = V$ and $V\sigma$ is ground. Then by Def. 3.1, $b\rho\sigma$ is ground in its input positions. Moreover, since σ does not instantiate the output arguments of $b\rho$, it follows that $b\rho\sigma$ is output-linear. Thus by Prop. 3.1, $b\rho\sigma$ is well and nicely moded.

By Lemma 3.5, no b-step in ξ_2, and hence no derivation step in ξ_3, can instantiate a variable in V. Since $dom(\sigma) = V$, it thus follows that from ξ_3 we can construct an input-consuming derivation

$$\xi_4 = \langle b, \rho\sigma\rangle; \ldots; \langle I_0, \theta_0\sigma\rangle; \tilde{\xi}_3\sigma \ldots$$

Since $b\rho\sigma$ is a well and nicely moded query and b is atom-terminating, ξ_4 is finite. Therefore ξ_3, ξ_2, and finally ξ are finite. □

The following theorem is a consequence of Lemma 4.1 and states that atom-terminating atoms on their own cannot produce an infinite derivation.

Theorem 4.2 Let P be a well and nicely moded program and Q a well and nicely moded query. An input-consuming derivation of $P \cup \{Q\}$ can be infinite only if it contains infinitely many steps where an atom is resolved that is not atom-terminating.

PROOF. We first show that for any well and nicely moded query Q', an input-consuming derivation of $P \cup \{Q'\}$ can be infinite only if it contains *at least one* step where an atom is resolved that is not atom-terminating ($*$). So let ξ' be an infinite input-consuming derivation of $P \cup \{Q'\}$.

If for each atom-terminating $b \in Q'$, the derivation ξ' contains only finitely many b-steps, then ξ' contains infinitely many a-steps for some $a \in Q'$ which is not atom-terminating. Hence the first a-step in ξ' is a step where an atom is resolved that is not atom-terminating. So in this case, ($*$) follows.

Otherwise, let $Q' = F, b, H$, where b is the first atom-terminating atom in Q' such that ξ' contains infinitely many b-steps. Then by Lemma 4.1, ξ' contains infinitely many a-steps, for some $a \in F$ that is not atom-terminating. Hence the first a-step in ξ' is a step where an atom is resolved that is not atom-terminating. So in this case, ($*$) follows as well.

Now let ξ be an infinite input-consuming derivation of $P \cup \{Q\}$. Assume, for the purpose of deriving a contradiction, that ξ contains only finitely many steps where an atom is resolved that is not atom-terminating. Let $\tilde{\xi}$ be a suffix of ξ containing no steps where an atom is resolved that is not atom-terminating. By Lemmas 3.2 and 3.4, the first query of $\tilde{\xi}$ is well and nicely moded. Moreover, $\tilde{\xi}$ is infinite, and so we have a contradiction to ($*$). Thus it follows that ξ contains infinitely many steps where an atom is resolved that is not atom-terminating, which completes the proof. $\square$

Theorem 4.2 provides us with the formal justification for restricting our attention to one-atom queries. Thus the question is how it can be shown that a predicate is atom-terminating.

5 Showing that a Predicate is Atom-Terminating

Termination proofs usually rely, more or less explicitly, on measuring the size of the *input* in a query [1, 3, 7, 8, 9, 10, 12]. We agree with Etalle et al. [12] that it is reasonable to make this dependency explicit. This gives rise to the concept of *moded level mapping* [12], which is an instance of *level mapping* [6]. $\mathbf{B}_P$ denotes the set of ground atoms using predicates occurring in P.

Definition 5.1 [moded level mapping] Let P be a program. $|.|$ is a **moded level mapping** if

1. it is a level mapping, that is a function $|.| : \mathbf{B}_P \rightsquigarrow \mathbb{N}$,
2. for any $\mathbf{t}$ and $\mathbf{u}$, $|p(\mathbf{s}, \mathbf{t})| = |p(\mathbf{s}, \mathbf{u})|$.

For $a \in \mathbf{B}_P$, $|a|$ is the **level** of a.

Thus the level of an atom only depends on the terms in the input positions.

The following concept is useful for proving termination for a whole program incrementally, by proving it for one predicate at a time [1].

Definition 5.2 [depends on] Let p, q be predicates in a program P. We say p **refers to** q if there is a clause in P with p in its head and q in its body, and p **depends on** q (written $p \sqsupseteq q$) if (p, q) is in the reflexive, transitive closure of *refers to*. We write $p \sqsupset q$ if $p \sqsupseteq q$ and $q \not\sqsupseteq p$, and $p \approx q$ if $p \sqsupseteq q$ and $q \sqsupseteq p$.

Abusing notation, we shall also use the above symbols for *atoms*, where $p(\mathbf{s}, \mathbf{t}) \sqsupseteq q(\mathbf{u}, \mathbf{v})$ stands for $p \sqsupseteq q$, and likewise for $\sqsupset$ and $\approx$. Furthermore, we denote the equivalence class of a predicate p with respect to $\approx$ as $[p]_\approx$.

The following definition provides us with a criterion to prove that a predicate is atom-terminating.

Definition 5.3 [ICD-acceptable] Let P be a program and $|.|$ a moded level mapping. A clause $C = h \leftarrow B$ is **acceptable for input-consuming derivations (with respect to $|.|$)** if for every substitution θ such that $C\theta$ is ground, and for every $a \in B$ such that $a \approx h$, we have $|h\theta| > |a\theta|$. We abbreviate *acceptable for input-consuming derivations* by **ICD-acceptable**.

A set of clauses is **ICD-acceptable with respect to** $|.|$ if each clause is ICD-acceptable with respect to $|.|$.

Let us compare this concept with some similar concepts in the literature: *recurrent* [6], *well-acceptable* [12] and *acceptable* [4, 10] programs.

Like Decorte and De Schreye [10] and Etalle et al. [12] but unlike Apt and Pedreschi [4] and Bezem [6], we require $|h\theta| > |a\theta|$ only for atoms a where $a \approx h$. This is consistent with the idea that termination should be proven incrementally: to show termination for a predicate p, it is assumed that all predicates q with $p \sqsupset q$ have already been shown to terminate. Therefore we can restrict our attention to the predicates q where $q \approx p$.

Like Bezem but unlike Apt and Pedreschi, Decorte and De Schreye and Etalle et al., our definition does not involve models or computed answer substitutions. Traditionally, the definition of acceptable programs is based on a model M of the program, and for a clause $h \leftarrow a_1, \ldots, a_n, \quad |h\theta| > |a_i\theta|$ is only required if $M \models (a_1, \ldots, a_{i-1})\theta$. The reason is that for LD-derivations, $a_1, \ldots, a_{i-1}$ must be completely resolved before a_i is selected. By the correctness of LD-resolution [16] and well-modedness [5], the accumulated answer substitution θ, just before a_i is selected, is such that $(a_1, \ldots, a_{i-1})\theta$ is ground and $M \models (a_1, \ldots, a_{i-1})\theta$.

Such considerations count for little when derivations are merely required to be input-consuming. This is illustrated in Ex. 1.2. In the third line of the derivation, $\texttt{permute}([1|\texttt{Z}''], \texttt{X}')$ is selected, although there is no instance of $\texttt{delete}([],\texttt{U}',\texttt{Z}'')$ in the model of the program. This problem has been described by saying that `delete` makes a *speculative output binding* [20, 24].

Theorem 5.1 Let P be a well and nicely moded program and p be a predicate in P. Suppose all predicates q with $p \sqsupset q$ are atom-terminating, and all clauses defining predicates $q \in [p]_\approx$ are ICD-acceptable. Then p, and hence every predicate in $[p]_\approx$, is atom-terminating.

PROOF. Suppose the set of clauses defining the predicates $q \in [p]_\approx$ is ICD-acceptable with respect to the moded level mapping $|.|$. For an atom a using a predicate in $[p]_\approx$, we define $||a|| = sup(\{|a\theta| \mid a\theta \text{ is ground}\})$, if the set $\{|a\theta| \mid a\theta \text{ is ground}\}$ is bounded. Otherwise $||a||$ is undefined. Observe that

$$\text{if } ||a|| \text{ is defined for an atom } a, \text{ then } ||a\theta|| \leq ||a|| \text{ for all } \theta. \qquad (*)$$

To measure the size of a query, we use the multiset containing the level of each atom whose predicate is in $[p]_\approx$. The multiset is formalised as a function $Size$, which takes as arguments a query and a natural number.

$$Size(Q)(n) = \#\{q(\mathbf{u},\mathbf{v}) \mid q(\mathbf{u},\mathbf{v}) \text{ is an atom in } Q, q \approx p \text{ and } ||q(\mathbf{u},\mathbf{v})|| = n\}$$

Note that if a query contains several identical atoms, each occurrence must be counted. We define $Size(Q) < Size(R)$ if and only if there is a number l such that $Size(Q)(l) < Size(R)(l)$ and $Size(Q)(l') = Size(R)(l')$ for all $l' > l$. Intuitively, a decrease with respect to $<$ is obtained when an atom in a query is replaced with a finite number of smaller atoms. By König's Lemma [14], all descending chains with respect to $<$ are finite.

Let $Q_0 = p(\mathbf{s},\mathbf{t})$ be a well and nicely moded query. Then $\mathbf{s}$ is ground and thus $||Q_0||$ is defined. Let $\xi = Q_0; Q_1; Q_2 \ldots$ be an input-consuming derivation of $P \cup \{Q_0\}$.

Since all predicates q with $p \sqsupset q$ are atom-terminating, it follows by Thm. 4.2 that there cannot be an infinite suffix of ξ without any steps where an atom $q(\mathbf{u},\mathbf{v})$ such that $q \approx p$ is resolved. We show that for all $i \geq 0$, if the selected atom in $Q_i; Q_{i+1}$ is $q(\mathbf{u},\mathbf{v})$ and $q \approx p$, then $Size(Q_{i+1}) < Size(Q_i)$, and otherwise $Size(Q_{i+1}) \leq Size(Q_i)$. This implies that ξ is finite, and, as the choice of the initial query $Q_0 = p(\mathbf{s},\mathbf{t})$ was arbitrary, p is atom-terminating.

Consider $i \geq 0$ and let $C = q(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1,\mathbf{v}_1), \ldots, q_m(\mathbf{u}_m,\mathbf{v}_m)$ be the clause, $q(\mathbf{u},\mathbf{v})$ the selected atom and θ the MGU used in $Q_i; Q_{i+1}$.

If $p \sqsupset q$, then $p \sqsupset q_j$ for all $j \in \{1,\ldots,m\}$ and hence by $(*)$ it follows that $Size(Q_{i+1}) \leq Size(Q_i)$.

Now consider $q \approx p$. Since C is a ICD-acceptable clause, $||q(\mathbf{v}_0,\mathbf{u}_{m+1})\theta|| > ||q_j(\mathbf{u}_j,\mathbf{v}_j)\theta||$ for all j with $q_j \approx p$. This together with $(*)$ implies $Size(Q_{i+1}) < Size(Q_i)$. $\square$

Obviously the above theorem applies in particular if there exists no q such that $p \sqsupset q$, in which case trivially all predicates q with $p \sqsupset q$ are atom-terminating.

Example 5.1 We now give a few examples. We denote the *term size* of a term t, that is the number of function and constant symbols that occur in t, as $TSize(t)$.

The clauses defining $\mathtt{append}(I, I, O)$ (Ex. 4.1) are ICD-acceptable, where $|\mathtt{append}(s_1, s_2, t)| = TSize(s_1)$. Thus $\mathtt{append}(I, I, O)$ is atom-terminating. The same holds for $\mathtt{append}(O, O, I)$, defining $|\mathtt{append}(t_1, t_2, s)| = TSize(s)$.

The clauses defining $\mathtt{delete}(I, O, O)$ (Ex. 1.2) are ICD-acceptable, where $|\mathtt{delete}(s, t_1, t_2)| = TSize(s)$. Thus $\mathtt{delete}(I, O, O)$ is atom-terminating. The same holds for $\mathtt{delete}(O, I, I)$, defining $|\mathtt{delete}(t, s_1, s_2)| = TSize(s_2)$.

In a similar way, we can show that $\mathtt{permute}(O, I)$ is atom-terminating.[3] However, $\mathtt{permute}(I, O)$ is not atom-terminating.

The book on the Gödel language [15, page 81] shows a program that contains a clause, which in Prolog would be written as

```
slowsort(X,Y) :-
  permute(X,Y),
  sorted(Y).
```

The meaning and the modes of the predicates should be obvious from their names, and there are delay declarations to ensure that derivations are input-consuming. The predicate `slowsort` is *not* atom-terminating. However it could be made atom-terminating by replacing `permute(X,Y)` with `permute(Y,X)`, so that `permute` is used in the mode in which it is atom-terminating.

Note that according to the Gödel specification, no guarantees are given about the selection rule that go beyond ensuring that derivations for the above program are input-consuming. Hence the program is not guaranteed to terminate even for a "well-behaved" query such as `slowsort([1,2],Y)`. Even though Hill and Lloyd do not claim that the program terminates, one would still expect it to do so. However, we can modify the program as stated, and guarantee that the modified program terminates using the method of this paper.

Figure 1 shows a fragment from a program for the n-queens problem. The mode is $\{\mathtt{nqueens}(I, O)$, $\mathtt{sequence}(I, O)$, $\mathtt{permute}(I, O)$, $\mathtt{safe}(I)$, $\mathtt{is}(O, I)$, $\mathtt{safe\_aux}(I, I, I)$, $\mathtt{no\_diag}(I, I, I)$, $\mathtt{=\backslash=}(I, I)\}$. Again using as level mapping the term size of one of the arguments, one can see that the clauses defining $\{\mathtt{no\_diag}, \mathtt{safe\_aux}, \mathtt{safe}\}$ are ICD-acceptable and thus these predicates are atom-terminating. Note that for efficiency reasons, this program relies on input-consuming derivations where atoms using `safe` are selected as early as possible [23].

As a more complex example, consider the following program, whose mode is $\{\mathtt{plus\_one}(I), \mathtt{minus\_two}(I), \mathtt{minus\_one}(I)\}$.

[3] Here we assume that the program is made well and nicely moded by interchanging the body atoms of the second clause.

```
nqueens(N,Sol) :-
  sequence(N,Seq),
  permute(Seq,Sol),
  safe(Sol).

safe([]).
safe([N|Ns]) :-
  safe_aux(Ns,1,N),
  safe(Ns).

safe_aux([],_,_).
safe_aux([M|Ms],Dist,N) :-
  no_diag(N,M,Dist),
  Dist2 is Dist+1,
  safe_aux(Ms,Dist2,N).

no_diag(N,M,Dist) :-
  Dist =\= N-M,
  Dist =\= M-N.
```

Figure 1: A program for n-queens

```
plus_one(X) :- minus_two(succ(X)).

minus_two(succ(X)) :- minus_one(X).
minus_two(0).

minus_one(succ(X)) :- plus_one(X).
minus_one(0).
```

We define

$$\begin{aligned}
|\texttt{plus_one}(s)| &= 3 * TSize(s) + 4 \\
|\texttt{minus_two}(s)| &= 3 * TSize(s) \\
|\texttt{minus_one}(s)| &= 3 * TSize(s) + 2
\end{aligned}$$

Then the program is ICD-acceptable and therefore all predicates are atom-terminating.

We see that whenever in some argument position of a clause head, there is a compound term of some recursive data structure, such as $[\texttt{X}|\texttt{Xs}]$, and all recursive calls in the body of the clause have a strict subterm of that term, such as `Xs`, in the same position — then the clause is ICD-acceptable using as level mapping the term size of that argument position. Since this situation occurs very often, it can be expected that an average program contains many atom-terminating predicates. However, it is unlikely that in any real program, *all* predicates are atom-terminating.

The last example shows that more complex scenarios than the one described above are possible, but we doubt that they would often occur in practice. Therefore level mappings such as the one used in the example will rarely be needed.

Consider again Def. 5.3. Given a clause $h \leftarrow a_1, \dots, a_n$ and an atom $a_i \approx h$, we require $|h\theta| > |a_i\theta|$ for all grounding substitutions θ, rather than only for θ such that $(a_1, \dots, a_{i-1})\theta$ is in a certain model of the program. This is of course a serious restriction. In Ex. 1.2, assuming mode $\texttt{permute}(I, O)$, there can be no moded level mapping such that $|\texttt{permute}(\texttt{Y}, [\texttt{U}|\texttt{X}])\theta| > |\texttt{permute}(\texttt{Z}, \texttt{X})\theta|$

for all θ. That however is not surprising since $\texttt{permute}(I, O)$ is not atom-terminating.

Similarly, we can show that there cannot be a moded level mapping such that the usual recursive clause for `quicksort`, in the usual mode, is ICD-acceptable, even though we conjecture that `quicksort` is atom-terminating. This shows a limitation of our method. The author is currently working on ways of overcoming this limitation, but the fact remains that many predicates are not atom-terminating.

6 Applying the Method

The requirement of input-consuming derivations merely reflects the very meaning of *input*: an atom must only consume its own input, not produce it. Thus if one accepts that (appropriately chosen) modes are useful for verification and reflect the programmer's intentions, then one should also accept this requirement and regard any violation of it as pathological. This does not exclude multiple modes, that is, the same program being used in a different mode at each run.

The requirement of input-consuming derivations is trivially met for LD-derivations of a well moded query and program,[4] since the leftmost atom in a well moded query is ground in its input positions. It can also be ensured by using delay declarations as in Gödel [15] that require the input arguments of an atom to be ground before this atom can be selected. Moreover, it might be ensured using *guards* as in GHC [26]. Finally, it can be ensured using delay declarations that check for partial instantiation of the input arguments, such as the `block` declarations of SICStus. Note that under certain conditions, delay declarations can ensure input-consuming derivations with respect to several, alternative modes [21, Chapter 7] [23].

Consequently, this paper is mainly aimed at logic programs with delay declarations, but unlike previous work [2, 17, 18, 19, 20, 23, 24], abstracts from the details of particular delay constructs. We only assume what we see as the basic purpose of delay declarations: ensuring that derivations are input-consuming.

As we have said in the introduction, the class of predicates for which all input-consuming derivations terminate is quite limited. In an average program, some predicates are atom-terminating but some are not. In general, one has to make stronger assumptions about the selection rule. We sketch three ways in which the method presented here might be incorporated into a more comprehensive method for proving termination. This boils down to the question: how do we deal with predicates that are not atom-terminating?

The first way has actually been developed already [23]. We have previously considered atom-terminating predicates in a more concrete setting than here

[4] In particular, this means that it is met in Mercury [25].

and called them *robust* predicates. The default left-to-right selection rule of most Prolog implementations is assumed. It is exploited that the textual position of atoms using robust predicates in clause bodies is irrelevant for termination. The other atoms must be placed such that the atoms producing their input occur earlier.

Secondly, we could build on a technique developed by Martin and King [19]. They consider coroutining derivations, but impose a bound on the depth of each sub-derivation by introducing auxiliary predicates with an additional argument that serves as depth counter. Applying the results of this paper, we only have to impose this depth bound for the predicates that are not atom-terminating. For the atom-terminating predicates, we can save the overheads involved in this technique.

Thirdly, we could use delay declarations as they are provided for example in Gödel [15]. For the atom-terminating predicates, it is sufficient to check for partial instantiation of the input positions using a `DELAY`...`UNTIL NONVAR`... declaration. For the other predicates, it must be ensured that the input positions are ground using a `DELAY`...`UNTIL GROUND`... declaration. Note that according to its specification, Gödel does not guarantee a (default) left-to-right selection rule, and therefore delay declarations are crucial for termination. Note also that a groundness test is usually more expensive than a test for partial instantiation. To the best of our knowledge, there has never been a systematic treatment of the question when `GROUND` declarations are needed, and when `NONVAR` declarations are sufficient.

7 Discussion

We have identified the class of predicates for which all input-consuming derivations are finite. An input-consuming derivation is a derivation where in each step, the input arguments of the selected atom are not instantiated. Predicates can be shown to be in that class using the notions of *level mapping* and *acceptable clause* in a very similar way to methods for LD-derivations [7, 10, 12].

Most previous approaches, including approaches for programs with delay declarations, can only show termination making stronger assumptions about the selection rule [17, 18, 19]. We have argued in the previous section that knowing the predicates that terminate under our weaker assumptions is useful even for programs where not *all* predicates have this property.

This paper builds on our own previous work [23], but attempts to formulate the results more abstractly, without getting involved in the details of particular delay constructs. For example, we previously imposed a restriction that all clause heads in a program must be input-linear, which is necessary so that `block` declarations can ensure input-consuming derivations. In this paper, we do not impose this restriction. Hence if input-consuming derivations can be ensured without imposing this restriction, say by using guards as in GHC [26], then the results of this paper could be applied to show termination.

We have claimed that most other approaches to termination rely on the idea that the size of an atom can be pinned down when the atom is selected. Technically, this usually means that the atom is *bounded* with respect to some level mapping [4, 6, 12, 19]. There are exceptions though [8, 10], where termination can be shown for the query, say, `append([X],[],Zs)` using as level mapping the term size of the first argument, even though the term size of `[X]` is not bounded. However, the method only works for LD-derivations and relies on the fact that any future instantiation of `X` cannot affect the derivation for `append([X],[],Zs)`. Therefore it is effectively possible to pin down the size of `append([X],[],Zs)`.

In contrast, we show that under certain conditions, it is enough to rely on a *relative* decrease in the size of the selected atom, even though this size cannot be pinned down. This is crucial to show termination of derivations with coroutining. More precisely, we exploit that an atom in a query cannot proceed indefinitely unless it is repeatedly fed by some other atom occurring earlier in the query. This implies that every derivation for the query is finite.

Bezem [6] has identified the class of strongly terminating programs, which are programs that terminate under *any* selection rule. While it is shown that every total recursive function can be computed by a strongly terminating program, this does not change the fact that few existing programs are strongly terminating. Transformations are proposed for three example programs to make them strongly terminating, but the transformations are complicated and ad-hoc.

On the whole, there seems to be a strong reluctance to give up the idea that the size of an atom must be pinned down when the atom is selected. This is true even for Bezem [6]. It is also true for Marchiori and Teusink [18], who assume a *local selection rule*, that is a rule under which only most recently introduced atoms can be resolved in each step. Martin and King [19] achieve a similar effect by bounding the depth of the computation introducing auxiliary predicates. It is more difficult to assess Lüttringhaus-Kappel [17] since his contribution is mainly to *generate* delay declarations automatically rather than *prove* termination.[5] However in some cases, the delay declarations that are generated require an argument of an atom to be a rigid list before that atom can be selected, which is similar to [18, 19]. Such uses of delay declarations go far beyond ensuring that derivations are input-consuming.

Of course, none of the above approaches [6, 17, 18, 19] can formally show termination under the weak assumptions we make here, even for derivations as trivial as the one in Ex. 1.1. Apt and Luitjes [2] give conditions for the termination of `append`, but those are ad-hoc and do not address the general problem. Naish [20] gives heuristics to ensure termination, but no formal results.

[5] For the reader familiar with that work, it is not said how programs are shown to be *safe*.

We have assumed that queries are well and nicely moded, which means that the atoms in the query are ordered[6] so that there is a left-to-right data-flow. As a topic for future work, we envisage to prove termination of programs where these conditions are relaxed, such as programs using *layered modes* [13]. We believe that the crucial idea will be the same as in this paper, namely that one must rely on a *relative* decrease in size of the selected atom in each derivation step, rather than an absolute one. Therefore this paper should provide a good basis for this extension.

Acknowledgements

The author would like to thank Florence Benoy for proofreading this paper, and Sandro Etalle and Pat Hill for some helpful comments. This work was funded by EPSRC Grant No. GR/K79635.

References

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

[2] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In V. S. Alagar and M. Nivat, editors, *Proceedings of AMAST'95*, LNCS, Berlin, 1995. Springer-Verlag. Invited Lecture.

[3] K. R. Apt and D. Pedreschi. Studies in pure Prolog: Termination. In J. W. Lloyd, editor, *Proceedings of the Symposium in Computational Logic*, LNCS, pages 150–176. Springer-Verlag, 1990.

[4] 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.

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

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

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

[8] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In *Proceedings of FGCS*, pages 481–488. ICOT Tokyo, 1992.

[9] S. Decorte and D. De Schreye. Automatic inference of norms: A missing link in automatic termination analysis. In *Proceedings of the 10th International Logic Programming Symposium*, pages 420–436. MIT Press, 1993.

[10] S. Decorte and D. De Schreye. Termination analysis: Some practical properties of the norm and level mapping space. In J. Jaffar, editor, *Proceedings of the 15th JICSLP*, pages 235–249. MIT Press, 1998.

[6] Or more generally: *can be* ordered (see [21, Sect. 5.3] or the discussion after Ex. 3.1).

[11] P. Dembinski and J. Małuszyński. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the 2nd International Logic Programming Symposium*, pages 29–38. MIT Press, 1985.

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

[13] S. Etalle and M. Gabbrielli. Layered modes. *Journal of Logic Programming*, 39:225–244, 1999.

[14] M. Fitting. *First-order Logic and Automated Theorem Proving*. Springer-Verlag, 1996.

[15] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[16] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[17] 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.

[18] E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In J. W. Lloyd, editor, *Proceedings of the 12th International Logic Programming Symposium*, pages 447–461. MIT Press, 1995.

[19] J. C. Martin and A. M. King. Generating efficient, terminating logic programs. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT'97*, LNCS, pages 273–284. Springer-Verlag, 1997.

[20] L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, University of Melbourne, 1992.

[21] J.-G. Smaus. *Modes and Types in Logic Programming*. PhD thesis, University of Kent at Canterbury, October 1999. Draft available from `www.cs.ukc.ac.uk/people/staff/jgs5/thesis.ps`.

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

[23] J.-G. Smaus, P. M. Hill, and A. M. King. Termination of logic programs with `block` declarations running in several modes. In C. Palamidessi, editor, *Proceedings of PLILP/ALP*, LNCS. Springer-Verlag, 1998.

[24] J.-G. Smaus, P. M. Hill, and A. M. King. Preventing instantiation errors and loops for logic programs with multiple modes using `block` declarations. In P. Flener, editor, *Proceedings of LOPSTR'98*, LNCS. Springer-Verlag, 1999.

[25] 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), 1996.

[26] K. Ueda. Guarded Horn clauses. In E. Wada, editor, *Proceedings of the 4th Japanese Conference on Logic Programming*, LNCS, pages 168–179. Springer-Verlag, 1986.