Well-Terminating, Input-Driven Logic Programs

Jan–Georg Smaus

University of Kent at Canterbury, Canterbury, CT2 7NF, United Kingdom,
telephone xx44/1227/827553, fax 762811, `j.g.smaus@ukc.ac.uk`

Abstract. We identify a class of predicates for which termination does not depend on left-to-right execution. All that is required is that derivations are *input-driven*, that is, in each derivation step, the input arguments of the selected atom do not become instantiated. The method of showing that a predicate is in that class is based on level mappings, closely following the traditional approach for LD derivations. Many predicates terminate under such weak assumptions. Knowing these predicates can be a very useful part of a more comprehensive method of showing termination, which would have to make more specific assumptions about the selection rule.

Keywords: Logic programming, termination, modes, selection rules.

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, 8–11, 13]. 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 their size when eventually they are selected, and that 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 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, 16–21]. The problem is that, depending on what kind of delay declarations and selection rule are used, it is 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, [17, 18] and to a limited extent [16] are based on the idea described above, whereas [19–21] avoid any explicit mention of "size" and instead try to reduce the problem to showing termination for LD derivations.

[1] The technical meaning of "pinning down the size" differs among different methods.

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 with the clause head does not bind the input arguments of the atom. We call derivations which meet this requirement *input-driven*.

This paper is about identifying predicates for which all input-driven 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 [17], delay declarations that test arguments for groundness or rigidness [16, 18], or the default left-to-right selection rule of most Prolog implementations [19–21]. 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-driven. We closely follow [13].

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 and the following input-driven derivation, where the selected atom is underlined:

$$
\begin{array}{l}
\underline{\mathtt{append}([1],[],\mathtt{As})}, \mathtt{append}(\mathtt{As},[],\mathtt{Bs}) \rightarrow \\
\underline{\mathtt{append}([],[],\mathtt{As}')}, \overline{\mathtt{append}([1|\mathtt{As}'],[],\mathtt{Bs})} \rightarrow \\
\underline{\mathtt{append}([],[],\mathtt{As}')}, \overline{\mathtt{append}(\mathtt{As}',[],\mathtt{Bs}')} \rightarrow \\
\underline{\mathtt{append}([],[],\mathtt{Bs}')} \rightarrow \Box.
\end{array}
$$

When $\mathtt{append}([1|\mathtt{As}'],[],\mathtt{Bs})$ 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 derivation associated with $\mathtt{append}([1|\mathtt{As}'],[],\mathtt{Bs})$ without knowing about other atoms which might instantiate $\mathtt{As}'$. However, the derivation could be infinite only if the derivation associated with $\mathtt{append}([],[],\mathtt{As}')$ was infinite. Our method is based on such a dependency between the atoms of a query.

Not surprisingly, the class of programs for which all input-driven derivations are finite is quite limited, although it is obviously larger than the class of strongly terminating programs. Realistically, a comprehensive method for proving termination would have to make stronger assumptions. However, within the framework of such a method, it can be useful to know for which predicates termination can already be ensured only assuming input-driven derivations. This is demonstrated in [21], but apart from that, we believe that it has not been recognised previously.

Example 1.2. Consider the following program which permutes a list. Assume that in both predicates, the first position is the only input position.

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

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

Then we have the following infinite input-driven derivation:

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

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. Section 6 discusses the results and the related work.

2 Preliminaries

Our notation follows [1, 13]. For the examples we use Prolog syntax. We recall some important notions. The set of variables in a syntactic object o is denoted as $vars(o)$. The domain of a substitution θ is denoted as $dom(\theta)$. The restriction of a substitution θ to the variables occurring in a syntactical object o is denoted as $\theta|o$. A syntactic object is **linear** if every variable occurs in it at most once.

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.

A **query** is a finite sequence of atoms. 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 the most general unifier 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-driven** if $dom(\sigma) \cap vars(\mathbf{s}\theta) = \emptyset$.

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 successive 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** $R \cup \{Q_0\theta_0\}$. We often 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-driven** derivation is a derivation consisting of input-driven derivation steps.

If $Q, a, R; (Q, B, R)\theta$ 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 $b \in Q, R$. 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. If, for a derivation $\ldots Q; \ldots; Q'; Q'' \ldots$, the selected atom in $Q'; Q''$ is a descendant of an atom a in Q, then $Q'; Q''$ is an **a-step**.

3 Modes

In this section we introduce the notions of well moded and nicely moded programs. Well-modedness has been used before to show termination of LD derivations [13]. In the context of arbitrary input-driven derivations, it is also crucial to require that programs are nicely moded.

Well-modedness has been introduced in [12] and widely used for verification since. In Mercury it is even mandatory that programs are well moded, which is one of the reasons for its remarkable performance [22].

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 $i \in \{1, \ldots, n\}$

$$vars(\mathbf{s}_i) \subseteq \bigcup_{j<i} vars(\mathbf{t}_j) \tag{1}$$

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

Another common concept for verification 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 $i \in \{1, \ldots, n\}$

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

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 P is **nicely moded** if all of its clauses are nicely moded.

Note that other authors have denoted the clause head of C as $p(\mathbf{s}_0, \mathbf{t}_{n+1})$ or $p(\mathbf{s}_0, \mathbf{t}_0)$, which allows for a more compact definition [13][2], furthermore suggesting that there is an analogy between (2) and (3) above [2]. We have chosen not

[2] We refer to the definition of *simply moded* here, but this is very similar to *nicely moded*.

to do this for two reasons. First, our notation is consistent with Def. 3.1. Secondly, the analogy is misleading. It would be more appropriate to see an analogy between (3) and the requirement that $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a linear vector of terms than between (3) and (2).

Example 3.1. The program in Ex. 1.2 is well and nicely moded. It is neither well moded nor nicely moded in reverse mode, 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 having multiple (renamed) versions of a predicate. This is why it is often assumed that each predicate has a fixed mode [2, 13, 19, 22]. However, this usually implies actual code duplication and is therefore a real loss of generality.

In this paper, assuming a fixed mode for each predicate is *not* a loss of generality, but merely a notational convenience. We consider arbitrary input-driven derivations. The textual position of an atom within a query is irrelevant for its selection. Any result that holds for a well moded program also holds for a 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. 3.1 is well moded and nicely moded in *both* modes. Whenever one considers specific selection rules where the textual position is relevant, one has to treat multiple modes explicitly [21].

The following lemma states a persistence property of well-modedness [2, Lemma 16]. It has been shown previously for LD derivations [5].

Lemma 3.1. Every resolvent of a well moded query Q and a well moded clause C, where $vars(C) \cap vars(Q) = \emptyset$, is well moded.

For nicely-modedness, there is a similar persistence property. It has been shown previously for LD resolvents [5] and arbitrary resolvents [2]. However in the latter case, it was required that the clause head is input-linear. For input-driven derivations, this is not necessary. 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.

Lemma 3.2. Every resolvent of a nicely moded query Q and a nicely moded clause C, where the derivation step is input-driven and $vars(C) \cap vars(Q) = \emptyset$, is nicely moded.

Proof. Let $C = h \leftarrow B$. We want to use [2, Lemma 11]. Therefore we must show that we can assume without loss of generality that h is input-linear. Let $C' = h' \leftarrow Eq, B$ be the clause obtained from C by repeatedly applying the following transformation: If the input arguments of the clause head contain two occurrences of a variable x, replace one occurrence with a fresh variable y and add the equation $x = y$ at the beginning of the clause body.

Then h' is input-linear, and C' is nicely moded, where the predicate $=$ in Eq is used in mode (I, I). Furthermore, if $Q = Q_1, a, Q_2$ and

$$\langle Q, \emptyset \rangle; \langle Q_1, B, Q_2,\ \theta \rangle$$

is an input-driven derivation step using C, then there is an input-driven derivation

$$\langle Q, \emptyset\rangle; \langle Q_1,Eq,B,Q_2,\ \theta'\rangle; \ldots; \langle Q_1,B,Q_2,\ \theta\rangle$$

using C' and the clause "$z = z$." (which is conceptually the definition of $=$).

By [2, Lemma 11], $(Q_1, Eq, B, Q_2)\theta'$ is nicely moded. Furthermore, since $\langle Q_1,Eq,B,Q_2,\ \theta'\rangle; \ldots; \langle Q_1,B,Q_2,\ \theta\rangle$ is input-driven, it follows that $Eq\theta'$ is a sequence of atoms of the form $s = s$. Therefore $\theta' = \theta$ and $(Q_1, B, Q_2)\theta$ is nicely moded. □

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

Lemma 3.3. 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-driven derivation step. Then $dom(\sigma) \cap vars(Q_1) = \emptyset$.

Proof. Since the derivation step is input-driven, it follows that $dom(\sigma)|Q \subseteq vars(\mathbf{t})$. Thus since Q is nicely moded, $dom(\sigma) \cap vars(Q_1) = \emptyset$. □

4 Controlled Coroutining

In this section we define *well-terminating* predicates, that is predicates for which all one-atom queries have finite derivations. As in [13], we then show that termination for one-atom queries implies termination for arbitrary queries.

For LD derivations, this is almost obvious and only requires that programs and queries are well moded [13, Lemma 4.2]. Given a 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 much less obvious.

Example 4.1. Consider the usual append program

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

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

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

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

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

This well-known termination problem of programs with coroutining has been identified as *circular modes* [19].

To avoid the problem, we require programs and queries to be nicely moded.

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

The following lemma says that a well-terminating atom cannot proceed indefinitely unless it is repeatedly fed by some other atom.

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 a well-terminating atom. An input-driven 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, we call an a-step for some $a \in F$ an F**-step**, and likewise for H. By Lemma 3.3, any H-step does not instantiate any descendant of b. Thus the H-steps can be disregarded, and without loss of generality, we assume H is empty. Let

$$\xi = \langle F,b,\ \emptyset\rangle; \ldots; \langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle \ldots$$

be an input-driven derivation such that $\langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle \ldots$ contains no F-steps (that is, ξ contains only finitely many F-steps). Since by Lemma 3.3, no b-step can instantiate any descendant of F, there exists an input-driven derivation

$$\xi_2 = \langle F,b,\ \emptyset\rangle; \ldots; \langle R, \rho\rangle; \ldots; \langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle \ldots$$

such that $\langle F,b,\ \emptyset\rangle; \ldots; \langle R, \rho\rangle$ contains only F-steps and $\langle R, \rho\rangle; \ldots; \langle Q_0, \theta_0\rangle$ contains only b-steps (that is, the F-steps are moved forward using the Switching Lemma [15]). Since $R = R', b$ for some R', there exists an input-driven derivation

$$\xi_3 = \langle b, \rho\rangle; \ldots; \langle I_0, \theta_0\rangle; \langle I_1, \theta_1\rangle \ldots$$

obtained from $\langle R, \rho\rangle; \ldots; \langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle \ldots$ by removing the prefix R' in each query.

Let $\mathbf{t}_1, \ldots, \mathbf{t}_m$ be the vector of output arguments of $R'\rho$ and σ a substitution such that $(\mathbf{t}_1, \ldots, \mathbf{t}_m)\sigma$ is ground. Then by Def. 3.1, $b\rho\sigma$ is a well moded query.

By Lemma 3.3, no b-step instantiates $\mathbf{t}_1, \ldots, \mathbf{t}_m$. Therefore from ξ_3 we can construct an input-driven derivation

$$\xi_4 = \langle b, \rho\sigma\rangle; \ldots; \langle I_0, \theta_0\sigma\rangle; \langle I_1, \theta_1\sigma\rangle \ldots$$

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

The following lemma is a consequence and states that well-terminating atoms on their own cannot produce an infinite derivation.

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

Proof. Let $Q = F, b, H$ where b is a well-terminating atom, and ξ an infinite derivation for Q. We show that ξ can have infinitely many b-steps only if ξ has infinitely many steps where an atom is resolved that is not well-terminating.

The proof is by induction on the length of F. If F is empty, the result follows from Lemma 4.1. Now suppose F contains at least one atom. By Lemma 4.1, ξ can have infinitely many b-steps only if for some $a \in F$, ξ has infinitely many a-steps. If a is not well-terminating, the result follows immediately. If a is well-terminating, let $F = F_1, a, F_2$. Since F_1 contains fewer atoms than F, the result follows from the inductive hypothesis. □

Lemma 4.2 provides us with the formal justification for restricting our attention to one-atom queries. However, it requires that programs and queries are nicely moded. This is not necessary for LD derivations [13].

We now define *well-terminating* programs. The definition differs from the corresponding one in [13] in that they consider only LD derivations.

Definition 4.2 (well-terminating program). Let P be a well and nicely moded program and Q a well and nicely moded query. P is **well-terminating** if all input-driven derivations of $P \cup \{Q\}$ are finite.

The following is an obvious corollary of Lemma 4.2.

Corollary 4.3. A program P is well–terminating if and only if all its predicates are well-terminating.

5 Showing Weak Termination

All of the mentioned approaches to termination [1, 3, 8–11, 13] more or less explicitly rely on measuring the size of the *input* in a query. We agree with Etalle et al. [13] that it is reasonable to make this dependency explicit. This gives rise to the concept of *moded level mapping* [13], which is an instance of *level mapping* first introduced in [6, 7]. $\mathbf{B}_P$ 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 $|.| : \mathbf{B}_P \to \mathbb{N}$,
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, adopted from [1], is useful for proving termination for a whole program incrementally, by proving it for one predicate at a time.

Definition 5.2 (depends on). Let p, q be predicates in a program P. We say that 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 well-terminating.

Definition 5.3 (well-acceptable). Let P be a program and $|.|$ a moded level mapping. A clause $C = h \leftarrow B$ is **well-acceptable (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 $|h\theta| > |a\theta|$.

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

Let us compare this concept to some similar concepts in the literature: *recurrent* [6], *well-acceptable* [13] and *acceptable* [4, 11] programs.

Like [11, 13] and unlike [4, 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 [6] and unlike [4, 11, 13], 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 $h \leftarrow a_1, \dots, a_n$, $|h\theta| > |a_i\theta|$ is only required if $M \models (a_1, \dots, a_{i-1})\theta$. The reason is that for LD derivations, $a_1, \dots, a_{i-1}$ must be completely resolved before a_i is selected. By the correctness of LD resolution [15] and well-modedness [5], the accumulated answer substitution θ, just before a_i is selected, is such that $(a_1, \dots, a_{i-1})\theta$ is ground and $M \models (a_1, \dots, a_{i-1})\theta$.

Such considerations count for little when derivations are merely required to be input-driven. This is illustrated in Ex. 1.2. In the third line of the derivation, `permute([1|Z''],X')` is selected, although there is no instance of `delete([],U',Z'')` in the model of the program. This problem has been described by saying that `delete` makes a *speculative output binding* [19]. Programs that do not make speculative output bindings are considered in [20].

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 well-terminating, and all clauses defining predicates $q \in [p]_\approx$ are well-acceptable. Then p, and hence every predicate in $[p]_\approx$, is well-terminating.

Proof. Suppose the set of clauses defining the predicates $q \in [p]_\approx$ is well-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. It is easy to see (König's Lemma [14]) that 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-driven derivation of $P \cup \{Q_0\}$.

Since all predicates q with $p \sqsupset q$ are well-terminating, it follows by Lemma 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 well-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 most general unifier 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 well-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 $Size(Q_{i+1}) < Size(Q_i)$. □

Example 5.1. We now give a few examples of well-terminating predicates. 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 $\texttt{append}(I,I,O)$ (Ex. 4.1) are well-acceptable, where $|\texttt{append}(s_1,s_2,t)| = TSize(s_1)$. Thus $\texttt{append}(I,I,O)$ is well-terminating. The same holds for $\texttt{append}(O,O,I)$, defining $|\texttt{append}(t_1,t_2,s)| = TSize(s)$.

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

In a similar way, we can show that $\texttt{permute}(O,I)$ is well-terminating. However, $\texttt{permute}(I,O)$ is not well-terminating.

Figure 1 shows a fragment from a program for the n-queens problem. The mode is $\{\texttt{nqueens}(I,O)$, $\texttt{sequence}(I,O)$, $\texttt{safe}(I)$, $\texttt{permute}(O,I)$, $\texttt{<}(I,I)$, $\texttt{is}(O,I)$, $\texttt{safe\_aux}(I,I,I)$, $\texttt{no\_diag}(I,I,I)$, $\texttt{=\textbackslash=}(I,I)\}$. Again using as level mapping the term size of one of the arguments, one can see that the clauses defining $\{\texttt{no\_diag}, \texttt{safe\_aux}, \texttt{safe}\}$ are well-acceptable and thus these predicates are well-terminating. This information is useful since this program relies on non-LD derivations for its performance [21].

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

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.
```

Fig. 1. A program for n-queens

As a more complex example, consider the following program, whose mode is $\{\texttt{plus\_one}(I), \texttt{minus\_two}(I), \texttt{minus\_one}(I), \}$. This example uses the successor notation for natural numbers.

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

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

minus_one(s(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 well-acceptable and thus well-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 well-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 well-terminating predicates. However, it is unlikely that in any real program, *all* predicates are well-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 $h \leftarrow a_1, \ldots, a_n$ 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 $(a_1, \ldots, a_{i-1})\theta$ 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. It might be possible to relax Def. 5.3 to allow more programs, but the fact remains that many predicates are not well-terminating.

6 Discussion

We have identified the class of programs for which all input-driven derivations are finite. An input-driven 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 [8, 11, 13].

This paper closely follows [13]. There a statement is shown which is essentially the converse of Thm. 5.1. It says that if a predicate is well-terminating, then there is a level mapping such that the clauses defining the predicate are well-acceptable. It would be interesting to show a similar result for arbitrary input-driven derivations, but we believe that it must be difficult, since our definition of acceptability is much more restrictive.

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, 13, 18]. This is different in [9, 11], 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. More concretely, we use 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 terminates.

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.

This paper is more abstract than the literature on programs with delay declarations [2, 16–21]. We are not concerned with the details of particular delay constructs. Instead, we only assume what we see as the basic purpose of delay declarations: ensuring that derivations are input-driven. Note that depending on what kind of constructs are used, ensuring that derivations are input-driven is actually quite subtle [21]. Nevertheless, delay declarations are clearly a powerful instrument for this purpose.

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 [6], where no assumptions at all are made about the selection rule. It is also true for [17], where a *local selection rule* is assumed, that is a rule under which only most recently introduced atoms can be resolved in each step. In [18], a similar effect is achieved by bounding the depth of the computation introducing auxiliary predicates. It is more difficult to assess [16] since the contribution there is mainly to *generate* delay declarations automatically rather than *prove* termination.[3] 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 [17, 18]. Such uses of delay declarations go far beyond ensuring that derivations are input-driven.

We do not claim to present a comprehensive method for showing termination. In an average program, some predicates are well-terminating but some are not. In general, one has to make stronger assumptions about the selection rule. Nevertheless, it is useful to know which predicates are well-terminating, essentially because it means that one has to make the stronger assumptions only for the predicates that are not well-terminating. For example, requiring ground or rigid arguments [16, 18] could be limited to atoms whose predicates are not well-terminating.

In [21], well-terminating predicates are considered in a more concrete setting than here and are called *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.

Acknowledgements

The author would like to thank Florence Benoy for proofreading this paper. 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 *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 Unversity Press, 1994.
5. K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 16(3):687–726, 1994.
6. M. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1 & 2):79–97, 1993.
7. 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. MIT Press, 1989.

[3] For the reader familiar with [16], it is not said how it is shown that programs are *safe*.

8. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.
9. 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.
10. S. Decorte and D. De Schreye. Automatic inference of norms: a missing link in automatic termination analysis. In D. Miller, editor, *Proceedings of the 10th International Logic Programming Symposium*, pages 420–436. MIT Press, 1993.
11. 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 Joint International Conference and Symposium on Logic Programming*, pages 235–249. MIT Press, 1998.
12. 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.
13. S. Etalle, A. Bossi, and N. Cocco. Well-terminating programs. *Journal of Logic Programming*, 1998. Accepted for publication.
14. M. Fitting. *First-order Logic and Automated Theorem Proving*. Springer-Verlag, 1996.
15. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
16. 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.
17. 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.
18. 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.
19. L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, University of Melbourne, 1992.
20. J.-G. Smaus, P. M. Hill, and A. M. King. Preventing instantiation errors and loops for logic programs with several modes using `block` declarations. In Pierre Flener, editor, *Pre-proceedings of the 8th International Workshop on Logic Program Synthesis and Transformation*, number UMCS-98-6-1, pages 72–29. University of Manchester, 1998. Extended abstract.
21. J.-G. Smaus, P. M. Hill, and A. M. King. Termination of logic programs with `block` declarations running in several modes. In C. Palamadessi, editor, *Proceedings of the 10th Symposium on Programming Language Implementations and Logic Programming*, LNCS. Springer-Verlag, 1998.
22. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, November 1996.