TLP **2** (2): 125–154, March 2002. © 2002 Cambridge University Press
DOI: 10.1017/S1471068401001326 Printed in the United Kingdom

Properties of Input-Consuming Derivations

ANNALISA BOSSI, SABINA ROSSI
Dipartimento di Informatica, Università di Venezia,
via Torino 155, 30172 Venezia, Italy
(*e-mail:* {bossi,srossi}@dsi.unive.it)

SANDRO ETALLE
Department of Computer Science, University of Maastricht,
P.O. Box 616, 6200 MD Maastricht, The Netherlands
(*e-mail:* etalle@cs.unimaas.nl)
and
CWI – Center for Mathematics and Computer Science,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

We study the properties of input-consuming derivations of moded logic programs. Input-consuming derivations can be used to model the behavior of logic programs using dynamic scheduling and employing constructs such as *delay declarations*. We consider the class of *nicely-moded* programs and queries. We show that for these programs a weak version of the well-known *switching lemma* holds also for input-consuming derivations. Furthermore, we show that, under suitable conditions, there exists an algebraic characterization of termination of input-consuming derivations.

KEYWORDS: dynamic scheduling, termination

1 Introduction

Most of the recent logic programming languages provide the possibility of employing *dynamic scheduling*, i.e., a runtime mechanism determining which atoms in a query are *selectable* and which ones are not. In fact, dynamic scheduling has proven to be useful in a number of applications; among other things, it allows one to model coroutining, as shown in Naish (1993) and Hill & Lloyd (1994), and parallel executions, as shown in Naish (1988).

Let us use the following simple examples to show how dynamic scheduling can be enforced by using *delay declarations* and how it can prevent nontermination and unnecessary computations. Consider the program APPEND

```
app([ ],Ys,Ys).
app([H|Xs],Ys,[H|Zs]) ← app(Xs,Ys,Zs).
```

together with the query

Q_1 := `app(Xs,[5,6],Ys), app([1,2],[3,4],Xs).`

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

In this query, if we select and resolve the leftmost atom, we could easily have to face one of the following two problems. First, the possibility of nontermination: This is the case if we repeatedly resolve the leftmost atom against the second clause. The second problem is that of inefficiency. If, for instance, in Q_1 we resolve the leftmost atom against the first clause, we obtain the query `app([1,2],[3,4],[ ])`. This will eventually fail, yielding to (unnecessary) backtracking. Notice that if one employs the rightmost selection rule, Q_1 would terminate with success and without backtracking. Basically, the problem when selecting `app(Xs,[5,6],Ys)`, is that we do not know which clause we should use for resolving it, and the only practical way for getting to know this is by waiting until the outermost functor of `Xs` is known: If it is the empty list `[ ]` we know that we should use the first clause, if it is the list-constructor symbol we know that we should use the second clause, if it is something else again, we know then that the query *fails*. Notice that the same problems arise for the query

$Q_2 :=$ `app([1,2],[3,4],Xs), app(Xs,[5,6],Ys).`

if the rightmost selection rule is considered.

This shows the usefulness of a mechanism for preventing the selection of those atoms which are not sufficiently instantiated. Such a mechanism is in fact offered by most modern languages: In GHC (Ueda, 1988) programs are augmented with *guards* in order to control the selection of atoms dynamically. Moded Flat GHC (Ueda & Morita, 1994) uses an extra condition on the input positions, which is extremely similar to the concept of input-consuming derivation step we refer to the sequel: The resolution of an atom with a definition must not instantiate the input arguments of the resolved atom. On the other hand, Gödel (Hill & Lloyd, 1994) and ECLiPSe (Wallace *et al.*, 1997) use *delay declarations*, and SICStus Prolog (1997) employs `block` declarations (which are a special kind of delay declarations). Both *delay* and `block` declarations check the partial instantiation of some arguments of calls. For instance, the standard delay declaration for `APPEND` is

$d_1 :=$ `delay app(Ls, _, _) until nonvar(Ls).`

This declaration forbids the selection of an atom of the form $\mathtt{app}(s,t,u)$ unless s is a non-variable term, which is precisely what we need in order to run the queries Q_1 or Q_2 efficiently.

The adoption of dynamic scheduling has the disadvantage that various program properties that have been proven for logic and pure Prolog programs do not apply any longer.

The goal of our research is the study of termination properties. This is motivated by the fact that most of the literature on termination of logic programs (see De Schreye and Decorte (1994) for a survey on this subject) assumes the standard Prolog selection rule, i.e. the leftmost one. Notable exceptions are Bezem (1993) and Cavedon (1989), who provide results for all selection rules. There are only few authors who tackled the specific problem of verifying the termination of logic programs with dynamic scheduling. Namely, Apt & Luitjes (1995), Marchiori and Teusink (1999) and Smaus (1999b). We compare our results with those in Apt &

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Luitjex (1995), Marchiori & Teusink (1999) and Smaus (1999b) in the concluding section.

Another feature of logic programs which does not hold in presence of dynamic scheduling is the well-known *switching lemma*, which is, for instance, at the base of the result on the independence of the selection rule. In this paper we show that – under certain conditions – a weak form of the well-known switching lemma holds.

To recuperate at least part of the declarative reading of logic programming, we follow here the same approach to dynamic scheduling as Smaus (1999b), and we substitute the use of delay declarations by the restriction to *input-consuming* derivations. The definition of input-consuming derivation is done in two phases. First we give the program a *mode*, that is, we partition the positions of each atom into *input* and *output* positions. Then, in presence of modes, *input-consuming* derivation steps are precisely those in which the input arguments of the selected atom will not be instantiated by the unification with the clause's head. If in a query no atom is resolvable via an input-consuming derivation step and a failure does not arise then we have a *deadlock* situation[1].

For example, the standard mode for the program `APPEND` reported above, when used for concatenating two lists, is `app(In,In,Out)`. Notice that in this case the delay declaration d_1 serves precisely the purpose of guaranteeing that if an atom of the form $\mathtt{app}(s, t, X)$ (with X being a variable) is selectable and unifiable with a clause head, then the resulting derivation step is input-consuming.

It is also worth remarking that, as a large body of literature shows, the vast majority of 'usual' programs are actually moded and are, in a well-defined sense consistent wrt. to their modes (e.g. well-moded, nicely-moded, simply-moded, etc.); see, for example, Apt & Pellegrini (1994) and Apt & Marchiori (1994), or more simply, the tables of programs we report in section 7, or consider for instance the logic programming language Mercury (Somogyi *et al.*, 1996), which requires that its programs are moded (and well-moded).

Contributions of this paper

In this paper we study some properties of input-consuming derivations. We first show that, if we restrict ourselves to programs and queries which are nicely-moded, then a weak form of the well-known switching lemma holds. Furthermore, we study the termination properties of input-consuming derivations. For this we define the class of *input terminating* programs which characterizes programs whose input-consuming derivations starting in a nicely-moded query are finite. In order to prove that a program is input terminating, we use the concept of *quasi recurrent* program (similar to, but noticeably less restrictive than the concept of semi-recurrent program introduced in Apt & Pedreschi (1994)). We show that if P is nicely-moded and quasi recurrent then all its input-consuming derivations starting from a nicely-moded query terminate.

[1] As we discuss in section 3.2, this notion of deadlock differs, in some way, from the usual one, which is given in the case of programs employing delay declarations.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Furthermore, we demonstrate that under mild additional constraints (namely, simply-modedness and input-recurrency) the above condition is both sufficient and *necessary* for ensuring that all input-consuming derivations starting from a nicely-moded query terminate.

This approach generalizes the method described in Smaus (1999b) in two ways: first because we also provide conditions which are both necessary and sufficient; and secondly, because we do not require programs and queries to be well-moded; we only assume that they are nicely-moded. This is actually crucial: When programs and queries are well-moded, derivations cannot *deadlock*. Thus, as opposed to Smaus (1999b), our results capture also termination by deadlock. For instance, we can easily prove that the query $\mathtt{app}(X, Y, Z)$ terminates. A more detailed comparison is presented in the concluding section.

We also show that the results presented in this paper can be extended to programs and queries which are *permutation* nicely- or simply-moded (Smaus *et al.*, 1998).

To evaluate the practicality of the results we present, we consider the programs from various well-known collections, and we check whether they satisfy the conditions of our main theorem.

The paper is organized as follows. Section 2 contains some preliminary notations and definitions. In section 3 input-consuming derivations are introduced and some properties of them are proven. In section 4 we prove that, for nicely-moded input-consuming programs, a left switching lemma holds. In section 5 a method for proving input termination of programs is presented, first in a non-modular way, then for modular programs. In section 6 we show that this method is necessary for the class of simply-moded and input-recursive programs. Section 7 discusses the applicability of our results through simple examples of programs and reports the results obtained by applying our method to various benchmarks. Finally, section 8 concludes the paper.

2 Preliminaries

The reader is assumed to be familiar with the terminology and the basic results of logic programs (Apt, 1990, 1997; Lloyd, 1987).

2.1 Terms and substitutions

Let $\mathscr{T}$ be the set of terms built on a finite set of *data constructors* $\mathscr{C}$ and a denumerable set of *variable symbols* $\mathscr{V}$. A *substitution* θ is a mapping from $\mathscr{V}$ to $\mathscr{T}$ such that $Dom(\theta) = \{X |\ \theta(X) \neq X\}$ is finite. For any syntactic object o, we denote by $Var(o)$ the set of variables occurring in o. A syntactic object is linear if every variable occurs in it at most once. We denote by ϵ the empty substitution. The *composition* $\theta\sigma$ of the substitutions θ and σ is defined as the functional composition, i.e. $\theta\sigma(X) = \sigma(\theta(X))$. We consider the pre-ordering $\leqslant$ (more general than) on substitutions such that $\theta \leqslant \sigma$ iff there exists γ such that $\theta\gamma = \sigma$. The result of the application of a substitution θ to a term t is said an *instance* of t and it is denoted by $t\theta$. We also consider the pre-ordering $\leqslant$ (more general than) on terms such that

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

$t \leqslant t'$ iff there exists θ such that $t\theta = t'$. We denote by $\approx$ the associated equivalence relation (*variance*). A substitution θ is a *unifier* of terms t and t' iff $t\theta = t'\theta$. We denote by $mgu(t, t')$ any *most general unifier* (*mgu*, in short) of t and t'. An mgu θ of terms t and t' is called relevant iff $Var(\theta) \subseteq Var(t) \cup Var(t')$.

2.2 Programs and derivations

Let $\mathscr{P}$ be a finite set of *predicate symbols*. An *atom* is an object of the form $p(t_1, \ldots, t_n)$ where $p \in \mathscr{P}$ is an n-ary predicate symbol and $t_1, \ldots, t_n \in \mathscr{T}$. Given an atom A, we denote by $Rel(A)$ the predicate symbol of A. A *query* is a finite, possibly empty, sequence of atoms $A_1, \ldots, A_m$. The empty query is denoted by $\Box$. Following the convention adopted in (Apt, 1997), we use bold characters to denote queries. A *clause* is a formula $H \leftarrow \mathbf{B}$ where H is an atom (the *head*) and $\mathbf{B}$ is a query (the *body*). When $\mathbf{B}$ is empty, $H \leftarrow \mathbf{B}$ is written $H \leftarrow$ and is called a *unit clause*. A *program* is a finite set of clauses. We denote atoms by $A, B, H, \ldots$, queries by $Q, \mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$, clauses by $c, d, \ldots$, and programs by P.

Computations are constructed as sequences of 'basic' steps. Consider a non-empty query $\mathbf{A}, B, \mathbf{C}$ and a clause c. Let $H \leftarrow \mathbf{B}$ be a variant of c variable disjoint from $\mathbf{A}, B, \mathbf{C}$. Let B and H unify with mgu θ. The query $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ is called a *resolvent of* $\mathbf{A}, B, \mathbf{C}$ *and* c *with selected atom* B *and mgu* θ. A *derivation step* is denoted by

$$\mathbf{A}, B, \mathbf{C} \overset{\theta}{\Longrightarrow}_{P,c} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$$

The clause $H \leftarrow \mathbf{B}$ is called its *input clause*. The atom B is called the *selected atom* of $\mathbf{A}, B, \mathbf{C}$.

If P is clear from the context or c is irrelevant then we drop the reference to them. A derivation is obtained by iterating derivation steps. A maximal sequence

$$\delta := Q_0 \overset{\theta_1}{\Longrightarrow}_{P,c_1} Q_1 \overset{\theta_2}{\Longrightarrow}_{P,c_2} \cdots Q_n \overset{\theta_{n+1}}{\Longrightarrow}_{P,c_{n+1}} Q_{n+1} \cdots$$

is called a *derivation of* $P \cup \{Q_0\}$ provided that for every step the standardization apart condition holds, i.e. the input clause employed is variable disjoint from the initial query Q_0 and from the substitutions and the input clauses used at earlier steps.

Derivations can be finite or infinite. If $\delta := Q_0 \overset{\theta_1}{\Longrightarrow}_{P,c_1} \cdots \overset{\theta_n}{\Longrightarrow}_{P,c_n} Q_n$ is a finite prefix of a derivation, also denoted $\delta := Q_0 \overset{\theta}{\longmapsto} Q_n$ with $\theta = \theta_1 \cdots \theta_n$, we say that δ is a *partial derivation* and θ is a *partial computed answer substitution* of $P \cup \{Q_0\}$. If δ is maximal and ends with the empty query then θ is called *computed answer substitution* (*c.a.s.*, for short). The length of a (partial) derivation δ, denoted by $len(\delta)$, is the number of derivation steps in δ.

The following definition of **B**-step is due to Smaus (1999a).

Definition 1 (**B**-*step*)
Let $\mathbf{A}, B, \mathbf{C} \overset{\theta}{\Longrightarrow} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ be a derivation step. We say that each atom in $\mathbf{B}\theta$ is a *direct descendant of* B, and for each atom E in $(\mathbf{A}, \mathbf{C})$, $E\theta$ is a *direct descendant of* E. We say that E is a descendant of F if the pair (E, F) is in the reflexive,

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

transitive closure of the relation *is a direct descendant of*. Consider a derivation $Q_0 \overset{\theta_1}{\Longrightarrow} \cdots \overset{\theta_i}{\Longrightarrow} Q_i \cdots \overset{\theta_j}{\Longrightarrow} Q_j \overset{\theta_{j+1}}{\Longrightarrow} Q_{j+1} \cdots$. We say that $Q_j \overset{\theta_{j+1}}{\Longrightarrow} Q_{j+1} \cdots$ is a **B**-step if **B** is a subquery of Q_i and the selected atom in Q_j is a descendant of an atom in **B**.

3 Modes and input-consuming derivations

In this section we introduce the concept of input-consuming derivation which is strictly related to the notion of mode; we discuss the relations between input-consuming derivations and programs using delay declarations; we recall the notion of nicely-moded program and state some properties.

3.1 Input-consuming derivations

Let us first recall the notion of mode. A *mode* is a function that labels as *input* or *output* the positions of each predicate in order to indicate how the arguments of a predicate should be used.

Definition 2 (*Mode*)
Consider an n-ary predicate symbol p. A *mode* for p is a function m_p from $\{1, \ldots, n\}$ to $\{In, Out\}$.

If $m_p(i) = In$ (resp. Out), we say that i is an *input* (resp. *output*) *position of* p (wrt. m_p). We assume that each predicate symbol has a unique mode associated to it; multiple modes may be obtained by simply renaming the predicates.

If Q is a query, we denote by $In(Q)$ (resp. $Out(Q)$) the sequence of terms filling in the input (resp. output) positions of predicates in Q. Moreover, when writing an atom as $p(\mathbf{s}, \mathbf{t})$, we are indicating with $\mathbf{s}$ the sequence of terms filling in the input positions of p and with $\mathbf{t}$ the sequence of terms filling in the output positions of p.

The notion of input-consuming derivation was introduced in Smaus (1999b), and is defined as follows.

Definition 3 (*Input-Consuming*)

- An atom $p(\mathbf{s}, \mathbf{t})$ is called *input-consuming resolvable wrt. a clause* $c := p(\mathbf{u}, \mathbf{v}) \leftarrow Q$ *and a substitution* θ iff $\theta = mgu(p(\mathbf{s}, \mathbf{t}), p(\mathbf{u}, \mathbf{v}))$ and $\mathbf{s} = \mathbf{s}\theta$.
- A derivation step

$$\mathbf{A}, B, \mathbf{C} \overset{\theta}{\Longrightarrow}_c (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$$

  is called *input-consuming* iff the selected atom B is input-consuming resolvable wrt. the input clause c and the substitution θ.
- A derivation is called *input-consuming* iff all its derivation steps are input-consuming.

The following lemma states that we are allowed to restrict our attention to input-consuming derivations with relevant mgu's.

Lemma 4
Let $p(\mathbf{s}, \mathbf{t})$ and $p(\mathbf{u}, \mathbf{v})$ be two atoms. If there exists an mgu θ of $p(\mathbf{s}, \mathbf{t})$ and $p(\mathbf{u}, \mathbf{v})$ such that $\mathbf{s}\theta = \mathbf{s}$, then there exists a *relevant* mgu ϑ of $p(\mathbf{s}, \mathbf{t})$ and $p(\mathbf{u}, \mathbf{v})$ such that $\mathbf{s}\vartheta = \mathbf{s}$.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Proof
Since $p(\mathbf{s}, \mathbf{t})$ and $p(\mathbf{u}, \mathbf{v})$ are unifiable, there exists a relevant mgu θ_{rel} of them (cfr. (Apt, 1997), Theorem 2.16). Now, θ_{rel} is a renaming of θ. Thus $\mathbf{s}\theta_{rel}$ is a variant of $\mathbf{s}$. Then there exists a renaming ρ such that $Dom(\rho) \subseteq Var(\mathbf{s}, \mathbf{t}, \mathbf{u}, \mathbf{v})$ and $\mathbf{s}\theta_{rel}\rho = \mathbf{s}$. Now, take $\vartheta = \theta_{rel}\rho$. □

From now on, we assume that all mgu's used in the input-consuming derivation steps are relevant.

Example 5
Consider the program REVERSE with accumulator in the modes defined below.

```
mode reverse(In, Out).
mode reverse_acc(In,Out,In)

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

The derivation δ of $\texttt{REVERSE} \cup \{\texttt{reverse}([\texttt{X1}, \texttt{X2}], \texttt{Zs})\}$ depicted below is input-consuming.

$$\begin{aligned}\delta := \quad & \texttt{reverse}([\texttt{X1}, \texttt{X2}], \texttt{Zs}) \Rightarrow \texttt{reverse_acc}([\texttt{X1}, \texttt{X2}], \texttt{Zs}, [\,]) \Rightarrow \\ & \texttt{reverse_acc}([\texttt{X2}], \texttt{Zs}, [\texttt{X1}]) \Rightarrow \texttt{reverse_acc}([\,], \texttt{Zs}, [\texttt{X2}, \texttt{X1}]) \Rightarrow \square.\end{aligned}$$

3.2 Input-consuming vs. delay declarations

Delay declarations are by far the most popular mechanism for implementing dynamic scheduling. However, being a non-logical mechanism, they are difficult to model and there are few proposals concerning their semantics (Marriott, 1997; Falaschi *et al.*, 1997).

An alternative approach to dynamic scheduling, which is much more declarative in nature, has been proposed by Smaus (1999b). It consists in the use of input-consuming derivations.

There is a main difference between the concept of delay declaration and the one of input-consuming derivation: While in the first case only the atom selectability is controlled, in the second one both the atom and the clause selectability are affected. In fact, in presence of delay declarations, if an atom is selectable then it can be resolved with respect to any program clause (provided it unifies with its head); on the contrary, in an input-consuming derivation, if an atom is selectable then it is input-consuming resolvable wrt. some, but not necessarily all, program clauses, i.e. only a restricted class of clauses can be used for resolution.

Also the concept of *deadlock* has to be understood in two different ways. For programs using delay declarations a deadlock situation occurs when no atom in a query satisfies the delay declarations (i.e., no atom is selectable), while for input-consuming derivations a deadlock occurs when no atom in a query is resolvable via an input-consuming derivation step and the derivation does not fail, i.e. there is

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

some atom in the query which unifies with a clause head but the unification is not input-consuming.

In spite of these differences, in many situations there is a strict relation between programs using delay declarations and input-consuming derivations. This relation is studied by Smaus in his PhD thesis (1999a). More precisely, Smaus proves a result that relates `block` declarations and input-consuming derivations. A `block` declaration is a special case of delay declaration and it is used to declare that certain arguments of an atom must be *non-variable* when the atom is selected for resolution. Smaus (1999a, Chapter 7) shows that `block` declarations can be used to ensure that derivations are input-consuming. In force of this result and of practical experience, we might claim that in most 'usual' moded programs using them, delay declarations are employed precisely for ensuring the input-consumedness of the derivations.

In fact, delay declarations are generally employed to guarantee that the interpreter will not use an 'inappropriate' clause for resolving an atom (the other, perhaps less prominent, use of delay declarations is to ensure absence of runtime errors, but we do not address this issue in this paper). This is achieved by preventing the selection of an atom until a certain degree of instantiation is reached. This degree of instantiation ensures then that the atom is unifiable only with the heads of the 'appropriate' clauses. In presence of modes, we can reasonably assume that this degree of instantiation is the one of the *input* positions, which are the ones carrying the information. Now, it is easy to see that a derivation step involving a clause c is input-consuming iff no further instantiation of the input positions of the resolved atom could prevent it from being resolvable with c. Therefore, c must belong to the set of 'appropriate' clauses for resolving it. Thus, the concepts of input-consuming derivation and of delay declarations are often employed for ensuring the same properties.

3.3 Nicely-moded programs

In the sequel of the paper, we will restrict ourselves to programs and queries which are nicely-moded. In this section we report the definition of this concept together with some basic important properties of nicely-moded programs.

Definition 6 (*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 sequence of terms and for all $i \in \{1, \ldots, n\}$

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

- A clause $c = p(\mathbf{s}_0, \mathbf{t}_0) \leftarrow Q$ is *nicely-moded* if Q is nicely-moded and

$$Var(\mathbf{s}_0) \cap \bigcup_{j=1}^{n} Var(\mathbf{t}_j) = \emptyset.$$

  In particular, every unit clause is nicely-moded.
- A program P is nicely-moded if all of its clauses are nicely-moded.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Note that a one-atom query $p(\mathbf{s}, \mathbf{t})$ is nicely-moded if and only if $\mathbf{t}$ is linear and $Var(\mathbf{s}) \cap Var(\mathbf{t}) = \emptyset$.

Example 7

- The program `APPEND` in the modes `app(In,In,Out)` is nicely-moded.
- The program `REVERSE` with accumulator in the modes depicted in Example 5 is nicely-moded.
- The following program `MERGE` is nicely-moded.

```
mode merge(In,In,Out).

merge(Xs,[ ],Xs).
merge([ ],Xs,Xs).
merge([X|Xs],[Y|Ys],[Y|Zs])  ← Y < X, merge([X|Xs],Ys,Zs).
merge([X|Xs],[Y|Ys],[X|Zs])  ← Y > X, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[X|Ys],[X|Zs])  ← merge(Xs,[X|Ys],Zs).
```

The following result is due to Smaus (1999a), and states that the class of programs and queries we are considering is persistent under resolution.

Lemma 8
Every resolvent of a nicely-moded query Q and a nicely-moded clause c, where the derivation step is input-consuming and $Var(Q) \cap Var(c) = \emptyset$, is nicely-moded.

The following Remark, also in (Smaus, 1999a), is an immediate consequence of the definition of input-consuming derivation step and the fact that the mgu's we consider are relevant.

Remark 9
Let the program P and the query $Q := \mathbf{A}, p(\mathbf{s}, \mathbf{t}), \mathbf{C}$ be nicely-moded. If $\mathbf{A}, p(\mathbf{s}, \mathbf{t}), \mathbf{C} \overset{\theta}{\Longrightarrow} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ is an input-consuming derivation step with selected atom $p(\mathbf{s}, \mathbf{t})$, then $\mathbf{A}\theta = \mathbf{A}$.

4 The left switching lemma

The *switching lemma* (see, for instance, Apt (1997, Lemma 3.32)) is a well-known result which allows one to prove the independence of the computed answer substitutions from the selection rule.

In the case of logic programs using dynamic scheduling, the switching lemma does not hold any longer. For example, in program `APPEND` reported in the introduction (together with the delay declaration d_1) we have that the rightmost atom of Q_2 is selectable only after the leftmost one has been resolved, i.e. the switching lemma cannot be applied.

Nevertheless, we can show that, for input-consuming derivations of nicely-moded programs, a weak version of the switching lemma still holds. Intuitively, we show that we can switch the selection of two atoms whenever this results in a left to right selection. For this reason, we call it *left switching lemma.*

First, we need one technical result, stating that the only variables of a query that can be 'affected' in an input-consuming derivation process are those occurring in

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

some output positions. Intuitively, this means that if the input arguments of a call are not 'sufficiently instantiated' then it is delayed until it allows for an input-consuming derivation step (if it is not the case then a deadlock situation will arise).

Lemma 10
Let the program P and the query Q be nicely-moded. Let $\delta := Q \overset{\theta}{\longmapsto} Q'$ be a partial input-consuming derivation of $P \cup \{Q\}$. Then, for all $x \in Var(Q)$ and $x \notin Var(Out(Q))$, $x\theta = x$.

Proof
Let us first establish the following claim.

Claim 11
Let $\mathbf{z}$ and $\mathbf{w}$ be two variable disjoint sequences of terms such that $\mathbf{w}$ is linear and $\theta = mgu(\mathbf{z}, \mathbf{w})$. If s_1 and s_2 are two variable disjoint terms occurring in $\mathbf{z}$ then $s_1\theta$ and $s_2\theta$ are variable disjoint terms.

Proof
The result follows from Lemmata 11.4 and 11.5 in Apt & Pedreschi (1994). □

We proceed with the proof of the lemma by induction on $len(\delta)$.

Base Case. Let $len(\delta) = 0$. In this case $Q = Q'$ and the result follows trivially.

Induction step. Let $len(\delta) > 0$. Suppose that $Q := \mathbf{A}, p(\mathbf{s}, \mathbf{t}), \mathbf{C}$ and

$$\delta := \mathbf{A}, p(\mathbf{s}, \mathbf{t}), \mathbf{C} \overset{\theta_1}{\Longrightarrow} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta_1 \overset{\theta_2}{\longmapsto} Q'$$

where $p(\mathbf{s}, \mathbf{t})$ is the selected atom of Q, $c := p(\mathbf{u}, \mathbf{v}) \leftarrow \mathbf{B}$ is the input clause used in the first derivation step, θ_1 is a relevant mgu of $p(\mathbf{s}, \mathbf{t})$ and $p(\mathbf{u}, \mathbf{v})$ and $\theta = \theta_1\theta_2$.

Let $x \in Var(\mathbf{A}, p(\mathbf{s}, \mathbf{t}), \mathbf{C})$ and $x \notin Var(Out(\mathbf{A}, p(\mathbf{s}, \mathbf{t}), \mathbf{C}))$. We first show that

$$x\theta_1 = x \tag{1}$$

We distinguish two cases:

(*a*) $x \in Var(\mathbf{s})$. In this case, property (1) follows from the hypothesis that δ is input-consuming.

(*b*) $x \notin Var(\mathbf{s})$. Since $x \in Var(\mathbf{A}, p(\mathbf{s}, \mathbf{t}), \mathbf{C})$, by standardization apart, we have that $x \notin Var(p(\mathbf{u}, \mathbf{v}))$. Moreover, since $x \notin Var(Out(\mathbf{A}, p(\mathbf{s}, \mathbf{t}), \mathbf{C}))$, it also holds that $x \notin Var(p(\mathbf{s}, \mathbf{t}))$. Then, property (1) follows from relevance of θ_1.

Now we show that

$$x\theta_2 = x \tag{2}$$

Again, we distinguish two cases:

(*c*) $x \notin Var((\mathbf{A}, \mathbf{B}, \mathbf{C})\theta_1)$. In this case, because of the standardization apart condition, x will never occur in $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta_1 \overset{\theta_2}{\longmapsto} Q'$. Hence, $x \notin Dom(\theta_2)$ and $x\theta_2 = x$.

(*d*) $x \in Var((\mathbf{A}, \mathbf{B}, \mathbf{C})\theta_1)$. In this case, in order to prove (2) we show that $x \notin Var(Out((\mathbf{A}, \mathbf{B}, \mathbf{C})\theta_1))$. The result then follows by the inductive hypothesis.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

From the standardization apart, relevance of θ_1 and the fact that the first derivation step is input-consuming, it follows that $Dom(\theta_1) \cap Var(Q) \subseteq Var(\mathbf{t})$.

From the hypothesis that Q is nicely-moded, $Var(\mathbf{t}) \cap Var(Out(\mathbf{A}, \mathbf{C})) = \emptyset$. Hence, $Var(Out(\mathbf{A}, \mathbf{C}))\theta_1 = Var(Out(\mathbf{A}, \mathbf{C}))$. Since $x \notin Var(Out(\mathbf{A}, \mathbf{C}))$, this proves that $x \notin Var(Out((\mathbf{A}, \mathbf{C})\theta_1))$.

It remains to be proven that $x \notin Var(Out(\mathbf{B}\theta_1)$. We distinguish two cases:

($d1$) $x \notin Var(\mathbf{s})$. Since $x \notin Var(p(\mathbf{s}, \mathbf{t}))$, the fact that $x \notin Var(Out(\mathbf{B}\theta_1)$ follows immediately by standardization apart condition and relevance of θ_1.

($d2$) $x \in Var(\mathbf{s})$. By known results (see Apt (1997, Corollary 2.25)), there exists two relevant mgu σ_1 and σ_2 such that

- $\theta_1 = \sigma_1\sigma_2$,
- $\sigma_1 = mgu(\mathbf{s}, \mathbf{u})$,
- $\sigma_2 = mgu(\mathbf{t}\sigma_1, \mathbf{v}\sigma_1)$.

From the relevance of σ_1 and the fact that, by nicely-modedness of Q, $Var(\mathbf{s}) \cap Var(\mathbf{t}) = \emptyset$, we have that $\mathbf{t}\sigma_1 = \mathbf{t}$, and by the standardization apart condition $Var(\mathbf{t}) \cap Var(\mathbf{v}\sigma_1) = \emptyset$. Now by nicely-modedness of c, $Var(\mathbf{u}) \cap Var(Out(\mathbf{B})) = \emptyset$. Since σ_1 is relevant, and by the standardization apart condition, it follows that

$$Var(\mathbf{u}\sigma_1) \cap Var(Out(\mathbf{B}\sigma_1)) = \emptyset \tag{3}$$

The proof proceeds now by contradiction. Suppose that $x \in Var(Out(\mathbf{B}\sigma_1\sigma_2))$. Since by hypothesis $x \in Var(\mathbf{s})$, and $\mathbf{s} = \mathbf{u}\sigma_1\sigma_2$, we have that $Var(\mathbf{u}\sigma_1\sigma_2) \cap Var(Out(\mathbf{B}\sigma_1\sigma_2)) \neq \emptyset$. By (3), this means that there exist two distinct variables z_1 and z_2 in $Var(\sigma_2)$ such that $z_1 \in Var(Out(\mathbf{B}\sigma_1))$, $z_2 \in Var(\mathbf{u}\sigma_1)$ and

$$Var(z_1\sigma_2) \cap Var(z_2\sigma_2) \neq \emptyset \tag{4}$$

Since, by the standardization apart condition and relevance of the mgu's, $Var(\sigma_2) \subseteq Var(\mathbf{v}\sigma_1) \cup Var(\mathbf{t})$ and $(Var(Out(\mathbf{B}\sigma_1)) \cup Var(\mathbf{u}\sigma_1)) \cap Var(\mathbf{t}) = \emptyset$, we have that z_1 and z_2 are two disjoint subterms of $\mathbf{v}\sigma_1$. Since $\sigma_2 = mgu(\mathbf{t}, \mathbf{v}\sigma_1)$, $\mathbf{t}$ is linear and disjoint from $\mathbf{v}\sigma_1$, (4) contradicts Claim 11. □

The following corollary is an immediate consequence of the above lemma and the definition of nicely-moded program.

Corollary 12

Let the program P and the one-atom query A be nicely-moded. Let $\delta := A \overset{\theta}{\longmapsto} Q'$ be a partial input-consuming derivation of $P \cup \{A\}$. Then, for all $x \in Var(In(A))$, $x\theta = x$.

Next is the main result of this section, showing that for input-consuming nicely-moded programs one half of the well-known switching lemma holds.

Lemma 13 (*Left-Switching*)

Let the program P and the query Q_0 be nicely-moded. Let δ be a partial input-consuming derivation of $P \cup \{Q_0\}$ of the form

$$\delta := Q_0 \overset{\theta_1}{\Longrightarrow}_{c_1} Q_1 \cdots Q_n \overset{\theta_{n+1}}{\Longrightarrow}_{c_{n+1}} Q_{n+1} \overset{\theta_{n+2}}{\Longrightarrow}_{c_{n+2}} Q_{n+2}$$

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

where

- Q_n is a query of the form $\mathbf{A}, B, \mathbf{C}, D, \mathbf{E}$,
- Q_{n+1} is a resolvent of Q_n and c_{n+1} wrt. D,
- Q_{n+2} is a resolvent of Q_{n+1} and c_{n+2} wrt. $B\theta_{n+1}$.

Then, there exist Q'_{n+1}, θ'_{n+1}, θ'_{n+2} and a derivation δ' such that

$$\theta_{n+1}\theta_{n+2} = \theta'_{n+1}\theta'_{n+2}$$

and

$$\delta' := Q_0 \overset{\theta_1}{\Longrightarrow}_{c_1} Q_1 \cdots Q_n \overset{\theta'_{n+1}}{\Longrightarrow}_{c_{n+2}} Q'_{n+1} \overset{\theta'_{n+2}}{\Longrightarrow}_{c_{n+1}} Q_{n+2}$$

where δ' is input-consuming and

- δ and δ' coincide up to the resolvent Q_n,
- Q'_{n+1} is a resolvent of Q_n and c_{n+2} wrt. B,
- Q_{n+2} is a resolvent of Q'_{n+1} and c_{n+1} wrt. $D\theta'_{n+1}$,
- δ and δ' coincide after the resolvent Q_{n+2}.

Proof
Let $B := p(\mathbf{s}, \mathbf{t})$, $D := q(\mathbf{u}, \mathbf{v})$, $c_{n+1} := q(\mathbf{u}', \mathbf{v}') \leftarrow \mathbf{D}$ and $c_{n+2} := p(\mathbf{s}', \mathbf{t}') \leftarrow \mathbf{B}$. Hence, $\theta_{n+1} = mgu(q(\mathbf{u}, \mathbf{v}), q(\mathbf{u}', \mathbf{v}'))$ and

$$\mathbf{u}\theta_{n+1} = \mathbf{u}, \text{ since } \delta \text{ is input-consuming.} \tag{5}$$

By (5) and the fact that Q_n is nicely-moded and θ_{n+1} is relevant, we have that $p(\mathbf{s}, \mathbf{t})\theta_{n+1} = p(\mathbf{s}, \mathbf{t})$. Then, $\theta_{n+2} = mgu(p(\mathbf{s}, \mathbf{t})\theta_{n+1}, p(\mathbf{s}', \mathbf{t}')) = mgu(p(\mathbf{s}, \mathbf{t}), p(\mathbf{s}', \mathbf{t}'))$ and

$$\mathbf{s}\theta_{n+2} = \mathbf{s}, \text{ since } \delta \text{ is input-consuming.} \tag{6}$$

Moreover,[2]

$$\theta_{n+1}\theta_{n+2} = mgu\{p(\mathbf{s}, \mathbf{t}) = p(\mathbf{s}', \mathbf{t}'), q(\mathbf{u}, \mathbf{v}) = q(\mathbf{u}', \mathbf{v}')\} = \theta_{n+2}\theta'_{n+2} \tag{7}$$

where

$$\theta'_{n+2} = mgu(q(\mathbf{u}, \mathbf{v})\theta_{n+2}, q(\mathbf{u}', \mathbf{v}')\theta_{n+2}) = mgu(q(\mathbf{u}, \mathbf{v})\theta_{n+2}, q(\mathbf{u}', \mathbf{v}')).$$

We construct the derivation δ' as follows.

$$\delta' := Q_0 \overset{\theta_1}{\Longrightarrow}_{c_1} Q_1 \cdots Q_n \overset{\theta'_{n+1}}{\Longrightarrow}_{c_{n+2}} Q'_{n+1} \overset{\theta'_{n+2}}{\Longrightarrow}_{c_{n+1}} Q_{n+2}$$

where

$$\theta'_{n+1} = \theta_{n+2} \tag{8}$$

By (6), $Q_n \overset{\theta'_{n+1}}{\Longrightarrow}_{c_{n+2}} Q'_{n+1}$ is an input-consuming derivation step. Observe now that

$$\begin{aligned} \mathbf{u}\theta'_{n+1}\theta'_{n+2} &= \mathbf{u}\theta_{n+2}\theta'_{n+2}, && \text{(by (8))} \\ &= \mathbf{u}\theta_{n+1}\theta_{n+2}, && \text{(by (7))} \\ &= \mathbf{u}\theta_{n+2}, && \text{(by (5))} \\ &= \mathbf{u}\theta'_{n+1}, && \text{(by (8)).} \end{aligned}$$

This proves that $Q'_{n+1} \overset{\theta'_{n+2}}{\Longrightarrow}_{c_{n+1}} Q'_{n+2}$ is an input-consuming derivation step. □

[2] We use the notation $mgu(E)$ to denote the mgu of a set of equations E, see Apt (1997).

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

This result shows that it is always possible to proceed left-to-right to resolve the selected atoms. Notice that this is different than saying that the leftmost atom of a query is always resolvable: it can very well be the case that the leftmost atom is suspended and the one next to it is resolvable. However, if the leftmost atom of a query is not resolvable then we can state that the derivation will not succeed, i.e. either it ends by deadlock, or by failure or it is infinite.

It is important to notice that if we drop the nicely-modedness condition the above lemma would not hold any longer. For instance, it does not apply to the query Q_1 of the introduction which is not nicely-moded. In fact, the leftmost atom of Q_1 is resolvable only after the rightmost one has been resolved at least once.

The following immediate corollary will be used in the sequel.

Corollary 14
Let the program P and the query $Q := \mathbf{A}, \mathbf{B}$ be nicely-moded. Suppose that

$$\delta := \mathbf{A}, \mathbf{B} \overset{\theta}{\longmapsto} \mathbf{C}_1, \mathbf{C}_2$$

that is a partial input-consuming derivation of $P \cup \{Q\}$ where $\mathbf{C}_1$ and $\mathbf{C}_2$ are obtained by partially resolving $\mathbf{A}$ and $\mathbf{B}$, respectively. Then there exists a partial input-consuming derivation

$$\delta' := \mathbf{A}, \mathbf{B} \overset{\theta_1}{\longmapsto} \mathbf{C}_1, \mathbf{B}\theta_1 \overset{\theta_2}{\longmapsto} \mathbf{C}_1, \mathbf{C}_2$$

where all the $\mathbf{A}$-steps are performed in the prefix $\mathbf{A}, \mathbf{B} \overset{\theta_1}{\longmapsto} \mathbf{C}_1, \mathbf{B}\theta_1$ and $\theta = \theta_1\theta_2$.

5 Termination

In this section we study the termination of input-consuming derivations. To this end we refine the ideas of Bezem (1993) and Cavedon (1989), who studied the termination of logic programs in a very strong sense, namely with respect to all selection rules, and of Smaus (1999b), who characterized terminating input-consuming derivations of programs which are both well and nicely-moded.

5.1 Input terminating programs

We first introduce the key notion of this section.

Definition 15 (*Input Termination*)
A program is called *input terminating* iff all its input-consuming derivations started in a nicely-moded query are finite.

The method we use to prove that a program is input terminating is based on the following concept of moded level mapping due to Etalle *et al.* (1999).

Definition 16 (*Moded Level Mapping*)
Let P be a program and $\mathscr{B}_P^e$ be the extended Herbrand base[3] for the language associated with P. A function $|\ |$ is a *moded level mapping for* P iff:

[3] The extended Herbrand base of P is the set of equivalence classes of all (possibly non-ground) atoms, modulo renaming, whose predicate symbol appears in P. As usual, an atom is identified with its equivalence class.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

- it is a function $|\ |: \mathscr{B}_P^{\mathscr{E}} \to \mathbf{N}$ from atoms to natural numbers;
- for any $\mathbf{t}$ and $\mathbf{u}$, $|p(\mathbf{s},\mathbf{t})| = |p(\mathbf{s},\mathbf{u})|$.

For $A \in \mathscr{B}_P^{\mathscr{E}}$, $|A|$ is the *level* of A.

The condition $|p(\mathbf{s},\mathbf{t})| = |p(\mathbf{s},\mathbf{u})|$ states that the level of an atom is independent from the terms in its output positions. There is actually a small yet important difference between this definition and that in Etalle *et al.* (1999), where the level mapping is defined on ground atoms only. Indeed, in Etalle *et al.* (1999) only well-moded atoms are considered, i.e. atoms with ground terms in the input positions. Here, instead, we are considering nicely-moded atoms whose input positions can be filled in by (possibly) non-ground terms.

Example 17
Let us denote by $TSize(t)$ the term size of a term t, that is the number of function and constant symbols that occur in t.

- A moded level mapping for the program `APPEND` reported in the introduction is as follows:

  $$|\mathtt{app}(x_s, y_s, z_s)| = TSize(x_s).$$

- A moded level mapping for the program `REVERSE` with accumulator of Example 5 is the following:

  $$|\mathtt{reverse}(x_s, y_s)| = TSize(x_s)$$
  $$|\mathtt{reverse_acc}(x_s, y_s, z_s)| = TSize(x_s).$$

5.2 *Quasi recurrency*

To give a sufficient condition for termination, we are going to employ a generalization of the concept of *recurrent* and of *semi-recurrent* program. The first notion (which in the case of normal programs, i.e. programs with negation, coincides with the one of *acyclic program*) was introduced in Bezem (1993) and Apt & Bezem (1991), and independently in Cavedon (1991), in order to prove universal termination for all selection rules together with other properties of logic programs. Later, Apt & Pedreschi (1994) provided the new definition of semi-recurrent program, which is equivalent to the one of recurrent program, but it is easier to verify in an automatic fashion. To proceed, we need a preliminary definition.

Definition 18
Let P be a program, p and q be relations. We say that p *refers to* q in P iff there is a clause in P with p in the head and q in the body. We say that p *depends on* q and write $p \sqsupseteq q$ in P iff (p,q) is in the reflexive and transitive closure of the relation *refers to*.

According to the above definition, $p \simeq q \equiv p \sqsubseteq q \wedge p \sqsupseteq q$ means that p and q are mutually recursive, and $p \sqsupset q \equiv p \sqsupseteq q \wedge p \not\simeq q$ means that p calls q as a subprogram. Notice that $\sqsupset$ is a well-founded ordering.

Finally, we can provide the key concept we are going to use in order to prove input termination.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Definition 19 (*Quasi Recurrency*)
Let P be a program and $|\,|:\mathscr{B}_P^e \to \mathbf{N}$ be a moded level mapping.

- A clause of P is called *quasi recurrent with respect to* $|\,|$ if for every instance of it, $H \leftarrow \mathbf{A}, B, \mathbf{C}$

$$\text{if } Rel(H) \simeq Rel(B) \text{ then } |H| > |B|. \tag{9}$$

- A program P is called *quasi recurrent with respect to* $|\,|$ if all its clauses are. P is called *quasi recurrent* if it is quasi recurrent wrt. some moded level mapping $|\,|:\mathscr{B}_P^e \to \mathbf{N}$.

The notion of quasi recurrent program differs from the concepts of recurrent and of semi-recurrent program in two ways. First, we require that $|H| > |B|$ only for those body atoms which mutually depend on $Rel(H)$; in contrast, both the concept of recurrent and of semi-recurrent program require that $|H| > |B|$ ($|H| \geqslant |B|$ in the case of semi-recurrency) also for the atoms for which $Rel(H) \not\simeq Rel(B)$. Secondly, every instance of a program clause is considered, not only ground instances as in the case of (semi-)recurrent programs. This allows us to treat directly any nicely-moded query without introducing the concept of *boundedness* (Apt & Pedreschi, 1994) or *cover*, as in Marchiori & Teusink (1999).

It is worth noticing that this concept almost coincides with the one of *ICD-acceptable* program introduced and used in Smaus (1999b). We decided to use a different name because we believe that referring to the word acceptable might lead to confusion: The concept of acceptable program was introduced by Apt & Pedreschi (1993, 1994) in order to prove termination of logic programs using the left-to-right selection rule. The crucial difference between recurrency and acceptability lies in the fact that the latter relies on a model M; this allows condition (9) to be checked only for those body atoms which are in a way 'reachable' wrt. M. Hence, every recurrent program is acceptable, but not vice-versa. As an aside, Marchiori & Teusink (1999) introduce the notion of *delay recurrent* program although their concept is based on the presence of a model M. Our definition does not rely on a model, and so it is much more related to the notion of recurrent than that of an acceptable program.

We can now state our first basic result on termination, in the case of non-modular programs.

Theorem 20
Let P be a nicely-moded program. If P is quasi recurrent then P is input terminating.

Proof
It will be obtained from the proof of Theorem 24 by setting $R = \emptyset$. □

Example 21
Consider the program `MERGE` defined in Example 7. Let $|\,|$ be the moded level mapping for `MERGE` defined by

$$|\texttt{merge}(x_s, y_s, z_s)| = TSize(x_s) + TSize(y_s).$$

It is easy to prove that `MERGE` is quasi recurrent wrt. the moded level mapping above. By Theorem 20, all input-consuming derivations of `MERGE` started with a query $\texttt{merge}(s, t, u)$, where u is linear and variable disjoint from s and t, are terminating.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

5.3 Modular termination

This section contains a generalization of Theorem 20 to the modular case, as well as the complete proofs for it. The following lemma is a crucial one.

Lemma 22
Let the program P and the query $Q := A_1, \ldots, A_n$ be nicely-moded. Suppose that there exists an infinite input-consuming derivation δ of $P \cup \{Q\}$. Then, there exist an index $i \in \{1, \ldots, n\}$ and substitution θ such that

1. there exists an input-consuming derivation δ' of $P \cup \{Q\}$ of the form
$$\delta' := A_1, \ldots, A_n \overset{\theta}{\longmapsto} \mathbf{C}, (A_i, \ldots, A_n)\theta \longmapsto \cdots$$
2. there exists an infinite input-consuming derivation of $P \cup \{A_i\theta\}$.

Proof
Let $\delta := A_1, \ldots, A_n \longmapsto \cdots$ be an infinite input-consuming derivation of $P \cup \{Q\}$. Then δ contains an infinite number of A_k-steps for some $k \in \{1, \ldots, n\}$. Let i be the minimum of such k. Hence δ contains a finite number of A_j-steps for $j \in \{1, \ldots, i-1\}$ and there exists $\mathbf{C}$ and $\mathbf{D}$ such that
$$\delta := A_1, \ldots, A_n \overset{\vartheta}{\longmapsto} \mathbf{C}, \mathbf{D} \longmapsto \cdots$$
where $A_1, \ldots, A_n \overset{\vartheta}{\longmapsto} \mathbf{C}, \mathbf{D}$ is a finite prefix of δ which comprises all the A_j-steps of δ for $j \in \{1, \ldots, i-1\}$ and $\mathbf{C}$ is the subquery of $\mathbf{C}, \mathbf{D}$ consisting of the atoms resulting from some A_j-step ($j \in \{1, \ldots, i-1\}$). By Corollary 14, there exists an infinite input-consuming derivation δ' such that
$$\delta' := A_1, \ldots, A_n \overset{\theta}{\longmapsto} \mathbf{C}, (A_i, \ldots, A_n)\theta \overset{\theta'}{\longmapsto} \mathbf{C}, \mathbf{D} \longmapsto \cdots$$
where $\vartheta = \theta\theta'$. This proves (i).

Now, let $\delta'' := \mathbf{C}, (A_i, \ldots, A_n)\theta \overset{\theta'}{\longmapsto} \mathbf{C}, \mathbf{D} \longmapsto \cdots$. Note that in δ'' the atoms of $\mathbf{C}$ will never be selected and, by Remark 9, will never be instantiated. Let δ''' be obtained from δ'' by omitting the prefix $\mathbf{C}$ in each query. Hence δ''' is an infinite input-consuming derivation of $P \cup \{(A_i, \ldots, A_n)\theta\}$ where an infinite number of $A_i\theta$-steps are performed. Again, By Remark 9, for every finite prefix of δ''' of the form
$$A_i\theta, (A_{i+1}, \ldots, A_n)\theta \overset{\sigma_1}{\longmapsto} \mathbf{D}_1, \mathbf{D}_2 \overset{\sigma_2}{\Longrightarrow} \mathbf{D}'_1, \mathbf{D}'_2$$
where $\mathbf{D}_1$ and $\mathbf{D}_2$ are obtained by partially resolving $A_i\theta$ and $(A_{i+1}, \ldots, A_n)\theta$, respectively, and $\mathbf{D}_1, \mathbf{D}_2 \overset{\sigma_2}{\Longrightarrow} \mathbf{D}'_1, \mathbf{D}'_2$ is an A_j-step for some $j \in \{i+1, \ldots, n\}$, we have that $\mathbf{D}'_1 = \mathbf{D}_1$. Hence, from the hypothesis that there is an infinite number of $A_i\theta$-steps in δ'', it follows that there exists an infinite input-consuming derivation of $P \cup \{A_i\theta\}$. This proves (ii). □

The importance of the above lemma is shown by the following corollary of it, which will allow us to concentrate on queries containing only one atom.

Corollary 23
Let P be a nicely-moded program. P is input terminating iff for each nicely-moded one-atom query A all input-consuming derivations of $P \cup \{A\}$ are finite.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

We can now state the main result of this section. Here and in what follows we say that a relation p is *defined in* the program P if p occurs in a head of a clause of P, and that P *extends* the program R if no relation defined in P occurs in R.

Theorem 24
Let P and R be two programs such that P extends R. Suppose that

- R is input terminating,
- P is nicely-moded and quasi recurrent wrt. a moded level mapping $|\ | : \mathscr{B}_P^{\mathscr{E}} \to \mathbf{N}$.

Then $P \cup R$ is input terminating.

Proof
First, for each predicate symbol p, we define $dep_P(p)$ to be the number of predicate symbols it depends on. More formally, $dep_P(p)$ is defined as the cardinality of the set $\{q|\ q \text{ is defined in } P \text{ and } p \sqsupseteq q\}$. Clearly, $dep_P(p)$ is always finite. Further, it is immediate to see that if $p \simeq q$ then $dep_P(p) = dep_P(q)$ and that if $p \sqsupset q$ then $dep_P(p) > dep_P(q)$.

We can now prove our theorem. By Corollary 23, it is sufficient to prove that for any nicely-moded one-atom query A, all input-consuming derivations of $P \cup \{A\}$ are finite.

First notice that if A is defined in R then the result follows immediately from the hypothesis that R is input terminating and that P is an extension of R. So we can assume that A is defined in P.

For the purpose of deriving a contradiction, assume that δ is an infinite input-consuming derivation of $(P \cup R) \cup \{A\}$ such that A is defined in P. Then

$$\delta := A \overset{\theta_1}{\Longrightarrow} (B_1, \ldots, B_n)\theta_1 \overset{\theta_2}{\Longrightarrow} \cdots$$

where $H \leftarrow B_1, \ldots, B_n$ is the input clause used in the first derivation step and $\theta_1 = mgu(A, H)$. Clearly, $(B_1, \ldots, B_n)\theta_1$ has an infinite input-consuming derivation in $P \cup R$. By Lemma 22, for some $i \in \{1, \ldots, n\}$ and for some substitution θ_2,

1. there exists an infinite input-consuming derivation of $(P \cup R) \cup \{A\}$ of the form
$$A \overset{\theta_1}{\Longrightarrow} (B_1, \ldots, B_n)\theta_1 \overset{\theta_2}{\longmapsto} \mathbf{C}, (B_i, \ldots, B_n)\theta_1\theta_2 \cdots;$$
2. there exists an infinite input-consuming derivation of $P \cup \{B_i\theta_1\theta_2\}$.

Notice also that $B_i\theta_1\theta_2$ is nicely-moded. Let now $\theta = \theta_1\theta_2$. Note that $H\theta \leftarrow (B_1, \ldots, B_n)\theta$ is an instance of a clause of P.

We show that (2) cannot hold. This is done by induction on $\langle dep_P(Rel(A)), |A| \rangle$ wrt. the ordering $\succ$ defined by: $\langle m, n \rangle \succ \langle m', n' \rangle$ iff either $m > m'$ or $m = m'$ and $n > n'$.

Base. Let $dep_P(Rel(A)) = 0$ ($|A|$ is arbitrary). In this case, A does not depend on any predicate symbol of P, thus all the B_i as well as all the atoms occurring in its descendents in any input-consuming derivation are defined in R. The hypothesis that R is input terminating contradicts (2) above.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Induction step. We distinguish two cases:

1. $Rel(H) \sqsupset Rel(B_i)$,
2. $Rel(H) \simeq Rel(B_i)$.

In case (*a*) we have that $dep_P(Rel(A)) = dep_P(Rel(H\theta)) > dep_P(Rel(B_i\theta))$. So, $\langle dep_P(Rel(A)), |A| \rangle = \langle dep_P(Rel(H\theta)), |H\theta| \rangle > \langle dep_P(Rel(B_i\theta)), |B_i\theta| \rangle$. In case (*b*), from the hypothesis that P is quasi recurrent wrt. $|\ |$, it follows that $|H\theta| > |B_i\theta|$.

Consider now the partial input-consuming derivation $A \overset{\theta}{\longmapsto} \mathbf{C}, (B_i, \ldots, B_n)\theta$. By Corollary 12 and the fact that $|\ |$ is a moded level mapping, it follows that $|A| = |A\theta| = |H\theta|$. Therefore, $\langle dep_P(Rel(A)), |A| \rangle = \langle dep_P(Rel(H\theta)), |H\theta| \rangle > \langle dep_P(Rel(B_i\theta)), |B_i\theta| \rangle$. In both cases, the contradiction follows by the inductive hypothesis. □

Example 25

The program FLATTEN using difference-lists is nicely-moded with respect to the modes described below, provided that one replaces "\" by ",", as we have done here.

```
mode flatten(In,Out).
mode flatten_dl(In,Out,In).
mode constant(In).
mode ≠(In,In).

flatten(Xs,Ys) ← flatten_dl(Xs,Ys,[ ]).

flatten_dl([ ],Ys,Ys).
flatten_dl(X,[X|Xs],Xs) ← constant(X), X ≠ [ ].
flatten_dl([X|Xs],Ys,Zs) ← flatten_dl(Xs,Y1s,Zs),
                           flatten_dl(X,Ys,Y1s).
```

Consider the moded level mapping for FLATTEN defined by

$|\texttt{flatten}(x_s, y_s)| = TSize(x_s)$
$|\texttt{flatten_dl}(x_s, y_s, z_s)| = TSize(x_s)$.

It is easy to see that the program FLATTEN is quasi recurrent wrt. the moded level mapping above. Hence, all input-consuming derivations of program FLATTEN started with a query flatten(s,t), where t is linear and variable disjoint from s, are terminating.

6 Termination: A necessary condition

Theorem 20 provides a sufficient condition for termination. The condition is not necessary, as demonstrated by the following simple example.

```
mode p(In,Out).

p(X,a) ← p(X,b).
p(X,b).
```

This program is clearly input terminating, however it is not quasi recurrent. If it was, we would have that $|\texttt{p(X, a)}| > |\texttt{p(X, b)}|$, for some moded level mapping $|\ |$ (otherwise

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

the first clause would not be quasi recurrent). On the other hand, since p(X, a) and p(X, b) differ only for the terms filling in their output positions, by definition of moded level mapping, $|p(X, a)| = |p(X, b)|$. Hence, we have a contradiction.

Nevertheless, as shown by other work (Bezem, 1993; Apt & Pedreschi, 1993; Etalle *et al.*, 1999), it is important to be able to give a characterization of termination, i.e. a condition which is *necessary* and sufficient to ensure termination. To this purpose is dedicated this section.

6.1 Simply-moded programs

As demonstrated by the example above, to provide a necessary condition for termination we need to further restrict the class of programs we consider. The first problem is that we should rule out those situations in which termination is guaranteed by the instantiation of the output positions of some selected atom, as it happens in the above example. For this we restrict to *simply-moded* programs which are nicely-moded programs with the additional condition that the output arguments of clause bodies are variables.

Definition 26 (*Simply-Moded*)

- A query Q (resp., a clause $c = H \leftarrow Q$) is *simply-moded* if it is nicely-moded and $Out(Q)$ is a linear sequence of variables.
- A program P is simply-moded iff all of its clauses are simply-moded.

It is important to notice that most programs are simply-moded (see the mini-survey at the end of Apt & Pedreschi (1993)), and that often non simply-moded programs can naturally be transformed into simply-moded ones.

Example 27

- The programs REVERSE of Example 5, MERGE of Example 7 and FLATTEN of Example 25 are all simply-moded.
- Consider the program LAST which extends REVERSE:

```
mode last(In,Out).
last(Ls,E) ← reverse(Ls,[E|_]).
```

  This program is not simply-moded since the argument filling in the output position in the body of the first clause is not a variable. However, it can be transformed into a simply-moded one as follows:

```
mode last(In,Out).
mode selectfirst(In,Out).

last(Ls,E) ← reverse(Ls,Rs), selectfirst(Rs,E).
selectfirst([E|_],E).
```

The following lemma, which is an immediate consequence of Lemma 30 in Apt & Luitjes (1995), shows the persistence of the notion of simply-modedness.

Lemma 28

Every resolvent of a simply-moded query Q and a simply-moded clause c, where the derivation step is input-consuming and $Var(Q) \cap Var(c) = \emptyset$, is simply-moded.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

6.2 Input-recursive programs

Unfortunately, the restriction to simply-moded programs alone is not sufficient to extend Theorem 20 by a necessary condition. Consider for instance the following program QUICKSORT:

```
        mode qs(In,Out).
        mode part(In,In,Out,Out).
        mode app(In,In,Out).

        qs([ ],[ ]).
c1 :=   qs([X|Xs],Ys)  ← part(X,Xs,Littles,Bigs),
          qs(Littles,Ls),
          qs(Bigs,Bs),
          app(Ls,[X|Bs],Ys).

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

This program is simply-moded and input terminating[4]. However it is not quasi recurrent. Indeed, there exist no moded level mapping $|\ |$ such that, for every variable-instance, $|\mathtt{qs([X|Xs],Ys)}| > |\mathtt{qs(Littles,Ls)}|$ and $|\mathtt{qs([X|Xs],Ys)}| > |\mathtt{qs(Bigs,Bs)}|$. This is due to the fact that, in clause c_1 there is no direct link between the input arguments of the recursive calls and those of the clause head. This motivates the following definition of *input-recursive* programs.

Definition 29 (*Input-Recursive*)
Let P be a program.

- A clause $H \leftarrow \mathbf{A}, B, \mathbf{C}$ of P is called *input-recursive* if

$$\text{if } Rel(H) \simeq Rel(B) \text{ then } Var(In(B)) \subseteq Var(In(H)).$$

- A program P is called *input-recursive* if all its clauses are.

Thus, we say that a clause is input-recursive if the set of variables occurring in the arguments filling in the input positions of each recursive call in the clause body is a subset of the set of variables occurring in the arguments filling in the input positions of the clause head. Input-recursive programs have strong similarities with primitive recursive functions.

Example 30

- The programs APPEND of the introduction, REVERSE of Example 5 and MERGE of Example 7 are all input-recursive.
- The program FLATTEN of Example 25 is not input-recursive. This is due to the presence of the fresh variable Y1s in a body atom of the last clause.
- QUICKSORT, is not input-recursive. In particular, clause c_1 is not input-recursive.

[4] Provided that one models the built-in predicates > and <= as being defined by (an infinite number of) ground facts of the form >(m,n) and <=(m,n). The problem here is that the definition of input-consuming derivation does not consider the presence of built-ins.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

6.3 Characterizing input terminating programs

We can now prove that by restricting ourselves to input-recursive and simply-moded programs, the condition of Theorem 20 is also a necessary one.

To prove this, we follow the approach of Apt and Pedreschi when characterizing terminating programs (Apt & Pedreschi, 1994). First we introduce the notion of *IC-tree* that corresponds to the notion of *S-tree* in Apt & Pedreschi (1994), and provides us with a representation for all input-consuming derivations of a program P with a query Q, then we define a level mapping which associates to every atom A the number of nodes of a given IC-tree and finally we prove that P is quasi recurrent wrt. such a level mapping.

Definition 31 (*IC-tree*)
An *IC-tree* for $P \cup \{Q\}$ is a tree whose nodes are labelled with queries such that

- its branches are input-consuming derivations of $P \cup \{Q\}$,
- every node Q has exactly one descendant for every atom A of Q and every clause c from P such that A is input-consuming resolvable wrt. c. This descendant is a resolvent of Q and c wrt. A.

In this tree, a node's children consist of all its resolvents, 'modulo renaming', via an input-consuming derivation step wrt. all the possible choices of a program clause and a selected atom.

Lemma 32 (*IC-tree 1*)
An IC-tree for $P \cup \{Q\}$ is finite iff all input-consuming derivations of $P \cup \{Q\}$ are finite.

Proof
By definition, the IC-trees are finitely branching. The claim now follows by König's Lemma. □

Notice that if an IC-tree for $P \cup \{Q\}$ is finite then all the IC-trees for $P \cup \{Q\}$ are finite.

For a program P and a query Q, we denote by $nodes_P^{ic}(Q)$ the number of nodes in an IC-tree for $P \cup \{Q\}$. The following properties of IC-trees will be needed.

Lemma 33 (*IC-tree 2*)
Let P be a program, Q be a query and T be a finite IC-tree for $P \cup \{Q\}$. Then

(i) for all non-root nodes Q' in T, $nodes_P^{ic}(Q') < nodes_P^{ic}(Q)$,
(ii) for all atoms A of Q, $nodes_P^{ic}(A) \leqslant nodes_P^{ic}(Q)$.

Proof
Immediate by Definition 31 of IC-tree. □

We can now prove the desired result.

Theorem 34
Let P be a simply-moded and input-recursive program. If P is input terminating then P is quasi recurrent.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Proof

We show that there exists a moded level mapping $|\ |$ for P such that P is quasi recurrent wrt. $|\ |$.

Given an atom A, we denote with A^* an atom obtained from A by replacing the terms filling in its output positions with fresh distinct variables. Clearly, we have that A^* is simply-moded. Then we define the following moded level mapping for P:

$$|A| = nodes_P^{ic}(A^*).$$

Notice that, the level $|A|$ of an atom A is independent from the terms filling in its output positions, i.e. $|\ |$ is a moded level mapping. Moreover, since P is input terminating and A^* is simply-moded (in particular, it is nicely-moded), all the input-consuming derivations of $P \cup \{A^*\}$ are finite. Therefore, by Lemma 32, $nodes_P^{ic}(A^*)$ is defined (and finite), and thus $|A|$ is defined (and finite) for every atom A.

We now prove that P is quasi recurrent wrt. $|\ |$. Let $c : H \leftarrow \mathbf{A}, B, \mathbf{C}$ be a clause of P and $H\theta \leftarrow \mathbf{A}\theta, B\theta, \mathbf{C}\theta$ be an instance of c (for some substitution θ). We show that if $Rel(H) \simeq Rel(B)$ then $|H\theta| > |B\theta|$.

Let $H = p(\mathbf{s}, \mathbf{t})$. Hence, $(H\theta)^* = p(\mathbf{s}\theta, \mathbf{x})$ where $\mathbf{x}$ is a sequence of fresh distinct variables. Consider a variant $c' : H' \leftarrow \mathbf{A}', B', \mathbf{C}'$ of c variable disjoint from $(H\theta)^*$. Let ρ be a renaming such that $c' = c\rho$. Clearly, $(H\theta)^*$ and H' unify. Let $\mu = mgu((H\theta)^*, H') = mgu((H\theta)^*, H\rho) = mgu(p(\mathbf{s}\theta, \mathbf{x}), p(\mathbf{s}, \mathbf{t})\rho)$. By properties of substitutions (see (Apt, 1997)), since $\mathbf{x}$ consists of fresh variables, there exists two relevant mgu σ_1 and σ_2 such that

- $\sigma_1 = mgu(\mathbf{s}\theta, \mathbf{s}\rho)$,
- $\sigma_2 = mgu(\mathbf{x}\sigma_1, \mathbf{t}\rho\sigma_1)$.

Since $\mathbf{s}\rho \leqslant \mathbf{s}\theta$, we can assume that $Dom(\sigma_1) \subseteq Var(\mathbf{s}\rho)$. Because of standardization apart, since $\mathbf{x}$ consists of fresh variables, $\mathbf{x}\sigma_1 = \mathbf{x}$ and thus $\sigma_2 = mgu(\mathbf{x}, \mathbf{t}\rho\sigma_1)$. Since $\mathbf{x}$ is a sequence of variables, we can also assume that $Dom(\sigma_2) \subseteq Var(\mathbf{x})$. Therefore $Dom(\mu) \subseteq Var(Out((H\theta)^*)) \cup Var(In(H\rho))$. Moreover, since $(\mathbf{A}', B', \mathbf{C}')\mu = (\mathbf{A}, B, \mathbf{C})\rho\mu$, we have that

$$(H\theta)^* \stackrel{\mu}{\Longrightarrow} (\mathbf{A}, B, \mathbf{C})\rho\mu$$

is an input-consuming derivation step, i.e. $(\mathbf{A}, B, \mathbf{C})\rho\mu$ is a descendant of $(H\theta)^*$ in an IC-tree for $P \cup \{(H\theta)^*\}$.

By definition of μ, $\mathbf{s}\theta = \mathbf{s}\rho\mu$; hence

$$(\rho\mu)_{|In(H)} = \theta_{|\mathbf{s}}. \tag{10}$$

Let now $B = p(\mathbf{u}, \mathbf{v})$. By (10) and the hypothesis that c is input-recursive, that is $Var(In(B)) \subseteq Var(In(H)) = Var(\mathbf{s})$, it follows that

$$\mathbf{u}\rho\mu = \mathbf{u}(\rho\mu)_{|In(H)} = \mathbf{u}\theta_{|\mathbf{s}} = \mathbf{u}\theta. \tag{11}$$

Moreover, since c' is simply-moded, $In(H\rho) \cap Out(B\rho) = \emptyset$. Hence, by definition of μ and standardization apart, $Dom(\mu) \cap Out(B\rho) = \emptyset$, i.e.,

$$\mathbf{v}\rho\mu = \mathbf{v}\rho. \tag{12}$$

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Therefore, by (11) and (12), $B\rho\mu = p(\mathbf{u},\mathbf{v})\rho\mu = p(\mathbf{u}\theta,\mathbf{v}\rho) = (B\theta)^*$, i.e.,

$$B\rho\mu = (B\theta)^*. \tag{13}$$

Hence,

$$\begin{array}{lcll} |H\theta| & = & nodes_P^{ic}((H\theta)^*) & \text{by definition of } |\ | \\ & > & nodes_P^{ic}((\mathbf{A},B,\mathbf{C})\rho\mu) & \text{by Lemma 33 (i)} \\ & \geqslant & nodes_P^{ic}(B\rho\mu) & \text{by Lemma 33 (ii)} \\ & = & nodes_P^{ic}((B\theta)^*) & \text{by (13)} \\ & = & |B\theta| & \text{by definition of } |\ |. \end{array}$$

□

7 Applicability

This section is intended to show through some examples the applicability of our results. Then, programs from various well-known collections are analyzed.

7.1 Examples

It is worth noticing that, since the definition of input-consuming derivation is independent from the textual order of the atoms in the clause bodies, the results we have provided (Theorems 20, 24 and 34) hold also in the case that programs and queries are *permutation* nicely- (or simply-) moded (Smaus *et al.*, 1998), that is programs and queries which would be nicely- (or simply-) moded after a permutation of the atoms in the bodies. Therefore, for instance, we can apply Theorems 20 and 24 to the program `FLATTEN` as it is presented in Apt (1997) (except for the replacement of '\' with ','), i.e.

```
flatten(Xs,Ys) ← flatten_dl(Xs,Ys,[ ]).

flatten_dl([ ],Ys,Ys).
flatten_dl(X,[X|Xs],Xs) ← constant(X), X ≠ [ ].
flatten_dl([X|Xs],Ys,Zs) ← flatten_dl(X,Ys,Y1s),
                           flatten_dl(Xs,Y1s,Zs).
```

where the atoms in the body of the last clause are permuted with respect to the version of Example 25.

Let us consider again the program `APPEND` of the introduction with its natural delay declaration:

```
mode app(In,In,Out)

app([ ],Ys,Ys).
app([H|Xs],Ys,[H|Zs]) ← app(Xs,Ys,Zs).

delay app(Xs,_,_) until nonvar(Xs).
```

Let $\mathcal{Q}$ be the set of one-atom queries of the form `app(`s`,`t`,`Z`)` where s and t are any terms and Z is a variable disjoint from s and t. Observe that $\mathcal{Q}$ is closed

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

under resolution: Each resolvent in a derivation starting in a query from $\mathcal{Q}$ is still a query from $\mathcal{Q}$. Moreover, because of the presence of the delay declaration, only atoms whose first argument is a non-variable term are allowed to be selected. Thus, selectable atoms have the form `app(`s`,`t`,`Z`)` where

(1) s is a non-variable term,
(2) t is any term and Z is a variable disjoint from s and t.

Any derivation of `APPEND` starting in a query of $\mathcal{Q}$ is similar to an input-consuming one. This follows from the fact that for any selectable atom A and clause's head H, there exists a mgu θ which does not affect the input arguments of A. In fact, let A be a selectable atom of $\mathcal{Q}$. If A unifies with the head of the first clause then, by (1), s is the empty list `[ ]` and $\theta = mgu(A, H) = \{\texttt{Ys}/t, Z/t\}$. Otherwise, If A unifies with the head of the second clause then, by (1), s is a term of the form $[s_1 \mid s_2]$ and $\theta = mgu(A, H) = \{\texttt{H}/s_1, \texttt{Xs}/s_2, \texttt{Ys}/t, Z/[s_1|\texttt{Zs}]\}$. By (2) it follows that, in both cases, $s\theta = s$ and $t\theta = t$, i.e. θ does not affect the input arguments of A.

Moreover, it is easy to check that `APPEND` is quasi recurrent wrt. the moded level mapping depicted in Example 17. Since it is nicely-moded, by applying Theorem 20 it follows that it is input terminating. By the arguments above, we can conclude that all the derivations of `APPEND` in presence of the delay declaration d_1 and starting in a (permutation) nicely-moded query are finite. Hence, in particular, we can state that all the derivations of `APPEND` starting in the query Q_1 of the introduction, which is not nicely-moded but it is permutation nicely-moded, are finite.

7.2 Benchmarks

To assess the applicability of our results, we have looked into four collections of logic programs, and we have checked those programs against the three classes of programs: (permutation) nicely-moded, input terminating and quasi recurrent programs. The results are reported in Tables 1 to 4. These tables clearly show that our results apply to the large majority of the programs considered.

In Table 1 the programs from Apt's collection are considered (Apt, 1997). The programs from the DPPD's collection, maintained by Leuschel and available at the URL: http://dsse.ecs.soton.ac.uk/~mal/systems/dppd.html, are referred to in Table 2. Table 3 considers various programs from Lindenstrauss's collection (see the URL: `http://www.cs.huji.ac.il/~naomil`). Finally, in Table 4 one finds the (almost complete) list of programs by F. Bueno, M. Garcia de la Banda and M. Hermenegildo that can be found at the URL: `http://www.clip.dia.fi.upm.es`.

For each program we specify the name and the modes of the main procedure. Then we report whether or not the program is (permutation) nicely-moded (**NM**), input terminating (**IT**), and quasi recurrent (**QR**). Notice that for programs which are not input terminating, because of Theorem 20, it does not make sense to check whether or not they are quasi recurrent. For this reason, we leave blank the cells in the column **QR** corresponding to non-input terminating programs.

Finally, Table 5 reports the list of programs from previous tables which have been found to be input terminating but not quasi recurrent. For these programs,

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Table 1. *Programs from Apt's collection*

	NM	IT	QR		NM	IT	QR
app(In,_,_)	yes	yes	yes	ordered(In)	yes	yes	yes
app(_,_,In)	yes	yes	yes	overlap(_,In)	yes	yes	yes
app(Out,In,Out)	yes	no		overlap(In,Out)	yes	no	
append3(In,In,In,Out)	yes	yes	yes	perm(_,In)	yes	yes	yes
color_map(In,Out)	yes	no		perm(In,Out)	yes	no	
color_map(Out,In)	yes	no		qsort(In,_)	yes	yes	no
color_map(In,In)	yes	yes	yes	qsort(Out,In)	yes	no	
dcsolve(In,_)	yes	no		reverse(In,_)	yes	yes	yes
even(In)	yes	yes	yes	reverse(Out,In)	yes	no	
fold(In,In,Out)	yes	yes	yes	select(_,In,_)	yes	yes	yes
list(In)	yes	yes	yes	select(_,_,In)	yes	yes	yes
lte(In,_)	yes	yes	yes	select(In,Out,Out)	yes	no	
lte(_,In)	yes	yes	yes	subset(In,In)	yes	yes	yes
map(In,_)	yes	yes	yes	subset (In,Out)	yes	no	
map(_,In)	yes	yes	yes	subset (Out,In)	yes	no	
member(_,In)	yes	yes	yes	sum(_,In,_)	yes	yes	yes
member(In,Out)	yes	no		sum(_,_,In)	yes	yes	yes
mergesort(In,_)	yes	yes	no	sum(In,Out,Out)	yes	no	
mergesort(Out,In)	yes	no		type(In,In,Out)	no	yes	no
mergesort_variant(_,_,In)	yes	yes	yes	type(In,Out,Out)	no	no	

the notion of quasi recurrency does not provide an exact characterization of input termination. In particular, Theorem 34 does not apply. In order to understand which of the hypothesis of the theorem does not hold, we report in Table 5 whether or not these programs are simply-moded (**SM**) and input-recursive (**IR**).

8 Conclusion and related work

In this paper we studied the properties of input-consuming derivations of nicely-moded programs.

This study is motivated by the widespread use of programs using dynamic scheduling controlled by delay declarations. In fact, as we have motivated in section 3.2, we believe that in most practical programs employing delay declarations these constructs are used for guaranteeing that the derivation steps are input-consuming.

In the first place, we showed that for nicely-moded programs a weak version of the well-known switching lemma holds: If, given a query $(\mathbf{A}, B, \mathbf{C}, D, \mathbf{E})$, D is selected before B in an input-consuming derivation, then the two resolution steps can be interchanged while maintaining that the derivation is input-consuming.

Secondly, we presented a method for proving termination of programs and queries which are (permutation) nicely-moded. We also showed a result characterizing a class of input terminating programs.

In the literature, the paper most related to the present one is certainly that by Smaus (1999b). Our results strictly generalize those in (Smaus, 1999b) in the fact

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Table 2. *Programs from DPPD's collection*

	NM	IT	QR		NM	IT	QR
applast(In,In,Out)	yes	yes	yes	match_app(In,Out)	yes	no	
applast(Out,_,_)	yes	no		max_lenth(In,Out,Out)	yes	yes	yes
applast(_,Out,_)	yes	no		memo_solve(In,Out)	yes	yes	no
contains(_,In)	yes	yes	yes	power(In,In,In,Out)	yes	yes	yes
contains(In,Out)	yes	no		prune(In,_)	yes	yes	yes
depth(In,In)	yes	yes	yes	prune(_,In)	yes	yes	yes
depth(In,Out)	yes	yes	no	relative (In,_)	yes	no	
depth(Out,In)	yes	no		relative(_,In)	yes	no	
duplicate(In,Out)	yes	yes	yes	rev_acc(In,In,Out)	yes	yes	yes
duplicate(Out,In)	yes	yes	yes	rotate(In,_)	yes	yes	yes
flipflip(In,Out)	yes	yes	yes	rotate(_,In)	yes	yes	yes
flipflip(Out,In)	yes	yes	yes	solve(_,_,_)	yes	no	
generate(In,In,Out)	yes	no		ssupply(In,In,Out)	yes	yes	yes
liftsolve(In,Out)	yes	no		trace(In,In,Out)	yes	yes	yes
liftsolve(Out,In)	yes	no		transpose(_,In)	yes	yes	yes
liftsolve(In,In)	yes	yes	yes	transpose(In,Out)	yes	no	
match_app(_,In)	yes	yes	yes	unify(In,In,Out)	yes	no	

Table 3. *Programs from Lindenstrauss's collection*

	NM	IT	QR		NM	IT	QR
ack(In,In,_)	yes	yes	no	least(In,_)	yes	yes	yes
concatenate(In,_,_)	yes	yes	yes	least(_,In)	yes	yes	yes
concatenate(_,_,In)	yes	yes	yes	normal_form(In,_)	yes	no	
concatenate(_,In,_)	yes	no		normal_form(_,In)	yes	no	
descendant(In,_)	yes	no		queens(_,Out)	yes	yes	no
descendant(_,In)	yes	no		queens(_,In)	yes	yes	yes
deep(In,_)	yes	yes	yes	poss(In)	yes	yes	yes
deep(Out,_)	yes	no		poss(Out)	yes	no	
credit(In,_)	yes	yes	yes	rewrite(In,_)	yes	yes	yes
credit(_,In)	yes	yes	yes	rewrite(_,In)	yes	yes	yes
holds(_,Out)	yes	no		transform(_,_,_,Out)	yes	no	
holds(_,In)	yes	yes	yes	transform(_,_,_, In)	yes	yes	yes
huffman(In,_)	yes	yes	no	twoleast(In,_)	yes	yes	yes
huffman(_,In)	yes	no		twoleast(_,In)	yes	yes	yes

that we drop the condition that programs and queries have to be well-moded. This is particularly important in the formulation of the queries. For instance, in the program `FLATTEN` of Example 25, our results show that every input-consuming derivation starting in a query of the form `flatten`(t, s) terminates provided that t is linear and disjoint from s, while the results of Smaus (1999b) apply only if t is a ground term. Note that well-moded queries (in well-moded programs) never

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Table 4. *Programs from Hermenegildo's collection*

		NM	IT	QR
aiakl.pl	init_vars(In,In,Out,Out)	yes	yes	yes
ann.pl	analyze_all(In,Out)	yes	yes	yes
bid.pl	bid(In,Out,Out,Out)	yes	yes	yes
boyer.pl	tautology(In)	yes	no	
browse.pl	investigate(In,Out)	yes	yes	yes
fib.pl	fib(In,Out)	yes	no	
fib_add.pl	fib(In,Out)	yes	yes	yes
hanoiapp.pl	shanoi(In,In,In,In,Out)	yes	no	
hanoiapp_suc.pl	shanoi(In,In,In,In,Out)	yes	yes	yes
mmatrix.pl	mmultiply(In,In,Out)	yes	yes	yes
occur.pl	occurall(In,In,Out)	yes	yes	yes
peephole.pl	peephole_opt(In,Out)	yes	yes	yes
progeom.pl	pds(In,Out)	yes	yes	yes
rdtok.pl	read_tokens(In,Out)	yes	no	
read.pl	parse(In,Out)	yes	no	
serialize.pl	serialize(In,Out)	yes	yes	no
tak.pl	tak(In,In,In,Out)	yes	no	
tictactoe.pl	play(In)	yes	no	
warplan.pl	plans(In,In)	yes	no	

Table 5. *Input terminating but non-quasi recurrent programs*

	SM	IR
mergesort(In,_)	yes	no
qsort(In,_)	yes	no
type(In,In,Out)	no	no
depth(In,Out)	yes	no
memo_solve(In,Out)	no	no
ack(In,In,_)	yes	no
huffman(In,_)	no	no
queens(_,Out)	no	no
serialize(In,Out)	no	no

terminate by deadlock, since the leftmost atom of each resolvent is ground in its input positions and hence selectable. This does not hold for nicely-moded queries which might deadlock. Our method thus allows us to cope also with this more difficult situation: For instance we can prove that all derivations of `APPEND` starting in `app(X,Y,Z)` are terminating. In practice the result of Smaus (1999b) identify a class of programs and queries which is both terminating and *deadlock free*. While deadlock is clearly an undesirable situation, there are various reasons why one might want to prove termination independently from the absence of deadlock: in the first

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

place, one might want to prove absence of deadlock using a different tool than by employing well-moded programs. Secondly, in some situations absence of deadlock might be difficult or impossible to prove, like in a modular context in which the code of some module is not known, hence not analyzable: consider for instance the query `generator_1(X1s), generator_2(X2s), append(X1s,X2s,Zs).`, where the generators are defined in different modules; our results allow us to demonstrate that if the `generators` terminate, then the above query terminates. On the other hand, one cannot determine whether it is deadlock free unless one has a more precise specification of the generators. Thirdly, it is well-known that one of the goals of dynamic scheduling is precisely enforcing termination; in this respect a deadlock can be regarded as the situation in which 'all else failed'. Our system allows us to check how effective dynamic scheduling is in enforcing termination.

Concluding our comparison with Smaus (1999b), for the class of (permutation) simply-moded and input-recursive programs, we provide an exact characterization of input termination. A similar result is not present in Smaus (1999b).

Apt & Luitjes (1995) have also tackled the problem of the termination of programs in presence of dynamic scheduling. The techniques employed in it are based on determinacy checks and the presence of successful derivations, thus are completely different from ours. It is nevertheless worth mentioning that Apt & Luitjes (1995) report a special *ad hoc* theorem to prove that, if u is linear and disjoint from s then the query $\mathtt{app}(s, t, u)$ terminates. This is reported in order to show the difficulties one encounters in proving termination in presence of dynamic scheduling. Now, under the further (mild) additional condition that u be disjoint from t, the termination of $\mathtt{app}(s, t, u)$ is a direct consequence of our main result.

Another related paper is the one by Marchiori & Teusink (1999). However, Marchiori and Teusink make a strong restriction on the selection rule, which has to be *local*; this restriction actually forbids any form of coroutining. Moreover, (Marchiori & Teusink, 1999) allows only *safe* delay declarations; we do not report here the definition of *safe* delay declaration, we just say that it is rather restrictive: For instance, the delay declaration we have used for `APPEND` is not safe (a safe one would be `delay app(X,_,_) until list(X)`). Actually, their requirements go beyond ensuring that derivations are input-consuming.

Applicability and effectiveness of our results have been demonstrated by matching our main definitions against the programs of four public program lists. These benchmarks showed that most of the considered programs are nicely-moded (for a suitable mode) and quasi recurrent (wrt. a suitable level mapping).

References

Apt, K. R. (1990) Introduction to Logic Programming. In: van Leeuwen, J. (editor), *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, pp. 495–574. Elsevier/MIT Press.

Apt, K. R. (1997) *From Logic Programming to Prolog*. Prentice Hall.

Apt, K. R. and Bezem, M. (1991) Acyclic programs. *New Generation Computing*, **9** (3&4), 335–363.

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Apt, K. R. and Luitjes, I. (1995) Verification of logic programs with delay declarations. In: Borzyszkowski, A. and Sokolowski, S. (editors), *Proc. 4th International Conference on Algebraic Methodology and Software Technology, (AMAST'95): Lecture Notes in Computer Science 936*, pp. 66–90. Springer-Verlag.

Apt, K. R. and Marchiori, E. (1994) Reasoning about Prolog programs: from Modes through Types to Assertions. *Formal Aspects of Computing*, **6** (6A), 743–765.

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

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

Apt, K. R. and Pellegrini, A. (1994) On the occur-check free Prolog programs. *ACM Toplas*, **16** (3), 687–726.

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

Cavedon, L. (1989) Continuity, consistency and completeness properties for logic programs. In: Levi, G. and Martelli, M. (editors), *Proc. 6th International Conference on Logic Programming*, pp. 571–584. MIT Press.

Cavedon, L. (1991) Acyclic programs and the completeness of SLDNF-resolution. *Theoretical Computer Science*, **86** (1), 81–92.

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

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

Falaschi, M., Gabbrielli, M., Marriott, K. and Palamidessi, C. (1997) Constraint logic programming with dynamic scheduling: a semantics based on closure operators. *Information & Computation*, **137** (1), 41–67.

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

Intelligent Systems Laboratory (1997) *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, P.O. Box 1263, S-164 29 Kista, Sweden. `http://www.sics.se/isl/sicstus/sicstus_toc.html`.

Lloyd, J. W. (1987) *Foundations of Logic Programming*. Springer-Verlag.

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

Marriott, K. (1997) Algebraic and logical semantics for CLP languages with dynamic scheduling. *J. Logic Programming*, **32** (1), 71–84.

Naish, L. (1988) Parallelizing NU-Prolog. In: Bowen, K. A. and Kowalski, R. A. (editors), *Proc. 5th International Conference and Symposium on Logic Programming*, pp. 1546–1564. MIT Press.

Naish, L. (1993) Coroutining and the construction of terminating logic programs. *Australian Computer Science Communications*, **15** (1), 181–190.

Smaus, J.-G. (1999a) *Modes and Types in Logic Programming*. PhD thesis, University of Kent at Canterbury. `http://www.cs.ukc.ac.uk/people/staff/jgs5/thesis.ps`.

Smaus, J.-G. (1999b) Proving termination of input-consuming logic programs. In: De Schreye, D. (editor), *Proc. 16th International Conference on Logic Programming*, pp. 335–349. MIT Press.

Smaus, J.-G., Hill, P. M. and King, A. M. (1998) Termination of logic programs with `block` declarations running in several modes. In: Palamidessi, C. (editor), *Proc. 10th Symposium on*

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press

Programming Language Implementations and Logic Programming: Lecture Notes in Computer Science 1490, pp. 73–88. Springer-Verlag.

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

Ueda, K. (1988) GHC - A language for a new age of parallel programming. In K. V. Nori and S. Kumar (editors), *Proc. 8th Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, 338*, pp. 364–376. Springer-Verlag.

Ueda, K. and Morita, M. (1994) Moded Flat GHC and its message-oriented implementation technique. *New Generation Computing*, **13** (1), 3–43.

Wallace M. G., Novello, S. and Schimpf, J. (1997) ECLiPSe : A platform for constraint logic programming. *ICL Systems J.*, **12** (1).

https://doi.org/10.1017/S1471068401001326 Published online by Cambridge University Press