Properties of Input-Consuming Derivations

ANNALISA BOSSI, SABINA ROSSI
Dipartimento di Informatica, Università di Venezia
via Torino 155, 30172 Venezia, Italy

SANDRO ETALLE
Department of Computer Science, University of Maastricht
P.O. Box 616, 6200 MD Maastricht, The Netherlands
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.

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; Hill and 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

\[
\text{app}([\ ], Ys, Ys).
\]
\[
\text{app}([H|Xs], Ys, [H|Zs]) \leftarrow \text{app}(Xs, Ys, Zs).
\]

together with the query

\[
Q_1 := \text{app}(Xs, [5, 6], Ys), \text{app}([1, 2], [3, 4], Xs).
\]

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 $\text{app}([1,2],[3,4],[\ ])$.

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 $\text{app}(\text{Xs}, [5,6], \text{Ys})$, 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 $\text{Xs}$ 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

$$Q_2 := \text{app}([1,2],[3,4],\text{Xs}), \text{app}(\text{Xs},[5,6],\text{Ys}).$$

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. Modeled Flat GHC (Ueda and 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 and 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

$$d_1 := \text{delay app(Ls, , ) until nonvar(Ls)}.$$

This declaration forbids the selection of an atom of the form $\text{app}(s, t, u)$ 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 and Luitjes (1995), Marchiori and Teusink (1999) and Smaus (1999b). We compare our results with the ones in (Apt and Luitjes, 1995; Marchiori and Teusink, 1999; 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.

In order 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 \texttt{APPEND} reported above, when used for concatenating two lists, is \texttt{app(In,In,Out)}. Notice that in this case the delay declaration \texttt{d\(_1\)} serves precisely the purpose of guaranteeing that if an atom of the form \texttt{app(s,t,X)} (with \texttt{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 and Pellegrini, 1994; Apt and Marchiori, 1994), or more simply, the tables of programs we report in Section 7, or consider for instance the logic programming language Mercury (Somogyi \textit{et al.}, 1996), which requires that its programs are moded (and well-moded).

\textbf{Contributions of this paper}

In this paper we study some properties of input-consuming derivations.

In the first place we 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 and Pedreschi, 1994)). We show that if \texttt{P} is nicely-moded and quasi recurrent then all its input-consuming derivations starting from a nicely-moded query terminate.

Furthermore, we demonstrate that under mild additional constraints (namely,

\(^1\) 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.
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 \texttt{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; Apt, 1997; Lloyd, 1987).

2.1 Terms and Substitutions

Let $\mathcal{T}$ be the set of terms built on a finite set of data constructors $\mathcal{C}$ and a denumerable set of variable symbols $\mathcal{V}$. A substitution $\theta$ is a mapping from $\mathcal{V}$ to $\mathcal{T}$ such that $\text{Dom}(\theta) = \{X \mid \theta(X) \neq X\}$ is finite. For any syntactic object $o$, we denote by $\text{Var}(o)$ the set of variables occurring in $o$. A syntactic object is linear if every variable occurs in it at most once. We denote by $\epsilon$ the empty substitution. The composition $\theta \sigma$ of the substitutions $\theta$ and $\sigma$ is defined as the functional composition, i.e., $\theta \sigma(X) = \sigma(\theta(X))$. We consider the pre-ordering $\leq$ (more general than) on substitutions such that $\theta \leq \sigma$ if there exists $\gamma$ such that $\theta \gamma = \sigma$. The result of the application of a substitution $\theta$ to a term $t$ is said an instance of $t$ and it is denoted by $t\theta$. We also consider the pre-ordering $\leq$ (more general than) on terms such that $t \leq t'$ iff there exists $\theta$ such that $t\theta = t'$. We denote by $\approx$ the associated
equivalence relation (variance). A substitution $\theta$ 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 $\theta$ of terms $t$ and $t'$ is called relevant iff $\text{Var}(\theta) \subseteq \text{Var}(t) \cup \text{Var}(t')$.

2.2 Programs and Derivations

Let $\mathcal{P}$ be a finite set of predicate symbols. An atom is an object of the form $p(t_1, \ldots, t_n)$ where $p \in \mathcal{P}$ is an $n$-ary predicate symbol and $t_1, \ldots, t_n \in T$. Given an atom $A$, we denote by $\text{Rel}(A)$ 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 $H \leftarrow B$ where $H$ is an atom (the head) and $B$ is a query (the body). When $B$ is empty, $H \leftarrow B$ 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 $Q, A, B, C, \ldots$, clauses by $c, d, \ldots$, and programs by $P$.

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

$$A, B, C \xrightarrow{\theta} P,c (A, B, C)\theta$$

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

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

$$\delta := Q_0 \xrightarrow{\theta_1} P,c_1 Q_1 \xrightarrow{\theta_2} P,c_2 \cdots \xrightarrow{\theta_n} P,c_n Q_n \xrightarrow{\theta_{n+1}} P,c_{n+1} Q_{n+1} \cdots$$

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

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

**Definition 1 (B-step)**

Let $A, B, C \xrightarrow{\theta} (A, B, C)\theta$ be a derivation step. We say that each atom in $B\theta$ is a direct descendant of $B$, and for each atom $E$ in $(A, C)$, $E\theta$ is a direct descendant of $E$. We say that $E$ is a descendant of $F$ if the pair $(E, F)$ is in the reflexive, transitive closure of the relation is a direct descendant of. Consider a derivation
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 \( \{\text{In}, \text{Out}\} \).

If \( m_p(i) = \text{In} \) (resp. \( \text{Out} \)), 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 \( \text{In}(Q) \) (resp. \( \text{Out}(Q) \)) the sequence of terms filling in the input (resp. output) positions of predicates in \( Q \). Moreover, when writing an atom as \( p(s, t) \), we are indicating with \( s \) the sequence of terms filling in the input positions of \( p \) and with \( t \) 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 \( p(s, t) \) is called input-consuming resolvable wrt. a clause \( c := p(u, v) \leftarrow Q \) and a substitution \( \theta \) iff \( \theta = \text{mgu}(p(s, t), p(u, v)) \) and \( s = s\theta \).
- A derivation step

\[
A, B, C \xrightarrow{\theta, c} (A, B, C)\theta
\]

is called input-consuming iff the selected atom \( B \) is input-consuming resolvable wrt. the input clause \( c \) and the substitution \( \theta \).
- 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 \( p(s, t) \) and \( p(u, v) \) be two atoms. If there exists an mgu \( \theta \) of \( p(s, t) \) and \( p(u, v) \) such that \( s\theta = s \), then there exists a relevant mgu \( \bar{\theta} \) of \( p(s, t) \) and \( p(u, v) \) such that \( s\bar{\theta} = s \).
Proof

Since \( p(s,t) \) and \( p(u,v) \) are unifiable, there exists a relevant mgu \( \theta_{rel} \) of them (cfr. (Apt, 1997), Theorem 2.16). Now, \( \theta_{rel} \) is a renaming of \( \theta \). Thus \( s\theta_{rel} \) is a variant of \( s \). Then there exists a renaming \( \rho \) such that \( \text{Dom}(\rho) \subseteq \text{Var}(s,t,u,v) \) and \( s\theta_{rel}\rho = s \). 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 \textsc{reverse} with accumulator in the modes defined below.

\begin{verbatim}
  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]).
\end{verbatim}

The derivation \( \delta \) of \textsc{reverse} \cup \{ \text{reverse([X1,X2],Zs)} \} depicted below is input-consuming.

\begin{verbatim}
\delta := reverse([X1,X2],Zs) \Rightarrow reverse_acc([X1,X2],Zs,[ ]) \Rightarrow
                reverse_acc([X2],Zs,[X1]) \Rightarrow reverse_acc([],Zs,[X2,X1]) \Rightarrow \square .
\end{verbatim}

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) and (Falaschi \textit{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...
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. In Chapter 7 of (Smaus, 1999a), Smaus 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 $Q := p_1(s_1, t_1), \ldots, p_n(s_n, t_n)$ is nicely-moded if $t_1, \ldots, t_n$ is a linear sequence of terms and for all $i \in \{1, \ldots, n\}$
  \[
  \text{Var}(s_i) \cap \bigcup_{j=1}^{n} \text{Var}(t_j) = \emptyset.
  \]

- A clause $c = p(s_0, t_0) ← Q$ is nicely-moded if $Q$ is nicely-moded and
  \[
  \text{Var}(s_0) \cap \bigcup_{j=1}^{n} \text{Var}(t_j) = \emptyset.
  \]

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

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

**Example 7**
- The program APPEND in the modes $\text{app}(\text{In}, \text{In}, \text{Out})$ 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(\text{In}, \text{In}, \text{Out}).
merge(\text{Xs}, [\ ], \text{Xs}).
merge([\ ], \text{Xs}, \text{Xs}).
merge([X|\text{Xs}], [Y|\text{Ys}], [Y|\text{Zs}]) \leftarrow Y < X, \text{merge}([X|\text{Xs}], Ys, Zs).
merge([X|\text{Xs}], [Y|\text{Ys}], [X|\text{Zs}]) \leftarrow Y > X, \text{merge}(\text{Xs}, [Y|\text{Ys}], Zs).
merge([X|\text{Xs}], [X|\text{Ys}], [X|\text{Zs}]) \leftarrow \text{merge}(\text{Xs}, [X|\text{Ys}], Zs).
```

The following result is due to Smaus (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 $\text{Var}(Q) \cap \text{Var}(c) = \emptyset$, 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 $Q := A, p(s, t), C$ be nicely-moded.
If $A, p(s, t), C \xrightarrow{\theta} (A, B, C)\theta$ is an input-consuming derivation step with selected atom $p(s, t)$, then $A\theta = A$.

### 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 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-modeled. Let $\delta := Q \xrightarrow{\theta} Q'$ be a partial input-consuming derivation of $P \cup \{Q\}$. Then, for all $x \in \text{Var}(Q)$ and $x \not\in \text{Var}(\text{Out}(Q))$, $x\theta = x$.

Proof
Let us first establish the following claim.

Claim 11
Let $z$ and $w$ be two variable disjoint sequences of terms such that $w$ is linear and $\theta = \text{mgu}(z, w)$. If $s_1$ and $s_2$ are two variable disjoint terms occurring in $z$ 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 and Pedreschi, 1994).

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

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

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

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

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

Let $x \in \text{Var}(A, p(s, t), C)$ and $x \not\in \text{Var}(\text{Out}(A, p(s, t), C))$. We first show that

$$x\theta_1 = x \quad (1)$$

We distinguish two cases.

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

(b) $x \not\in \text{Var}(s)$. Since $x \in \text{Var}(A, p(s, t), C)$, by standardization apart, we have that $x \not\in \text{Var}(p(u, v))$. Moreover, since $x \not\in \text{Var}(\text{Out}(A, p(s, t), C))$, it also holds that $x \not\in \text{Var}(p(s, t))$. Then, property (1) follows from relevance of $\theta_1$.

Now we show that

$$x\theta_2 = x \quad (2)$$

Again, we distinguish two cases:

(c) $x \not\in \text{Var}((A, B, C)\theta_1)$. In this case, because of the standardization apart condition, $x$ will never occur in $(A, B, C)\theta_1 \xrightarrow{\theta_2} Q'$. Hence, $x \not\in \text{Dom}(\theta_2)$ and $x\theta_2 = x$.

(d) $x \in \text{Var}((A, B, C)\theta_1)$. In this case, in order to prove (2) we show that $x \not\in \text{Var}(\text{Out}((A, B, C)\theta_1))$. The result then follows by the inductive hypothesis.
where

be a partial input-consuming derivation of $P$. Let the program

Corollary 12

definition of nicely-moded program.

$x$ proves that

Hence, $\text{Var}(\text{Out}(A, C))\theta_1 = \text{Var}(\text{Out}(A, C))$. Since $x \notin \text{Var}(\text{Out}(A, C))$, this proves that $x \notin \text{Var}(\text{Out}(A, C))$.

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

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

(d2) $x \in \text{Var}(s)$. By known results (see (Apt, 1997), Corollary 2.25), there exists two relevant mgu $\sigma_1$ and $\sigma_2$ such that

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

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

$$\text{Var}(u\sigma_1) \cap \text{Var}(\text{Out}(B\sigma_1)) = \emptyset$$  \hfill (3)

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

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

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

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 $\delta := A \xrightarrow{\theta} Q'$ be a partial input-consuming derivation of $P \cup \{A\}$. Then, for all $x \in \text{Var}(\text{In}(A))$, $x\theta = x$.

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 $\delta$ be a partial input-consuming derivation of $P \cup \{Q_0\}$ of the form

$$\delta := Q_0 \xrightarrow{\theta_1} Q_1 \cdots Q_n \xrightarrow{\theta_{n+1}} Q_{n+1} \xrightarrow{\theta_{n+2}} Q_{n+2}$$

where
then, there exist $Q'_{n+1}$, $\theta'_{n+1}$, $\theta'_{n+2}$ and a derivation $\delta'$ such that

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

and

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

where $\delta'$ is input-consuming and

- $\delta$ and $\delta'$ 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. $D\theta_{n+1}$,
- $\delta$ and $\delta'$ coincide after the resolvent $Q_{n+2}$.

**Proof**

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

$$u \theta_{n+1} = u,$$

since $\delta$ is input-consuming. (5)

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

and

$$s \theta_{n+2} = s,$$

since $\delta$ is input-consuming. (6)

Moreover,$^2$

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

where

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

We construct the derivation $\delta'$ as follows.

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

where

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

(8)

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

$$u \theta'_{n+1} \theta_{n+2} = u \theta_{n+2} \theta'_{n+2}, \quad \text{(by (8))}$$

$$= u \theta'_{n+1} \theta_{n+2}, \quad \text{(by (7))}$$

$$= u \theta'_{n+2}, \quad \text{(by (5))}$$

$$= u \theta'_{n+1}, \quad \text{(by (8))}.$$

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

$^2$ We use the notation $\text{mgu}(E)$ to denote the mgu of a set of equations $E$, see (Apt, 1997).
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 $Q := A, B$ be nicely-moded. Suppose that

$$
\delta := A, B \mapsto^\theta C_1, C_2
$$

that is a partial input-consuming derivation of $P \cup \{Q\}$ where $C_1$ and $C_2$ are obtained by partially resolving $A$ and $B$, respectively. Then there exists a partial input-consuming derivation

$$
\delta' := A, B \mapsto_{\theta_1} C_1, B \mapsto_{\theta_2} C_1, C_2
$$

where all the $A$-steps are performed in the prefix $A, B \mapsto_{\theta_1} C_1, B \mapsto_{\theta_2} C_1, C_2$ 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 in order 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 (Mode Level Mapping)**

Let $P$ be a program and $B^e_P$ be the extended Herbrand base 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.
• it is a function $|\cdot| : \mathcal{B}_p^\pi \rightarrow \mathbb{N}$ from atoms to natural numbers;
• for any $t$ and $u$, $|p(s, t)| = |p(s, u)|$.

For $A \in \mathcal{B}_p^\pi$, $|A|$ is the level of $A$.

The condition $|p(s, t)| = |p(s, u)|$ 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 the one in (Etalle et al., 1999): In (Etalle et al., 1999) 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 $\text{TSize}(t)$ 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:

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

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

$$|\text{reverse}(x_s, y_s)| = \text{TSize}(x_s)$$
$$|\text{reverse, acc}(x_s, y_s, z_s)| = \text{TSize}(x_s).$$

### 5.2 Quasi Recurrency

In order 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; Apt and 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 and 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. In order 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, $p \approx q \equiv p \subseteq q \land p \sqsupseteq q$ means that $p$ and $q$ are mutually recursive, and $p \sqsupset q \equiv p \sqsupseteq q \land p \not\approx q$ 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.
Definition 19 (Quasi Recurrency)
Let $P$ be a program and $|: \mathbb{B}_P^f \rightarrow \mathbb{N}$ be a moded level mapping.

- A clause of $P$ is called quasi recurrent with respect to $|$ if for every instance of it, $H \leftarrow A, B, C$
  
  \[
  \text{if } \text{Rel}(H) \simeq \text{Rel}(B) \text{ then } |H| > |B|. \tag{9}
  \]

- 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 $|: \mathbb{B}_P^f \rightarrow \mathbb{N}$.

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 $|H| > |B|$ (or $|H| \geq |B|$ 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 and Pedreschi, 1994) or cover as in (Marchiori and Teusink, 1999).

It is worthwhile 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 and 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 and 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 the one of 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 $R = \emptyset$. \qed

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

\[
|\text{merge}(x_s, y_s, z_s)| = \text{TSize}(x_s) + \text{TSize}(y_s).
\]
It is easy to prove that \textsc{merge} is quasi recurrent wrt. the moded level mapping above. By Theorem 20, all input-consuming derivations of \textsc{merge} started with a query \textsc{merge}(s,t,u), where \textit{u} is linear and variable disjoint from \textit{s} and \textit{t}, are terminating.

### 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 \(\delta\) of \(P \cup \{Q\}\). Then, there exist an index \(i \in \{1, \ldots, n\}\) and substitution \(\theta\) such that

1. there exists an input-consuming derivation \(\delta'\) of \(P \cup \{Q\}\) of the form
   \[\delta' := A_1, \ldots, A_n \xrightarrow{\theta} C, (A_1, \ldots, A_n)\theta \quad \cdots\]

2. there exists an infinite input-consuming derivation of \(P \cup \{A, \theta\}\).

**Proof**

Let \(\delta := A_1, \ldots, A_n \quad \cdots\) be an infinite input-consuming derivation of \(P \cup \{Q\}\). Then \(\delta\) contains an infinite number of \(A_j\)-steps for some \(k \in \{1, \ldots, n\}\). Let \(i\) be the minimum of such \(k\). Hence \(\delta\) contains a finite number of \(A_j\)-steps for \(j \in \{1, \ldots, i - 1\}\) and there exists \(C\) and \(D\) such that

\[\delta := A_1, \ldots, A_n \xrightarrow{\theta} C, D \quad \cdots\]

where \(A_1, \ldots, A_n \xrightarrow{\theta} C, D\) is a finite prefix of \(\delta\) which comprises all the \(A_j\)-steps of \(\delta\) for \(j \in \{1, \ldots, i - 1\}\) and \(C\) is the subquery of \(C, D\) 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 \(\delta'\) such that

\[\delta' := A_1, \ldots, A_n \xrightarrow{\theta} C, (A_1, \ldots, A_n)\theta \xrightarrow{\theta'} C, D \quad \cdots\]

where \(\theta = \theta'\). This proves (i).

Now, let \(\delta'' := C, (A_1, \ldots, A_n)\theta \xrightarrow{\theta'} C, D \quad \cdots\). Note that in \(\delta''\) the atoms of \(C\) will never be selected and, by Remark 9, will never be instantiated. Let \(\delta''\) be obtained from \(\delta''\) by omitting the prefix \(C\) in each query. Hence \(\delta''\) is an infinite input-consuming derivation of \(P \cup \{A_1, \ldots, A_n\}\theta\) where an infinite number of \(A_i\theta\)-steps are performed. Again, By Remark 9, for every finite prefix of \(\delta''\) of the form

\[A_1\theta, (A_{i+1}, \ldots, A_n)\theta \xrightarrow{\sigma_1} D_1, D_2 \xrightarrow{\sigma_2} D_1', D_2'\]

where \(D_1\) and \(D_2\) are obtained by partially resolving \(A_i\theta\) and \((A_{i+1}, \ldots, A_n)\theta\), respectively, and \(D_1, D_2 \xrightarrow{\sigma_2} D_1', D_2'\) is an \(A_j\)-step for some \(j \in \{i+1, \ldots, n\}\), we have that \(D_1' = D_1\). Hence, from the hypothesis that there is an infinite number of \(A_i\theta\)-steps in \(\delta''\), it follows that there exists an infinite input-consuming derivation of \(P \cup \{A, \theta\}\). This proves (ii). \(\Box\)
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 if for each nicely-moded one-atom query $A$ all input-consuming derivations of $P \cup \{ A \}$ are finite.

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 $| | : B_e^P \rightarrow N$.

Then $P \cup R$ is input terminating.

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

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 $\delta$ is an infinite input-consuming derivation of $(P \cup R) \cup \{ A \}$ such that $A$ is defined in $P$. Then

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

where $H \leftarrow B_1, \ldots, B_n$ is the input clause used in the first derivation step and $\theta_1 = \text{mgu}(A, H)$. 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 $\theta_2$,

1. there exists an infinite input-consuming derivation of $(P \cup R) \cup \{ A \}$ of the form
   $$A \xrightarrow{\theta_1} (B_1, \ldots, B_n) \theta_1 \xrightarrow{\theta_2} \cdots$$

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 $\langle \text{dep}_P(\text{Rel}(A)), |A| \rangle$.
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 \( \text{dep}_P(\text{Rel}(A)) = 0 \) (\( |A| \) 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.

**Induction step.** We distinguish two cases:

1. \( \text{Rel}(H) \sqsupseteq \text{Rel}(B_i) \),
2. \( \text{Rel}(H) \simeq \text{Rel}(B_i) \).

In case (a) we have that \( \text{dep}_P(\text{Rel}(A)) = \text{dep}_P(\text{Rel}(H\theta)) > \text{dep}_P(\text{Rel}(B_i\theta)) \). So, \( \langle \text{dep}_P(\text{Rel}(A)), |A| \rangle = \langle \text{dep}_P(\text{Rel}(H\theta)), |H\theta| \rangle > \langle \text{dep}_P(\text{Rel}(B_i\theta)), |B_i\theta| \rangle \). 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 \( A \xrightarrow{\theta} C, (B_i, \ldots, B_n)\theta \). By Corollary 12 and the fact that \( | | \) is a moded level mapping, it follows that \( |A| = |A\theta| = |H\theta| \). Therefore, \( \langle \text{dep}_P(\text{Rel}(A)), |A| \rangle = \langle \text{dep}_P(\text{Rel}(H\theta)), |H\theta| \rangle > \langle \text{dep}_P(\text{Rel}(B_i\theta)), |B_i\theta| \rangle \). In both cases, the contradiction follows by the inductive hypothesis. \(\Box\)

**Example 25**
The program \textsc{Flatten} using difference-lists is nicely-modeled with respect to the modes described below, provided that one replaces “\( \\backslash \)” by “,”, as we have done here.

\[
\begin{align*}
\text{mode flatten}(\text{In}, \text{Out}). \\
\text{mode flatten}_\text{dl}(\text{In}, \text{Out}, \text{In}). \\
\text{mode constant}(\text{In}). \\
\text{mode } \neq (\text{In}, \text{In}). \\
\text{flatten}(\text{Xs}, \text{Ys}) \leftarrow \text{flatten}_\text{dl}(\text{Xs}, \text{Ys}, [ ]). \\
\text{flatten}_\text{dl}([ ], \text{Ys}, \text{Ys}). \\
\text{flatten}_\text{dl}(\text{X}[\text{Xs}], \text{Ys}) \leftarrow \text{constant}(	ext{X}), \text{X} \neq [ ]. \\
\text{flatten}_\text{dl}([\text{X}][\text{Xs}], \text{Ys}, \text{Zs}) \leftarrow \text{flatten}_\text{dl}(\text{Xs}, \text{Ys}, \text{Zs}), \\
\text{flatten}_\text{dl}(\text{X}, \text{Ys}, \text{Ys}).
\end{align*}
\]

Consider the moded level mapping for \textsc{Flatten} defined by

\[
\begin{align*}
|\text{flatten}(x_s, y_s)| &= T\text{Size}(x_s) \\
|\text{flatten}_\text{dl}(x_s, y_s, z_s)| &= T\text{Size}(x_s).
\end{align*}
\]

It is easy to see that the program \textsc{Flatten} is quasi recurrent wrt. the moded level mapping above. Hence, all input-consuming derivations of program \textsc{Flatten} started with a query \text{flatten}(s, t), where \( t \) is linear and variable disjoint from \( s \), are terminating.
Theorem 20 provides a sufficient condition for termination. The condition is not necessary, as demonstrated by the following simple example.

\begin{verbatim}
mode p(In,Out).
p(X,a) ← p(X,b).
p(X,b).
\end{verbatim}

This program is clearly input terminating, however it is not quasi recurrent. If it was, we would have that $|p(X,a)| > |p(X,b)|$, for some moded level mapping $| |$ (otherwise 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 works, e.g., (Bezem, 1993; Apt and 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, in order 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 $c = H ← Q$) 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 and Pedreschi, 1993)) and that often non simply-moded programs can naturally be transformed into simply-moded ones.

**Example 27**

- The programs \textsc{reverse} of Example 5, \textsc{merge} of Example 7 and \textsc{flatten} of Example 25 are all simply-moded.
- Consider the program \textsc{last} which extends \textsc{reverse}:

\begin{verbatim}
mode last(In,Out).
last(Ls,E) ← reverse(Ls,[E| ]).
\end{verbatim}

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 and 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 \( \text{Var}(Q) \cap \text{Var}(c) = \emptyset \), is simply-moded.

### 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 \textsc{Quicksort}:

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

\[ c_1 := \]

\[ qs([\ ],[\ ]), \]
\[ qs([X|Xs],Ys) ← \text{part}(X,Xs,Littles,Bigs), \]
\[ qs(Littles,\text{Ls}), \]
\[ qs(Bigs,\text{Bs}), \]
\[ \text{app}((Ls,[X|Bs],Ys). \]
\[ \text{part}(X,[\ ],[\ ],[\ ]), \]
\[ \text{part}(X,[Y|Xs],[Y|\text{Ls}],B) ← X>Y, \text{part}(X,Xs,\text{Ls},B). \]
\[ \text{part}(X,[Y|Xs],[Ls,[Y|Bs]) ← X=Y, \text{part}(X,Xs,\text{Ls},B). \]

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, \( |qs([X|Xs],Ys)| > |qs(Littles,\text{Ls})| \) and \( |qs([X|Xs],Ys)| > |qs(Bigs,\text{Bs})|. \)

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 \textit{input-recursive} programs.

**Definition 29 (Input-Recursive)**

Let \( P \) be a program.

- A clause \( H ← A, B, C \) of \( P \) is called \textit{input-recursive} if

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

\(^4\) Provided that one models the built-in predicates \( > \) and \( \leq \) as being defined by (an infinite number of) ground facts of the form \( >(m,n) \) and \( \leq(m,n) \). The problem here is that the definition of input-consuming derivation does not consider the presence of built-ins.
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 $Y_1$s in a body atom of the last clause.
- **QUICKSORT**, is not input-recursive. In particular, clause $c_1$ is not input-recursive.

### 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 and Pedreschi, 1994). First we introduce the notion of IC-tree that corresponds to the notion of S-tree in (Apt and 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 $\text{nodes}^i_{IC}(P)$ 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 $T$, $nodes_{P}^{Q'}(Q') < nodes_{P}^{Q}(Q)$,
(ii) for all atoms $A$ of $Q$, $nodes_{P}^{Q}(A) \leq nodes_{P}^{Q}(Q)$.

Proof

Immediate by Definition 31 of IC-tree. \qed

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.

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 $P$:

$$|A| = nodes_{P}^{Q'}(A^*)$$

Notice that, the level $|A|$ of an atom $A$ is independent from the terms filling in its output positions, i.e., $|A|$ 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, $nodes_{P}^{Q'}(A^*)$ is defined (and finite), and thus $|A|$ is defined (and finite) for every atom $A$.

We now prove that $P$ is quasi recurrent wrt. $|A|$.

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

Let $H = p(s, t)$. Hence, $(H\theta)^* = p(s\theta, x)$ where $x$ is a sequence of fresh distinct variables. Consider a variant $c' : H' \leftarrow A', B', C'$ of $c$ variable disjoint from $(H\theta)^*$. Let $\rho$ be a renaming such that $c' = cp$. Clearly, $(H\theta)^*$ and $H'$ unify. Let $\mu = mgu((H\theta)^*, H') = mgu((H\theta)^*, H\rho) = mgu(p(s\theta, x), p(s, t)\rho)$. By properties of substitutions (see [Apt, 1997]), since $x$ consists of fresh variables, there exists two relevant mgu $\sigma_1$ and $\sigma_2$ such that

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

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

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

By definition of \(\mu\), \(s\theta = s_{\rho \mu}\); hence

\[
(p_{\mu})|_{\text{In}(H)} = \theta_s.
\]  

(10)

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

\[
u_{\rho \mu} = u(p_{\mu})|_{\text{In}(H)} = u_{\theta_s} = u_{\theta}.
\]

(11)

Moreover, since \(c'\) is simply-moded, \(\text{In}(H_{\rho}) \cap \text{Out}(B_{\rho}) = \emptyset\). Hence, by definition of \(\mu\) and standardization apart, \(\text{Dom}(\mu) \cap \text{Out}(B_{\rho}) = \emptyset\), i.e.,

\[
u_{\rho \mu} = v_{\rho}.
\]

(12)

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

\[
B_{\rho \mu} = (B\theta)^*.
\]

(13)

Hence,

\[
|H\theta| = \text{nodes}_{ic}^\mu((H\theta)^*) \quad \text{by definition of } ||
\]

\[
> \text{nodes}_{ic}^\mu((A, B, C)_{\rho \mu}) \quad \text{by Lemma 33 (i)}
\]

\[
\geq \text{nodes}_{ic}^\mu(B_{\rho \mu}) \quad \text{by Lemma 33 (ii)}
\]

\[
= \text{nodes}_{ic}^\mu((B\theta)^*) \quad \text{by (13)}
\]

\[
= |B\theta| \quad \text{by definition of } | |.
\]

\[\square\]

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.,

\[
\text{flatten}(Xs, Ys) \leftarrow \text{flatten}_{dl}(Xs, Ys, [ ]). \]
\]

\[
\text{flatten}_{dl}([ ], Ys, Ys). \]

\[
\text{flatten}_{dl}(X, [X|Xs], Ys) \leftarrow \text{constant}(X), X \neq [ ]. \]

\[
\text{flatten}_{dl}([X|Xs], Ys, Zs) \leftarrow \text{flatten}_{dl}(X, Ys, Y1s), \quad \text{flatten}_{dl}(Xs, Y1s, Zs). \]
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:

\begin{verbatim}
mode app(In,In,Out)
app([],Ys,Ys).
app([H|Xs],Ys,[H|Zs]) ← app(Xs,Ys,Zs).
delay app(Xs,_,_) until nonvar(Xs).
\end{verbatim}

Let \( Q \) be the set of one-atom queries of the form \( \text{app}(s,t,Z) \) where \( s \) and \( t \) are any terms and \( Z \) is a variable disjoint from \( s \) and \( t \). Observe that \( Q \) is closed under resolution: Each resolvent in a derivation starting in a query from \( Q \) is still a query from \( Q \). 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 \( \text{app}(s,t,Z) \) 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 \( Q \) 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 \( \theta \) which does not affect the input arguments of \( A \). In fact, let \( A \) be a selectable atom of \( Q \). If \( A \) unifies with the head of the first clause then, by (1), \( s \) is the empty list \([\ ]\) and \( \theta = \text{mgu}(A,H) = \{Ys/t,Z/t\} \). Otherwise, If \( A \) unifies with the head of the second clause then, by (1), \( s \) is a term of the form \([s_1|s_2]\) and \( \theta = \text{mgu}(A,H) = \{H/s_1,Xs/s_2,Ys/t,Z/[s_1|Zs]\} \). By (2) it follows that, in both cases, \( s\theta = s \) and \( t\theta = t \), i.e., \( \theta \) 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

In order 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, see (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, 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 Works

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 \((A, B, C, D, E)\), \(D\) 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 (Smaus, 1999b). Our results strictly generalize those in (Smaus, 1999b) in the fact 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 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 allows us thus to cope also with this more difficult situation: For instance we can prove that all derivations of APPEND starting
in \texttt{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 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 \texttt{generator\_1}(X_1s), \texttt{generator\_2}(X_2s), \texttt{append}(X_1s,X_2s,Zs), where the generators are defined in different modules; our results allow us to demonstrate that if the \texttt{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 and 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 and Luitjes, 1995) reports a special ad-hoc theorem, in order to prove that, if \( u \) is linear and disjoint from \( s \) then the query \texttt{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 \texttt{app}(s,t,u) is a direct consequence of our main result.

Another related paper is the one by Marchiori and 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 and 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 \texttt{APPEND} is not safe (a safe one would be \texttt{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


### Table 1. Programs from Apt’s Collection

<table>
<thead>
<tr>
<th>NM</th>
<th>IT</th>
<th>QR</th>
<th>NM</th>
<th>IT</th>
<th>QR</th>
</tr>
</thead>
<tbody>
<tr>
<td>app(In,<em>,</em>)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>ordered(In)</td>
<td>yes</td>
</tr>
<tr>
<td>app(_,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>overlap(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>app(Out,In,Out)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>overlap(In,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>append(In,In,Out)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>perm(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>color_map(In,Out)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>perm(In,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>color_map(In,Out)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>qsort(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>color_map(_,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>qsort(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>dcsolve(<em>,</em>)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>reverse(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>even(In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>reverse(_,Out,In)</td>
<td>yes</td>
</tr>
<tr>
<td>fold(_,In,Out)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>select(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>list(_,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>select(_,Out,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>fold(<em>,</em>,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>select(In,Out,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>map(_,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>select(_,In,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>map(<em>,</em>,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>select(_,Out,In)</td>
<td>yes</td>
</tr>
<tr>
<td>member(_,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>sum(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>member(_,In)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>sum(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>mergesort(_,In)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>sum(_,Out,In)</td>
<td>yes</td>
</tr>
<tr>
<td>mergesort(_,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>sum(_,Out,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>mergesort_variant(_,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>type(_,Out)</td>
<td>yes</td>
</tr>
</tbody>
</table>

### Table 2. Programs from DPPD’s Collection

<table>
<thead>
<tr>
<th>NM</th>
<th>IT</th>
<th>QR</th>
<th>NM</th>
<th>IT</th>
<th>QR</th>
</tr>
</thead>
<tbody>
<tr>
<td>applast(_,In,Out)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>match_app(_,In,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>applast(Out,<em>,</em>)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>max_length(_,In,Out,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>applast(<em>,Out,</em>)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>memo_solve(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>contains(_,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>power(_,In,In,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>contains(_,In)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>prune(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>depth(_,In,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>prune(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>depth(_,In,Out)</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>relative(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>depth(_,Out,In)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>relative(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>duplicate(_,In,Out)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>rev_acc(_,In,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>duplicate(_,Out,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>rotate(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>flipflip(_,In,Out)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>rotate(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>flip(_,In,Out)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>solve(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>generate(_,In,Out)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>ssupply(_,In,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>lift(_,In,Out)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>trace(_,In,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>lift(_,Out,In)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>transpose(_,In)</td>
<td>yes</td>
</tr>
<tr>
<td>lift(_,Out,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>transpose(_,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>match_app(<em>,</em>,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>unify(_,In,Out)</td>
<td>yes</td>
</tr>
</tbody>
</table>
### Table 3. Programs from Lindenstrauss’s Collection

<table>
<thead>
<tr>
<th>Program</th>
<th>NM</th>
<th>IT</th>
<th>QR</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ack(In,In,)</code></td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><code>concatenate(In,In,)</code></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>concatenate(_,In,)</code></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code> descendant(In,)</code></td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>deep(In,)</code></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>deep(Out,)</code></td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>credit(In,)</code></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>credit(_,In,)</code></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>holds(_,Out) </code></td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>holds(_,In) </code></td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><code>huffman(In,)</code></td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><code>huffman(_,In) </code></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

### Table 4. Programs from Hermenegildo’s Collection

<table>
<thead>
<tr>
<th>Program</th>
<th>NM</th>
<th>IT</th>
<th>QR</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aiakl.pl</code></td>
<td>init_vars(In,In,Out,Out)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>ann.pl</code></td>
<td>analyze_all(In,Out)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>bid.pl</code></td>
<td>bid(In,Out,Out,Out)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>boyer.pl</code></td>
<td>tautology(In)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><code>browse.pl</code></td>
<td>investigate(In,Out)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>fib.pl</code></td>
<td>fib(In,Out)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><code>fib_add.pl</code></td>
<td>fib(In,Out)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>hanoiapp.pl</code></td>
<td>shanoi(In,In,In,In,Out)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><code>hanoiapp_suc.pl</code></td>
<td>shanoi(In,In,In,In,Out)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>mmatrix.pl</code></td>
<td>mmultiply(In,In,Out)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>occur.pl</code></td>
<td>occurall(In,Out)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>peephole.pl</code></td>
<td>peephole_opt(In,Out)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>progeom.pl</code></td>
<td>pds(In,Out)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>rdtok.pl</code></td>
<td>read_tokens(In,Out)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><code>read.pl</code></td>
<td>parse(In,Out)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><code>serialize.pl</code></td>
<td>serialize(In,Out)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>tak.pl</code></td>
<td>tak(In,In,In,Out)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><code>ttactoee.pl</code></td>
<td>play(In)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><code>warplan.pl</code></td>
<td>plans(In,In)</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
Table 5. Input terminating but non-quasi recurrent Programs

<table>
<thead>
<tr>
<th>Function</th>
<th>SM</th>
<th>IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>mergesort(In,_)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>qsort(In,_)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>type(In,In,Out)</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>depth(In,Out)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>memo_solve(In,Out)</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>ack(In,In,_)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>huffman(In,_)</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>queens(_,Out)</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>serialize(In,Out)</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>