Properties of input-consuming derivations

Bossi, A.; Etalle, S.; Rossi, S.

Published in:
Workshop on Verification of Logic Programs (in connection with ICLP'99, International Conference on Logic Programming, Las Cruces NM, USA, December 1, 1999)

DOI:
10.1016/S1571-0661(04)00116-1

Published: 01/01/1999

Citation for published version (APA):
Properties of Input-Consuming Derivations

A. Bossi\textsuperscript{a}, S. Etalle\textsuperscript{b} and S. Rossi\textsuperscript{a}

\textsuperscript{a} Dipartimento di Informatica, Università di Venezia
30173 Venezia, Italy

\textsuperscript{b} Department of Computer Science, University of Maastricht
6200 MD Maastricht, The Netherlands

Abstract
We study the properties of input-consuming derivations of moded logic programs. Input-consuming derivations do not employ a fixed selection rule, and can be used to model the behavior of logic programs using dynamic scheduling and employing constructs such as \textit{delay declarations}.

We consider the class of \textit{nicely-moded} programs and queries. We show that for these programs one part of the well-known Switching Lemma holds also for input-consuming derivations. Furthermore, we provide conditions which guarantee that all input-consuming derivations starting in a Nicely-Moded query are finite. The method presented here is easy to apply and generalizes other related works.

1 Introduction
Most of the recent logic programming languages provide the possibility of employing a \textit{dynamic} selection rule, that is, a search strategy which is not bound to the selection of the leftmost atom in the query. In fact, dynamic selection rules have proven to be extremely useful in a number of applications; among other things they allow the coroutines of different “processes” and thus to model parallelism by means of interleaving.

Clearly, to be practical, a dynamic selection rule must be restricted in some way. Should one not do so, the computation could easily diverge or yield to unnecessary backtracking. In order to show a trivial example of this, let us consider the standard program \textsc{append}

\texttt{a1: app([ ],Ys,Ys).}
\texttt{a2: app([H|Xs],Ys,[H|Zs]) ← app(Xs,Ys,Zs).}

Together with the contrived query

\texttt{q1: app([1,2],[3,4],Xs), app(Xs,[5,6],Ys).}

In this query, if we selected and resolved the rightmost atom, we could easily

©1999 Published by Elsevier Science B. V. Open access under CC BY-NC-ND license.
have to face one of the following two problems: First, the possibility of non-
termination; if we resolve it using the second clause \( a_2 \), we obtain the query 
\( \text{app}([1, 2], [3, 4], [X|Xs]) \), \( \text{app}(Xs', [5, 6], Ys') \); from here, we can again 
resolve the rightmost atom against \( a_2 \). It is clear that this process can be 
repeated ad infinitum, yielding an infinite computation. The second problem is 
that of inefficiency. If for instance in \( q_1 \) we resolve the rightmost atom against 
\( a_1 \), we obtain the query \( \text{app}([1, 2], [3, 4], []) \). This will eventually fail, 
yielding to (unnecessary) backtracking. Notice that if one employs Prolog’s 
selection rule \( q_1 \) would terminate (with success), without backtracking.

Basically, the problem when selecting \( \text{app}(Xs, [5, 6], Ys) \), is that we don’t 
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 \texttt{nil} symbol we know that we should use \( a_1 \), if it is the 
list-construct symbol we know that we should use \( a_2 \), if it is something else 
again, we know then that the query can fail.

As a matter of fact, languages using dynamic scheduling do provide ways 
of restricting the selection rule, and avoid the selection of those atoms which 
are not sufficiently instantiated. To this end, different languages use different 
mechanisms. For instance, Gödel [17] and Eclipse [25], use \textit{delay declarations}, 
while SICStus Prolog [18] employs \textit{block declarations}. Both check the partial 
instantiation of the input arguments of calls. In GHC [24] programs are 
augmented with \textit{guards} in order to control the selection of atoms dynamically. 
The common underlying idea of all the above solutions is to allow one to “de-
lay” the selection of certain atoms in the query until their arguments become 
sufficiently instantiated. For instance a standard delay declaration for the \textit{app} 
relation symbol would be the following

\[
d1: \text{delay app(Xs,-,-) until nonvar(Xs)}
\]

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 above query without overhead.

Let us now fix some notation: we call \textit{dynamic scheduling} a non-fixed selec-
tion rule which employs suspension mechanisms such as delay declarations. 
As opposed to this, an \textit{unrestricted} selection rule employs no such mecha-
nisms. Notice that a program employing dynamic scheduling might deadlock, 
while this is not the case with an unrestricted selection rule.

\textbf{Properties of Programs using dynamic scheduling}

The additional flexibility introduced by the adoption of dynamic scheduling 
has the disadvantage that a good deal of the properties of programs using 
either a fixed selection rule or an unrestricted one do not hold any longer.

Initial goal of our research was the study of termination properties of this 
class of programs. The study is motivated by the fact that most results on 
termination of logic programs (see [13] for a survey on this subject) does not
apply when a dynamic selection rule is employed. Notable exceptions are Bezem's [9] and and Cavedon's [11], which however provide results for unrestricted selection rules. We know of few authors who tackled the specific problem of termination of logic programs with a dynamic selection rule. Apt and Luitjes's [4] exploits properties of a restricted class of SLD-derivations to prove termination of logic programs augmented with delay declarations that imply determinacy and matching. Marchiori and Teusink's [20] introduces the class of delay recurrent programs and proves that programs in this class terminate for all local delay selection rule. More recently, Smaus's [21] studies the termination of input-consuming derivations of well and nicely modeled programs. We compare our results with the ones in [4,20,21] in the conclusions.

During our study it became also clear that more fundamental properties of logic programs do not hold, or hold only partially, in this modified setting. Among them, 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 condition - one can guarantee that one side of the switching lemma applies to programs with dynamic scheduling as well.

**Modes and input-consuming derivations**

The first obstacle one encounters in the study of properties of programs employing delay declarations is that these constructs are meta-logical, therefore many of the the nice declarative properties of logic programming languages are immediately lost. For instance, to our knowledge nobody has so far provided a model-theoretical semantics for them.

In order to recuperate at least part of the declarative side of logic programming, one has to find an “algebraic” way of representing delay declarations. For this purpose, we follow here the same approach of [21] 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 occurring in 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. An input-consuming derivation contains only input-consuming steps. If in a query no atom is resolvable via an input-consuming step we have a deadlock situation.

For example, the standard mode for the program APPEND reported above, when used for concatenating two lists is $\text{app}(\text{In}, \text{In}, \text{Out})$. Notice that in this case the delay declaration $d_1$ serves precisely for guaranteeing that if an atom of the form $\text{app}(\text{ts,us,x})$ (x being a variable) is selectable and resolvable, then it is so via an input-consuming derivation step. It is also worth remarking that, as a large body of literature shows, the vast majority of “usual” programs are actually moded; see for example [7,8], or more simply, the tables of programs we report in Section 6, or consider for instance the
logic programming language Mercury [23], which requires that its programs are moded (and well-moded).

Of course, in presence of modes, one has to require that the programs and queries we consider should be coherent wrt the chosen modes. In the literature there are several definition for this purpose (e.g. the ones of “well-moded”, “nicely moded” and “simply moded” programs, see for instance [8]). Each definition has its own restrictions and yields to a set of properties. In this paper we consider programs which are nicely moded [7] with respect to a given mode. One might wonder to which extent “usual” programs are nicely moded. Again, in Section 6 we provide arguments showing that the large majority of moded programs is nicely-moded.

*Delay declarations vs input-consuming derivations*

We claim that in most programs using delay declarations, delay declarations are in fact used to enforce that the derivation steps are input-consuming. This claim is of crucial importance for the relevance of our study: it allows us to assert that our result can be applied to programs employing dynamic scheduling. Therefore, we now spend a few paragraphs “proving” it.

It is clear that the scope of a delay declaration is to guarantee that the interpreter will not select the “wrong” clause to resolve a goal. In fact, if the interpreter always selected the appropriate clause, by the known results over independence from the selection rule one would not have to worry about the order of the selection of the atoms in the query. Notice in fact that in the above example concerning query q1, nothing forbids us to employ the fixed selection rule that keeps selecting the rightmost. However, if we don’t want to waste computation time with this we have to make sure that for the first four resolutions steps starting from q1 we employ clause a2 and for the fifth we resolve the selected atom against a1. This is because x8 will eventually be instantiated to a list of length four. The problem is that at the moment we have to resolve q1 we don’t have knowledge of that, so we should wait until the instantiation of x8 provides us with some more informations.

As we mentioned, delay declarations are used to prevent 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, the degree of instantiation we are interested in is clearly the one of the input positions, which are the one carrying the information.

Now, take an atom p(s, t), where s and t denote the sequences of terms filling in its input and output positions, respectively. Suppose that it is resolvable with a clause c (moreover by means of an input-consuming derivation step). Then, for every instance s' of s, we have that the atom p(s', t) is as well resolvable with c by means of an input-consuming derivation step. In other words, no further instantiation of the input positions of p(s, t) can rule out c as a possible clause for resolving it. Thus c must be one of the “right”
clauses for resolving $p(s, t)$ and we can say that $p(s, t)$ is in its input positions “sufficiently instantiated” to be resolved with $c$.

On the other hand, following the same reasoning, it is easy to see that if $p(s, t)$ is resolvable with $c$ but not via an input-consuming derivation step, then there exists an instance $s'$ of $s$, such that $p(s', t)$ is not resolvable via $c$. In this case we can say that $p(s, t)$ is not instantiated enough to know whether $c$ is one of the “right” clauses for resolving it.

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 to programs and queries which are nicely-moded, then a one way switching-lemma holds and a simple method for proving termination can be applied.

Furthermore, we study their termination properties, for this we define the class of input terminating programs which characterizes programs whose input-consuming derivations are finite. In order to prove that a program is input terminating we use the concept of quasi recurrent program which is much less restrictive than the similar concept of semi-recurrent program introduced by Apt and Pedreschi in [6]. We show that if $P$ is nicely-moded and quasi recurrent then all its input-consuming derivations starting from a nicely-moded query terminate.

Our work generalizes the method described by Smaus in [21] for proving the termination of input-consuming derivations of queries and programs which are both well- and nicely-moded. First, as opposed to [21], we do not require programs and queries to be well-moded; we only assume that they are nicely-moded. Second, our concept of quasi recurcency provides a condition to hold for all instances of a clause while the notion of ICD-acceptability proposed by Smaus only considers clause ground instances. This small generalization allows us to prove termination of input-consuming derivations of queries where the input arguments are not necessarily ground. For example, we can prove termination of all the input-consuming derivations of the program APPEND starting from a query append$(s, t, u)$ provided that $u$ is linear and variable disjoint from $s$ and $t$. With the method of [21] one can only prove termination of those input-consuming derivations where the initial query satisfies the additional condition that $s$ and $t$ are ground.

We also show that the results presented in this paper can be extended to programs and queries which are permutation nicely-moded [22].

Finally, we apply our method to many benchmarks from well-known collections to show applicability and effectiveness of the results presented in this paper.

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 proved. In Section 4 we prove that, for nicely-moded input-consuming programs one side of the Switching Lemma
holds. In Section 5 a method for proving termination of programs is presented, first in a non-modular way, then for modular programs. Section 6 reports the results obtained by applying our method to various benchmarks. Finally, Section 7 concludes the paper.

2 Preliminaries

The reader is assumed to be familiar with the terminology and the basic results of the semantics of logic programs [1,2,19].

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 $\text{Dom}(\theta) = \{X | \theta(X) \neq X\}$ 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 $\theta \leq \sigma$ iff 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 $\text{mgu}(t = t')$ 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 $t_1, \ldots, t_n \in \mathcal{T}$. Given an atom $A$, we denote by $\text{Rel}(A)$ the predicate symbol in $A$. A query is a possibly empty finite sequence of atoms $A_1, \ldots, A_m$. The empty query is denoted by $\Box$. Following the convention adopted by Apt in [2], we use bold characters to denote (possibly empty) sequences of atoms. 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 $\text{mgu} \theta$. The query $(A,B,C) \theta$ is called a resolvent of $A, B, C$ and $c$ with respect to $B$, with an mgu $\theta$. A
derivation step is denoted by

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

\( H \leftarrow B \) is called its input clause. The atom \( B \) is called the selected atom of \( A, B, C \). The atoms in \( B \theta \) are the ones resulting from the derivation step.

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

\[ \delta := Q_0 \xrightarrow{\theta_1} P_{c_1} Q_1 \xrightarrow{\theta_2} P_{c_2} \cdots Q_n \xrightarrow{\theta_{n+1}} P_{c_{n+1}} Q_{n+1} \cdots \]

of derivation steps 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 \( \delta := Q_0 \leftarrow Q_n \) 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 is the only non-standard one we need in the sequel.

**Definition 2.1 (B-step)** Consider a (partial) derivation \( \delta \) starting in \( A, B, C \). A B-step of \( \delta \) is either the step in which \( B \) is selected (if it exists) or a derivation step in which an atom resulting from a B-step is selected.

3 Modes

3.1 Basic Definitions

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 3.1 (Mode)** Consider an \( n \)-ary predicate symbol \( p \). By a mode for \( p \) we mean 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 \( p \) (with respect to \( m_p \)). 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 set 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 [21] and is defined as follows.

**Definition 3.2 (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
  \[ A, B, C \xrightarrow{\theta \epsilon_c} (A, B, C)\theta \]
  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 3.3** 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 $\vartheta$ of $p(s, t)$ and $p(u, v)$ such that $s\vartheta = s$.

**Proof.** Since $p(s, t)$ and $p(u, v)$ are unifiable, there exists a relevant mgu $\theta_{\text{rel}}$ of them (cfr. [2], Theorem 2.16). Now, $\theta_{\text{rel}}$ is a renaming of $\theta$. Thus $s\theta_{\text{rel}}$ is a variant of $s$. Then there exists a renaming $\rho$ such that $\text{Dom}(\rho) \subseteq \text{Var}(s, t, u, v)$ and $s\theta_{\text{rel}}\rho = s$. Now, take $\vartheta = \theta_{\text{rel}}\rho$. \(\Box\)

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

**Example 3.4** Consider the program `REVERSE` with accumulator in the modes defined below.

```prolog
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]).
```

Consider also the query `reverse([X1,X2], Zs)`. The derivation $\delta$ of $\text{REVERSE} \cup \{ reverse([X1,X2], Zs) \}$ depicted below is input-consuming.

$\delta := reverse([X1,X2], Zs) \Rightarrow reverse_acc([X1,X2], Zs, []) \Rightarrow reverse_acc([X2], Zs, [X1]) \Rightarrow reverse_acc([], Zs, [X2,X1]) \Rightarrow \Box$. 

8
3.2 *Nicely-Moded Programs*

In this the sequel of the paper we’ll restrict 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 3.5 (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 vector of terms and for all \( i \in \{1, \ldots, n\} \)

\[
\text{Var}(s_i) \cap \bigcup_{j=i}^{n} \text{Var}(t_j) = \emptyset.
\]

- A clause \( c = p(s_0, t_0) \leftarrow Q \) is *nicely-moded* if \( Q \) is nicely-moded and

\[
\text{Var}(s_0) \cap \bigcup_{j=1}^{n} \text{Var}(t_j) = \emptyset.
\]

- 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 3.6**

- The program `REVERSE` with accumulator in the modes depicted in the Example 3.4 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]) \leftarrow Y < X, merge([X|Xs],Ys,Zs).
merge([X|Xs],[Y|Ys],[X|Zs]) \leftarrow Y > X, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[X|Ys],[X|Zs]) \leftarrow merge(Xs,[X|Ys],Zs).
```

The following result is due to Smaus [21], and states that the class of programs and queries we are considering is closed under resolution.

**Lemma 3.7 [21]** 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 [21]) is an immediate consequence of the definition of input-consuming derivation step and the fact that the mgu’s we consider are relevant.

**Remark 3.8 [21]** Let the program \( P \) and the query \( Q := A, p(s, t), C \) be
nicely-moded. If $A, p(s, t), C \xrightarrow{\theta} A, B, C$ 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, [2, Lemma 3.32]) is a well-known result holding for logic programs using unrestricted selection rules, and which allows one to prove several fundamental results such as for instance the independence of the answer substitutions from the selection rule.

In the case of logic programs using a dynamic restricted selection rule such as the one we are considering here, the switching lemma does not hold any longer. For example, in program \textsc{Append} reported in the introduction (together with delay declaration d1) we have that the rightmost atom of q1 is selectable only after the leftmost one has been resolved; which is in contradiction with the switching lemma.

Nevertheless we can show that for nicely moded programs one “side” of the switching lemma still holds.

First, we need one technical result, stating that the only variables of a query that can be “affected” in the derivation process are those occurring in some output positions. The proof is omitted, and reported in [10].

\textbf{Lemma 4.1} Let the program $P$ and the query $Q$ be nicely-moded. 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 $x \notin \text{Var}(\text{Out}(Q))$, $x\theta = x$.

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

\textbf{Corollary 4.2} 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.

\textbf{Lemma 4.3} (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

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

where

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

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

$$
\theta_{n+1}\theta_{n+2} = \theta'_{n+1}\theta'_{n+2}
$$
and
\[ \delta' := Q_0 \xrightarrow{\theta_1} c_1 Q_1 \cdots Q_n \xrightarrow{\theta_{n+1}} c_{n+2} Q'_{n+1} \xrightarrow{\theta'_{n+2}} c_{n+1} Q_{n+2} \]

where
- \( \delta' \) is input-consuming,
- \( \delta \) and \( \delta' \) coincide up to the resolvent \( Q_n \),
- \( Q_{n+1} \) is a resolvent of \( Q_n \) and \( c_{n+2} \) wrt \( A \),
- \( Q_{n+2} \) is a resolvent of \( Q'_{n+1} \) and \( c_{n+1} \) wrt \( B\theta'_{n+1} \).
- \( \delta \) and \( \delta' \) coincide after the resolvent \( Q_{n+2} \).

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

1. \( u\theta_{n+1} = u \), since \( \delta \) is input-consuming.

By (1) 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

2. \( s\theta_{n+2} = s \), since \( \delta \) is input-consuming.

Moreover,

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

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,

\[ \delta' := Q_0 \xrightarrow{\theta_1} c_1 Q_1 \cdots Q_n \xrightarrow{\theta_{n+1}} c_{n+2} Q'_{n+1} \xrightarrow{\theta'_{n+2}} c_{n+1} Q_{n+2} \]

where

4. \( \theta'_{n+1} = \theta_{n+2} \)

By 2, \( Q_n \xrightarrow{\theta_{n+1}} c_{n+2} Q'_{n+1} \) is an input-consuming derivation step. Observe now that
\[ u\theta'_{n+1}\theta'_{n+2} = u\theta_{n+2}\theta'_{n+2}, \text{ (by (4))} \]
\[ = u\theta_{n+1}\theta_{n+2}, \text{ (by (3))} \]
\[ = u\theta_{n+2}, \text{ (by (1))} \]
\[ = u\theta'_{n+2}, \text{ (by (4))} \]

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

This result shows that it is always possible to proceed left-to-right to resolve the selected atoms. Notice however that this is different than saying that the leftmost atom of a query should always be resolvable: it can very
well be the case that the leftmost atom is suspended and the one next to it is resolvable.

It is important to notice that if we drop the nicely-modedness condition the above Lemma would not hold any longer: it is not difficult to write a classical coroutining program which is not nicely-moded for which the above lemma does not apply (see for instance the program reader-writer in [16]).

**Corollary 4.4** Let the program $P$ and the query $Q := A, B$ be nicely-moded. Suppose that

$$\delta := A, B \xrightarrow{\theta} C_1, C_2$$

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

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

where all the $A$-steps are performed in the prefix $A, B \xrightarrow{\theta_1} C_1, B\theta_1$ of $\delta'$ and $\theta = \theta_1\theta_2$.

## 5 Termination

In this section we study the termination of input-consuming derivations of logic programs. To this end we refine the ideas of Bezem [9] and Cavedon [11] who studied the termination of logic programs in a very strong sense, namely with respect to unrestricted selection rules, and of Smaus [21] 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 5.1 (Input Termination)** A program is called input terminating iff all its input-consuming derivations started with a nicely-moded query are finite.

The method we are going to use in order to prove that a program is input-terminating is based on the following concept of moded level mapping introduced by Etalle et al. in [15].

**Definition 5.2 (Moded Level Mapping)** Let $P$ be a program and $\mathcal{B}_P^c$ be the extended Herbrand Base for the language associated with $P$. A function $|$ is a moded level mapping for $P$ iff:

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

For $A \in \mathcal{B}_P^c$, $|A|$ 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 [15]: in [15] the level mapping is defined on ground atoms only. Therefore this is actually an extension of the definition of [15].

**Example 5.3** 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 `REVERSE` with accumulator of the Example 3.4 is

$$\begin{align*}
|\text{reverse}(x,s,y)| &= TSize(x) \\
|\text{reverse}_\text{acc}(x,s,y,z)| &= TSize(x)
\end{align*}$$

5.2 Quasi Recurrence

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 coincides with the one of *acyclic program*) was introduced by Apt and Bezem [9,3] and independently by Cavedon [12] in order to prove universal termination for unrestricted selection rules together with other properties of logic programs. Later, Apt and Pedreschi [6] 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 5.4** 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 $p \sqsupset q$ in $P$ iff $(p,q)$ is in the reflexive and transitive closure of the relation refers to.

According to the above definition, $p \sim q \equiv p \sqsupset q \land p \sqsupset q$ means that $p$ and $q$ are mutually recursive, and $p \sqsupset q \equiv p \sqsupset q \land p \not\sqsupset q$ 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 5.5 (Quasi Recurrence)** Let $P$ be a program and $\mid \mid : \mathcal{B}_P \rightarrow \mathbb{N}$ be a moded level mapping.

- A clause of $P$ is called *quasi recurrent with respect to* $\mid \mid$ iff for every instance of it, $H \leftarrow A, B, C$

  $$(5) \text{ if Rel}(H) \sim \text{Rel}(B) \text{ then } |H| > |B|.$$  

- A program $P$ is called *quasi recurrent with respect to* $\mid \mid$ iff all its clauses are. $P$ is called *quasi recurrent* iff it is quasi recurrent with respect to some moded level mapping $\mid \mid : \mathcal{B}_P \rightarrow \mathbb{N}$.
The notion of quasi recurrent program differs from the concepts of recurrent and of semi-recurrence introduced in two ways. First, we require that $|H| > |B|$ only for those body atoms which recursively depend on $\text{Rel}(H)$; on the other hand, both concept of recurrent and semi-recurrent program require that $|H| > |B|$ ($|H| \geq |B|$ in the case of semi-recurrency) also for the atoms for which $\text{Rel}(H) \not\subset \text{Rel}(B)$. Secondly, every instance of a program clause is considered, not only ground instances as in the case of (semi-)recurrent programs.

It is worthwhile noticing that this concept almost coincides with the one of ICD-acceptable introduced and used in [21]. We allowed ourself 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 in Apt and Pedreschi [5,6] in order to prove termination of logic programs using the fixed left-to-right selection rule. The crucial difference between recurrency and of acceptability lies in the fact that the latter relies on the presence of a model $M$; this allows condition (5) to be checked only for those body atoms which are in a way “reachable” wrt $M$. For this reason every recurrent program is acceptable but not vice-versa. Since the above definition does not rely on the presence of a model, it is clearly much more related to the notion of recurrent that to the one of acceptable program.

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

**Theorem 5.6** 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 5.10 by setting $R = \emptyset$. □

**Example 5.7** Consider the program `MERGE` defined in the Example 3.6. Let $\| \|$ be the moded level mapping for `MERGE` defined by

$$\text{merge}(\text{xs}, \text{ys}, \text{zs}) = \text{TSize}(\text{xs}) + \text{TSize}(\text{ys})$$

It is easy to prove that `MERGE` is quasi recurrent with respect to the moded level mapping above. By Theorem 5.6, all input-consuming derivations of the program `MERGE` started with a query `merge(u, s, t)` where $t$ is linear and variable disjoint from $u$ and $s$ are terminating.

### 5.3 Modular Termination

This section contains a generalization of Theorem 5.6 to the modular case, as well as the complete proofs for it.

The following lemma is a crucial one. The proof is omitted, and reported in [10].

**Lemma 5.8** 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

- there exists an input-consuming derivation $\delta'$ of $P \cup \{Q\}$ of the form
  
  \[ \delta' := A_1, \ldots, A_n \xrightarrow{\theta} C_i(A_i, \ldots, A_n) \theta \rightsquigarrow \cdots \]

- there exists an infinite input-consuming derivation of $P \cup \{A_i \theta\}$.

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

**Corollary 5.9** 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.

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 $R$ iff no relation defined in $P$ occurs in $R$.

**Theorem 5.10** Let $P$ and $R$ be two programs such that $P$ extends $R$. Suppose that

(i) $R$ is input-terminating,

(ii) $P$ is nicely-moded and quasi recurrent with respect to a moded level mapping $\parallel : \mathcal{B}^e_P \rightarrow \mathbb{N}$.

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 \{\emph{q}| q is defined in P and p $\sqsubseteq$ 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 $p \sqsubseteq q$ then $dep_P(p) > dep_P(q)$.

We can now prove our theorem. By Corollary 5.9, 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 $P$.

Let $\delta$ be 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 $\theta_1 = mg(A = H)$. Clearly, $(B_1, \ldots, B_n) \theta_1$ has an infinite input-consuming derivation in $P \cup R$. By Lemma 5.8, 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

$$A \xrightarrow{\theta_1} (B_1, \ldots, B_n)\theta_1 \xrightarrow{\theta_2} C, (B_1, \ldots, B_n)\theta_1\theta_2 \cdots;$$

(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 now show that (2) cannot hold; for this we proceed by induction on $\langle \text{dep}_P(\text{Rel}(A)), |A|\rangle$ with respect to 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 $\text{dep}_P(\text{Rel}(A)) = 0$ (any $|A|$). 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 descendants in any input-consuming derivation are defined in $R$. The hypothesis that $R$ is input-terminating contradicts point (2) above.

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

(a) $\text{Rel}(H) \sqsubset \text{Rel}(B_i)$,
(b) $\text{Rel}(H) \simeq \text{Rel}(B_i)$.

In case (a) we have that $\text{dep}_P(\text{Rel}(A)) = \text{dep}_P(\text{Rel}(H\theta)) \succ \text{dep}_P(\text{Rel}(B_i\theta))$. So, $\langle \text{dep}_P(\text{Rel}(A)), |A| \rangle = \langle \text{dep}_P(\text{Rel}(H\theta)), |H\theta| \rangle \succ \langle \text{dep}_P(\text{Rel}(B_i\theta)), |B_i\theta| \rangle$.

In case (b), from the hypothesis that $P$ is quasi recurrent w.r.t. $| |$, it follows that $|H\theta| \succ |B_i\theta|$. Consider the partial input-consuming derivation $A \xrightarrow{\theta} C, (B_1, \ldots, B_n)\theta$, by Corollary 4.2 and the fact that $| |$ is a moded level mapping, we have that $|A| = |A\theta| = |H\theta|$. Hence, $\langle \text{dep}_P(\text{Rel}(A)), |A| \rangle = \langle \text{dep}_P(\text{Rel}(H\theta)), |H\theta| \rangle \succ \langle \text{dep}_P(\text{Rel}(B_i\theta)), |B_i\theta| \rangle$. In both cases, the contradiction follows by the inductive hypothesis. \qed

**Example 5.11** The program FLATTEN using difference-lists is nicely-moded wrt the modes described below, provided that we replace “\" replaced by “,”, as we've done here.

```
mode flatten(In,Out).
mode flatten_dl(In,Out,In).
mode constant(In).
mode \neq(In,In).

flatten(Xs, Ys) ← flatten_dl(Xs, Ys, [ ]).
flatten_dl([], Ys, Ys).
flatten_dl([ X | Xs ], Xs) ← constant(X), X \neq [ ].
flatten_dl([ X | Xs ], Ys, Zs) ← flatten_dl(Xs, Ys1, Zs),
                   flatten_dl(X, Ys, Ys1).
```

Consider the moded level mapping for FLATTEN defined by

$|\text{flatten}(Xs, Ys)| = T\text{Size}(Xs)$

$|\text{flatten}_\text{dl}(Xs, Ys, Zs)| = T\text{Size}(Xs)$.

It is easy to see that the program FLATTEN is quasi recurrent with respect to
the moded level mapping above. Hence, all input-consuming derivations of \texttt{FLATTEN} starting from a query \texttt{flatten}(u,s) where \( s \) is linear and variable disjoint from \( u \) are terminating.

\textit{Permutation Nicely-Moded}

At this point it is worth noticing that, since the programs we are considering do not use a fixed selection rule the result we have provided (Theorems 5.6 and 5.10) hold also in the case that programs and queries are permutation nicely-moded \cite{22}, that is programs and queries for which would be nicely-moded after a permutation of the atoms in the bodies. Therefore, for instance, we can treat the program \texttt{FLATTEN} as it is presented in \cite{2} (except for the replacement of "\\" with ","), i.e.,

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

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

where the atoms in the body of the last clause are permuted with respect to the version of the Example 5.11.

\section{Applicability}

In order to assess the applicability of the results we report here, we have looked into four collection of logic programs, and we have checked those programs against the three basic definitions we employ here: the ones of nicely-moded, input terminating and quasi recurrent program. The results are reported in the Tables 1...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 \cite{2} and \cite{6}). 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 we find 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 nicely-moded (NM), input-terminating (IT), and quasi recurrent (QR).
7 Conclusion

In this paper we studied the properties of nicely-moded programs using an input-consuming selection rule, which is a restricted non-fixed selection rule.

This study is motivated by the widespread use of programs using dynamic selection rule together with delay declarations. In fact, as we have motivated in the introduction we strongly believe that most programs employing a dynamic selection rule together with selection restrictions such as delay declarations, the use of such restrictions mainly reduces to guaranteeing that the derivation steps are input-consuming.

In the first place we showed that, for nicely-moded programs one side of the well-known switching lemma holds.

Secondly, we presented a method for proving termination of programs and queries which are (permutation) nicely-moded. Our results strictly improve on [21] 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 above program flatten, 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 [21] apply only if t is a ground term.

Other related papers are the ones of Apt and Luitjes [4] and of Marchiori and Teusink [20]. The most important difference between this paper and [4] lies in the fact that the latter demands like [21] that the programs and queries are well-moded. On the other hand, in [20] the author make a strong restriction on the selection rule, which has to be local; this restriction actually forbids any form of coroutining.

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

Future work concerns the automatization of our method; this depends on the capability of automatically inferring moded level mappings. It is well-known the relation between norms, which define the size of terms, and level mappings: roughly, level mappings are obtained by extending norms to function from atoms to natural numbers. Decorte, De Schreye and Fabris’ [14] presents two techniques for the automatic inference of norms. We argue that the same techniques can be applied to automatize termination proofs based on our approach.

References


<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>append(In,\ldots)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>ordered(In)</td>
<td>yes</td>
</tr>
<tr>
<td>append(_\ldots,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>overlap(_\ldots,In)</td>
<td>yes</td>
</tr>
<tr>
<td>append(Out,In,Out)</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>overlap(In,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>append3(In,In,In,Out)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>perm(_\ldots,In)</td>
<td>yes</td>
</tr>
<tr>
<td>color_map(In,Out)</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>perm(In,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>color_map(Out,In)</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>qsort(In,\ldots)</td>
<td>yes</td>
</tr>
<tr>
<td>color_map(In,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>qsort(Out,In)</td>
<td>yes</td>
</tr>
<tr>
<td>dresolve(In,\ldots)</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>reverse(In,\ldots)</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,In,Out)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>select(_\ldots,\ldots)</td>
<td>yes</td>
</tr>
<tr>
<td>list(In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>select(_\ldots,In)</td>
<td>yes</td>
</tr>
<tr>
<td>lte(In,\ldots)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>select(In,Out,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>lte(_\ldots,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>subset(In,In)</td>
<td>yes</td>
</tr>
<tr>
<td>map(In,\ldots)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>subset(Out,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>map(_\ldots,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>subset(Out,In)</td>
<td>yes</td>
</tr>
<tr>
<td>member(_\ldots,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>mem(_\ldots,In)</td>
<td>yes</td>
</tr>
<tr>
<td>member(In,Out)</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>mem(_\ldots,In)</td>
<td>yes</td>
</tr>
<tr>
<td>mergesort(In,\ldots)</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>mem(In,Out,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>mergesort(Out,In)</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>type(In,Out)</td>
<td>no</td>
</tr>
<tr>
<td>mergesort_variant(_\ldots,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>type(In,Out)</td>
<td>no</td>
</tr>
</tbody>
</table>

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>applast(In,In,Out)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>matchืน(In)</td>
<td>yes</td>
</tr>
<tr>
<td>applast(Out,..,..)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>matchืน(In,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>applast(.,Out,..)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>max lenght(In,Out,Out)</td>
<td>yes</td>
</tr>
</tbody>
</table>
| contains(.,In)     | yes | yes | yes | memo solve(In,Out) | yes | yes | no  
| contains(In,Out)   | yes | no  | yes | power(In,In,Out) | yes | yes | yes |
| depth(In,In)       | yes | yes | yes | prune(In) | yes | yes | yes |
| depth(In,Out)      | yes | no  | yes | prune(In) | yes | yes | yes |
| depth(Out,In)      | yes | no  | yes | relative(In) | yes | no  |
| duplicate(In,Out)  | yes | yes | yes | relative(.,In) | yes | no  |
| duplicate(Out,In)  | yes | yes | yes | rev acc(In,In,Out) | yes | yes | yes |
| flipflip(In,Out)   | yes | yes | yes | rotate(In) | yes | yes | yes |
| flipflip(Out,In)   | yes | yes | yes | rotate(.,In) | yes | yes | yes |
| generate(In,In,Out)| yes | no  | yes | solve(.,..) | yes | no  |
| liftsolve(In,Out)  | yes | no  | yes | supply(In,In,Out) | yes | yes | yes |
| liftsolve(Out,In)  | yes | no  | yes | trace(In,In,Out) | yes | yes | yes |
| liftsolve(In,In)   | yes | yes | yes | transpose(In) | yes | yes | yes |
| match(In,..)       | yes | no  | yes | transpose(In,Out) | yes | no  |
| match(.,In)        | yes | yes | no  | unify(In,In,Out) | yes | no  |

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>ack(In,..)</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>least(In)</td>
<td>yes</td>
</tr>
<tr>
<td>concatenate(In,..)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>least(.,In)</td>
<td>yes</td>
</tr>
<tr>
<td>concatenate(.,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>normal form(In)</td>
<td>yes</td>
</tr>
<tr>
<td>concatenate(.,.,In)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>normal form(.,In)</td>
<td>yes</td>
</tr>
<tr>
<td>descendant(In)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>queens(.,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>descendant(.,In)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>queens(.,.,In)</td>
<td>yes</td>
</tr>
<tr>
<td>deep(In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>poss(In)</td>
<td>yes</td>
</tr>
<tr>
<td>deep(Out)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>poss(Out)</td>
<td>yes</td>
</tr>
<tr>
<td>credit(In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>rewrite(In)</td>
<td>yes</td>
</tr>
<tr>
<td>credit(.,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>rewrite(.,In)</td>
<td>yes</td>
</tr>
<tr>
<td>holds(.,Out)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>transform(.,.,Out)</td>
<td>yes</td>
</tr>
<tr>
<td>holds(.,In)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>transform(.,.,.,In)</td>
<td>yes</td>
</tr>
<tr>
<td>huffman(In)</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>total least(In)</td>
<td>yes</td>
</tr>
<tr>
<td>huffman(.,In)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>total least(.,In)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3
Programs from Lindenstrauss's Collection
<table>
<thead>
<tr>
<th></th>
<th>NM</th>
<th>IT</th>
<th>QR</th>
</tr>
</thead>
<tbody>
<tr>
<td>aiak1.pl</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>ann.pl</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>bid.pl</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>boyer.pl</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>browse.pl</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>fib.pl</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>fib_add.pl</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>hanoiapp.pl</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>hanoiapp_muc.pl</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>mmatrix.pl</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>occur.pl</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>peephole.pl</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>progeom.pl</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>rdtk.pl</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>read.pl</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>serialize.pl</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>tak.pl</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>tic tac toe.pl</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>warplan.pl</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Table 4
Programs from Hermenegildo’s Collection