URL: http://www.elsevier.nl/locate/entcs/volume30.html 21 pages

Properties of Input-Consuming Derivations

A. Bossi [a], S. Etalle [b] and S. Rossi [a]

[a] *Dipartimento di Informatica, Università di Venezia*
30173 Venezia, Italy

[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 *delay declarations*.

We consider the class of Nicely-Moded programs and queries. We show that for these programs one part of the well-known Switching Lemma holds. 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 *dynamic* selection rule, that is, a selection rule which is more flexible than Prolog's standard left-to-right one. In fact, dynamic selection rules have proven to be useful in a number of applications; among other things they allow the coroutining of different "processes" and thus to model parallelism by means of interleaving.

Clearly, even a dynamic selection rule must be restricted in some way. Should one not do so, the computation could easily diverge. To this end, different languages use different mechanisms. For instance, in Gödel [12] and in Eclipse [21], *delay declarations* are used to ensure that only atoms which are ground in their input arguments are selected. In GHC [19] programs are augmented with *guards* in order to control the selection of atoms dynamically (moreover, moded flat GHC [20] uses an extra condition on the input positions, which is extremely similar to the one we'll adopt in the sequel). `block` declarations that check the partial instantiation of the input arguments of calls are used in SICStus. The common underlying idea of all the above solutions is

to allow one to "delay" the selection of certain atoms in the query until their arguments become sufficiently instantiated.

The additional flexibility introduced by the adoption of a dynamic selection mechanism has the disadvantage that the most of the literature on termination of logic programs (see [15] for a survey on the subject) does not apply when a dynamic selection rule is employed. Notable exceptions are Bezem's [7] and and Cavedon's [8], which 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 [3] 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 [14] 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 [16] studies the termination of input-consuming derivations of well and nicely moded programs.

Goal of this paper is to study the dynamic behavior of programs using dynamic scheduling, and to provide sufficient conditions which guarantee their termination.

The first obstacle we encounter is of providing an "algebraic" way of representing delay declarations. For this purpose, we follow here the same approach of [16] 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* derivations are precisely those in which only atoms whose input arguments will not be instantiated by the unification are allowed to be selected.

We claim that in most "usual" moded programs using a dynamic selection rule, delay declarations are employed precisely for ensuring the input-consumedness of the derivations. Clearly, this thesis cannot be proven, yet it is for instance substantiated by the fact that concept of input-consuming resolution is very similar to the selection mechanism employed in Moded Flat GHC [20], and by the arguments in [16]. In Section 3 we provide further technical arguments sustaining this thesis.

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

In order to study termination properties, 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 *weakly semi-recurrent* program which is much less restrictive than the similar concept of semi-recurrent program introduced by Apt and Pedreschi in [4]. We show that if P is nicely-moded and weakly semi-recurrent

then all its input-consuming derivations starting from a nicely-moded query terminate.

Our work generalizes the method described by Smaus in [16] for proving the termination of input-consuming derivations of well and nicely-moded programs and queries. First, as opposed to [16], we do not require programs and queries to be well-moded; we only assume that they are nicely-moded. Second, our concept of weak semi-recurrency 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 $\texttt{append}(s, t, u)$ provided that u is linear and variable disjoint from s and t. With the method of [16] 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 show that the results presented in this paper can be extended to programs and queries which are *permutation nicely-moded* [17].

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 a method for proving termination of programs is presented, first in a non-modular way, then for modular programs. Section 5 reports the results obtained by applying our method to various benchmarks. Finally, Section 6 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 [1,2,13].

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

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

2.2 Programs and Derivations

Let $\mathcal{P}$ 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 $Rel(A)$ 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 $H \leftarrow \mathbf{B}$ where H is an atom (the *head*) and $\mathbf{B}$ is a query (the *body*). When $\mathbf{B}$ is empty, $H \leftarrow \mathbf{B}$ is written $H \leftarrow$ and is called a *unit clause*. A *program* is a finite set of clauses. We denote atoms by $A, B, H, \ldots$, queries by $Q, \mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$, clauses by $c, d, \ldots$, and programs by P.

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

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

$H \leftarrow \mathbf{B}$ is called its *input clause*. The atom B is called the *selected atom* of $\mathbf{A}, B, \mathbf{C}$. If P is clear from the context or c is irrelevant then we drop a reference to them. A derivation is obtained by iterating derivation steps. A maximal sequence

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

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 $\delta := Q_0 \stackrel{\theta_1}{\Longrightarrow}_{P,c_1} \cdots \stackrel{\theta_n}{\Longrightarrow}_{P,c_n} Q_n$ is a finite prefix of a derivation, also denoted $\delta := Q_0 \stackrel{\theta}{\longmapsto} Q_n$ with $\theta = \theta_1 \cdots \theta_n$, we say that δ is a *partial derivation* and θ is a *partial computed answer substitution* (*p.c.a.s.*, for short) of $P \cup \{Q_0\}$. If δ is maximal and ends with the empty query then θ is called *computed answer substitution* (*c.a.s.*, for short). The length of a (partial) derivation δ, denoted by $len(\delta)$, is the number of derivation steps in δ. We call B-step any derivation step in a derivation δ of $\mathbf{A}, B, \mathbf{C}$ in which the selected atom is B or any other atom obtained by resolving B.

3 Input-Consuming Derivations

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 $\{In, Out\}$.

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

The notion of input-consuming derivation was introduced in [16] and is defined as follows.

Definition 3.2 [Input-Consuming]

- An atom $p(\mathbf{s}, \mathbf{t})$ is called *input-consuming resolvable wrt a clause* $c := p(\mathbf{u}, \mathbf{v}) \leftarrow Q$ *and a substitution* θ iff $\theta = mgu(p(\mathbf{s}, \mathbf{t}) = p(\mathbf{u}, \mathbf{v}))$ and $\mathbf{s} = \mathbf{s}\theta$.
- A derivation step
$$\mathbf{A}, B, \mathbf{C} \stackrel{\theta}{\Longrightarrow}_c (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$$
is called *input-consuming* iff the selected atom B is input-consuming resolvable wrt the input clause c and the substitution θ.
- A derivation is called *input-consuming* iff all its derivation steps are input-consuming.

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

Lemma 3.3 *Let* $p(\mathbf{s}, \mathbf{t})$ *and* $p(\mathbf{u}, \mathbf{v})$ *be two atoms. If there exists an mgu* θ *of* $p(\mathbf{s}, \mathbf{t})$ *and* $p(\mathbf{u}, \mathbf{v})$ *such that* $\mathbf{s}\theta = \mathbf{s}$ *then there exists a* relevant *mgu* ϑ *of* $p(\mathbf{s}, \mathbf{t})$ *and* $p(\mathbf{u}, \mathbf{v})$ *such that* $\mathbf{s}\vartheta = \mathbf{s}$.

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

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

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

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

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

Consider also the query reverse([X1, X2], Zs). The derivation δ of 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]) $\Rightarrow \Box$.

Input-Consuming vs. Delay Declarations

In the introduction we have stated the claim that in most "usual" moded programs using a dynamic selection rule, delay declarations are employed precisely for ensuring the input-consumedness of the derivations. As we have already mentioned, this thesis is already substantiated by the fact that concept of input-consuming resolution is very similar to the selection mechanism employed in Moded Flat GHC [20], and by the arguments in [16].

We now want to add another argument sustaining it.

First, as a large body of literature shows, the vast majority of "usual" programs are actually moded (see for example [5,6] or consider for instance the strictly moded logic programming language Mercury [18]).

Secondly, 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 "right" 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. Typically, delay declarations are used to prevent the selection of an atom until a certain degree of instantiation is reached. This degree of instantiation ensures that the atom is unifiable only with the heads of the "right" 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 $p(\mathbf{s}, \mathbf{t})$ and suppose that it is resolvable with a clause c by means of an input-consuming derivation step. Then, for every instance $\mathbf{s}'$ of $\mathbf{s}$, we have that the atom $p(\mathbf{s}', \mathbf{t})$ is as well resolvable with a clause c by means of an input-consuming derivation step. In other words, no further instantiation of the input positions of $p(\mathbf{s}, \mathbf{t})$ can rule out c as a possible clause for resolving it. Thus c must be one of the "right" clauses for resolving $p(\mathbf{s}, \mathbf{t})$ and we can say that $p(\mathbf{s}, \mathbf{t})$ 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 $p(\mathbf{s}, \mathbf{t})$ is resolvable with c but not via an input-consuming derivation step, then there exists an instance $\mathbf{s}'$ of $\mathbf{s}$, such that $p(\mathbf{s}', \mathbf{t})$ is not resolvable via c. In this case we can say that $p(\mathbf{s}, \mathbf{t})$ is not instantiated enough to know whether c is one of the "right" clauses for resolving it.

3.2 Nicely-Moded Programs

In this the sequel of the paper we'll restrict to programs and queries which are Nicely-Moded. We report here the definition of this concept together with some important properties of nicely-moded programs.

Definition 3.5 [Nicely-Moded]

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

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

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

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

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

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

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

We now start investigating the properties of nicely-moded programs employing input-consuming selection rules.

The following result is due to Smaus [16], and states that the class of programs and queries we are considering is closed under resolution.

Lemma 3.7 *[16] Every resolvent of a nicely-moded query Q and a nicely-moded clause c, where the derivation step is input-consuming and $Var(Q) \cap Var(c) = \emptyset$, is nicely-moded.*

The following Remark (also in [16]) 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 [16] Let the program P and the query $Q := \mathbf{A}, p(\mathbf{s},\mathbf{t}), \mathbf{C}$ be nicely-moded. If $\mathbf{A}, p(\mathbf{s},\mathbf{t}), \mathbf{C} \stackrel{\theta}{\Longrightarrow} \mathbf{A}, \mathbf{B}, \mathbf{C}$ is an input-consuming derivation step with selected atom $p(\mathbf{s},\mathbf{t})$, then $\mathbf{A}\theta = \mathbf{A}$.

We now 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.

Lemma 3.9 *Let the program P and the query Q be nicely-moded. Let $\delta := Q \stackrel{\theta}{\longmapsto} Q'$ be a partial input-consuming derivation of $P \cup \{Q\}$. Then, for all $x \in Var(Q)$ and $x \notin Var(Out(Q))$, $x\theta = x$.*

Proof. Let us first establish the following claim.

Claim 3.10 *Let $\mathbf{z}$ and $\mathbf{w}$ be two variable disjoint sequences of terms such that $\mathbf{w}$ is linear and $\theta = mgu(\mathbf{z} = \mathbf{w})$. If s_1 and s_2 are two variable disjoint terms occurring in $\mathbf{z}$ then $s_1\theta$ and $s_2\theta$ are variable disjoint terms.*

Proof. The result follows from Lemmata 11.4 and 11.5 in [4]. □

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

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

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

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

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

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

(1) $x\theta_1 = x$.

We distinguish two cases.

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

(b) $x \notin Var(\mathbf{s})$. Then, by the choice of x, $x \notin Var(p(\mathbf{s},\mathbf{t}))$. In this case, property (1) follows from the standardization apart condition and relevance of θ_1.

Now we show that

(2) $x\theta_2 = x$.

Again, we distinguish two cases:

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

(d) $x \in Var((\mathbf{A},\mathbf{B},\mathbf{C})\theta_1)$. In this case, in order to prove (2) we show that $x \notin Var(Out((\mathbf{A},\mathbf{B},\mathbf{C})\theta_1))$. The result then follows by the inductive hypothesis.
From the standardization apart condition, relevance of θ_1 and (1), it follows that $Dom(\theta_1) \cap Var(Q) \subseteq Var(\mathbf{t})$.
From the hypothesis that Q is nicely-moded, $Var(\mathbf{t}) \cap Var(Out(\mathbf{A},\mathbf{C})) = \emptyset$. Hence, $Var(Out(\mathbf{A},\mathbf{C}))\theta_1 = Var(Out(\mathbf{A},\mathbf{C}))$. Since $x \notin Var(Out(\mathbf{A},\mathbf{C}))$, this proves that $x \notin Var(Out((\mathbf{A},\mathbf{C})\theta_1))$.
It remains to prove that $x \notin Var(Out(\mathbf{B}\theta_1)$. We distinguish two further cases.

($d1$) $x \notin Var(\mathbf{s})$. In this case, $x \notin Var(Out(\mathbf{B}\theta_1)$ follows immediately by the standardization apart condition and the relevance of θ_1.

($d2$) $x \in Var(\mathbf{s})$. By known results (see [2], Corollary 2.25), there exists two relevant mgu σ_1 and σ_2 such that

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

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

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

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

(4) $Var(z_1\sigma_2) \cap Var(z_2\sigma_2) \neq \emptyset$.

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

$\mathbf{v}\sigma_1$), $\mathbf{t}$ is linear and disjoint from $\mathbf{v}\sigma_1$, (4) contradicts Claim 3.10. □

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

Corollary 3.11 *Let the program P and the one-atom query A be nicely-moded. Let $\delta := A \overset{\theta}{\longmapsto} Q'$ be a partial input-consuming derivation of $P \cup \{A\}$. Then, for all $x \in Var(In(A))$, $x\theta = x$.*

The Left-Switching Lemma

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. This shows that it is always possible to proceed left-to-right to resolve the selected atoms [1].

Lemma 3.12 (Left-Switching) *Let the program P and the query Q_0 be nicely-moded. Let δ be a partial input-consuming derivation of $P \cup \{Q_0\}$ of the form*

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

where

- *Q_n is a query of the form $\mathbf{A}, A, \mathbf{B}, B, \mathbf{C}$,*
- *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}, θ'_{n+1}, θ'_{n+2} and a derivation δ' such that

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

and

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

where

- *δ' is input-consuming,*
- *δ and δ' 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 $B\theta'_{n+1}$.*
- *δ and δ' coincide after the resolvent Q_{n+2}.*

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

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

[1] Notice that this is however 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 resolvable and the one next to it is resolvable.

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

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

Moreover,

$$\begin{aligned}(3)\quad \theta_{n+1}\theta_{n+2} &= mgu\{p(\mathbf{s},\mathbf{t}) = p(\mathbf{s}',\mathbf{t}'), q(\mathbf{u},\mathbf{v}) = q(\mathbf{u}',\mathbf{v}')\}\\ &= \theta_{n+2}\theta'_{n+2}\end{aligned}$$

where

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

We construct the derivation δ' as follows.

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

where

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

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

Observe now that

$$\begin{aligned}\mathbf{u}\theta'_{n+1}\theta'_{n+2} &= \mathbf{u}\theta_{n+2}\theta'_{n+2}, \text{ (by (4))}\\ &= \mathbf{u}\theta_{n+1}\theta_{n+2}, \text{ (by (3))}\\ &= \mathbf{u}\theta_{n+2}, \quad \text{(by (1))}\\ &= \mathbf{u}\theta'_{n+1}, \quad \text{(by (4))}\end{aligned}$$

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

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 [11]).

Corollary 3.13 *Let the program P and the query $Q := \mathbf{A},\mathbf{B}$ be nicely-moded. Suppose that*

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

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

Then there exists a partial input-consuming derivation

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

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

4 Termination

In this section we study the termination of input-consuming derivations of logic programs. To this end we refine the ideas of Bezem [7] and Cavedon [8] who studied the termination of logic programs in a very strong sense, namely with respect to all selection rules, and of Smaus [16] who carachterized terminating input-consuming derivations of well and nicely-moded programs.

4.1 Input Terminating Programs

We first introduce the key notion of this section.

Definition 4.1 [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 [10].

Definition 4.2 [Moded Level Mapping] Let P be a program and $\mathcal{B}_P^{\mathcal{E}}$ 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 $|\ | : \mathcal{B}_P^{\mathcal{E}} \to \mathbf{N}$ from atoms to natural numbers;
- for any $\mathbf{t}$ and $\mathbf{u}$, $|p(\mathbf{s}, \mathbf{t})| = |p(\mathbf{s}, \mathbf{u})|$.

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

The condition $|p(\mathbf{s}, \mathbf{t})| = |p(\mathbf{s}, \mathbf{u})|$ states that the level of an atom is independent from the terms filling in its output positions. There is actually a small yet important difference between this definition and the one in [10]: in [10] the level mapping is defined on ground atoms only. Therefore this is actually an extension of the definition of [10].

Example 4.3 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

$$|\texttt{reverse}(\texttt{Xs}, \texttt{Ys})| = TSize(\texttt{Xs})$$
$$|\texttt{reverse\_acc}(\texttt{Xs}, \texttt{Ys}, \texttt{Zs})| = TSize(\texttt{Xs})$$

where `Xs` is the first input argument.

4.2 Weak Semi-Recurrency

In order to give a sufficient condition for termination, we are going to employ a generalization of the concept of semi-recurrent program defined by Apt and Pedreschi in [4]. First, we need a preliminary definition.

Definition 4.4 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 $p \sqsubseteq q$ in P iff (p, q) is in the reflexive and transitive closure of the relation *refers to*.

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

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

Definition 4.5 [Weak Semi-Recurrency] Let P be a program and $|\ |: \mathcal{B}_P^{\mathcal{E}} \rightarrow \mathbf{N}$ be a moded level mapping.

- A clause of P is called *weakly semi-recurrent with respect to* $|\ |$ iff for every instance of it, $H \leftarrow \mathbf{A}, B, \mathbf{C}$
$$\text{if } Rel(H) \simeq Rel(B) \text{ then } |H| > |B|.$$
- A program P is called *weakly semi-recurrent with respect to* $|\ |$ iff all its clauses are. P is called *weakly semi-recurrent* iff it is weakly semi-recurrent with respect to some moded level mapping $|\ | : \mathcal{B}_P^{\mathcal{E}} \rightarrow \mathbf{N}$.

The notion of weak semi-recurrency differs from the concept of semi-recurrency introduced by Apt and Pedreschi in [4] in two ways. First, our definition provides a condition to hold for every instance of a program clause not only for ground instances as in [4]. Second, we do not require any decreasing neither non increasing of the level mapping between the head H of a rule instance and every corresponding non recursive body atom B: indeed, the additional condition $|H| \geq |B|$ is required in [4] for any body atom B such that $Rel(H) \not\simeq Rel(B)$.

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

Theorem 4.6 *Let P be a nicely-moded program. If P is weakly semi-recurrent then P is input-terminating.*

Proof. It will be obtained from the proof of Theorem 4.10 by setting $R = \emptyset$.□

Example 4.7 Consider the program `MERGE` defined in the Example 3.6. Let $|\ |$ be the moded level mapping for `MERGE` defined by

$$|\texttt{merge(Xs, Ys, Zs)}| = \mathit{TSize}(\texttt{Xs}) + \mathit{TSize}(\texttt{Ys}).$$

It is easy to prove that MERGE is weakly semi-recurrent with respect to the moded level mapping above. By Theorem 4.6, all input-consuming derivations of the program MERGE started with a query $\texttt{merge}(u, s, t)$ where t is linear and variable disjoint from u and s are terminating.

4.3 Modular Termination

This section contains a generalization of Theorem 4.6 to the modular case, as well as the complete proofs for it.

The following lemma is a crucial one.

Lemma 4.8 *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 δ of $P \cup \{Q\}$. Then, there exist $i \in \{1, \ldots, n\}$ and substitution θ such that*

- *there exists an input-consuming derivation δ' of $P \cup \{Q\}$ of the form*

$$\delta' := A_1, \ldots, A_n \stackrel{\theta}{\longmapsto} \mathbf{C}, (A_i, \ldots, A_n)\theta \longmapsto \cdots;$$

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

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

$$\delta := A_1, \ldots, A_n \stackrel{\vartheta}{\longmapsto} \mathbf{C}, \mathbf{D} \longmapsto \cdots$$

where $A_1, \ldots, A_n \stackrel{\vartheta}{\longmapsto} \mathbf{C}, \mathbf{D}$ is a finite prefix of δ which comprises all the A_j-steps for $j \in \{1, \ldots, i-1\}$ and $\mathbf{C}$ results from their resolution. By Corollary 3.13, there exists an infinite input-consuming derivation δ' such that

$$\delta' := A_1, \ldots, A_n \stackrel{\theta}{\longmapsto} \mathbf{C}, (A_i, \ldots, A_n)\theta \stackrel{\theta'}{\longmapsto} \mathbf{C}, \mathbf{D} \longmapsto \cdots$$

where $\vartheta = \theta\theta'$. Let $\delta'' := \mathbf{C}, (A_i, \ldots, A_n)\theta \stackrel{\theta'}{\longmapsto} \mathbf{C}, \mathbf{D} \longmapsto \cdots$. Note that in δ'' the atoms of $\mathbf{C}$ will never be selected and, by Remark 3.8, will never be instantiated. Hence there exists an infinite input-consuming derivation δ''' of $P \cup \{(A_i, \ldots, A_n)\theta\}$ where an infinite number of $A_i\theta$-steps are performed. Again, By Remark 3.8, for every finite prefix of δ''' of the form

$$A_i\theta, (A_{i+1}, \ldots, A_n)\theta \stackrel{\sigma_1}{\longmapsto} \mathbf{D}_1, \mathbf{D}_2 \stackrel{\sigma_2}{\Longrightarrow} \mathbf{D}'_1, \mathbf{D}'_2$$

where $\mathbf{D}_1$ and $\mathbf{D}_2$ are obtained by partially resolving $A_i\theta$ and $(A_{i+1}, \ldots, A_n)\theta$, respectively, and $\mathbf{D}_1, \mathbf{D}_2 \stackrel{\sigma_2}{\Longrightarrow} \mathbf{D}'_1, \mathbf{D}'_2$ is an A_j-step for some $j \in \{i+1, \ldots, n\}$, we have that $\mathbf{D}'_1 = \mathbf{D}_1$. Hence, from the hypothesis that there is an infinite number of $A_i\theta$-steps in δ'', it follows that there exists an infinite input-consuming derivation of $P \cup \{A_i\theta\}$. □

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 4.9 *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 4.10 *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 weakly semi-recurrent with respect to a moded level mapping $|\ |: \mathcal{B}_P^{\mathcal{E}} \to \mathbf{N}$.*

Then $P \cup R$ is input-terminating.

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

We can now prove our theorem. By Corollary 4.9, it is sufficient to prove that for every 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 δ be an infinite input-consuming derivation of $P \cup R \cup \{A\}$ such that A is defined in P. Then

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

where $H \leftarrow B_1, \ldots, B_n$ is the input clause used in the first derivation step and $\theta_1 = mgu(A = H)$. Clearly, $(B_1, \ldots, B_n)\theta_1$ has an infinite input-consuming derivation in $P \cup R$. By Lemma 4.8, for some $i \in \{1, \ldots, n\}$ and for some substitution θ_2,

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

We proceed by proving that (2) is a contradiction.

Let $\theta = \theta_1\theta_2$. Note that $H\theta \leftarrow (B_1, \ldots, B_n)\theta$ is an instance of a clause of P. The proof follows by induction on $\langle dep_P(Rel(A)), |A| \rangle$ 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 $dep_P(Rel(A)) = 0$ and $|A| = 0$. 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 point (2) above.

Induction step. Let $dep_P(Rel(A)) > 0$ and $|A| > 0$. We distinguish two cases:

(*a*) $Rel(H) \sqsupset Rel(B_i)$,
(*b*) $Rel(H) \simeq Rel(B_i)$.

In case (*a*) we have that $dep_P(Rel(A)) = dep_P(Rel(H\theta)) > dep_P(Rel(B_i\theta))$. So, $\langle dep_P(Rel(A)), |A| \rangle = \langle dep_P(Rel(H\theta)), |H\theta| \rangle \succ \langle dep_P(Rel(B_i\theta)), |B_i\theta| \rangle$. In case (*b*), from the hypothesis that P is weakly semi-recurrent w.r.t. $|\ |$, it follows that $|H\theta| \succ |B_i\theta|$. Consider the partial input-consuming derivation $A \overset{\theta}{\longmapsto} \mathbf{C}, (B_i, \ldots, B_n)\theta$, by Corollary 3.11 and the fact that $|\ |$ is a moded level mapping, we have that $|A| = |A\theta| = |H\theta|$. Hence, $\langle dep_P(Rel(A)), |A| \rangle = \langle dep_P(Rel(H\theta)), |H\theta| \rangle \succ \langle dep_P(Rel(B_i\theta)), |B_i\theta| \rangle$.
In both cases, the contradiction follows by the inductive hypothesis. □

Example 4.11 The program `FLATTEN` using difference-lists is nicely-moded in the modes described below (with "\" replaced by ",").

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

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

Consider the moded level mapping for `FLATTEN` defined by

$$|\texttt{flatten(Xs, Ys)}| = TSize(\texttt{Xs})$$
$$|\texttt{flatten\_dl(Xs, Ys \ Zs)}| = TSize(\texttt{Xs}).$$

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

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 4.6 and 4.10) hold also in the case that programs and queries are *permutation nicely-moded* [17], 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 FLATTEN as it is presented in [2], i.e.,

$$\begin{aligned}
&\texttt{flatten}(\texttt{Xs},\texttt{Ys}) \leftarrow \texttt{flatten_dl}(\texttt{Xs},\texttt{Ys} \setminus [\,]).\\
&\texttt{flatten_dl}([\,],\texttt{Ys} \setminus \texttt{Ys}).\\
&\texttt{flatten_dl}(\texttt{X},[\texttt{X}|\texttt{Xs}] \setminus \texttt{Xs}) \leftarrow \texttt{constant}(\texttt{X}),\texttt{X} \neq [\,].\\
&\texttt{flatten_dl}([\texttt{X}|\texttt{Xs}],\texttt{Ys} \setminus \texttt{Zs}) \leftarrow \texttt{flatten_dl}(\texttt{X},\texttt{Ys} \setminus \texttt{Y1s}),\\
&\qquad\qquad\qquad\qquad\qquad\quad \texttt{flatten_dl}(\texttt{Xs},\texttt{Y1s} \setminus \texttt{Zs}).
\end{aligned}$$

where the atoms in the body of the last clause are permuted with respect to the version of the Example 4.11.

5 Applicability

In this section we report the results that we obtained by applying the termination criterion presented in this paper to several benchmarks from well-known collections.

In Table 1 benchmarks from Apt's collection are considered (see [2] and [4]). Benchmarks 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 benchmarks from Lindenstrauss's collection (see the URL: http://www.cs.huji.ac.il/ naomil). Finally, Table 4 concerns with benchmarks from 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 benchmark we specify the name and the modes of the main procedure. In the tables below **NM** stays for nicely-moded and the corresponding entry is yes when we can find some modes for the subprocedures with respect to which the whole program is nicely moded. The next to columns refer to such a modes: **IT** stays for input terminating and **WSR** stays for weakly semi-recurrent.

6 Conclusion

We presented a method for proving termination of programs and queries which are (permutation) nicely-moded. Since input-consuming derivations do not use any fixed selection rule, our method can be applied for proving termination of programs which employ a dynamic selection rule. Our results strictly improve on [16] in the fact that we drop the condition that programs and

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

Table 1
Benchmarks from Apt's Collection

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 [16] apply only if `t` is a ground term.

As side-effect of our investigation, we also showed that for this class of programs one side of the well-known Switching Lemma holds.

Applicability and effectiveness of our approach has been demonstrated by applying it to several benchmarks for most of which we can prove weakly semi-recurrency.

Automatization of our method 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's [9] 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.

	NM	IT	WSR		NM	IT	WSR
applast(In,In,Out)	yes	yes	yes	match_app(_,In)	yes	yes	yes
applast(Out,_,_)	yes	no		match_app(In,Out)	yes	no	
applast(_,Out,_)	yes	no		max_lenth(In,Out,Out)	yes	yes	yes
contains(_,In)	yes	yes	yes	meno_solve(In,Out)	yes	yes	no
contains(In,Out)	yes	no		power(In,In,In,Out)	yes	yes	yes
depth(In,In)	yes	yes	yes	prune(In,_)	yes	yes	yes
depth(In,Out)	yes	yes	no	prune(_,In)	yes	yes	yes
depth(Out,In)	yes	no		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		solve(_,_,_)	yes	no	
liftsolve(In,Out)	yes	no		ssupply(In,In,Out)	yes	yes	yes
liftsolve(Out,In)	yes	no		trace(In,In,Out)	yes	yes	yes
liftsolve(In,In)	yes	yes	yes	transpose(_,In)	yes	yes	yes
match(In,_)	yes	no		transpose(In,Out)	yes	no	
match(_,In)	yes	yes	no	unify(In,In,Out)	yes	no	

Table 2
Benchmarks from DPPD's Collection

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

Table 3
Benchmarks from Lindenstrauss's Collection

References

[1] Apt, K. R., *Introduction to Logic Programming*, in J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and

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

Table 4
Benchmarks from Hermenegildo's Collection

Semantics, Elsevier, Amsterdam and The MIT Press, Cambridge, (1990), 495–574

[2] Apt, K. R., "From Logic Programming to Prolog", Prentice Hall, 1997

[3] Apt, K. R. and Luitjes, I., *Verification of logic programs with delay declarations*, in A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, (AMAST'95)*, Lecture Notes in Computer Science, Berlin, 1995

[4] Apt, K. R. and Pedreschi, D., *Modular termination proofs for logic and pure Prolog programs* in G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994

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

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

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

[8] Cavedon, L., *Continuity, consistency and completeness properties for logic programs*, in G. Levi and M. Martelli, editors, International Conference on Logic Programming, pages 571–584. MIT press, 1989

[9] Decorte, S. and De Schreye, D. and Fabris, M., *Automatic inference of norms: a missing link in automatic termination analysis*, in D. Miller, editor, Proc. Tenth International Logic Programming Symposium, number 526 in Lecture Notes in Computer Science, pages 420–436. Springer-Verlag, 1993

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

[11] Etalle, S., and Gabbrielli, M., and Marchiori, E., *A Transformation System for CLP with Dynamic Scheduling and CCP*, in C. Consel, editor, ACM–SIGPLAN Symposium on Partial Evaluation and Semantic Based Program Manipulation. ACM Press, 1997

[12] Hill, P. M., and Lloyd, J. W., "The Gödel programming language" The MIT Press, 1994

[13] Lloyd, J. W., "Foundations of Logic Programming", Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, 1987, Second edition

[14] Marchiori, E. and Teusink, F., *Proving termination of logic programs with delay declarations*, in J. Lloyd, editor, Proc. Twelfth International Logic Programming Symposium. MIT Press, 1995.

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

[16] Smaus, J. G., *Proving termination of input-consuming logic programs*, in D. De Schreye, editor, 16th International Conference on Logic Programming. MIT press, 1999.

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

[18] Somogyi, Z., Henderson, F. and Conway, T., *Mercury: an efficient purely declarative logic programming language*, in Australian Computer Science Conference, 1995. available at http://www.cs.mu.oz.au/mercury/papers.html

[19] Ueda, K., *Guarded Horn Clauses, a parallel logic programming language with the concept of a guard*, in M. Nivat and K. Fuchi, editors, Programming of Future Generation Computers, pages 441–456. North Holland, Amsterdam, 1988

[20] Ueda, K. and Morita, M., *Moded flat ghc and its message-oriented implementation technique*, New Generation Computing, 13(1) (1994) 3–43

[21] Wallace, M. and Veron, A., *Two problems – two solutions: One system – ECLiPSe*, in Proceedings IEE Colloquium on Advanced Software Technologies for Scheduling, London, April 1993