

# Termination of Simply Moded Logic Programs with Dynamic Scheduling

ANNALISA BOSSI

Università Ca' Foscari di Venezia

SANDRO ETALLE

Universiteit Twente and CWI Amsterdam

SABINA ROSSI\*

Università Ca' Foscari di Venezia

JAN-GEORG SMAUS

Universität Freiburg

March 6, 2003

## Abstract

In logic programming, *dynamic scheduling* indicates the feature by means of which the choice of the atom to be selected at each resolution step is done at runtime and does not follow a fixed selection rule such as the left-to-right one of Prolog. *Input consuming derivations* were introduced to model dynamic scheduling while abstracting from the technical details. In this article, we provide a sufficient and necessary criterion for termination of input consuming derivations of simply moded logic programs. The termination criterion we propose is based on a denotational semantics for partial derivations which is defined in the spirit of model-theoretic semantics previously proposed for left-to-right derivations.

## 1 Introduction

Algorithm = Logic + Control.

The programmer should be responsible for the logic part. The control should be taken care of by the logic programming system.

In reality, logic programming is far from this ideal. Without the programmer being aware of the control and writing programs accordingly, most logic programs would be hopelessly inefficient or even non-terminating.

---

\*Contact author's address: Sabina Rossi, Dipartimento di Informatica, Università Ca' Foscari di Venezia, via Torino 155, 30172 Venezia, Italy.

This article is accepted for publication in the ACM Transactions on Computational Logic, and is based on an earlier conference paper [8].

One aspect of control in logic programs is the *selection rule*, stating which atom in a query is selected in each derivation step. The standard selection rule in logic programming languages is the fixed left-to-right rule of Prolog. While this rule is appropriate for many applications, there are situations, e.g., in the context of parallel executions or the test-and-generate paradigm, that require a more flexible control mechanism, namely, *dynamic scheduling*, where the selectable atoms are determined at runtime.

To demonstrate that on the one hand, the left-to-right selection rule is sometimes inappropriate, but that on the other hand, the selection mechanism must be controlled in some way, consider the following programs APPEND and IN\_ORDER:

```
% append(Xs,Ys,Zs) ← Zs is the result of concatenating the lists Xs and Ys
append([H|Xs],Ys,[H|Zs]) ← append(Xs,Ys,Zs).
append([],Ys,Ys).

% in_order(Tree,List) ← List is an ordered list of the nodes of Tree
in_order(tree(Label,Left,Right),Xs) ←
    in_order(Left,Ls),
    in_order(Right,Rs),
    append(Ls,[Label|Rs],Xs).
in_order(void,[]).
```

together with the query

```
Q: read_tree(Tree), in_order(Tree,List), write_list(List).
```

where `read_tree` and `write_list` are defined elsewhere. If `read_tree` cannot read the whole tree at once – say, it receives the input from a stream – it would be nice to be able to run the “processes” `in_order` and `write_list` on the available input. This can only be done properly if one uses a dynamic selection rule (Prolog’s rule would call `in_order` only after `read_tree` has finished, while other fixed rules would immediately diverge and/or have an unwanted behavior<sup>1</sup>. Such a mechanism is provided in modern logic programming languages in the form of *delay declarations* (also called *when declarations*) [19]. In the above program, in order to avoid nontermination one can declare that predicates `in_order`, `append` and `write_list` can be selected only if their first argument is not just a variable. Formally,

```
delay in_order(T,_) until nonvar(T).
delay append(Ls,_,_) until nonvar(Ls).
delay write_list(Ls,_) until nonvar(Ls).
```

---

<sup>1</sup>For instance, the fixed rule that selects always the second atom in a clause body, and that selects the first one only when the body contains only one atom can lead to nontermination, as the query `in_order(Tree, List)` can easily diverge. The same applies to the rule that always selects the rightmost atom in a query, with the extra problem that `write_list(List)` would be called with a non-instantiated argument: if `write_list` is non-backtrackable (as many IO predicates are) this would imply that this selection rule yields a wrong output.

These declarations prevent `in_order`, `append` and `write_list` from being selected “too early”, i.e., when their arguments are not “sufficiently instantiated”. Note that instead of having interleaving “processes”, one can also select several atoms in *parallel*, as long as the delay declarations are respected. This approach to parallelism has been first proposed by Naish [21] and – as observed by Apt and Luitjes [4] – “has an important advantage over the ones proposed in the literature in that it allows us to parallelize programs written in a large subset of Prolog by merely adding to them delay declarations, so *without modifying* the original program”.

Compared to other mechanisms for user-defined control, e.g., using the cut operator in connection with built-in predicates that test for the instantiation of a variable (`var` or `ground`), delay declarations are more compatible with the declarative character of logic programming. Nevertheless, many important declarative properties that have been proven for logic programs do not apply to programs with delay declarations. The problem is mainly related to the fact that delay declarations might cause *deadlock* situations, in which no atom in the query respects its delay declaration. For instance, for such programs the well-known equivalence between model-theoretic and operational semantics does not hold. As an example, consider the query `append(X,Y,Z)` with the execution mechanism described above: it does not succeed (it *deadlocks*) and this is in contrast with the fact that (infinitely many) instances of `append(X,Y,Z)` are contained in the least Herbrand model of `APPEND`.

In order to provide a characterization of dynamic scheduling that is reasonably abstract and hence amenable to semantic analysis, Smaus [26] introduced *input consuming derivations*. The definition of input consuming program relies on the concept of *mode*. A *moded program* is a program in which each atom’s arguments are partitioned into *input* and *output* ones. Output arguments are those which can be produced by the computation process, while input arguments should be only consumed. Roughly speaking, in an input consuming program only atoms whose input arguments are not instantiated through the unification step are allowed to be selected.

In [8] we have demonstrated that – in many cases – the adoption of the “natural” delay declarations is equivalent to considering only input consuming derivations. This is the case – for instance – for the programs mentioned above (together with their natural mode `append(I,I,O)`<sup>2</sup>, `in_order(I,O)`): under normal circumstances, the adoption of the just stated delay declarations enforces nothing but a restriction to input consuming derivations. In both cases, whether we consider selection rules defined in terms of a programming language construct such as delay declarations, or whether we consider input consuming derivations, we speak of *LP with dynamic scheduling*.

**The contribution** The adoption of dynamic scheduling has as ultimate goal that of ensuring the termination of the program under construction, by prevent-

---

<sup>2</sup>In this mode, the first two positions are considered *input positions*, while the rightmost one is an *output* one.

ing possible diverging derivations. Nevertheless, while for pure Prolog programs (i.e., logic programs employing the fixed leftmost selection rule) there exist results characterizing when a program is terminating [5], no such characterization has been found yet for programs with dynamic scheduling. In addition, there are relatively few contributions concerning the termination of programs with dynamic scheduling.

In this paper we tackle the problem of establishing the termination of input consuming logic programs. For this, we restrict our attention to the class of *simply moded* programs, which are programs that are, in a well-defined sense, consistent with respect to the intended producer/consumer behavior (modes). As also shown by the benchmarks reported in [8], most practical programs are simply moded.

The main contribution of this paper is a full characterization of the class of simply moded input terminating logic programs, i.e., simply moded programs whose input consuming derivations starting from a simply moded query are finite.

In order to provide such a result, we had to define a new declarative semantics that allows us to capture the inter-argument relationships of input-consuming programs. Since dynamic scheduling also allows for parallelism, in this context it is important to model the result of *partial* (i.e., incomplete) derivations. In fact, partial computed answer substitutions may activate suspended processes by means of interleaving therefore influencing the termination of the system. To capture this appropriately, we defined a denotational semantics modeling computed answer substitutions of incomplete derivations and enjoying a model-theoretical reading as well as a natural bottom-up constructive definition. We demonstrate that this semantics is correct and fully abstract with respect to the computed substitutions of partial derivations.

A first attempt to tackle this problem has been presented in [27] and extended in [7] where we defined the class of *input terminating* programs, i.e., programs whose input consuming derivations are finite, and characterize the subclass of simply moded *quasi recurrent* programs. It is worth stressing that this latter class includes only programs whose termination does not depend on the so-called *inter-argument relationships* and therefore it does not include programs such that `quicksort`, `transpose`, `list_tree`. Further comparisons are reported in the concluding section.

**Structure of the paper** The rest of this paper is organized as follows. The next section introduces some preliminaries. Section 3 shows some useful properties of input consuming derivations. Section 4 provides a result on denotational semantics for partial input consuming derivations. Section 5 provides a sufficient and necessary criterion for termination of programs using input consuming partial derivations. In Section 6 we report additional examples. Section 7 discusses related work and draws some conclusions.

## 2 Preliminaries

The reader is assumed to be familiar with the terminology and the basic results of logic programs and their semantics [1, 2, 15]. In this section we introduce a few notions that will be used in the sequel.

### 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*  $\mathcal{V}$ . 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. A *substitution*  $\theta$  is a mapping from  $\mathcal{V}$  to  $\mathcal{T}$ . Given a *substitution*  $\sigma = \{x_1/t_1, \dots, x_n/t_n\}$ , we say that  $\{x_1, \dots, x_n\}$  is its *domain* (denoted by  $Dom(\sigma)$ ), and  $Var(\{t_1, \dots, t_n\})$  is its *range* (denoted by  $Ran(\sigma)$ ). Note that  $Var(\sigma) = Dom(\sigma) \cup Ran(\sigma)$ . We denote by  $\epsilon$  the empty substitution:  $Dom(\epsilon) = Ran(\epsilon) = \emptyset$ . Given a *substitution*  $\sigma$  and a syntactic object  $E$ , we denote by  $\sigma|_E$  the restriction of  $\sigma$  to the variables in  $Var(E)$ , i.e.,  $\sigma|_E(x) = \sigma(x)$  if  $x \in Var(E)$ , otherwise  $\sigma|_E(x) = x$ . If  $t_1, \dots, t_n$  is a permutation of  $x_1, \dots, x_n$  then we say that  $\sigma$  is a *renaming*. The *composition* of substitutions is denoted by juxtaposition, i.e.,  $\theta\sigma(x) = \sigma(\theta(x))$ . 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 say that  $t$  is a *variant* of  $t'$ , written  $t \approx t'$ , if  $t$  and  $t'$  are instances of each other. A substitution  $\theta$  is a *unifier* of terms  $t$  and  $t'$  if  $t\theta = t'\theta$ . We denote by  $mgu(t, t')$  any *most general unifier* (*mgu*, in short) of  $t$  and  $t'$ . An *mgu*  $\theta$  of terms  $t$  and  $t'$  is called *relevant* iff  $Var(\theta) \subseteq Var(t) \cup Var(t')$ .

### 2.2 Programs and Derivations

Let  $\mathcal{P}$  be a finite set of *predicate symbols*. An *atom* is an object of the form  $p(t_1, \dots, t_n)$  where  $p \in \mathcal{P}$  is an  $n$ -ary predicate symbol and  $t_1, \dots, t_n \in \mathcal{T}$ . Given an atom  $A$ , we denote by  $Rel(A)$  the predicate symbol of  $A$ . A *query* is a finite, possibly empty, sequence of atoms  $A_1, \dots, A_m$ . The empty query is denoted by  $\square$ . Following the convention adopted in [2], we use boldface characters to denote sequences of objects: so, for instance,  $\mathbf{t}$  denotes a sequence of terms, while  $\mathbf{B}$  is a query (i.e., a possibly empty sequence 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 simply written  $H$  and is called a *unit clause*. A *program* is a finite set of clauses. We denote atoms by  $A, B, H, \dots$ , queries by  $Q, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{R}, \dots$ , clauses by  $c, d, \dots$ , 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*  $\theta$ . The query  $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$  is called a *resolvent* of  $\mathbf{A}, B, \mathbf{C}$  and  $c$  with selected atom  $B$  and *mgu*  $\theta$ . A *derivation step* is denoted by  $\mathbf{A}, B, \mathbf{C} \xrightarrow{\theta}_{P, c} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ . The clause  $H \leftarrow \mathbf{B}$  is called its *input clause*. The atom  $B$  is called the *selected atom* of  $\mathbf{A}, B, \mathbf{C}$ .

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

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

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 by  $\delta : Q_0 \xrightarrow{\theta} Q_n$  with  $\theta = \theta_1 \cdots \theta_n$ , we say that  $\delta$  is a *partial derivation* and  $\theta$  is a *partial computed answer substitution of  $P \cup \{Q_0\}$* . If  $\delta$  is maximal and ends with the empty query then  $\theta$  is called *computed answer substitution (c.a.s., for short)*. In this case we say that the derivation is *successful*. A finite derivation is called *failed* if it ends with a non-empty query  $Q$  and there is no input clause whose head unifies with the selected atom of  $Q$ . The length of a (partial) derivation  $\delta$ , denoted by  $len(\delta)$ , is the number of derivation steps in  $\delta$ .

The following definition of **D**-step is due to Smaus [26].

**Definition 2.1 ((D-step))**

- Let  $\mathbf{A}, \mathbf{B}, \mathbf{C} \xrightarrow{\theta} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$  be a derivation step. We say that each atom in  $\mathbf{B}\theta$  is a direct descendant of  $\mathbf{B}$ , and for each atom  $E$  in  $(\mathbf{A}, \mathbf{C})$ ,  $E\theta$  is a direct descendant of  $E$ . We say that  $E$  is a descendant of  $F$  if the pair  $(E, F)$  is in the reflexive, transitive closure of the relation is a direct descendant of.
- Consider a derivation  $Q_0 \xrightarrow{\theta_1} \cdots \xrightarrow{\theta_i} Q_i \cdots \xrightarrow{\theta_j} Q_j \xrightarrow{\theta_{j+1}} Q_{j+1} \cdots$ . We say that  $Q_j \xrightarrow{\theta_{j+1}} Q_{j+1} \cdots$  is a **D**-step if  $\mathbf{D}$  is a subquery of  $Q_i$  and the selected atom in  $Q_j$  is a descendant of an atom in  $\mathbf{D}$ .

Intuitively, a **D**-step occurring in a derivation  $\delta$  is a derivation step that concerns the derivation of the subquery  $\mathbf{D}$  of some query in  $\delta$ .

### 2.3 Moded Programs

Modes are a common tool for verification. 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. A program (resp. a query, an atom) is called *moded* whenever it is provided with a mode.

**Definition 2.2 ((mode))** A mode for a predicate symbol  $p$  of arity  $n$ , is a function  $m_p$  from  $\{1, \dots, n\}$  to  $\{I, O\}$ .

If  $m_p(i) = I$  (resp.  $O$ ), we say that  $i$  is an *input* (resp. *output*) position of  $p$  (with respect to  $m_p$ ). In examples, we often indicate the mode by writing the atom  $p(m_p(1), \dots, m_p(n))$ , e.g., `append(I, I, O)`.

We assume that each predicate symbol has a unique mode associated to it; multiple modes may be obtained by simply renaming the predicates. We denote by  $In(Q)$  (resp.  $Out(Q)$ ) the sequence of terms filling in the input (resp. output) positions of predicates in  $Q$ . Moreover, when writing an atom as  $p(\mathbf{s}, \mathbf{t})$ , we are indicating that  $\mathbf{s}$  is the sequence of terms filling in its input positions and  $\mathbf{t}$  is the sequence of terms filling in its output positions.

In the literature, several correctness criteria concerning the modes have been proposed, e.g., nicely and well-modedness [2]. In the sequel of the paper we will restrict ourselves to programs and queries which are *simply moded* [3].

**Definition 2.3 ((simply moded))** *A clause  $H \leftarrow B_1, \dots, B_n$  is simply moded if*

- $Out(B_1, \dots, B_n)$  is a linear vector of variables,
- $Var(In(H)) \cap Var(Out(B_1, \dots, B_n)) = \emptyset$ ,
- for all  $i \in [1..n]$ ,  $Var(Out(B_i)) \cap Var(In(B_1, \dots, B_i)) = \emptyset$ .

*A query  $\mathbf{B}$  is simply moded if the clause  $q \leftarrow \mathbf{B}$  is simply moded, where  $q$  is any variable-free atom. A program is simply moded if all of its clauses are.*

Thus a clause is simply moded if the output positions of body atoms are filled in by distinct variables, and every variable occurring in an output position of a body atom does not occur in an earlier input position. In particular, every unit clause is simply moded.

#### Example 2.4

- The program APPEND of the introduction in the mode `append(I, I, O)` is simply moded.
- The following program REVERSE with accumulator in the mode defined below is simply moded.

```

mode reverse(I, O).
mode reverse_acc(I, O, I)

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

```

In Definition 2.3, if we drop the condition that output positions of body atoms are filled in by variables then we obtain the definition of *nicely moded* programs and queries. Therefore the class of simply moded programs is properly contained in the class of nicely moded programs.

## 2.4 Input Consuming Derivations

The notion of input consuming derivation was introduced in [26] as formalism for describing dynamic scheduling in an abstract way and is defined as follows.

**Definition 2.5** ((input consuming))

- A derivation step  $\mathbf{A}, B, \mathbf{C} \xrightarrow{\theta} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$  is input consuming if  $In(B)\theta = In(\mathbf{B})$ .
- A derivation is input consuming if all its derivation steps are input consuming.

**Example 2.6** Consider the program REVERSE with accumulator in the modes defined above. The derivation  $\delta$  of  $\text{REVERSE} \cup \{\text{reverse}([X1, X2], Zs)\}$  depicted below is input consuming.

$$\delta: \text{reverse}([X1, X2], Zs) \Rightarrow \text{reverse\_acc}([X1, X2], Zs, [ ]) \Rightarrow \text{reverse\_acc}([X2], Zs, [X1]) \Rightarrow \text{reverse\_acc}([ ], Zs, [X2, X1]) \Rightarrow \square.$$

Allowing only input consuming derivations is a form of dynamic scheduling, since whether or not an atom can be selected depends on its degree of instantiation at runtime. Given a non-empty query, if no atom is resolvable via an input consuming derivation step and no failure arises, then we say that the query *deadlocks*.

In previous works many important properties of input consuming derivations have been proven by considering various classes of programs and queries. In this article, we focus on the simply moded ones, but we consider results that hold only for this class as well as results that hold for larger classes, e.g., the class of nicely moded programs and queries.

The following lemma is a straightforward consequence of [4, Lemma 30].

**Lemma 2.7** *In a input consuming derivation, every resolvent of a nicely (resp. simply) moded query and a nicely (resp. simply) moded clause is nicely (resp. simply) moded.*

The following result has been proven in [7] for nicely moded programs and queries. It states that the only variables of a nicely moded query that can be “affected” through the computation of an input consuming derivation with a nicely moded program are those occurring in some output positions.

**Lemma 2.8** *Let the program  $P$  and the query  $Q$  be nicely moded. Let also  $Q \xrightarrow{\theta} Q'$  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 next lemma shows that input consuming derivations are invariant under renaming.



**Lemma 2.9** *Let  $P$  be a program,  $Q$  be a query and  $\delta : Q \xrightarrow{\theta} Q'$  be a (partial) input consuming derivation of  $P \cup \{Q\}$ . Then, for any renaming  $\rho$  there exists a (partial) input consuming derivation  $\delta' : Q\rho \xrightarrow{\vartheta} Q'\rho$  where  $\vartheta = \rho^{-1}\theta\rho$ .*

**Proof.** *First notice that*

- (1) *if  $c$  is a clause renamed apart with respect to a query  $Q$  then  $c\rho$  is renamed apart with respect to  $Q\rho$ ,*
- (2) *if  $A$  and  $H$  are unifiable with mgu  $\theta$  then  $A\rho$  and  $H\rho$  are unifiable with mgu  $\rho^{-1}\theta\rho$ ,*
- (3) *if  $\text{In}(A\theta) = \text{In}(A)$  then  $\text{In}(A\rho\rho^{-1}\theta\rho) = \text{In}(A\theta\rho) = \text{In}(A\rho)$ .*

*Consider now the list of clauses  $c_1, \dots, c_n$  employed in  $\delta$  and the corresponding list of mgus,  $\theta_1, \dots, \theta_n$ , where  $\theta = \theta_1 \cdot \dots \cdot \theta_n$ . By (1) and (2) we can construct a derivation  $\delta'$  starting from  $Q\rho$  with input clauses  $c_1\rho, \dots, c_n\rho$  and unifiers  $\rho^{-1}\theta_1\rho, \dots, \rho^{-1}\theta_n\rho$ . We obtain a derivation  $\delta' : Q\rho \xrightarrow{\vartheta} Q'\rho$  which is input consuming (by point (3) above) and whose computed answer substitution is  $\vartheta = (\rho^{-1}\theta_1\rho)(\rho^{-1}\theta_2\rho) \cdots (\rho^{-1}\theta_n\rho) = \rho^{-1}\theta_1 \cdots \theta_n\rho = \rho^{-1}\theta\rho$ .  $\square$*

We recall below the Left-Switching Lemma that has been proven in [7].

**Lemma 2.10 ((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*

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

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  $c_{n+1}$  with respect to  $B$ ,
- $Q_{n+2}$  is a resolvent of  $Q_{n+1}$  and  $c_{n+2}$  with respect to  $A\theta_{n+1}$ .

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

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

and

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

where  $\delta'$  is input consuming and

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

Lemma 2.10 suggests the following definition which introduces a way of ordering the selected atoms in an input consuming derivation of a simply moded query.

**Definition 2.11** *A partial derivation  $\delta : Q_0 \Longrightarrow Q_1 \cdots \Longrightarrow Q_n$  of a simply moded query  $Q_0$  proceeds left-to-right if whenever an atom  $B$  is selected in a resolvent  $Q_i : \mathbf{A}, B, \mathbf{C}$  then no  $\mathbf{A}$ -step is performed in the rest of the derivation  $Q_{i+1} \longrightarrow Q_n$ .*

The next corollary is an immediate consequence of the Left-Switching Lemma. Intuitively, it says that any resolvent in an input consuming derivation of a simply moded query can be obtained by an input consuming derivation which proceeds left-to-right.

**Corollary 2.12** *Let the program  $P$  and the query  $\mathbf{A}, \mathbf{B}$  be simply moded. Suppose that  $\delta : \mathbf{A}, \mathbf{B} \xrightarrow{\theta} \mathbf{C}$  is a (partial) input consuming derivation of  $P \cup \{\mathbf{A}, \mathbf{B}\}$ . Then there exist  $\mathbf{C}_1$  and  $\mathbf{C}_2$  and a (partial) input consuming derivation  $\delta'$  that proceeds left-to-right of the form*

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

*such that  $len(\delta) = len(\delta')$ ,  $\mathbf{C} = \mathbf{C}_1, \mathbf{C}_2$ ,  $\theta = \theta_1\theta_2$ , all the  $\mathbf{A}$ -steps are performed in the prefix  $\mathbf{A}, \mathbf{B} \xrightarrow{\theta_1} \mathbf{C}_1, \mathbf{B}\theta_1$ , all the  $\mathbf{B}$ -steps are performed in the suffix  $\mathbf{C}_1, \mathbf{B}\theta_1 \xrightarrow{\theta_2} \mathbf{C}_1, \mathbf{C}_2$  and  $\mathbf{C}_1\theta_2 = \mathbf{C}_1$ .*

**Proof.** By repeatedly applying the Left-Switching Lemma,  $\delta$  is equivalent to a derivation  $\delta'$  in which all the  $\mathbf{A}$ -steps are carried out before the  $\mathbf{B}$ -steps.  $\mathbf{C}_1, \mathbf{B}\theta_1$  is the resolvent that we obtain after carrying out the  $\mathbf{A}$ -steps. By the persistence of simply-moded queries (Lemma 2.7),  $\mathbf{C}_1, \mathbf{B}\theta_1$  is simply-moded. Therefore, by Lemma 2.8,  $\theta_2$  has no influence on  $\mathbf{C}_1$  (i.e.,  $\mathbf{C}_1\theta_2 = \mathbf{C}_1$ ).  $\square$

### 3 Simply Local Substitutions

When input consuming derivations are applied to simply moded programs and queries, important properties follow from the way clauses become instantiated during the derivation process. We introduce *simply local* substitutions to reflect this instantiation mechanism. A clause  $c := H \leftarrow B_1, \dots, B_n$  becomes instantiated by its “caller” (the atom that is resolved using  $c$ ) and its “callees” (the clauses used to resolve the body atoms of  $c$ ). Thus, a simply local substitution is defined as the composition of several substitutions,  $\sigma_0, \sigma_1 \dots, \sigma_n$ , one for each atom in the given clause, such that  $\sigma_0$  binds the input variables of the head of the clause, and each  $\sigma_i$  ( $i > 0$ ) creates a binding between the output variables and the input terms of  $B_i$  (instantiated by the previous substitutions  $\sigma_0, \dots, \sigma_{i-1}$ ). The definition involves variable sets  $v_0, v_1, \dots, v_n$ . Intuitively, the variables in  $v_0$  come from the “caller” and the variables in  $v_1, \dots, v_n$  come from the “callees”.

**Definition 3.1 ((simply local substitution))** Let  $\theta$  be a substitution. We say that  $\theta$  is simply local with respect to the clause  $H \leftarrow B_1, \dots, B_n$  if there exist substitutions  $\sigma_0, \sigma_1, \dots, \sigma_n$  and disjoint sets of fresh (with respect to  $c$ ) variables  $v_0, v_1, \dots, v_n$  such that  $\theta = \sigma_0 \sigma_1 \dots \sigma_n$  where

- $Dom(\sigma_0) \subseteq Var(In(H))$  and  $Ran(\sigma_0) \subseteq v_0$ ,
- for  $i \in [1..n]$ ,  
 $Dom(\sigma_i) \subseteq Var(Out(B_i))$  and  $Ran(\sigma_i) \subseteq Var(In(B_i)\sigma_0\sigma_1\dots\sigma_{i-1}) \cup v_i$ .

The substitution  $\theta$  is simply local with respect to a query  $\mathbf{B}$  if  $\theta$  is simply local with respect to the clause  $q \leftarrow \mathbf{B}$  where  $q$  is any variable-free atom.

Given a simply local substitution  $\theta$ , we call the set of fresh variables of  $\theta$  the union of the sets  $v_0, v_1, \dots, v_n$  introduced in the above definition.

Note that in the case of a simply local substitution with respect to a query,  $\sigma_0$  is the empty substitution, since  $Dom(\sigma_0) \subseteq Var(q)$  where  $q$  is an (imaginary) variable-free atom.

**Example 3.2** Consider the program APPEND with the modes  $append(I, I, O)$  and its recursive clause

$$c : append([H|Xs], Ys, [H|Zs]) \leftarrow append(Xs, Ys, Zs).$$

The substitution  $\theta = \{Xs/\[], Ys/W, Zs/W\}$  is simply local with respect to  $c$ . In fact, let  $\sigma_0 = \{Xs/\[], Ys/W\}$  and  $\sigma_1 = \{Zs/W\}$  be two substitutions and  $v_0 = \{W\}$  and  $v_1 = \emptyset$  be two disjoint sets of fresh (with respect to  $c$ ) variables. According to Definition 3.1, we have that  $\theta = \sigma_0 \sigma_1$ ,  $Dom(\sigma_0) \subseteq Var(In(append([H|Xs], Ys, [H|Zs])))$ ,  $Ran(\sigma_0) \subseteq v_0$ ,  $Dom(\sigma_1) \subseteq Var(Out(append(Xs, Ys, Zs)))$  and also  $Ran(\sigma_1) \subseteq Var(In(append(Xs, Ys, Zs))\sigma_0) \cup v_1$ .

Consider now the query

$$Q : append([a, X, c], Ys, Zs), append(Zs, [b], Ls).$$

The substitution  $\theta = \{Zs/[a, X, c|Ys]\}$  is simply local with respect to  $Q$ . In fact  $\theta = \sigma_1 \sigma_2$  where  $\sigma_1 = \{Zs/[a, X, c|Ys]\}$  and  $\sigma_2$  is the empty substitution, and  $v_1$  and  $v_2$  are empty sets of variables.

The following property follows immediately from Definition 3.1.

**Proposition 3.3** Let the clause  $c$  be simply moded and  $\rho$  be a renaming. If the substitution  $\theta$  is simply local with respect to  $c$  then the substitution  $\rho^{-1}\theta\rho$  is simply local with respect to  $c\rho$ .

The next lemma provides us with a means of composing substitutions which are simply local with respect to pieces of queries provided that they satisfy the following *variable compatible* property.

**Definition 3.4** Let  $\vartheta_1$  be a substitution simply local with respect to  $\mathbf{A}$  and  $\vartheta_2$  be simply local with respect to  $\mathbf{B}\vartheta_1$ . Then  $\vartheta_1$  and  $\vartheta_2$  are variable compatible with respect to  $\mathbf{A}$  and  $\mathbf{B}$  if

- the set of fresh variables of  $\vartheta_1$  is disjoint from the set of fresh variables of  $\vartheta_2$ ,
- $\text{Var}(\mathbf{A}, \mathbf{B})$  is disjoint from the set of fresh variables of  $\vartheta_1$  and  $\vartheta_2$ .

When two substitutions are variable compatible then we have a way of combining them as described below.

**Lemma 3.5** *Let the query  $\mathbf{A}, \mathbf{B}$  be simply moded. There exists a substitution  $\theta$  simply local with respect to  $\mathbf{A}, \mathbf{B}$  iff  $\theta = \vartheta_1 \vartheta_2$  where*

- $\vartheta_1 = \theta|_{\mathbf{A}} = \theta|_{\text{Out}(\mathbf{A})}$  is simply local with respect to  $\mathbf{A}$ ,
- $\vartheta_2 = \theta|_{\mathbf{B}} = \theta|_{\text{Out}(\mathbf{B})}$  simply local with respect to  $\mathbf{B} \vartheta_1$ ,
- $\vartheta_1$  and  $\vartheta_2$  are variable compatible with respect to  $\mathbf{A}$  and  $\mathbf{B}$ .

**Proof.** Let  $\mathbf{A} = A_1, \dots, A_i$  and  $\mathbf{B} = A_{i+1}, \dots, A_n$ .

$\Rightarrow$ ) Let  $\theta = \sigma_1 \cdots \sigma_n$  be according to Definition 3.1. By definition of simply local substitution and properties of simply moded queries, for every  $k, j \in [1..n]$  and  $k \neq j$ ,  $\text{Dom}(\sigma_k) \cap \text{Dom}(\sigma_j) = \emptyset$ ,  $\text{Out}((A_{k+1}, \dots, A_n) \sigma_1 \cdots \sigma_k) = \text{Out}(A_{k+1}, \dots, A_n)$  and  $((A_1, \dots, A_k) \sigma_1 \cdots \sigma_k) \sigma_{k+1} \cdots \sigma_n = (A_1, \dots, A_k) \sigma_1 \cdots \sigma_k$ . Thus  $\theta|_{\mathbf{A}} = \sigma_1 \cdots \sigma_i$  is simply local with respect to  $\mathbf{A}$  and  $\sigma_{i+1} \cdots \sigma_n$  is simply local with respect to  $(A_{i+1}, \dots, A_n) \sigma_1 \cdots \sigma_i$ .

$\Leftarrow$ ) Let  $\vartheta_1 = \sigma_1 \cdots \sigma_i$  and  $\vartheta_2 = \sigma_{i+1} \cdots \sigma_n$ . To prove that  $\vartheta_1 \vartheta_2 = \sigma_1 \cdots \sigma_n$  is simply local with respect to  $\mathbf{A}, \mathbf{B}$ , we observe that by definition of simply local substitution and properties of simply moded queries,  $\text{Out}((A_{i+1}, \dots, A_n) \sigma_1 \cdots \sigma_i) = \text{Out}(A_{i+1}, \dots, A_n)$  and hence for all  $j \in [i+1..n]$ ,  $\text{Dom}(\sigma_j) \subseteq \text{Var}(\text{Out}(A_j))$ . The fact that  $\vartheta_1$  and  $\vartheta_2$  are variable compatible ensures that the composition  $\vartheta_1 \vartheta_2$  satisfies the requirement on fresh variables in the definition of simply local substitution.  $\square$

Analogously, one can prove the following result which allows us to combine simply local substitutions applied to a clause rather than to a query.

**Lemma 3.6** *Let the clause  $c : H \leftarrow \mathbf{B}$  be simply moded. There exists a substitution  $\theta$  simply local with respect to  $c$  iff  $\theta = \vartheta_0 \vartheta_1$  where*

- $\vartheta_0 = \theta|_H = \theta|_{\text{In}(H)}$  is simply local with respect to  $H \leftarrow$ ,
- $\vartheta_1 = \theta|_{\mathbf{B}} = \theta|_{\text{Out}(\mathbf{B})}$  simply local with respect to  $\mathbf{B} \vartheta_0$ ,
- $\vartheta_0$  and  $\vartheta_1$  are variable compatible with respect to  $H$  and  $\mathbf{B}$ .

The following definition introduces a property of mgu's which can be naturally satisfied by input consuming derivations, as shown in the subsequent lemma. The proof of the lemma is reported in the appendix.

**Definition 3.7 ((simply local mgu))** *Let the atoms  $A$  and  $H$  be variable disjoint,  $A$  be simply moded and  $\theta$  be a mgu of  $A$  and  $H$  such that  $\text{In}(A\theta) = \text{In}(A)$ . We say that  $\theta$  is a simply local mgu of  $A$  and  $H$  if  $\theta = \sigma_0 \sigma_1$  where  $\sigma_0$  is simply local with respect to the clause  $H \leftarrow$  and  $\sigma_1$  is simply local with respect to the atom  $A$ .*

**Lemma 3.8** *Let the atoms  $A$  and  $H$  be variable disjoint and  $A$  be simply moded. Suppose that there exists  $\vartheta = \text{mgu}(A, H)$  such that  $\text{In}(A\vartheta) = \text{In}(A)$ . Then there exists a simply local mgu  $\theta$  of  $A$  and  $H$ .*

Note that the previous Lemma 3.8 together with Theorem 3.18 in [2] (on derivations with different mgu's), ensures us that as long as we are interested in properties which are invariant under renaming, we can safely assume that all the mgu's employed in an input consuming derivation of a simply moded program with a simply moded query are simply local.

**Example 3.9** Consider the predicate  $\text{p}/2$  in the mode  $\text{p}(I, O)$  and the atoms

$$A = \text{p}(\mathbf{f}(\mathbf{X}, \mathbf{Y}), \mathbf{Z}) \quad H = \text{p}(\mathbf{W}, \mathbf{U}).$$

Note that there exists an mgu  $\vartheta$  of  $A$  and  $H$  such that  $\text{In}(A\vartheta) = \text{In}(A)$ . In fact, there are actually two relevant mgus which enjoy this property:

$$\vartheta_1 = \{\mathbf{W}/\mathbf{f}(\mathbf{X}, \mathbf{Y}), \mathbf{U}/\mathbf{Z}\} \quad \vartheta_2 = \{\mathbf{W}/\mathbf{f}(\mathbf{X}, \mathbf{Y}), \mathbf{Z}/\mathbf{U}\}$$

but only the second one is simply local.

Note that when two atoms  $A$  and  $H$  are variable disjoint and  $\vartheta$  is a simply local mgu of  $A$  and  $H$  then the variables in  $\text{Out}(A)$  do not occur anymore in  $A\vartheta$ .

The next lemma shows a persistence property of simply local substitutions. It provides one of the key intuitions for the development of the bottom-up semantics of the next section. Its proof is reported in the appendix.

**Lemma 3.10** *Let  $Q : A, \mathbf{R}$  be a simply moded query,  $Q' : (\mathbf{B}, \mathbf{R})\vartheta$  and  $Q \xrightarrow{\vartheta} Q'$  be an input consuming derivation step obtained by using the simply moded clause  $c : H \leftarrow \mathbf{B}$  and the simply local mgu  $\vartheta$ . Let  $\theta$  be a substitution simply local with respect to  $Q'$  such that the set of fresh variables of  $\theta$  is disjoint from  $\text{Var}(Q)$  and  $\text{Var}(c)$ . Then  $(\vartheta\theta)|_Q$  is simply local with respect to  $Q$ .*

## 4 A Denotational Semantics for Partial Derivations

As we mentioned in the introduction, input consuming derivations can be used to model parallelism, and in this context it is very important to model the results of partial computations. Indeed, standard semantics for concurrent logic languages such as CCP [13, 25] and GHC [31] often capture such intermediate results, or in any case, the results of non-successful computations [11]. In fact, input consuming programs can have a reactive nature: the (partial) result of a computation may trigger another computation by instantiating sufficiently the input positions of another atom so that it becomes resolvable. Because of this, when one wants to characterize for instance termination, the adoption of a semantics modeling intermediate results becomes essential.

In this section we define a denotational semantics that models partial computed answer substitutions of input consuming derivations of simply moded programs and queries. We will later see how this semantics allows us to characterize termination of input consuming derivations.

#### 4.1 Immediate consequence operator

In predicate logic, an interpretation states which formulas are true and which ones are not. For our purposes, it is convenient to formalize this by defining an interpretation  $I$  as a set of atoms closed under variance. Based on this notion and simply local substitutions, we now define a restricted notion of model.

**Definition 4.1 ((simply local model))** *Let  $M$  be an interpretation. We say that  $M$  is a simply local model of a clause  $c : H \leftarrow B_1, \dots, B_n$  if for every substitution  $\theta$  simply local with respect to  $c$ ,*

$$\text{if } B_1\theta, \dots, B_n\theta \in M \text{ then } H\theta \in M. \quad (1)$$

*$M$  is a simply local model of a program  $P$  if it is a simply local model of each clause of it.*

Clearly a simply local model is not necessarily a model in the classical sense, since the substitution  $\theta$  in (1) is required to be simply local. For example, given the program  $\{q(1), p(x) \leftarrow q(x)\}$  with modes  $q(I), p(O)$ , a model must contain the atom  $p(1)$ , whereas a simply local model does not necessarily contain  $p(1)$ , since  $\{x/1\}$  is not simply local with respect to  $p(x) \leftarrow q(x)$ . On the other hand, any *term model* (see [2]) is a simply local model, while there are Herbrand models which are not simply local.

We now show that there exists a minimal simply local model and that it is bottom-up computable. For this we need the following operator  $T_P^{SL}$  on interpretations.

**Definition 4.2 (( $T_P^{SL}$  operator))** *Given a program  $P$  and an interpretation  $I$ , we define*

$$T_P^{sl}(I) = \{H\theta \mid \exists c : H \leftarrow B_1, \dots, B_n \text{ variant of a clause in } P, \\ \theta \text{ is simply local with respect to } c, \\ B_1\theta, \dots, B_n\theta \in I\}$$

and

$$T_P^{SL}(I) = (T_P^{sl} + id)(I) = I \cup T_P^{sl}(I).$$

It is easy to show that both  $T_P^{sl}$  and  $T_P^{SL}$  are monotonic and continuous on the lattice where interpretations are ordered by set inclusion. We consider powers of an operator  $T$  which are defined in the standard way as follows:  $T \uparrow 0(I) = I$ ,  $T \uparrow (i+1)(I) = T(T \uparrow i(I))$ , and  $T \uparrow \omega(I) = \bigcup_{i=0}^{\infty} T \uparrow i(I)$ .

We now show that if  $I$  consists of simply moded atoms then  $T_P^{SL} \uparrow \omega(I)$  is a simply local model of  $P$  containing  $I$ . In the following we denote by  $SM_P$  the set of all simply moded atoms of the extended Herbrand universe of  $P$ . The proof of the next proposition is reported in the appendix.

**Proposition 4.3** *Let  $P$  be simply moded and  $I \subseteq SM_P$  be an interpretation. Then  $T_P^{SL} \uparrow \omega(I)$  is the least simply local model of  $P$  containing  $I$ .*

The following lemma relates partial input consuming derivations of simply moded programs and queries with powers of the  $T_P^{SL}$  operator. It is the key result to relate the operational semantics of partial input consuming derivations to the denotational semantics introduced below. The proof is reported in the appendix.

**Lemma 4.4** *Let the program  $P$  and the query  $\mathbf{A}$  be simply moded and  $I \subseteq SM_P$  be an interpretation. The following statements are equivalent:*

- (i) *there exists an input consuming derivation  $\delta : \mathbf{A} \xrightarrow{\vartheta}_P \mathbf{C}$  with  $\mathbf{C} \subseteq I$ ,*
- (ii) *there exists a substitution  $\theta$  simply local with respect to  $\mathbf{A}$ , such that  $\mathbf{A}\theta \subseteq T_P^{SL} \uparrow \omega(I)$ ,*

where  $\mathbf{A}\vartheta$  and  $\mathbf{A}\theta$  are variant.

## 4.2 Modeling the results of partial derivations

The results of partial input consuming derivations of simply moded queries in simply moded programs are captured by the following operational semantics.

**Definition 4.5 ((partial c.a.s. semantics))** *Let the program  $P$  be simply moded.*

$$\mathcal{O}_{SM_P}(P) = \{A\theta \mid A \text{ is simply moded and there exists } A \xrightarrow{\theta}_P \mathbf{C} \text{ with } \mathbf{C} \subseteq SM_P\}.$$

The next theorem shows that the denotational semantics provided by the least simply local model of  $P$  containing  $SM_P$  is correct and fully abstract with respect to the operational semantics of partial computed answer substitutions  $\mathcal{O}_{SM_P}(P)$ . The proof follows immediately by Lemma 4.4 above.

**Theorem 4.6** *Let  $P$  be simply moded. Then  $\mathcal{O}_{SM_P}(P) = T_P^{SL} \uparrow \omega(SM_P)$ .*

In the following we denote by  $PM_P^{SL}$  the least simply local model of  $P$  containing  $SM_P$ .

**Example 4.7** Consider again program APPEND.  $PM_{\text{APPEND}}^{SL}$  is obtained by repeatedly applying the  $T_P^{SL}$  operator, starting from any simply moded atom, i.e., an atom of the form `append(s, t, x)` where  $s$  and  $t$  are arbitrary terms but  $x$  is a variable not occurring in  $s$  or in  $t$ . Hence,

$$\begin{aligned} PM_{\text{APPEND}}^{SL} &= \{\text{append}([t_1, \dots, t_m], t, [t_1, \dots, t_m|t])\} \\ &\cup \{\text{append}(s, t, x) \mid x \text{ is a fresh variable}\} \\ &\cup \{\text{append}([t_1, \dots, t_m|s], t, [t_1, \dots, t_m|x]) \mid x \text{ is a fresh variable}\} \end{aligned}$$

where  $s, t, t_1, \dots, t_m$  are arbitrary terms, and  $m \geq 0$

Consider the query  $\text{append}([\mathbf{a}, \mathbf{b}, \mathbf{c}|\mathbf{X}], \mathbf{Y}, \mathbf{Z})$ . The substitution  $\theta = \{\mathbf{Z}/[\mathbf{a}, \mathbf{b}|\mathbf{Z}']\}$  is simply local with respect to that query and  $\text{append}([\mathbf{a}, \mathbf{b}, \mathbf{c}|\mathbf{X}], \mathbf{Y}, [\mathbf{a}, \mathbf{b}|\mathbf{Z}']) \in PM_{\text{APPEND}}^{SL}$ . Using Theorem 4.6, we can conclude that the query has a partial derivation with computed answer  $\theta$ . Following the same reasoning, we can also conclude that the query has a partial derivation with computed answer  $\theta' = \{\mathbf{Z}/[\mathbf{a}|\mathbf{Z}']\}$ .

## 5 Termination

In this section, we show how the denotational semantics can be used to give a characterization of termination of input consuming derivations, in a similar way as this has been done previously for LD-derivations [5, 24].

Input consuming derivations were originally conceived as an abstract and “reasonably strong” assumption about the selection rule in order to prove termination [27]. The first result in this area was a sufficient criterion applicable to well- and nicely moded programs. This was improved upon by dropping the requirement of well-modedness, which means that one also captures termination by deadlock.

The previous approaches are applicable as long as each recursive clause in the program is *direct recursive*, i.e., the structure upon which the recursion is carried out is passed directly from the clause head to the recursive call in the body. Typically, this means that the clause has the form  $p(\dots, s, \dots) \leftarrow \mathbf{A}, p(\dots, t, \dots), \mathbf{C}$ , where  $t$  is a proper subterm of  $s$ .

In this section we define the class of simply acceptable programs which includes programs whose termination cannot be proven without taking into account inter-argument relations. This means that for a clause  $p(\dots) \leftarrow \mathbf{A}, p(\dots), \mathbf{C}$ , we need to take into account how  $\mathbf{A}$  and  $\mathbf{C}$  might instantiate the body atom  $p(\dots)$  in order to establish termination. In this case, simply local models and simply local substitutions convey the needed information.

### 5.1 Simply Acceptable Programs

Note that programs without recursion terminate trivially. In order to deal with mutually recursive procedures we need the following standard definitions [2].

**Definition 5.1** *Let  $P$  be a program,  $p$  and  $q$  be relations. We say that  $p$  refers to  $q$  in  $P$  if there is a clause in  $P$  with  $p$  in the head and  $q$  in the body;  $p$  depends on  $q$  in  $P$ , and we write  $p \sqsupseteq q$ , if  $(p, q)$  is in the reflexive and transitive closure of the relation refers to;  $p$  and  $q$  are mutually recursive, written  $p \simeq q$ , if  $p$  and  $q$  depend on each other (i.e.,  $p \sqsupseteq q$  and  $q \sqsupseteq p$ ). We also write  $p \sqsubset q$  when  $p \sqsupseteq q$  and  $q \not\sqsupseteq p$ .*

To prove termination, it is common to use some measure of size for atoms in a query, often called *level mapping*. To show termination of moded programs, it is natural to use *moded level mappings*, where it is made explicit that the level of an atom depends only on its input positions. This concept was originally



defined for ground atoms [12]. Generalizing the definition to arbitrary atoms is crucial for showing termination of input consuming derivations.

**Definition 5.2 ((moded level mapping))** *A function  $|\cdot|$  is a moded level mapping if it maps atoms into  $\mathbf{N}$  and for any two atoms  $A$  and  $B$ , if  $A$  and  $B$  have the same predicate symbol and the same terms in their input positions, then  $|A| = |B|$ .*

In other words, the level of an atom has to be independent from the terms occurring in its output positions. For our purposes it is not necessary to require that the level mapping is invariant under renaming, yet this being the most common case.

We now provide the central definitions of this section.

**Definition 5.3 ((input terminating))** *A program is called input terminating with respect to a given class  $C$  of queries if all its input consuming derivations starting in any query in  $C$  are finite.*

In particular, we say that  $P$  is input terminating with respect to simply moded queries if for each simply moded query  $Q$ , all input consuming derivations of  $P \cup \{Q\}$  terminate.

The basic notion for proving input termination is *simply acceptability*, which is in analogy to acceptability [5].

**Definition 5.4 ((simply acceptable))** *Let  $P$  be a program and  $M$  a simply local model of  $P$  containing  $SM_P$ . A clause  $c$  is simply acceptable with respect to the moded level mapping  $|\cdot|$  and  $M$  if for every variant  $H \leftarrow \mathbf{A}, B, \mathbf{C}$  of  $c$  and every substitution  $\theta$  simply local with respect to  $c$ ,*

$$\text{if } \mathbf{A}\theta \in M \text{ and } \text{Rel}(H) \simeq \text{Rel}(B) \text{ then } |H\theta| > |B\theta|.$$

*The program  $P$  is simply acceptable with respect to  $M$  if there exists a moded level mapping  $|\cdot|$  such that each clause of  $P$  is simply acceptable with respect to  $|\cdot|$  and  $M$ . We also say that  $P$  is simply acceptable if it is simply acceptable with respect to some  $M$  and moded level mapping  $|\cdot|$ .*

The difference between acceptability and simply acceptability is that acceptability is based on the classical notion of model and consequently on ground instances of a clause, whereas simply acceptability is based on simply local models containing  $SM_P$ . These models allow us to model correctly the behaviour induced by the dynamic scheduling and to capture the results of partial computations. Another important difference with acceptability is that the level mapping decreasing is now required for mutually recursive calls only.

It is important to realize why we need to model partial results. Consider the following program

```

q(a) ← q(a).
p(a) ← fail.

mode q(I)
mode p(O)

```

Notice that the query  $q(\mathbf{X})$  terminates by deadlock, while  $q(\mathbf{a})$  loops. Now consider the query  $p(\mathbf{X}), q(\mathbf{X})$ . This query can yield to a nonterminating computation because the query  $p(\mathbf{X})$ , *before failing*, reports the partial answer  $\{\mathbf{X}/\mathbf{a}\}$ . If – in order to prove termination – we referred to a classical model (modeling only successful derivations) then we would not be able to see that the above program could diverge, because we would not consider  $\{\mathbf{X}/\mathbf{a}\}$  as a possible answer substitution.

In the next two sections, we prove that simply acceptability is a sufficient and necessary criterion for input termination with respect to simply moded queries.

## 5.2 Sufficiency of Simply Acceptability

The following corollary of [7, Lemma 22] allows us to restrict our attention to queries containing only one atom.

**Corollary 5.5** *Let  $P$  be a simply moded program.  $P$  is input terminating with respect to simply moded queries if and only if for each simply moded atomic query  $A$  all input consuming derivations of  $P \cup \{A\}$  are finite.*

From now on, we say that a relation  $p$  is *defined in* the program  $P$  if  $p$  occurs in a head of a clause of  $P$ , and that  $P$  *extends* the program  $R$  if no relation defined in  $P$  occurs in  $R$ .

The following theorem shows that simply acceptability is a sufficient criterion for input termination with respect to simply moded queries, and can be used in a modular way.

**Theorem 5.6** *Let  $P$  and  $R$  be two simply moded programs such that  $P$  extends  $R$ . Let  $M$  be a simply local model of  $P \cup R$  containing  $SM_P$ . Suppose that*

- $R$  is input terminating with respect to simply moded queries,
- $P$  is simply acceptable with respect to  $M$  (and a moded level mapping  $||$ ).

*Then  $P \cup R$  is input terminating with respect to simply moded queries.*

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

*We can now prove our theorem. By Corollary 5.5, it is sufficient to prove that for any simply moded atomic 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 with respect to simply moded queries and that  $P$  is an extension of  $R$ . So we can assume that  $A$  is defined in  $P$ .*

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

$$\delta : A \xrightarrow{\vartheta_1} (B_1, \dots, B_n)\vartheta_1 \xrightarrow{\vartheta_2} \dots$$

where  $c : H \leftarrow B_1, \dots, B_n$  is the input clause used in the first derivation step and  $\vartheta_1 = \text{mgu}(A, H)$ . Clearly,  $(B_1, \dots, B_n)\vartheta_1$  has an infinite input consuming derivation in  $P \cup R$ . By Corollary 2.12 and Lemma 3.8, for some  $i \in [1..n]$  and for some substitution  $\vartheta'_2$ ,

1. there exists an infinite input consuming derivation of  $(P \cup R) \cup \{A\}$  of the form

$$A \xrightarrow{\vartheta_1} (B_1, \dots, B_n)\vartheta_1 \xrightarrow{\vartheta'_2} C, (B_i, \dots, B_n)\vartheta_1\vartheta'_2 \cdots;$$

2. there exists an infinite input consuming derivation of  $P \cup \{B_i\vartheta_1\vartheta'_2\}$

both employing only simply local mgu's.

Let  $\theta = (\vartheta_1\vartheta'_2)|_c$ . It is not difficult to see that  $\theta$  is simply local with respect to  $c$  (this is a consequence of Proposition A.1, reported in the appendix). Consider the instance  $H\theta \leftarrow (B_1, \dots, B_n)\theta$  of  $c$ . By Theorem 4.6,  $(B_1, \dots, B_{i-1})\theta \in M$ .

We show that (2) cannot hold, by induction on  $\langle \text{dep}_P(\text{Rel}(A)), |A| \rangle$  with respect to the ordering  $\succ$  defined by:  $\langle m, n \rangle \succ \langle m', n' \rangle$  if either  $m > m'$  or  $m = m'$  and  $n > n'$ .

*Base.* Let  $\text{dep}_P(\text{Rel}(A)) = 0$  ( $|A|$  is arbitrary). In this case,  $A$  does not depend on any predicate symbol of  $P$ , thus all the  $B_i$  as well as all the atoms occurring in its descendants in any input consuming derivation are defined in  $R$ . The hypothesis that  $R$  is input terminating with respect to simply moded queries contradicts (2) above.

*Induction step.* We distinguish two cases:

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

In case (a) we have  $\text{dep}_P(\text{Rel}(A)) = \text{dep}_P(\text{Rel}(H\theta)) > \text{dep}_P(\text{Rel}(B_i\theta))$ . Therefore,

$$\langle \text{dep}_P(\text{Rel}(A)), |A| \rangle = \langle \text{dep}_P(\text{Rel}(H\theta)), |H\theta| \rangle \succ \langle \text{dep}_P(\text{Rel}(B_i\theta)), |B_i\theta| \rangle.$$

In case (b), from the hypothesis that  $P$  is simply acceptable with respect to  $| \cdot |$  and  $M$ ,  $\theta$  is simply local with respect to  $c$  and  $(B_1, \dots, B_{i-1})\theta \in M$ , it follows that  $|H\theta| > |B_i\theta|$ . Consider the partial input consuming derivation  $A \xrightarrow{\theta} C, (B_i, \dots, B_n)\theta$ . By Lemma 2.8 and the fact that  $| \cdot |$  is a moded level mapping, we have  $|A| = |A\theta| = |H\theta|$ . 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.  $\square$

The above theorem suggests proving termination in a modular way, i.e., extending a program that is already known to be input terminating with respect to simply moded queries by a program that is simply acceptable. Of course, this theorem holds in particular if the former program is empty.

**Theorem 5.7** *Let  $P$  be a simply moded program. If  $P$  is simply acceptable then it is input terminating with respect to simply moded queries.*

**Proof.** *The proof follows from Theorem 5.6, by setting  $R = \emptyset$ .*  $\square$

```

% quicksort(Xs, Ys) ← Ys is an ordered permutation of Xs.
   quicksort(Xs, Ys) ← quicksort_dl(Xs, Ys, []).

c1: quicksort_dl([X|Xs], Ys, Zs) ←
     partition(Xs, X, Littles, Bigs),
     quicksort_dl(Bigs, Ys1, Zs),
     quicksort_dl(Littles, Ys, [X|Ys1]).
   quicksort_dl([], Xs, Xs).

c2: partition([X|Xs], Y, [X|Ls], Bs) ← X =< Y, partition(Xs, Y, Ls, Bs).
c3: partition([X|Xs], Y, Ls, [X|Bs]) ← X > Y, partition(Xs, Y, Ls, Bs).
   partition([], Y, [], []).

```

Figure 1: The QUICKSORT program

**Example 5.8** Fig. 1 shows `quicksort` using a form of difference lists [30, program 15.3] (we permuted two body atoms for the sake of clarity). This program is simply moded with respect to the mode

$$\{\text{quicksort}(I, O), \text{quicksort\_dl}(I, O, I), \text{partition}(I, I, O, O), =<(I, I), >(I, I)\}.$$

We show that it is simply acceptable. We start by defining the level mapping. Define function *len* as

$$\begin{aligned} \text{len}([h|t]) &= 1 + \text{len}(t), \\ \text{len}(a) &= 0 \quad \text{if } a \text{ is not of the form } [h|t]. \end{aligned}$$

We use the following moded level mapping (where positions with `_` are irrelevant):

$$\begin{aligned} |\text{quicksort\_dl}(l, \_, \_)| &= \text{len}(l), \\ |\text{partition}(l, \_, \_, \_)| &= \text{len}(l). \end{aligned}$$

The level mapping of all other atoms can be set to 0. Concerning the simply local model, the crucial aspect with respect to termination is that it has to express the dependency between the list lengths of the arguments of `partition`. To this end, the simplest solution is to choose it so that  $M$  restricted to `partition` contains exactly the atoms of the form `partition`( $t_1, t_2, t_3, t_4$ ) where

$$\text{len}(t_1) \geq \text{len}(t_3) \text{ and } \text{len}(t_1) \geq \text{len}(t_4). \quad (2)$$

The presence (absence) of other atoms is irrelevant for showing simple-acceptability, so the simplest way of building a simply local model is that of adding all other atoms not defining `partition`. Let

$$\begin{aligned} M &= \{ \text{partition}(t_1, t_2, t_3, t_4) \mid \text{len}(t_1) \geq \text{len}(t_3) \text{ and } \text{len}(t_1) \geq \text{len}(t_4) \} \\ &\cup \{ \text{quicksort\_dl}(r, s, t) \mid \text{for all } r, s, t \} \\ &\cup \{ \text{quicksort}(r, s) \mid \text{for all } r, s \} \\ &\cup \{ =<(r, s), >(r, s) \mid \text{for all } r, s \}. \end{aligned}$$

Notice that  $M$  includes all simply moded atoms. It is easy to show that the program is simply acceptable with respect to  $M$  and  $||$  and hence input terminating with respect to simply moded queries. In fact:

- Consider  $c1$ , the first clause defining `quicksort_dl`. For every substitution  $\theta$ , simply local with respect to  $c1$ , we have to show that
  - If  $\text{partition}(Xs, X, Littles, Bigs)\theta \in M$ , then
 
$$|\text{quicksort\_dl}([X|Xs], Ys, Zs)\theta| > |\text{quicksort\_dl}(Bigs, Ys1, Zs)\theta|.$$
 This follows immediately from the definition of level mapping  $||$  and the fact that since  $\text{partition}(Xs, X, Littles, Bigs)\theta \in M$ , we have  $\text{len}(Bigs)\theta \leq \text{len}(Xs)\theta$ .
  - If  $(\text{partition}(Xs, X, Littles, Bigs), \text{quicksort\_dl}(Bigs, Ys1, Zs))\theta \in M$ , then  $|\text{quicksort\_dl}([X|Xs], Ys, Zs)\theta| > |\text{quicksort\_dl}(Littles, Ys, [X|Ys1])\theta|$ .
 This is analogous to the previous point and follows by the definition of  $||$  and the fact that since  $\text{partition}(Xs, X, Littles, Bigs)\theta \in M$ ,  $\text{len}(Littles)\theta \leq \text{len}(Xs)\theta$ .
- Next, we consider  $c2$ . We have to show that for each simply local substitution  $\theta$  such that  $(X =< Y)\theta \in M$ ,
 
$$|\text{partition}([X|Xs], Y, [X|Ls], Bs)\theta| > |\text{partition}(Xs, Y, Ls, Bs)\theta|.$$
 This follows directly from the definition of  $||$  (the fact that  $(X =< Y)\theta \in M$  is not used here).
- Finally, we consider the other clauses. Clause  $c3$  is handled as  $c2$ , while all other ones are not recursive (not even mutually), and therefore they are trivially simply acceptable.
 

There is one aspect we have neglected so far, namely that the program contains calls to (built-in) predicates `=<` and `>` without defining clauses. However, these predicates are conceptually defined by fact clauses such as `1>0.`, which are trivially simply acceptable.

By Theorem 5.7 we have that every query of the form `quicksort(t, x)`, where  $x$  is a variable disjoint from  $t$ , yields a finite input consuming derivation. In particular, Theorem 5.7 shows that the query `quicksort(Y, X)` yields terminating input consuming derivations. These derivations terminate by deadlock, while by dropping the requirement of input consuming resolution steps it is easy to build a non-terminating derivation starting in that query. This shows that Theorem 5.7 allows us to capture termination by deadlock, as further confirmed by the necessity results we will provide in the next section.

It is worth remarking that with the tool of [7] it is not possible to prove that `QUICKSORT` is input terminating (with respect to simply moded queries). This is because in that paper the concept of quasi-recurrent program, which has the same role as that of simply acceptable program, does not take into account

the presence of inter-argument relationships, (which in the above example are present in the form of equation (2)).

The following contrived example shows the necessity of referring to simply local substitutions.

**Example 5.9** Consider the program

**c4:**  $q(\mathbf{a}) \leftarrow q(\mathbf{X})$ .

together with the mode  $q(D)$ . Every simply moded query terminates (either by failure or by deadlock). Take the level mapping  $|q(t)| = 1$  if  $t$  is not a variable and  $|q(x)| = 0$  otherwise. We now show that **c4** is simply acceptable with respect to  $| \cdot |$  and any simply local model  $M$ . In fact, for every  $\theta$  simply local with respect to **c4** we have that  $q(\mathbf{X})\theta = q(\mathbf{X})$ : since  $Out(q(\mathbf{X})) = \emptyset$ , we have that  $\mathbf{X} \notin Dom(\theta)$ . Moreover trivially  $q(\mathbf{a})\theta = q(\mathbf{a})$ . Therefore  $|q(\mathbf{a})\theta| > |q(\mathbf{X})\theta|$ , which implies simply acceptability.

Notice that if we drop the requirement that  $\theta$  must be simply local then we would have no guarantee that  $|q(\mathbf{a})\theta| > |q(\mathbf{X})\theta|$ , and so we would not be able to demonstrate termination: simply let  $\theta = \{\mathbf{X}/\mathbf{a}\}$ .

In the next section we will show that simply acceptability is a *necessary* condition for termination.

### 5.3 Necessity of Simply Acceptability

We now prove the converse of Theorem 5.7, namely that our criterion for proving input termination with respect to simply moded queries is also necessary. For this we need some new definitions as well as some new preliminary results in the spirit of those in [5].

The first definition concerns a concept analogous to that of SLD-trees in the context of input consuming derivations.

**Definition 5.10 ((IC-tree))** Let  $P$  be a program and  $Q$  be a query. An IC-tree for  $P \cup \{Q\}$  is a tree such that

- its root is  $Q$ ,
- every node  $Q'$  has exactly one descendant  $Q''$  for every atom  $A$  of  $Q'$  and every clause  $c$  such that  $Q''$  is an input consuming resolvent of  $Q'$  with respect to  $A$  and  $c$ .

Informally, an IC-tree for  $P \cup \{Q\}$  groups all the input consuming derivations of  $P \cup \{Q\}$  modulo the choices of the renaming of the program clauses used and the choices of the mgu's.

Notice that it can happen that a node contains no selectable atom, in which case it has no children.

Branches of IC-trees are input consuming derivations. Therefore we can characterize input termination in terms of IC-trees.

**Lemma 5.11** *A IC-tree for  $P \cup \{Q\}$  is finite iff all input consuming consuming derivations of  $P \cup \{Q\}$  are finite.*

**Proof.** *By definition, the IC-trees are finitely branching. The claim now follows by the classical result of König.  $\square$*

Analogously to the case of acceptability, we measure atoms by counting the number of nodes in the corresponding IC-tree. For a program  $P$  and a query  $Q$ , we denote by  $\text{nodes}_P^{ic}(Q)$  the number of nodes in an IC-tree for  $P \cup \{Q\}$ . We need one last property of IC-trees.

**Lemma 5.12** *Let the program  $P$  and the query  $\mathbf{A}, B$  be simply moded. Suppose that  $P$  is input terminating with respect to simply moded queries and that  $\mathbf{A}\theta \in PM_P^{SL}$ , where  $\theta$  is a simply local substitution with respect to  $\mathbf{A}$ . Then  $\text{nodes}_P^{ic}(\mathbf{A}, B) \geq \text{nodes}_P^{ic}(B\theta)$ .*

**Proof.** *Consider an IC-tree  $T$  for  $P \cup \{\mathbf{A}, B\}$ . By the hypothesis that  $\mathbf{A}\theta \in PM_P^{SL}$ , it follows that there exists a substitution  $\vartheta$  such that – by Lemma 4.4 –  $\mathbf{A} \xrightarrow{\vartheta}_P \mathbf{C}$  is a (partial) input consuming derivation and  $\mathbf{A}\theta \approx \mathbf{A}\vartheta$ . Hence there exists an input consuming derivation  $\mathbf{A}, B \xrightarrow{\vartheta}_P \mathbf{C}, B\vartheta$  and  $B\theta \approx B\vartheta$ . Clearly, by definition of IC-tree,  $\text{nodes}_P^{ic}(\mathbf{A}, B) \geq \text{nodes}_P^{ic}(B\vartheta) = \text{nodes}_P^{ic}(B\theta)$ . Hence the thesis.  $\square$*

We are now in the position to prove that the class of simply acceptable programs comprises all the programs input terminating with respect to simply moded queries.

**Theorem 5.13** *Let  $P$  be a simply moded program. If  $P$  is input terminating with respect to simply moded queries then  $P$  is simply acceptable.*

*In particular, it is simply acceptable with respect to  $PM_P^{SL}$  and a moded level mapping which is invariant under renaming.*

**Proof.** *show that there exists a moded level mapping  $|\cdot|$  for  $P$  such that  $P$  is simply acceptable with respect to  $|\cdot|$  and  $PM_P^{SL}$ . We recall that  $PM_P^{SL}$  is the least simply local model of  $P$  containing  $SM_P$ .*

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

$$|A| = \text{nodes}_P^{ic}(A^*).$$

*Notice that the level  $|A|$  of an atom  $A$  is independent from the terms filling in its output positions, i.e.,  $|\cdot|$  is a moded level mapping. Moreover, since  $P$  is input terminating with respect to simply moded queries and  $A^*$  is simply moded, all the input consuming derivations of  $P \cup \{A^*\}$  are finite. Therefore, by Lemma 5.11,  $\text{nodes}_P^{ic}(A^*)$  is defined (and finite), and thus  $|A|$  is defined (and finite) for every atom  $A$ .*

*We now prove that  $P$  is simply acceptable with respect to  $|\cdot|$  and  $PM_P^{SL}$ .*

Let  $c : H \leftarrow \mathbf{A}, B, \mathbf{C}$  be a clause of  $P$  and  $H\theta \leftarrow \mathbf{A}\theta, B\theta, \mathbf{C}\theta$  be an instance of  $c$  where  $\theta$  is a simply local substitution with respect to  $c$ . We show that

$$\text{if } \mathbf{A}\theta \in PM_P^{SL} \text{ and } \text{Rel}(H) \simeq \text{Rel}(B) \text{ then } |H\theta| > |B\theta|.$$

Consider a variant  $c' : H' \leftarrow \mathbf{A}', B', \mathbf{C}'$  of  $c$  variable disjoint from  $(H\theta)^*$ . Let  $\rho$  be a renaming such that  $c' = c\rho$ . Clearly,  $(H\theta)^*$  and  $H'$  unify. Let  $\mu = \text{mgu}((H\theta)^*, H') = \text{mgu}((H\theta)^*, H\rho)$  be a simply local mgu of  $(H\theta)^*$  and  $H'$ . Then we have that  $\text{Dom}(\mu) \subseteq \text{Var}(\text{Out}((H\theta)^*)) \cup \text{Var}(\text{In}(H\rho))$ . Hence  $(\mathbf{A}', B', \mathbf{C}')\mu = (\mathbf{A}, B, \mathbf{C})\rho\mu$ , and

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

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

Moreover,  $(\mathbf{A}, B, \mathbf{C})\rho\mu \approx (\mathbf{A}, B, \mathbf{C})(\rho\mu)|_{\text{In}(H)} = (\mathbf{A}, B, \mathbf{C})\theta|_{\text{In}(H)}$ .

Let  $\theta = \theta|_{\text{In}(H)}\theta|_{\text{Out}(\mathbf{A})}\theta|_{\text{Out}(B, \mathbf{C})}$ . Hence, by Lemmas 3.5 and 3.6,  $\theta|_{\text{Out}(\mathbf{A})}$  is simply local with respect to  $\mathbf{A}\theta|_{\text{In}(H)}$ . Therefore, we have that

$$\begin{aligned} |H\theta| &= \text{nodes}_P^{ic}((H\theta)^*) && \text{(by definition of } | \cdot | \text{)} \\ &> \text{nodes}_P^{ic}((\mathbf{A}, B, \mathbf{C})\theta|_{\text{In}(H)}) && \text{(by definition of IC-tree)} \\ &\geq \text{nodes}_P^{ic}((\mathbf{A}, B)\theta|_{\text{In}(H)}) && \text{(by definition of IC-tree)} \\ &\geq \text{nodes}_P^{ic}((B\theta|_{\text{In}(H)})\theta|_{\text{Out}(\mathbf{A})}) && \text{(by Lemma 5.12)} \\ &= \text{nodes}_P^{ic}((B\theta)^*) && \text{(since } \theta \text{ is simply local with respect to } c \text{)} \\ &= |B\theta| && \text{(by definition of } | \cdot | \text{)}. \end{aligned}$$

□

## 5.4 A Characterization

Summarizing, we have characterized input termination by simply acceptability.

**Theorem 5.14** *A simply moded program  $P$  is simply acceptable if and only if it is input terminating with respect to simply moded queries. In particular, if  $P$  is input terminating with respect to simply moded queries, then it is simply acceptable with respect to  $PM_P^{SL}$  and a moded level mapping which is invariant under renaming.*

**Proof.** *Theorem 5.7 and Theorem 5.13.* □

The following example shows how we can use Theorem 5.14 to reason about termination of a program.

**Example 5.15** Consider the following program PERMUTE.

```
% permute(Xs, Ys) ← Ys is a permutation of the list Xs
c1: permute([X|Xs], Ys) ← insert(Zs, X, Ys), permute(Xs, Zs).
    permute([], []).
```



`% insert(Xs,X,Ys) ← Ys` is the result of inserting `X` into the list `Xs`

```
c2: insert([U|Xs],X,[U|Zs]) ← insert(Xs,X,Zs).
    insert(Xs,X,[X|Xs]).
```

First, let us consider it together with the mode `permute(O,I),insert(O,O,I)`. Notice that the program is simply-moded. It is immediate to check that the program is not input terminating in this mode: by repeatedly selecting the right-most atom, the query `permute(Xs,Ys)` generates an infinite input consuming derivation. This is basically due to the fact that `c1` has a variable in its input position. Therefore, the recursive call in the body can always be selected.

This suggests that one could obtain input termination by replacing `c1` by:

```
c1': permute([X|Xs],[Y|Ys]) ← insert(Zs,X,[Y|Ys]), permute(Xs,Zs).
```

Call the resulting program `PERMUTE2`. This program is still nonterminating (the query `permute(Xs,[Y|Ys])` has an infinite input consuming derivation). However, this is not so obvious, and in essence, it has first been observed by Naish [20], in the context of programs with delay declarations. We can use Theorem 5.13 to demonstrate that and to understand why `PERMUTE2` does not input terminate. We show that the program cannot be simply acceptable with respect to  $PM_{\text{PERMUTE2}}^{SL}$  and a moded level mapping which is invariant under renaming. By applying  $T_P^{SL}$  once to the the simply moded atom `insert(Xs',X',Zs')` (where `Xs',X',Zs'` are fresh variables), one sees that `insert([U'|Xs'],X',[U'|Zs'])`  $\in PM_{\text{PERMUTE2}}^{SL}$ . The substitution  $\{Y/U', Ys/Zs', Zs/[U'|Xs'], X/X'\}$  is simply local with respect to `c1'`. Therefore, for `c1'` to be simply acceptable, by Theorem 5.13, there would have to be a moded level mapping invariant under renaming such that  $|\text{permute}([X'|Xs],[U'|Zs'])| > |\text{permute}(Xs,[U'|Xs'])|$ . This is a contradiction since a *moded* level mapping depends only on the input arguments (the second argument of `permute`).

Naish [20] suggested to obtain a terminating program by replacing `c2` with its *most specific* variant:

```
c2': insert([U|Xs],X,[U|[H|T]]) ← insert(Xs,X,[H|T]).
```

Call the resulting program `PERMUTE3`. We show that `PERMUTE3` is input terminating.<sup>3</sup> Note that `PERMUTE3` is simply moded, and consider the following level mapping:

$$\begin{aligned} |\text{permute}(-,l)| &= \text{len}(l), \\ |\text{insert}(-,-,l)| &= \text{len}(l). \end{aligned}$$

Concerning the simply local model, the crucial aspect with respect to termination is that it has to express the dependency between the lengths of the third

---

<sup>3</sup>This has not been observed previously and seems to contradict our previous claim that Naish's proposal for obtaining a terminating program does not work [28]. However, this claim referred to Naish's program with *delay declarations*, and, as we observed in [26], these delay declarations do not ensure input consuming derivations.

and first arguments of `insert`. We define:

$$M = \left\{ \begin{array}{l} \text{permute}(l, m) \quad | \quad \text{for all } l, m \\ \cup \quad \text{insert}(m, a, l) \quad | \quad \begin{array}{l} \text{either } \text{insert}(m, a, l) \text{ is simply moded} \\ \text{or } \text{len}(l) > \text{len}(m) \end{array} \end{array} \right\}$$

Notice that this model contains also non-ground atoms. We have to verify that  $M$  is a simply-local model. The only non-trivial proof obligation concerns `c2'`. Now for any, not even necessarily simply local, substitution  $\theta$ ,  $\text{insert}(Xs, X, [H|T])\theta \in M$  implies  $\text{insert}([U|Xs], X, [U|[H|T]])\theta \in M$ . Hence  $M$  is a simply-local model.

We show that `PERMUTE3` is simply acceptable with respect to  $M$  and  $|\cdot|$ . Concerning `c1'`, we must show that for every substitution  $\theta$ , simply local with respect to `c1'`,  $\text{insert}(Zs, X, [Y|Ys])\theta \in M$  implies  $|\text{permute}([X|Xs], [Y|Ys])\theta| > |\text{permute}(Xs, Zs)\theta|$ . By the definitions of  $M$  and  $|\cdot|$ , this even holds for arbitrary  $\theta$ . For the remaining clauses, it is immediate to check that they are simply-acceptable. It follows that `PERMUTE3` is input terminating with respect to simply moded queries.

To conclude, consider the program `PERMUTE4`: that is, `PERMUTE` together with the modes `permute(I, O)`, `insert(I, I, O)`. In this case, in order to make the program simply moded we have to permute the two body atoms of the first `permute` clause (but see the remark below) i.e., `permute` is redefined as

$$\begin{array}{l} \text{permute}([X|Xs], Ys) \leftarrow \text{permute}(Xs, Zs), \text{insert}(Zs, X, Ys). \\ \text{permute}([], []). \end{array}$$

Notice that the program is now input terminating with respect to simply moded queries. This is in fact the *natural* mode of the `PERMUTE` program. To demonstrate the termination one can apply Theorem 5.7 using *any* simply local model together with the following moded level mapping:

$$\begin{array}{l} |\text{permute}(l, -)| = \text{len}(l), \\ |\text{insert}(l, -, -)| = \text{len}(l). \end{array}$$

In `PERMUTE4` we reordered the body atoms of a program, but this was actually an unnecessary operation.

**Remark 5.16** *Everything we state in this article that applies to the class of simply-moded programs (resp. queries) applies to the class of permutation simply moded programs (queries) as well, i.e., to those programs and queries that are simply moded possibly after a permutation of body atoms. For the sake of notation simplicity, we avoid to refer to this in a structural way.*

## 6 Other Examples

In this section we provide additional explanatory examples.

**Example 6.1** Consider the following program `LISTTREE` for converting a list  $l$  into a binary tree  $t$  with labeled nodes, so that  $t$  contains as labels exactly the elements of  $l$ , in the same left-to-right order (in can also be used to convert  $t$  into  $l$ ).

```

% list_tree(L,T) ← L is a list and T is a binary tree with labelled nodes
% containing the same elements in a left-to-right order

list_tree([],void).
c1: list_tree([H|T],tree(TA,X,TB)) ←
    extract([H|T],LA,X,LB),
    list_tree(LA,TA),
    list_tree(LB,TB).

% extract(Xs,Ys,X,Zs) ← Xs is the result of concatenating Ys, [X] and Zs
c2: extract([X|L],[],X,L).
c3: extract([X|[H|T]], [X|S],Y,R) ← extract([H|T],S,Y,R).

mode list_tree(I,O)
mode extract(I,O,O,O)

```

This program is simply moded. We now show that it is simply acceptable; for this we employ the following moded level mapping:

$$\begin{aligned}
|\text{list\_tree}(l,-)| &= \text{len}(l), \\
|\text{extract}(l,-,-,-)| &= \text{len}(l).
\end{aligned}$$

Concerning the simply local model, the crucial aspect with respect to termination is that it has to express the dependency between the lengths of the arguments of `extract`. We define

$$\begin{aligned}
M = \{ & \text{list\_tree}(l,t) & | & \text{for all } l,t\} \\
& \cup \{ \text{extract}(l,l_1,x,l_2) & | & \text{either } l_1,l_2,l \text{ are distinct variables,} \\
& & & \text{or } \text{len}(l) > \text{len}(l_1) \text{ and } \text{len}(l) > \text{len}(l_2) \}.
\end{aligned}$$

We have to verify that  $M$  is indeed a simply-local model.

First, we have to show that  $M$  is a simply-local model of the clauses defining `list_tree`. This is trivial, since  $M$  contains all instances of `list_tree(X,Y)`.

Secondly, we have to show that  $M$  is a simply-local model of `c2`. We have to show that for each  $\theta$  simply-local with respect to `c2`  $\text{extract}([X|L],[],X,L)\theta \in M$ . But this holds by the model definition and the fact that for any substitution  $\theta$ , we have that  $\text{len}([X|L]\theta) > \text{len}([],\theta)$  and  $\text{len}([X|L]\theta) > \text{len}(L\theta)$ .

Thirdly, we have to show that  $M$  is a simply-local model of `c3`. Consider any substitution  $\theta$  such that  $\text{extract}([H|T],S,Y,R)\theta \in M$ . Since  $[H|T]\theta$  cannot be a variable, by the definition of  $M$ ,  $\text{len}([X|[H|T]]\theta) > \text{len}([X|S]\theta)$  and  $\text{len}([X|[H|T]]\theta) > \text{len}(R\theta)$ ; thus  $\text{extract}([X|[H|T]], [X|S],Y,R)\theta \in M$ . Therefore  $M$  is a simply-local model of `c3`.

Finally, we show that the program is simply acceptable with respect to  $M$  and  $||$  and hence input terminating with respect to simply moded queries. The

only non-trivial case is clause c1. For every simply local substitution  $\theta$ , we must show that

1. If  $\text{extract}([\text{H|T}], \text{LA}, \text{X}, \text{LB})\theta \in M$   
then  $|\text{list\_tree}([\text{H|T}], \text{tree}(\text{TA}, \text{X}, \text{TB}))\theta| > |\text{list\_tree}(\text{LA}, \text{TA})\theta|$ .
2. If  $\text{extract}([\text{H|T}], \text{LA}, \text{X}, \text{LB})\theta, \text{list\_tree}(\text{LA}, \text{TA})\theta \in M$   
then  $|\text{list\_tree}([\text{H|T}], \text{tree}(\text{TA}, \text{X}, \text{TB}))\theta| > |\text{list\_tree}(\text{LB}, \text{TB})\theta|$ .

Both implications follow immediately from the definition of  $|\cdot|$  and of  $M$ .

Observe that it is essential that we have the non-variable term  $[\text{H|T}]$  in c1, rather than simply a variable. Also, in c3, we must have  $[\text{H|T}]$  rather than simply a variable. Otherwise, the program would not be input terminating.

**Example 6.2** Consider the following program **TRANSPPOSE** for transposing a matrix. A matrix is represented as a list of lists:  $[[\mathbf{a}, \mathbf{b}, \mathbf{c}], [\mathbf{1}, \mathbf{2}, \mathbf{3}]]$  is a matrix with two rows and 3 columns. Note the degenerate cases:  $[[\ ]]$  is the matrix with 0 columns and 2 rows, while  $[\ ]$  is not a matrix (though it could be regarded as any matrix with 0 rows but an unknown number of columns).

```
% transpose(M,N) ← N is the transposed matrix of matrix M.
transpose(M, []) ← no_cols_matrix(M).
c1: transpose([R|Rs], [C|Cs]) ← cut_col([R|Rs], C, M2),
    transpose(M2, Cs).

% cut_col(M,C,N) ← C is the first column of the matrix M
% and N is obtained by removing C from M
c2: cut_col([], [], []).
c3: cut_col([[E|Es]|Rs], [E|C2], [Es|Rs2]) ← cut_col(Rs, C2, Rs2).

% no_cols_matrix(M) ← matrix M has zero width (no columns)
no_cols_matrix([]).
c4: no_cols_matrix([[[]]|Rs]) ← no_cols_matrix(Rs).

mode transpose(I, O)
mode cut_col(I, O, O)
mode no_cols_matrix(D)
```

This program is simply moded. We now show that it is simply acceptable. The moded level mapping uses  $len$  and the usual term size norm and is defined as follows:

$$\begin{aligned} |\text{transpose}(m, \_)| &= \text{size}(m), \\ |\text{cut\_col}(m, \_, \_)| &= \text{len}(m), \\ |\text{no\_cols\_matrix}(m)| &= \text{len}(m). \end{aligned}$$

where  $\text{size}(f(t_1, \dots, t_n)) = 1 + \text{size}(t_1) + \dots + \text{size}(t_n)$  for  $n \geq 0$ , and  $\text{size}(t) = 0$  if  $t$  is a variable.

Concerning the simply local model, the crucial aspect with respect to termination is that it has to express the dependency between the row widths of the arguments of `cut_col`. More specifically, in clause `c1`, `[R|Rs]` is a matrix (a list of rows), and `M2` is obtained from `[R|Rs]` by cutting off the first element in each row. This decrease in row width is crucial for termination. We define

$$\begin{aligned}
M = & \{ \text{transpose}(m, n) \mid \text{for all } m, n \} \\
& \cup \{ \text{cut\_col}(m, r, n) \mid \text{either } \text{cut\_col}(m, r, n) \text{ is simply-moded} \\
& \qquad \qquad \qquad \text{or } m = n = [] \\
& \qquad \qquad \qquad \text{or } \text{size}(m) > \text{size}(n) \} \\
& \cup \{ \text{no\_cols\_matrix}(m) \mid \text{for all } m \}.
\end{aligned}$$

We now verify that  $M$  is a simply-local model. We have non-trivial proof obligations for `c2` and `c3`. Concerning `c2`, `cut_col([], [], []) ∈ M` by construction. Concerning `c3`, consider an arbitrary (not even necessarily simply-local) substitution  $\theta$  such that `cut_col(Rs, C2, Rs2)θ ∈ M`. There are three cases.

- If `cut_col(Rs, C2, Rs2)θ` is simply-moded, then `Rs2θ` is a variable, thus  $\text{size}([\text{E|Es}|Rs]\theta) > \text{size}([\text{Es}|Rs2]\theta)$  and therefore  $\text{cut\_col}([\text{E|Es}|Rs], [\text{E|C2}], [\text{Es}|Rs2])\theta \in M$ .
- If  $\text{Rs}\theta \equiv \text{Rs2}\theta \equiv []$ , then  $\text{cut\_col}([\text{E|Es}|Rs], [\text{E|C2}], [\text{Es}|Rs2])\theta \equiv \text{cut\_col}([\text{E|Es}], [\text{E|C2}], [\text{Es}])\theta$ , and since  $\text{size}([\text{E|Es}]\theta) > \text{size}([\text{Es}]\theta)$ , it follows that  $\text{cut\_col}([\text{E|Es}], [\text{E|C2}], [\text{Es}])\theta \in M$ .
- If  $\text{size}(\text{Rs}\theta) > \text{size}(\text{Rs2}\theta)$ , then  $\text{size}([\text{E|Es}|Rs]\theta) > \text{size}([\text{Es}|Rs2]\theta)$ , thus  $\text{cut\_col}([\text{E|Es}|Rs], [\text{E|C2}], [\text{Es}|Rs2])\theta \in M$ .

Thus in all cases,  $\text{cut\_col}([\text{E|Es}|Rs], [\text{E|C2}], [\text{Es}|Rs2])\theta \in M$ . Therefore  $M$  is a model of `c3`. We now show that the program is simply acceptable with respect to  $M$  and `| |` and hence input terminating with respect to simply moded queries. Consider `c1`: for every substitution  $\theta$ , simply local with respect to `c1`, we have to show that if  $\text{cut\_col}([\text{R|Rs}], \text{C}, \text{M2})\theta \in M$ , then  $|\text{transpose}([\text{R|Rs}], [\text{C|Cs}])\theta| > |\text{transpose}(\text{M2}, \text{Cs})\theta|$ . This holds by the definition of  $M$ . Next, consider `c3`. For every substitution  $\theta$ , it is easy to see that  $|\text{cut\_col}([\text{E|Es}|Rs], [\text{E|C}], [\text{Es}|Rs2])\theta| > |\text{cut\_col}(\text{Rs}, \text{C}, \text{Rs2})\theta|$ . Equivalently, for clause `c4`, it is immediate to check that for any  $\theta$ ,  $|\text{no\_cols\_matrix}([\text{[]|Rs}])\theta| > |\text{no\_cols\_matrix}(\text{Rs})\theta|$ . All other clauses are trivially simply acceptable. Hence the thesis.

## 6.1 Delay Declarations

In practical systems, dynamic selection rules are implemented by means of constructs such as *delay declarations* and *block declarations*. Delay declarations, advocated by van Emden and de Lucena [33] were introduced explicitly in logic programming by Naish [22].

In a previous paper [8] we have argued that in most cases delay declarations are employed exactly to guarantee that the derivations are input consuming. We have also provided a technical result establishing that under some syntactically checkable conditions the use of delay declarations is equivalent to restricting to input consuming derivations. This allows one to apply Theorems 4.6 and 5.14 to a large class of programs employing delay declarations, thereby providing such programs with a model-based semantics for partial derivations, and a result characterizing their termination.

In this section we report some examples showing the analogies between the use of delay declarations and the restriction to input consuming derivations. Just for this subsection, we assume the reader to be familiar with the notion and the notation of delay declarations.

**Example 6.3** Consider again APPEND, in mode `append(I, I, O)` with the delay declarations we mentioned in the introduction, namely

```

delay append(Ls, -, _) until nonvar(Ls).
append([H|Xs], Ys, [H|Zs]) ← append(Xs, Ys, Zs).
append([], Ys, Ys).

```

In practice, this delay declaration can be seen as a compiler directive stating that the selection rule is allowed to select an atom of the form `append( $t_1, t_1, t_3$ )` iff  $t_1$  is a non-variable term. A derivation that respects this directive is called *delay-respecting*.

This is the natural delay declaration of the program and achieves the purpose that most natural queries are forced to terminate<sup>4</sup>. Now, it is easy to check that every SLD derivation starting in a simply moded query is similar to an input consuming derivation *if and only if* it is delay-respecting.

Thus, for APPEND we can say that input consuming derivations model in a correct and complete way the operational behavior determined by the above delay declaration. Formally, when we consider simply moded queries, we have that:

- we can employ Theorem 5.14 to demonstrate termination,
- by Theorem 4.6,  $PM_P^{SL}$  characterizes the behavior of APPEND in terms of the intermediate computed answer substitutions.

**Example 6.4** Consider PERMUTE4, i.e., PERMUTE of Example 5.15, with the modes `permute(I, O)`, `insert(I, I, O)`. Consider the following delay declarations for it:

```

delay permute(Xs, _) until nonvar(Xs)
delay insert(Xs, -, _) until nonvar(Xs)

```

---

<sup>4</sup>An interesting example suggested by Apt of a contrived query that does not terminate in combination of the above program is `append([X|Xs], [], Xs)`. Notice that this query is not simply moded. This demonstrates also the need for restricting to a class of “well formed” programs and queries such as that of simply moded ones.

The meaning of these declarations is equivalent to that of the previous example. It is not difficult to see that for the above program, for every derivation starting in a simply-moded query, the derivation is input consuming if and only if it is delay-respecting.

**Example 6.5** Consider again QUICKSORT. In the context of dynamic scheduling, its standard delay declarations are:

```

delay quicksort(Xs,_) until nonvar(Xs).
delay quicksort_dl(Xs,_,_) until nonvar(Xs).
delay partition(Xs,_,_,_) until nonvar(Xs).
delay =<(X,Y) until ground(X) and ground(Y).
delay >(X,Y) until ground(X) and ground(Y)

```

While the first three declarations are equivalent to those used above, the last two state that an atom of the form  $a =< b$  (resp.  $a > b$ ) can be selected iff both  $a$  and  $b$  are ground terms.

Now, if we think of the built-ins  $>$  and  $=<$  as being defined by a program containing infinitely many ground facts of the form  $>(n, m)$ , with  $n$  and  $m$  being two appropriate integers, the derivations respecting the above delay declarations are exactly the input consuming ones.

## 7 Conclusion and Related Works

In this article, we have studied the termination of input consuming programs. In order to do this, we have provided a denotational semantics for input consuming derivations that models the results of *incomplete* derivations. This semantics uses a variant of the well-known  $T_P$ -operator.

In a previous paper [6] we have introduced a different semantics for input consuming programs. The two semantics, however, are quite orthogonal to each other: while that of [6] models exclusively the result of successful derivations and requires the program to be *well-moded* and *nicey-moded*, the semantics used here models the results of also incomplete derivations and requires programs and queries to be simply moded.

As mentioned in Subsection 4.2, in the context of parallelism and concurrency [21], one can have derivations that never *succeed*, and yet compute substitutions. Thus we have provided a denotational semantics for such programs/programming languages, which goes beyond the usual success-based SLD-resolution mechanism of logic programming.

Input consuming derivations bear a certain resemblance with derivations in the language of *Moded (Flat) GHC* [32]. Actually, input-consuming programs can be seen as a simplified version of moded GHC, and the results we provide here can be thus applied to some moded GHC programs. We want to note however that Moded (F)GHC is a full-fledged programming paradigm, while input-consuming programs are meant for abstraction purposes. In fact, Moded (F)GHC enjoys a more complex computational mechanisms: In (F)GHC, a

clause has the form  $H \leftarrow \mathbf{G} \mid \mathbf{B}$ , where  $\mathbf{G}$  is called a *guard*. An atom  $A$  can be resolved using  $H \leftarrow \mathbf{G} \mid \mathbf{B}$  only when  $A$  is an instance of  $H$  and  $\mathbf{G}\theta$  is entailed, where  $\theta$  is an mgu of  $A$  and  $H$ . The atom  $A$  can become instantiated only later via explicit unifications occurring in  $\mathbf{B}$ . In Moded (F)GHC, there are (non-trivial) conditions on clauses ensuring that when an argument position in  $A$  is input, then the clause used to resolve  $A$  will never (not even via later resolution steps) cause any bindings to that position.

Falaschi *et al.* [14] have defined a denotational semantics for CLP programs with dynamic scheduling of a somewhat different kind: the semantics of a query is given by a set of closure operators; each operator is a function modeling a possible effect of resolving the query on a program state (i.e., constraint on the program variables). Their semantics is the analogue of the bottom-up  $S$ -semantics [9] for usual logic programs, where atoms are mapped to their set of answers. In this respect, it corresponds to the semantics defined in [6]. The approach presented here is more suited to termination proofs since we deal with partial answers.

Concerning termination, we have provided a necessary and sufficient criterion for it, applicable to a wide class of programs, namely the class of simply moded programs. In previous papers [7, 27], we have already addressed the problem of the termination of input consuming programs. The results we present here constitute a big improvement with respect to [7, 27] in that we can now capture (by means of the model) the inter-argument relationships in the bodies of the clauses. This improvement allows us to give a *necessary* and sufficient condition for termination. In fact, we can now prove the termination of programs employing a non-trivial recursion scheme such as QUICKSORT, PERMUTE3, TRANSPOSE; this was not possible using previous sufficient conditions of [7, 27] (though, with the tools of [7, 27] we could prove the termination of PERMUTE4, which employs *direct recursion*).

Finally, we have provided some examples showing analogies between the use of delay declarations and input consuming derivations. A technical result demonstrating equivalence (under some syntactically-checkable assumption) is given in [8].

To conclude, we discuss some other works about termination of programs with dynamic scheduling. First note that those works are usually about termination of programs with delay declarations, whereas we consider the more abstract notion of input consuming derivations. As has been argued before [27], this allows us to see more clearly which programs terminate under which assumptions about the selection rule.

Apt and Luitjes [4] give conditions for the termination of `append`, but those are ad-hoc and do not address the general problem. Naish [20] gives heuristics to ensure termination, but no formal results.

There are several works in this area making assumptions about the selection rule that are stronger than assuming input consuming derivations [16, 17, 18].

Marchiori and Teusink [17] assume a *local selection rule*, that is a rule under which only most recently introduced atoms can be resolved in each step. Moreover, it is assumed that an atom is only selected once it is *bounded* with respect



to a level mapping, that is, any instance of the selected atom has a level that is below a certain bound. This is in contrast to our approach where any selected atom, even one that is non-ground in its input, has a well-defined level, but this level is not stable under instantiation.

Martin and King [18] achieve a similar effect by bounding the depth of the computation introducing auxiliary predicates.

It is more difficult to assess Lüttringhaus-Kappel [16] since his contribution is mainly to *generate* delay declarations automatically rather than *prove* termination. However in some cases, the delay declarations that are generated require an argument of an atom to be a rigid list before that atom can be selected, which is similar to the above approaches [17, 18]. Such uses of delay declarations go well beyond ensuring that derivations are input consuming.

Some authors have considered a selection rule stating that in each derivation step, the leftmost selectable atom is selected [4, 10, 20]. Due to the problem of simultaneously reawaken atoms, this rule is actually not exactly the one implemented in most Prolog versions, but this has been corrected by proposing the *left-based* derivations [29]. Here it is enough to recall that such derivations “prefer” to select atoms that occur on the left of a query, which is an assumption made in addition to input consuming derivations. As already shown (Left-Switching Lemma) for nicely or simply moded programs and queries this assumption does not influence the set of computed answer substitutions but may affect partial computed answer as well as termination.

A survey classifying logic programs according to the selection rules for which they terminate can be found in [23]. Among others, this survey considers input termination and termination with respect to local selection rules as mentioned above [17].

The specific problem of termination of input consuming derivations has been treated also in [7] where nicely moded programs have been studied. By applying those results to simply moded programs we obtain a characterization of a proper subset of input terminating and simply moded programs. This class does not contain programs like `quicksort` whose termination proof needs information on partial computed answer substitutions.

## APPENDIX

PROOF OF LEMMA 3.8 First notice that, since  $A$  is a simply-moded atom,  $Var(In(A)) \cap Var(Out(A)) = \emptyset$ ; therefore, by properties of mgu’s (see [2, Corollary 2.25]), there exist substitutions  $\sigma_0$  and  $\sigma_1$  such that

- $\sigma_0 = mgu(In(A), In(H))$ ,
- $\sigma_1 = mgu(Out(A)\sigma_0, Out(H)\sigma_0)$ ,
- $\sigma_0\sigma_1 = mgu(A, H)$ ,

and all those mgu’s are relevant. Since, by hypothesis,  $\vartheta = mgu(A, H)$  and  $In(A\vartheta) = In(A)$ ,  $In(A)$  is an instance of  $In(H)$ . In particular,  $In(H)\sigma_0 = In(A)$  and thus

- $Dom(\sigma_0) \subseteq Var(In(H))$ ,
- $Ran(\sigma_0) \subseteq Var(In(A))$ .

Since  $Var(In(A))$  is fresh with respect to  $H$ , this means that  $\sigma_0$  is simply local with respect to the clause  $H \leftarrow$ . Moreover, by relevance of  $\sigma_0$ , simple modedness of  $A$  and the fact that  $A$  and  $H$  are variable disjoint, it follows that  $Dom(\sigma_0) \cap Var(Out(A)) = \emptyset$ . Hence,  $\sigma_1 = mgu(Out(A), Out(H)\sigma_0)$ . By simple modedness of  $A$ , the fact that  $Out(A)$  is sequence of distinct variables and that  $\sigma_1$  is relevant, we can assume that  $Out(A)\sigma_1 = Out(H)\sigma_0$  and thus

- $Dom(\sigma_1) \subseteq Var(Out(A))$ ,
- $Ran(\sigma_1) \subseteq Var(Out(H)\sigma_0) \subseteq (Ran(\sigma_0) \cup Var(Out(H))) \subseteq (Var(In(A)) \cup Var(Out(H)))$ .

Since  $Var(Out(H))$  is fresh with respect to  $A$ , this means that  $\sigma_1$  is simply local with respect to the query  $A$ .  $\square$

PROOF OF LEMMA 3.10 Since both  $Q$  and  $c$  are simply moded, by Lemma 2.7 also  $Q'$  is simply moded. Then by Lemma 3.5 there exist  $\alpha$  and  $\beta$  such that

- (a)  $\theta = \alpha\beta$ ;
- (b)  $\alpha = \theta|_{\mathbf{B}\vartheta}$  is simply local with respect to  $\mathbf{B}\vartheta$ ;
- (c)  $\beta$  is simply local with respect to  $\mathbf{R}\vartheta\alpha$ ;
- (d)  $\alpha$  and  $\beta$  are variable compatible with respect to  $\mathbf{B}\vartheta$  and  $\mathbf{R}\vartheta$ .

The proof proceeds by proving that

- (a1)  $(\vartheta\theta)|_Q = (\vartheta\alpha)|_A\beta$ ;
- (b1)  $(\vartheta\alpha)|_A$  is simply local with respect to  $A$ ;
- (c1)  $\beta$  is simply local with respect to  $\mathbf{R}(\vartheta\alpha)|_A$ ;
- (d1)  $(\vartheta\alpha)|_A$  and  $\beta$  are variable compatible with respect to  $A$  and  $\mathbf{R}$ .

The result will follow by applying again Lemma 3.5.

(a1) follows from the fact that  $(\vartheta\alpha)|_A\beta = (\vartheta\alpha)|_Q\beta = (\vartheta\alpha\beta)|_Q = (\vartheta\theta)|_Q$ .

To prove (b1) we prove that

- (b11)  $Dom(\vartheta\alpha)|_A \subseteq Var(Out(A))$
- (b12)  $Ran(\vartheta\alpha)|_A \subseteq Var(In(A)) \cup V$  where  $V \cap Var(A) = \emptyset$ .

(b11)  $Dom(\vartheta\alpha)|_A \subseteq Dom(\vartheta|_A) \cup Dom(\alpha|_A)$ . Now,  $Dom(\vartheta|_A) \subseteq Var(Out(A))$ , since  $\vartheta$  is a simply local mgu of  $A$  and  $H$ , and  $Dom(\alpha|_A) \subseteq Var(Out(\mathbf{B}\vartheta)) \cap Var(A)$ , since  $\alpha$  is simply local with respect to  $\mathbf{B}\vartheta$ . Then,  $Dom(\alpha|_A) \subseteq Var(Out(\mathbf{B})) \cap Var(A)$ , since  $Dom(\vartheta) \cap Var(Out(\mathbf{B})) = \emptyset$ . But,  $Var(Out(\mathbf{B})) \cap Var(A) = \emptyset$ , by standardization apart.

(b12) Since  $Ran(\vartheta|_A) \subseteq Var(Out(H))$ , then  $Ran((\vartheta\alpha)|_A) \subseteq Var(Out(H)) \cup Var(In(A)) \cup Var(\mathbf{B}) \cup V_1 \subseteq Var(In(A)) \cup V_1 \cup Var(c)$  where  $V_1$  is the set of

fresh variables of  $\alpha$  and  $V_1 \cup \text{Var}(c)$  is disjoint from  $A$  by standardization apart and lemma's hypothesis.

(c1) holds since  $\beta$  is simply local with respect to  $\mathbf{R}\vartheta\alpha$  and  $\mathbf{R}(\vartheta\alpha)|_A = \mathbf{R}(\vartheta\alpha)|_Q = \mathbf{R}\vartheta\alpha$ .

Finally, (d1) follows from (d), the assumption on the fresh variables of  $\theta$  (which implies that the sets of fresh variables of  $\alpha$  and  $\beta$  are disjoint from  $\text{Var}(Q)$  and  $\text{Var}(c)$ ) and the fact that  $\vartheta$  is a simply local mgu.  $\square$

PROOF OF LEMMA 4.3 We first prove that  $T_P^{SL} \uparrow \omega(I)$  is a fixpoint of  $T_P^{SL}$ . In fact

$$\begin{aligned} T_P^{SL}(T_P^{SL} \uparrow \omega(I)) &= T_P^{SL} \uparrow \omega(I) \cup T_P^{sl}(T_P^{SL} \uparrow \omega(I)) \\ &= \bigcup_{i \geq 0} T_P^{SL} \uparrow i(I) \cup \bigcup_{i \geq 0} T_P^{sl}(T_P^{SL} \uparrow i(I)) \\ &= \bigcup_{i \geq 0} (T_P^{SL} \uparrow i(I) \cup T_P^{sl}(T_P^{SL} \uparrow i(I))) \\ &= \bigcup_{i \geq 0} T_P^{SL} \uparrow i(I) \\ &= T_P^{SL} \uparrow \omega(I). \end{aligned}$$

We now prove that  $T_P^{SL} \uparrow \omega(I)$  is the least fixpoint of  $T_P^{SL}$  containing  $I$ .

Let  $J$  be a fixpoint of  $T_P^{SL}$  containing  $I$ , i.e.,  $I \subseteq J = T_P^{SL}(J)$ . We prove that  $T_P^{SL} \uparrow \omega(I) \subseteq J$ . More precisely, we prove by induction on  $i$ , that for all  $i \geq 0$ ,  $T_P^{SL} \uparrow i(I) \subseteq J$ .

*Base.*  $i = 0$ . In this case  $T_P^{SL} \uparrow 0(I) = I \subseteq J$ .

*Induction step.*  $i > 0$ . In this case  $T_P^{SL} \uparrow i(I) = T_P^{SL}(T_P^{SL} \uparrow i - 1(I))$ . By the inductive hypothesis,  $T_P^{SL} \uparrow i - 1(I) \subseteq J$ . By monotonicity of  $T_P^{SL}$ ,  $T_P^{SL} \uparrow i(I) = T_P^{SL}(T_P^{SL} \uparrow i - 1(I)) \subseteq T_P^{SL}(J) = J$ .

By definition of simply local models and of  $T_P^{SL}$ , we have that  $J$  is a simply local model of  $P$  containing  $I$  iff  $T_P^{SL}(J) \subseteq J$  and  $I \subseteq J$ . This proves that  $T_P^{SL} \uparrow \omega(I)$  is the least simply local model of  $P$  containing  $I$ .  $\square$

PROOF OF LEMMA 4.4 (i)  $\Rightarrow$  (ii). We first assume that  $\delta$  proceeds left-to-right and employs only simply local mgu's and prove that:  $\vartheta|_{\mathbf{A}}$  is simply local with respect to  $\mathbf{A}$  and  $\mathbf{A}\vartheta \subseteq T_P^{SL} \uparrow \omega(I)$ . The general case follows from Corollary 2.12 and Theorem 3.18 in [2] on derivations employing different mgu's.

We proceed by induction on the length of  $\delta$ .

*Base.*  $\text{len}(\delta) = 0$ . In this case  $\mathbf{A} = \mathbf{C} \subseteq I$  and  $\vartheta = \epsilon$  (the empty substitution). The thesis follows from the fact that, by definition of  $T_P^{SL}$ ,  $I \subseteq T_P^{SL} \uparrow \omega(I)$ .

*Induction step.*  $\text{len}(\delta) > 0$ . Let  $\mathbf{A} = \mathbf{L}, A, \mathbf{R}$  and  $A$  be the leftmost atom of  $\mathbf{A}$  such that there is some  $A$ -step in  $\delta$  (and hence there are no  $\mathbf{L}$ -steps in  $\delta$ ). Assume also that  $c : H \leftarrow \mathbf{B}$  is the input clause used in the first derivation step of  $\delta$  and  $\vartheta_1$  is the simply local mgu of  $A$  and  $H$  used in this step. By Corollary 2.12,

$$\delta : \mathbf{A} \xrightarrow{\vartheta_1} (\mathbf{L}, \mathbf{B}, \mathbf{R})\vartheta_1 \xrightarrow{\vartheta_2} \mathbf{L}, \mathbf{C}'$$

such that  $\mathbf{C} = \mathbf{L}, \mathbf{C}'$ ,  $\vartheta = \vartheta_1\vartheta_2$  and  $\mathbf{L}\vartheta_1 = \mathbf{L}\vartheta_1\vartheta_2 = \mathbf{L}$ .

Hence

$$\mathbf{L} \subseteq I \subseteq T_P^{SL} \uparrow \omega(I) \quad (3)$$

and there exists the input consuming derivation:  $\delta' : (\mathbf{B}, \mathbf{R})\vartheta_1 \xrightarrow{\vartheta_2} \mathbf{C}'$  where  $len(\delta') = len(\delta) - 1$  and  $(\mathbf{B}, \mathbf{R})\vartheta_1$  is simply moded.

By the inductive hypothesis,  $\vartheta_2|_{(\mathbf{B}, \mathbf{R})\vartheta_1}$  is simply local with respect to  $(\mathbf{B}, \mathbf{R})\vartheta_1$  and

$$(\mathbf{B}, \mathbf{R})\vartheta_1\vartheta_2 \subseteq T_P^{SL} \uparrow \omega(I). \quad (4)$$

Note also that since  $\vartheta_1\vartheta_2$  is computed in a derivation of  $(A, \mathbf{R})$ , by standardization apart and Lemma 3.10 we have that

$$(\vartheta_1\vartheta_2|_{(\mathbf{B}, \mathbf{R})\vartheta_1})|_{(A, \mathbf{R})} = (\vartheta_1\vartheta_2)|_{(A, \mathbf{R})} \text{ is simply local with respect to } (A, \mathbf{R}). \quad (5)$$

Since  $(\vartheta_1\vartheta_2)|_{\mathbf{L}} = \varepsilon$  and  $(\vartheta_1\vartheta_2)|_{(A, \mathbf{R})}$  is simply local with respect to  $(A, \mathbf{R})$  and the fact that variable compatibility is guaranteed by standardization apart, by Lemma 3.5

$$(\vartheta_1\vartheta_2)|_{(\mathbf{L}, A, \mathbf{R})} \text{ is simply local with respect to } (\mathbf{L}, A, \mathbf{R}). \quad (6)$$

To conclude the proof it remains to shown that

$$A\vartheta_1\vartheta_2 \subseteq T_P^{SL} \uparrow \omega(I). \quad (7)$$

Then, the result will follow from (3), (4), (6) and (7).

In order to prove (7) note that  $\vartheta_1$  is a simply local mgu of  $A$  and  $H$ , so  $(\vartheta_1)|_H$  is simply local with respect to  $H \leftarrow$ . Moreover, by Lemma 3.5,  $(\vartheta_2)|_{\mathbf{B}\vartheta_1}$  is simply local with respect to  $\mathbf{B}\vartheta_1$ . Note also that, by standardization apart,  $\vartheta_1|_H$  and  $\vartheta_2|_{\mathbf{B}\vartheta_1}$  are variable compatible with respect to  $H$  and  $\mathbf{B}$ . Hence, by Lemma 3.6,  $(\vartheta_1)|_H(\vartheta_2)|_{\mathbf{B}\vartheta_1} = (\vartheta_1\vartheta_2)|_c$  is simply local with respect to  $c$ .

By Definition 4.2 and property (4), this proves that

$$H(\vartheta_1)|_H(\vartheta_2)|_{\mathbf{B}\vartheta_1} = H\vartheta_1\vartheta_2 = A\vartheta_1\vartheta_2 \subseteq T_P^{SL} \uparrow \omega(I)$$

(ii)  $\Rightarrow$  (i). Let  $\mathbf{A}\theta \subseteq T_P^{SL} \uparrow \omega(I)$  with  $\mathbf{A} : A_1, \dots, A_n$ . Let  $k$  be the minimum index such that  $\mathbf{A}\theta \in T_P^{SL} \uparrow k(I)$ . The proof proceeds by induction on  $k$ .

*Base.*  $k = 0$ . In this case,  $\mathbf{A}\theta \subseteq T_P^{SL} \uparrow 0(I) = I$  with  $\theta$  simply local with respect to  $\mathbf{A}$ . Since both  $\mathbf{A}$  and  $\mathbf{A}\theta$  consist of simply moded atoms, and  $\theta$  is a simply local substitution with respect to  $\mathbf{A}$ , it follows that  $\theta$  is just a renaming of the output variables of  $\mathbf{A}$ . The thesis follows by taking  $\vartheta$  to be the empty substitution and  $\delta$  to be the derivation of length zero.

*Induction step.*  $k > 0$ . We proceed by induction on  $n$ , the number of atoms in the query.

*Base.*  $n = 1$ . In this case  $\mathbf{A} = A$ ,  $\theta$  is simply local with respect to  $A$  and  $\mathbf{A}\theta \in T_P^{SL} \uparrow k(I)$ . By definition of  $T_P^{SL}$  and Proposition 3.3, there exist a variant  $c : H \leftarrow \mathbf{B}$  of a clause of  $P$  variable disjoint from  $A$  and a substitution  $\vartheta$  such that

$$\vartheta \text{ is simply local with respect to } c \quad (8)$$

$$\mathbf{B}\vartheta \subseteq T_P^{SL} \uparrow (k-1)(I) \quad (9)$$

$$A\theta = H\vartheta. \quad (10)$$

By (8) and Lemma 3.6 there exist  $\sigma_0$  and  $\sigma_1$  such that  $\vartheta = \sigma_0\sigma_1$ ,  $\sigma_0 = \vartheta|_H$  is simply local with respect to  $H \leftarrow$  and  $\sigma_1$  is simply local with respect to  $\mathbf{B}\sigma_0$ .

Hence, by (9) and the inductive hypothesis on  $k$ , there exists an input consuming derivation

$$\delta' : \mathbf{B}\sigma_0 \xrightarrow{\vartheta_2} \mathbf{C}$$

where  $\mathbf{C} \subseteq I$  and  $\mathbf{B}\sigma_0\vartheta_2 \approx \mathbf{B}\sigma_0\sigma_1$ .

Note also that  $H\sigma_0\sigma_1 \approx H\sigma_0\vartheta_2$ , since the only variables of  $H\sigma_0$  which can be affected by  $\sigma_1$  or  $\vartheta_2$  are those occurring also in  $\mathbf{B}\sigma_0$ .

Finally, note that by Proposition 3.3 we can assume  $\text{Var}(A) \cap \text{Var}(c) = \emptyset$  and then by (10) and the fact that  $\theta$  is simply local with respect to  $A$  (which implies that  $\text{In}(A) = \text{In}(A\theta)$ ),  $\theta\sigma_0$  is a simply local mgu of  $A$  and  $H$ , and

$$\delta : A \xrightarrow{\theta\sigma_0} \mathbf{B}\sigma_0 \xrightarrow{\vartheta_2} \mathbf{C}$$

is an input consuming derivation where  $A\theta\sigma_0\vartheta_2 = H\sigma_0\vartheta_2 \approx H\vartheta = A\theta$ .

*Induction step.*  $n > 1$ . In this case  $\mathbf{A} = A, \mathbf{R}$  and  $\mathbf{A}\theta \in T_P^{SL} \uparrow k(I)$ . By Lemma 3.5 there exist  $\theta_1$  and  $\theta_2$  such that  $\theta = \theta_1\theta_2$ ,  $\theta_1 = \theta|_A$  is simply local with respect to  $A$  and  $\theta_2$  is simply local with respect to  $\mathbf{R}\theta_1$ , and  $\theta_1$  and  $\theta_2$  are variable compatible with respect to  $A$  and  $\mathbf{R}$ . By the inductive hypothesis on  $n$ ,

$$\delta_1 : A \xrightarrow{\vartheta_1} \mathbf{C}_1$$

where  $\mathbf{C}_1 \subseteq I$  and  $A\vartheta_1$  and  $A\theta|_A = A\theta$  are variant.

Again by the inductive hypothesis on  $n$ , there exists an input consuming derivation

$$\delta'_2 : \mathbf{R}\theta_1 \xrightarrow{\vartheta'_2} \mathbf{C}'_2$$

where  $\mathbf{C}'_2 \subseteq I$  and  $\mathbf{R}\theta_1\vartheta'_2 \approx \mathbf{R}\theta_1\theta_2$ . Since  $\mathbf{R}\theta_1 \approx \mathbf{R}\vartheta_1$ , by Lemma 2.9 there exists

$$\delta_2 : \mathbf{R}\vartheta_1 \xrightarrow{\vartheta_2} \mathbf{C}_2$$

where  $\mathbf{C}_2 \subseteq I$  and  $\mathbf{R}\vartheta_1\vartheta_2 \approx \mathbf{R}\theta_1\vartheta'_2$ . Without loss of generality, we can assume that the input clauses used in  $\delta_2$  are standardized apart with respect to  $\delta_1$ .

Then there exist  $\delta$ ,

$$\delta : A, \mathbf{R} \xrightarrow{\vartheta_1} \mathbf{C}_1, \mathbf{R}\vartheta_1 \xrightarrow{\vartheta_2} \mathbf{C}_1\mathbf{C}_2$$

such that  $\mathbf{A}\vartheta_1\vartheta_2$  and  $\mathbf{A}\theta$  are variant.  $\square$

The following result is a corollary of the above proof. It states that the relation between computed answers of input consuming derivations employing simply local mgu's and simply local substitutions.

**Proposition A.1** *Let  $\mathbf{A}$  be a simply moded query and  $P$  a simply moded program. Let  $\delta : \mathbf{A} \xrightarrow{\vartheta_1} \mathbf{C}_1 \xrightarrow{\vartheta_2} \mathbf{C}_2$  be an input consuming derivation in  $P$  that proceeds left-to-right and employs only simply local mgu's. Let  $c : H \leftarrow \mathbf{B}$  be the input clause used in the first derivation step of  $\delta$  and  $\vartheta_1$  be a simply local mgu employed in this step. Then  $(\vartheta_1\vartheta_2)|_{\mathbf{A}}$  is simply local with respect to  $\mathbf{A}$  and  $(\vartheta_1\vartheta_2)|_c$  is simply local with respect to  $c$ .*

**Proof.** It follows from (5) in the proof of Lemma 4.4 (the proof above).  $\square$

## References

- [1] K. R. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier and The MIT Press, Amsterdam and Cambridge, MA, 1990.
- [2] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, London, 1997.
- [3] K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS'93)*, volume 711 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, Germany, 1993. Springer-Verlag.
- [4] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the 4th International Conference on Algebraic Methodology and Software Technology, (AMAST'95)*, volume 936 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, Germany, 1995. Springer-Verlag.
- [5] K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, Oxford, UK, 1994.
- [6] A. Bossi, S. Etalle, and S. Rossi. Semantics of well-moded input-consuming logic programs. *Computer Languages*, 26(1):1–25, 2000.
- [7] A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. *Theory and Practice of Logic Programming*, 2(2):125–154, 2002.
- [8] A. Bossi, S. Etalle, S. Rossi, and J.-G. Smaus. Semantics and termination of simply-moded logic programs with dynamic scheduling. In D. Sands, editor, *Proceedings of the European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 402–416, Genova, Italy, 2001. Springer-Verlag.

- [9] A. Bossi, M. Gabrielli, G. Levi, and M. Martelli. The S-semantics approach: Theory and applications. *The Journal of Logic Programming*, 19 & 20:149–198, May 1994.
- [10] J. Boye. *Directional Types in Logic Programming*. PhD thesis, Linköpings Universitet, 1996.
- [11] F.S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development, (TAPSOFT/CAAP)*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319, Brighton, UK, 1991. Springer-Verlag.
- [12] S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *Journal of Logic Programming*, 38(2):243–257, 1999.
- [13] S. Etalle, M. Gabbrielli, and M. C. Meo. Transformations of ccp programs. *ACM Transactions on Programming Languages and Systems*, 23(3):304–395, 2002.
- [14] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Constraint logic programming with dynamic scheduling: a semantics based on closure operators. *Information and Computation*, 137(1):41–67, 1997.
- [15] J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, Berlin, Germany, 1987. Second edition.
- [16] S. Lüttringhaus-Kappel. Control generation for logic programs. In D.S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 478–495, Budapest, Hungary, 1993. The MIT Press.
- [17] E. Marchiori and F. Teusink. Termination of logic programs with delay declarations. *Journal of Logic Programming*, 39(1–3):95–124, 1999.
- [18] J. C. Martin and A. M. King. Generating efficient, terminating logic programs. In M. Bidoit and M. Dauchet, editors, *Proceedings of the 7th International Joint Conference on Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 273–184, Lille, France, 1997. Springer-Verlag.
- [19] L. Naish. *Negation and control in Prolog*, volume 238 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1986.
- [20] L. Naish. Coroutining and the construction of terminating logic programs. *Australian Computer Science Communications*, 15(1):181–190, 1993.
- [21] L. Naish. Parallelizing NU-Prolog. In K. A. Bowen and R. A. Kowalski, editors, *Proceedings of the 5th International Conference/Symposium on Logic Programming*, pages 1546–1564, Seattle, Washington, August 1988. The MIT Press.

- [22] L. Naish. An introduction to MU-Prolog. Technical Report 82/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, March 1982 (Revised July 1983).
- [23] D. Pedreschi, S. Ruggieri, and J.-G. Smaus. Classes of terminating logic programs. *Theory and Practice of Logic Programming*, 3(2):369–418, 2002.
- [24] S. Ruggieri. Termination of Constraint Logic Programs. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *Lecture Notes in Computer Science*, pages 838–848, Bologna, Italy, 1997. Springer-Verlag.
- [25] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, California, 1990. ACM, New York.
- [26] J.-G. Smaus. *Modes and Types in Logic Programming*. PhD thesis, University of Kent at Canterbury, 1999. Available from <http://www.cs.ukc.ac.uk/pubs/1999/986/>.
- [27] J.-G. Smaus. Proving termination of input-consuming logic programs. In D. De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming*, pages 335–349, Las Cruces, New Mexico, USA, 1999. The MIT Press.
- [28] J.-G. Smaus, P. M. Hill, and A. M. King. Termination of logic programs with `block` declarations running in several modes. In C. Palamidessi, editor, *Proceedings of the 10th Symposium on Programming Language Implementations and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 73–88, Pisa, Italy, 1998. Springer-Verlag.
- [29] J.-G. Smaus, P. M. Hill, and A. M. King. Verifying termination and error-freeness of logic programs with `block` declarations. *Theory and Practice of Logic Programming*, 1(4):447–486, 2001.
- [30] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, MA, 1986.
- [31] K. Ueda and K. Furukawa. Transformation rules for GHC Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 582–591, Tokyo, Japan, 1988. Institute for New Generation Computer Technology, Tokyo, OHMSHA Ltd. Tokyo and Springer-Verlag.
- [32] K. Ueda and M. Morita. Moded flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.
- [33] M. H. van Emden and G. J. de Lucena. Predicate logic as a language for parallel programming. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, London, 1982. Academic Press.