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
and
JAN-GEORG SMAUS
Universität Freiburg

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.

Categories and Subject Descriptors: F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Logic programming*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program verification*

General Terms: Languages, Verification, Termination

Additional Key Words and Phrases: Logic programs, dynamic scheduling, simply moded, termination

1. INTRODUCTION

Logic programming is based on giving a computational interpretation to a fragment of first order logic. Kowalski [Kowalski 1979] advocates the separation of the *logic* and *control* aspects of a logic program and has coined the famous formula

Algorithm = Logic + Control.

Contact author's address: Sabina Rossi, Dipartimento di Informatica, Università Ca' Foscari di Venezia, via Torino 155, 30172 Venezia, Italy.
This article is based on an earlier conference paper [Bossi et al. 2001].

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.

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 paradigms, 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[1]. Such a mechanism is provided in modern logic programming languages in the form of *delay declarations* (also called *when declarations* [Naish 1986]). 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).
```

[1]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 to a wrong output.

```
    delay write_list(Ls,_) until nonvar(Ls).
```

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 [Naish 1988] and – as observed by Apt and Luitjes [Apt and Luitjes 1995] – "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 [Smaus 1999a] introduced *input consuming derivations*, a formalism very similar to the one of *Moded GHC* [Ueda and Morita 1994]. 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 [Bossi et al. 2001] 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)[2], `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 preventing pos-

[2]In this mode, the first two positions are considered *input positions*, while the rightmost one is an *output* one.

sible 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 [Apt and Pedreschi 1994], 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 [Bossi et al. 2001], 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 [Smaus 1999b] and extended in [Bossi et al. 2002] 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 stretching 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 [Apt 1990; 1997; Lloyd 1987]. 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* θ 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 ϵ the empty substitution: $Dom(\epsilon) = Ran(\epsilon) = \emptyset$. Given a *substitution* σ and a syntactic object E, we denote by $\sigma_{|E}$ the restriction of σ 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 σ 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 θ 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 θ 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 θ of terms t and t' is called *relevant* iff $Var(\theta) \subseteq Var(t) \cup Var(t')$.

2.2 Programs and Derivations

Let $\mathcal{P}$ be a finite set of *predicate symbols*. An *atom* is an object of the form $p(t_1, \dots, t_n)$ 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 [Apt 1997], 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 θ. The query $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ is called a *resolvent of* $\mathbf{A}, B, \mathbf{C}$ *and* c *with selected atom* B *and mgu* θ. A *derivation step* is denoted by $\mathbf{A}, B, \mathbf{C} \overset{\theta}{\Longrightarrow}_{P,c} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$. The clause $H \leftarrow \mathbf{B}$ is called its *input clause*. The atom B is called the *selected atom* of $\mathbf{A}, B, \mathbf{C}$.

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

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

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

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

The following definition of **D**-step is due to Smaus [Smaus 1999a].

Definition 2.1 *(D-step).*

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

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

Intuitively, a **D**-step occurring in a derivation δ is a derivation step that concerns the derivation of the subquery **D** of some query in δ.

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., $\texttt{append}(I, I, O)$.

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 [Apt 1997]. In the sequel of the paper we will restrict ourselves to programs and queries which are *simply moded* [Apt and Etalle 1993].

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 [Smaus 1999a] as formalism for describing dynamic scheduling in an abstract way and is defined as follows.

Definition 2.5 *(input consuming).*

— A derivation step $\mathbf{A}, B, \mathbf{C} \overset{\theta}{\Longrightarrow} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ is *input consuming* if $In(B)\theta = In(B)$.
— 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 δ of `REVERSE` $\cup$ {`reverse([X1,X2],Zs)`} depicted below is input consuming.

```
δ:  reverse([X1,X2],Zs) ⇒ reverse_acc([X1,X2],Zs,[ ]) ⇒
    reverse_acc([X2],Zs,[X1]) ⇒ reverse_acc([ ],Zs,[X2,X1]) ⇒ □.
```

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 [Apt and Luitjes 1995, 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 [Bossi et al. 2002] 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 $x \in Var(Q)$ and $x \notin Var(Out(Q))$, $x\theta = x$.*

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 ρ 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 θ then $A\rho$ and $H\rho$ are unifiable with mgu $\rho^{-1}\theta\rho$,
(3) if $In(A\theta) = In(A)$ then $In(A\rho\rho^{-1}\theta\rho) = In(A\theta\rho) = In(A\rho)$.

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

We recall below the Left-Switching Lemma that has been proven in [Bossi et al. 2002].

Lemma 2.10 (Left-Switching). *Let the program P and the query Q_0 be nicely moded. Let δ be a partial input consuming derivation of $P \cup \{Q_0\}$ of the form*

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

where

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

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

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

and

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

where δ' is input consuming and

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

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 δ' 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, δ is equivalent to a derivation δ' 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, θ_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 $c := H \leftarrow B_1, \ldots, B_n$ 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, $\sigma_0, \sigma_1 \ldots, \sigma_n$, one for each atom in the given clause, such that σ_0 binds the input variables of the head of the clause, and each σ_i $(i > 0)$ creates a binding between the output variables and the input terms of B_i (instantiated by the previous substitutions $\sigma_0, \ldots, \sigma_{i-1}$). The definition involves variable sets $v_0, v_1, \ldots, v_n$. Intuitively, the variables in v_0 come from the "caller" and the variables in $v_1, \ldots, v_n$ come from the "callees".

Definition 3.1 *(simply local substitution).* Let θ be a substitution. We say that θ is *simply local* wrt. the clause $H \leftarrow B_1, \ldots, B_n$ if there exist substitutions $\sigma_0, \sigma_1 \ldots, \sigma_n$ and disjoint sets of fresh (wrt. c) variables $v_0, v_1, \ldots, v_n$ such that $\theta = \sigma_0\sigma_1 \cdots \sigma_n$ 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 $Ran(\sigma_i) \subseteq Var(In(B_i)\sigma_0\sigma_1 \cdots \sigma_{i-1}) \cup v_i$.

The substitution θ is *simply local* wrt. a query $\mathbf{B}$ if θ is simply local wrt. the clause $q \leftarrow \mathbf{B}$ where q is any variable-free atom.

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

Note that in the case of a simply local substitution wrt. a query, σ_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

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

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

Consider now the query

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

The substitution $\theta = \{\texttt{Zs/[a,X,c|Ys]}\}$ is simply local wrt. Q. In fact $\theta = \sigma_1\sigma_2$ where $\sigma_1 = \{\texttt{Zs/[a,X,c|Ys]}\}$ and σ_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 ρ be a renaming. If the substitution θ is simply local wrt. c then the substitution $\rho^{-1}\theta\rho$ is simply local wrt. $c\rho$.*

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 ϑ_1 be a substitution simply local wrt. $\mathbf{A}$ and ϑ_2 be simply local wrt. $\mathbf{B}\vartheta_1$. Then ϑ_1 and ϑ_2 are *variable compatible* wrt. $\mathbf{A}$ and $\mathbf{B}$ if

— the set of fresh variables of ϑ_1 is disjoint from the set of fresh variables of ϑ_2,
— $Var(\mathbf{A}, \mathbf{B})$ is disjoint from the set of fresh variables of ϑ_1 and ϑ_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 θ simply local wrt. $\mathbf{A}, \mathbf{B}$ iff $\theta = \vartheta_1\vartheta_2$ where*

— *$\vartheta_1 = \theta_{|\mathbf{A}} = \theta_{|Out(\mathbf{A})}$ is simply local wrt. $\mathbf{A}$,*
— *$\vartheta_2 = \theta_{|\mathbf{B}} = \theta_{|Out(\mathbf{B})}$ simply local wrt. $\mathbf{B}\vartheta_1$,*
— *ϑ_1 and ϑ_2 are variable compatible wrt. $\mathbf{A}$ and $\mathbf{B}$.*

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

$\Rightarrow$) 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 $k \neq j$, $Dom(\sigma_k) \cap Dom(\sigma_j) = \emptyset$, $Out((A_{k+1}, \ldots, A_n)\sigma_1 \cdots \sigma_k) = Out(A_{k+1}, \ldots, A_n)$ and $((A_1, \ldots, A_k)\sigma_1 \cdots \sigma_k)\sigma_{k+1} \cdots \sigma_n = (A_1, \ldots, A_k)\sigma_1 \cdots \sigma_k$. Thus $\theta_{|\mathbf{A}} = \sigma_1 \cdots \sigma_i$ is simply local wrt. $\mathbf{A}$ and $\sigma_{i+1} \cdots \sigma_n$ is simply local wrt. $(A_{i+1}, \ldots, A_n)\sigma_1 \cdots \sigma_i$.

$\Leftarrow$) 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 wrt. $\mathbf{A}, \mathbf{B}$, it is sufficient to observe that by definition of simply local substitution and properties of simply moded queries, $Out((A_{i+1}, \ldots, A_n)\sigma_1 \cdots \sigma_i) = Out(A_{i+1}, \ldots, A_n)$ and hence for all $j \in [i+1..n]$, $Dom(\sigma_j) \subseteq Var(Out(A_j))$. The fact that ϑ_1 and ϑ_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. □

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 θ simply local wrt. c iff $\theta = \vartheta_0\vartheta_1$ where*

— *$\vartheta_0 = \theta_{|H} = \theta_{|In(H)}$ is simply local wrt. $H \leftarrow$,*
— *$\vartheta_1 = \theta_{|\mathbf{B}} = \theta_{|Out(\mathbf{B})}$ simply local wrt. $\mathbf{B}\vartheta_0$,*
— *ϑ_0 and ϑ_1 are variable compatible wrt. 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 θ be a mgu of A and H such that $In(A\theta) = In(A)$. We say that θ is a *simply local mgu* of A and H if $\theta = \sigma_0\sigma_1$ where σ_0 is simply local wrt. the clause $H \leftarrow$ and σ_1 is simply local wrt. the atom A.

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

Note that previous Lemma 3.8 together with Theorem 3.18 in [Apt 1997] (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 $\mathtt{p}/2$ in the mode $\mathtt{p}(I, O)$ and the atoms

$$A = \mathtt{p(f(X,Y),Z)} \qquad H = \mathtt{p(W,U)}.$$

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

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

but only the second one is simply local. Note also that when A and H are variable disjoint and ϑ is a simply local mgu of A and H then the variables in $Out(A)$ do not occur anymore in $A\vartheta$.

The next lemma shows a persistency property of simply local substitutions. It provides one of the key intuitions for the development of the bottom-up semantics of 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 $Q \overset{\vartheta}{\Longrightarrow} Q'$ be an input consuming derivation step obtained by using the simply moded clause $c : H \leftarrow \mathbf{B}$ and the simply local mgu ϑ. Let θ be a substitution simply local wrt. Q' such that the set of fresh variables of θ is disjoint from $Var(Q)$ and $Var(c)$. Then $(\vartheta\theta)_{|Q}$ is simply local wrt. 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 [Etalle et al. 2002; Saraswat and Rinard 1990] and GHC [Ueda and Furukawa 1988] often capture such intermediate results, or in any case, the results of non-successful computations [de Boer and Palamidessi 1991]. 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 $c : H \leftarrow B_1, \ldots, B_n$ if for every substitution θ simply local wrt. c,

$$\text{if } B_1\theta, \ldots, B_n\theta \in M \text{ then } H\theta \in M. \tag{1}$$

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 θ 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 wrt. `p(X)` $\leftarrow$ `q(X)`. On the other hand, any term model (see [Apt 1997]) 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

$$\begin{aligned} T_P^{sl}(I) = \{H\theta \mid\ & \exists\, c : H \leftarrow B_1, \ldots, B_n \text{ variant of a clause in } P, \\ & \theta \text{ is simply local wrt. } c, \\ & B_1\theta, \ldots, B_n\theta \in I\} \end{aligned}$$

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 θ simply local wrt. $\mathbf{A}$, 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.

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

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

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

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

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

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 [Apt and Pedreschi 1994; Ruggieri 1997].

Input consuming derivations were originally conceived as an abstract and "reasonably strong" assumption about the selection rule in order to prove termination [Smaus 1999b]. 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 $p(\ldots, s, \ldots) \leftarrow \mathbf{A}, p(\ldots, t, \ldots), \mathbf{C}$, 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 $p(\ldots) \leftarrow \mathbf{A}, p(\ldots), \mathbf{C}$, we need to take into account how $\mathbf{A}$ and $\mathbf{C}$ might instantiate the body atom $p(\ldots)$ 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 [Apt 1997].

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 $p \sqsupset q$ 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 [Etalle et al. 1999]. Generalizing the definition to arbitrary atoms is crucial for showing termination of input consuming derivations.

Definition 5.2 *(moded level mapping).* A function $|\ |$ 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 wrt.* a given class C of queries if all its input consuming derivations starting in query in C are finite.

In particular, we say that P is input terminating wrt. 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 [Apt and Pedreschi 1994].

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 wrt. the moded level mapping* $|\ |$ *and* M if for every variant $H \leftarrow \mathbf{A}, B, \mathbf{C}$ of c and every substitution θ simply local wrt. c,

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

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

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(X)` terminates by deadlock, while `q(a)` loops. Now consider the query `p(X),q(X)`. This query can yield to a nonterminating computation because the query `p(X)`, *before failing*, reports the partial answer {`X/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 {`X/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 wrt. simply moded queries.

5.2 Sufficiency of Simply Acceptability

The following corollary of [Bossi et al. 2002, 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 wrt. 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 wrt. 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 wrt. simply moded queries,*
— *P is simply acceptable wrt. M (and a moded level mapping $|\ |$).*

Then $P \cup R$ is input terminating wrt. simply moded queries.

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

We can now prove our theorem. By Corollary 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 wrt. 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 δ is an infinite input consuming derivation of $(P \cup R) \cup \{A\}$ such that A is defined in P. Then

$$\delta : A \overset{\vartheta_1}{\Longrightarrow} (B_1, \ldots, B_n)\vartheta_1 \overset{\vartheta_2}{\Longrightarrow} \cdots$$

where $c : H \leftarrow B_1, \ldots, B_n$ is the input clause used in the first derivation step and $\vartheta_1 = mgu(A, H)$. Clearly, $(B_1, \ldots, B_n)\vartheta_1$ 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 ϑ'_2,

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

$$A \overset{\vartheta_1}{\Longrightarrow} (B_1, \ldots, B_n)\vartheta_1 \overset{\vartheta'_2}{\longrightarrow} \mathbf{C}, (B_i, \ldots, B_n)\vartheta_1\vartheta'_2 \cdots;$$

(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 θ is simply local wrt. c (this is a consequence of Proposition A.1, reported in the appendix). Consider the instance $H\theta \leftarrow (B_1, \ldots, B_n)\theta$ of c. By Theorem 4.6, $(B_1, \ldots, B_{i-1})\theta \in M$.

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

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

Induction step. We distinguish two cases:

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

In case (a) we have $dep_P(Rel(A)) = dep_P(Rel(H\theta)) > dep_P(Rel(B_i\theta))$. Therefore,

$$\langle dep_P(Rel(A)), |A|\rangle = \langle dep_P(Rel(H\theta)), |H\theta|\rangle \succ \langle dep_P(Rel(B_i\theta)), |B_i\theta|\rangle.$$

In case (b), from the hypothesis that P is simply acceptable wrt. $|\ |$ and M, θ is simply local wrt. c and $(B_1, \ldots, B_{i-1})\theta \in M$, it follows that $|H\theta| > |B_i\theta|$. Consider the partial input consuming derivation $A \overset{\theta}{\longrightarrow} \mathbf{C}, (B_i, \ldots, B_n)\theta$. By Lemma 2.8 and the fact that $|\ |$ is a moded level mapping, we have that $|A| = |A\theta| = |H\theta|$. Hence, $\langle dep_P(Rel(A)), |A|\rangle = \langle dep_P(Rel(H\theta)), |H\theta|\rangle \succ \langle dep_P(Rel(B_i\theta)), |B_i\theta|\rangle$.

In both cases, the contradiction follows by the inductive hypothesis. $\square$

```
%   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,[],[]).
```

Fig. 1. The `QUICKSORT` program

The above theorem suggests proving termination in a modular way, i.e., extending a program that is already known to be input terminating wrt. 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 wrt. simply moded queries.*

PROOF. The proof follows from Theorem 5.6, by setting $R = \emptyset$. □

EXAMPLE 5.8. Figure 1 shows `quicksort` using a form of difference lists [Sterling and Shapiro 1986, program 15.3] (we permuted two body atoms for the sake of clarity). This program is simply moded wrt. the mode

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

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

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

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

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

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 $\texttt{partition}(t_1, t_2, t_3, t_4)$ where

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

The presence (or 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

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

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

— Consider `c1`, the first clause defining `quicksort_dl`. For every substitution θ, simply local wrt. `c1`, we have to show that
- If $\texttt{partition}(\texttt{Xs},\texttt{X},\texttt{Littles},\texttt{Bigs})\theta \in M$, then
$|\texttt{quicksort\_dl}([\texttt{X}|\texttt{Xs}],\texttt{Ys},\texttt{Zs})\theta| > |\texttt{quicksort\_dl}(\texttt{Bigs},\texttt{Ys1},\texttt{Zs})\theta|$.
This follows immediately from the definition of level mapping $|\ |$ and the fact that since $\texttt{partition}(\texttt{Xs},\texttt{X},\texttt{Littles},\texttt{Bigs})\theta \in M$, we have $len(\texttt{Bigs})\theta \leq len(\texttt{Xs})\theta$.
- If $(\texttt{partition}(\texttt{Xs},\texttt{X},\texttt{Littles},\texttt{Bigs}), \texttt{quicksort\_dl}(\texttt{Bigs},\texttt{Ys1},\texttt{Zs}))\theta \in M$, then
$|\texttt{quicksort\_dl}([\texttt{X}|\texttt{Xs}],\texttt{Ys},\texttt{Zs})\theta| > |\texttt{quicksort\_dl}(\texttt{Littles},\texttt{Ys},[\texttt{X}|\texttt{Ys1}])\theta|$.
This is analogous to the previous point and follows by the definition of $|\ |$ and the fact that since $\texttt{partition}(\texttt{Xs},\texttt{X},\texttt{Littles},\texttt{Bigs})\theta \in M$, $len(\texttt{Littles})\theta \leq len(\texttt{Xs})\theta$.

— Next, we consider `c2`. We have to show that for each simply local substitution θ such that $(\texttt{X =< Y})\theta \in M$,
$|\texttt{partition([X|Xs],Y,[X|Ls],Bs)}\theta| > |\texttt{partition(Xs,Y,Ls,Bs)}\theta|$.
This follows directly from the definition of $|\ |$ (the fact that $(\texttt{X =< Y})\theta \in M$ 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 $\texttt{quicksort}(t,x)$, 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 [Bossi et al. 2002] it is not possible to prove that `QUICKSORT` is input terminating (wrt. 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(a) ← q(X).
```

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

Notice that if we drop the requirement that θ must be simply local then we would have no guarantee that $|\mathtt{q(a)}\theta| > |\mathtt{q(X)}\theta|$: simply let $\theta = \{\mathtt{X/a}\}$.

5.3 Necessity of Simply Acceptability

We now prove the converse of Theorem 5.7, namely that our criterion for proving input termination wrt. 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 [Apt and Pedreschi 1994].

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' wrt. 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. □

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 $nodes^{ic}_P(Q)$ 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 wrt. simply moded queries and that $\mathbf{A}\theta \in PM^{SL}_P$, where θ is a simply local substitution wrt. $\mathbf{A}$. Then $nodes^{ic}_P(\mathbf{A}, B) \geq nodes^{ic}_P(B\theta)$.*

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 ϑ 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, $nodes_P^{ic}(\mathbf{A}, B) \geq nodes_P^{ic}(B\vartheta) = nodes_P^{ic}(B\theta)$. Hence the thesis. □

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

THEOREM 5.13. *Let P be a simply moded program. If P is input terminating wrt. simply moded queries then P is simply acceptable.*
In particular, it is simply acceptable wrt. PM_P^{SL} and a moded level mapping which is invariant under renaming.

PROOF. We show that there exists a moded level mapping $|\ |$ for P such that P is simply acceptable wrt. $|\ |$ 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:

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

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

We now prove that P is simply acceptable wrt. $|\ |$ 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 θ is a simply local substitution wrt. c. We show that

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

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

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

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

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

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

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

□

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 wrt. simply moded queries. In particular, if P is input terminating wrt. simply moded queries, then it is simply acceptable wrt. PM_P^{SL} and a moded level mapping which is invariant under renaming.*

Proof. By 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 rightmost 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 [Naish 1993], 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 wrt. $PM_{\texttt{PERMUTE2}}^{SL}$ and a moded level mapping which is invariant under renaming. By applying T_P^{SL} once to the the simply moded atom $\texttt{insert(Xs}',\texttt{X}',\texttt{Zs}')$ ($\texttt{Xs}',\texttt{X}',\texttt{Zs}'$ are fresh variables), one sees that $\texttt{insert}([\texttt{U}'|\texttt{Xs}'],\texttt{X}',[\texttt{U}'|\texttt{Zs}']) \in PM^{SL}_{\texttt{PERMUTE2}}$. The substitution $\{\texttt{Y}/\texttt{U}',\texttt{Ys}/\texttt{Zs}',\texttt{Zs}/[\texttt{U}'|\texttt{Xs}'],\texttt{X}/\texttt{X}'\}$ is simply local wrt. `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 $|\texttt{permute}([\texttt{X}'|\texttt{Xs}],[\texttt{U}'|\texttt{Zs}'])| > |\texttt{permute}(\texttt{Xs},[\texttt{U}'|\texttt{Xs}'])|$. This is a contradiction since a *moded* level mapping depends only on the input arguments (the second argument of `permute`).

Naish [Naish 1993] 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.[3] Note that `PERMUTE3` is simply moded, and consider the following level mapping:

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

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 and first arguments of `insert`. We define:

$$\begin{aligned} M = &\{\texttt{permute}(l,m) \mid \text{for all } l,m\}\\ \cup &\{\texttt{insert}(m,a,l) \mid \text{either } \texttt{insert}(m,a,l) \text{ is simply moded}\\ &\qquad\qquad\qquad\quad \text{or } len(l) > len(m) \} \end{aligned}$$

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 θ, $\texttt{insert}(\texttt{Xs},\texttt{X},[\texttt{H}|\texttt{T}])\theta \in M$ implies $\texttt{insert}([\texttt{U}|\texttt{Xs}],\texttt{X},[\texttt{U}|[\texttt{H}|\texttt{T}]])\theta \in M$. Hence M is a simply-local model.

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

To conclude, consider the program `PERMUTE4`: that is, `PERMUTE` together with the modes $\texttt{permute}(I,O)$, $\texttt{insert}(I,I,O)$. 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

```
permute([X|Xs],Ys) ← permute(Xs,Zs), insert(Zs,X,Ys).
permute([],[]).
```

Notice that the program is now input terminating wrt. simply moded queries. This is in fact the *natural* mode of the `PERMUTE` program. To demonstrate the

[3]We noted in [Smaus et al. 1998] that Naish's proposal for obtaining a terminating program does not work: For example, the query `permute(Xs,[1,2])` still loops. Indeed, following Naish's proposal we get an input terminating program. The problem is that his delay declarations do not ensure input consuming derivations, as noted in [Smaus 1999a].

termination one can apply Theorem 5.7 using *any* simply local model together with the following moded level mapping:

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

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:

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

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

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

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 however 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 θ simply-local wrt. `c2` `extract([X|L],[],X,L)`$\theta \in M$. But this holds by the model definition and the fact that for any substitution θ, we have that $len($`[X|L]`$\theta) > len($`[]`$\theta)$ and $len($`[X|L]`$\theta) > len($`L`$\theta)$.

Thirdly, we have to show that M is a simply-local model of `c3`. Consider any substitution θ such that `extract([H|T],S,Y,R)`$\theta \in M$. Since `[H|T]`θ cannot be a variable, by the definition of M, $len($`[X|[H|T]]`$\theta) > len($`[X|S]`$\theta)$ and $len($`[X|[H|T]]`$\theta) > len($`R`$\theta)$; thus `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 wrt. M and $|\ |$ and hence input terminating wrt. simply moded queries. The only non-trivial case is clause `c1`. For every simply local substitution θ, we must show that

(1) If `extract([H|T],LA,X,LB)`$\theta \in M$
    then $|$`list_tree([H|T],tree(TA,X,TB))`$\theta| > |$`list_tree(LA,TA)`$\theta|$.
(2) If `extract([H|T],LA,X,LB)`θ, `list_tree(LA,TA)`$\theta \in M$
    then $|$`list_tree([H|T],tree(TA,X,TB))`$\theta| > |$`list_tree(LB,TB)`$\theta|$.

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

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

Example 6.2. Consider the following program `TRANSPOSE` for transposing a matrix. A matrix is represented as a list of lists: `[[a,b,c],[1,2,3]]` 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(I).
```

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:

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

where $\mathit{size}(f(t_1,\dots,t_n)) = 1 + \mathit{size}(t_1) + \cdots + \mathit{size}(t_n)$ for $n \geq 0$, and $\mathit{size}(t) = 0$ 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

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

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

— If $\texttt{cut\_col}(\texttt{Rs},\texttt{C2},\texttt{Rs2})\theta$ is simply-moded, then
$\texttt{Rs2}\theta$ is a variable, thus
$\mathit{size}(\texttt{[[E|Es]|Rs]}\theta) > \mathit{size}(\texttt{[Es|Rs2]}\theta)$ and therefore
$\texttt{cut\_col([[E|Es]|Rs],[E|C2],[Es|Rs2])}\theta \in M$.

— If $\texttt{Rs}\theta \equiv \texttt{Rs2}\theta \equiv \texttt{[]}$, then
$\texttt{cut\_col([[E|Es]|Rs],[E|C2],[Es|Rs2])}\theta \equiv \texttt{cut\_col([[E|Es]],[E|C2],[Es])}\theta$,
and since $\mathit{size}(\texttt{[[E|Es]]}\theta) > \mathit{size}(\texttt{[Es]}\theta)$, it follows that
$\texttt{cut\_col([[E|Es]],[E|C2],[Es])}\theta \in M$.

— If $\mathit{size}(\texttt{Rs}\theta) > \mathit{size}(\texttt{Rs2}\theta)$, then
$\mathit{size}(\texttt{[[E|Es]|Rs]}\theta) > \mathit{size}(\texttt{[Es|Rs2]}\theta)$, thus
$\texttt{cut\_col([[E|Es]|Rs],[E|C2],[Es|Rs2])}\theta \in M$.

Thus in all cases, $\texttt{cut\_col([[E|Es]|Rs],[E|C2],[Es|Rs2])}\theta \in M$. Therefore M is a model of `c3`. We now show that the program is simply acceptable wrt. M and $|\ |$ and hence input terminating wrt. simply moded queries. Consider `c1`: for every substitution θ, simply local wrt. `c1`, we have to show that if $\texttt{cut\_col([R|Rs],C,M2)}\theta \in M$, then $|\texttt{transpose([R|Rs],[C|Cs])}\theta| > |\texttt{transpose(M2,Cs)}\theta|$. This holds by the definition of M. Next, consider `c3`. For every substitution θ, it is easy to see that $|\texttt{cut\_col([[E|Es]|Rs],[E|C],[Es|Rs2])}\theta| > |\texttt{cut\_col(Rs,C,Rs2)}\theta|$. Equivalently, for clause `c4`, it is immediate to check that for any θ, $|\texttt{no\_cols\_matrix([[]|Rs])}\theta| > |\texttt{no\_cols\_matrix(Rs)}\theta|$. 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 [van Emden and de Lucena 1982] were introduced explicitly in logic programming by Naish [Naish 1983].

In a previous paper [Bossi et al. 2001] 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[4]. 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 its behavior 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:

[4]An interesting example suggested by K. R. 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.

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

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 [Bossi et al. 2000] we have introduced a different semantics for input consuming programs. The two semantics, however, are quite orthogonal to each other: while that of [Bossi et al. 2000] models exclusively the result of successful derivations and requires the program to be *well-moded* and *nicely-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 [Naish 1988], 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* [Ueda and Morita 1994]. 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 $H \leftarrow \mathbf{G} \,|\, \mathbf{B}$, 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 θ 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.* [Falaschi et al. 1997] 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 for usual logic programs, where atoms are mapped to their set of answers. In this respect, it corresponds to the semantics defined in [Bossi et al. 2000]. 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 termination, applicable to a wide class of programs, namely the class of simply moded programs. In previous papers, [Bossi et al. 2002; Smaus 1999b] we have already addressed the problem of the termination of input consuming programs. The results we present here constitute a big improvement wrt. [Bossi et al. 2002; Smaus 1999b] 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 [Bossi et al. 2002; Smaus 1999b] (though, with the tools of [Bossi et al. 2002; Smaus 1999b] 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 [Bossi et al. 2001].

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 [Smaus 1999b], this allows us to see more clearly which programs terminate under which assumptions about the selection rule.

Apt and Luitjes [Apt and Luitjes 1995] give conditions for the termination of `append`, but those are ad-hoc and do not address the general problem. Naish [Naish 1993] 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 [Lüttringhaus-Kappel 1993; Marchiori and Teusink 1999; Martin and King 1997].

Marchiori and Teusink [Marchiori and Teusink 1999] 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 [Martin and King 1997] achieve a similar effect by bounding the depth of the computation introducing auxiliary predicates.

It is more difficult to assess Lüttringhaus-Kappel [Lüttringhaus-Kappel 1993] 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 [Marchiori and Teusink 1999; Martin and King 1997]. 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 [Apt and Luitjes 1995; Boye 1996; Naish 1993]. 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 [Smaus et al. 2001]. 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 [Pedreschi et al. 2002]. Among others, this survey considers input termination and termination wrt. local selection rules as mentioned above [Marchiori and Teusink 1999].

The specific problem of termination of input consuming derivations has been treated also in [Bossi et al. 2002] 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 [Apt 1997, Corollary 2.25]), there exist substitutions σ_0 and σ_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 wrt. H, this means that σ_0 is simply local wrt. the clause $H \leftarrow$. Moreover, by relevance of σ_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 σ_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 wrt. A, this means that σ_1 is simply local wrt. 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 α and β such that

(a) $\theta = \alpha\beta$;
(b) $\alpha = \theta_{|\mathbf{B}\vartheta}$ is simply local wrt. $\mathbf{B}\vartheta$;
(c) β is simply local wrt. $\mathbf{R}\vartheta\alpha$;
(d) α and β are variable compatible wrt. $\mathbf{B}\vartheta$ 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 wrt. A;
$(c1)$ β is simply local wrt. $\mathbf{R}(\vartheta\alpha)_{|A}$;
$(d1)$ $(\vartheta\alpha)_{|A}$ and β are variable compatible wrt. 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 ϑ is a simply local mgu of A and H, and $Dom(\alpha_{|A}) \subseteq Var(Out(\mathbf{B}\vartheta)) \cap Var(A)$, since α is simply local wrt. $\mathbf{B}\vartheta$. 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 $Ran(\vartheta_{|A}) \subseteq Var(Out(H))$, $Ran((\vartheta\alpha)_{|A}) \subseteq Var(Out(H)) \cup Var(In(A)) \cup Var(\mathbf{B}) \cup V_1 \subseteq Var(In(A)) \cup V_1 \cup Var(c)$ where V_1 is the set of fresh variables of α and $V_1 \cup Var(c)$ is disjoint from A by standardization apart and lemma's hypothesis.

$(c1)$ holds since β is simply local wrt. $\mathbf{R}\vartheta\alpha$ 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 θ (which implies that the sets of fresh variables of α and β are are disjoint from $Var(Q)$ and $Var(c)$) and the fact that ϑ 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. □

PROOF OF LEMMA 4.4. (i) $\Rightarrow$ (ii). We first assume that δ proceeds left-to-right and employs only simply local mgu's and prove that: $\vartheta_{|\mathbf{A}}$ is simply local wrt. $\mathbf{A}$ and $\mathbf{A}\vartheta \subseteq T_P^{SL} \uparrow \omega(I)$. The general case follows from Corollary 2.12 and Theorem 3.18 in [Apt 1997] on derivations employing different mgu's.

We proceed by induction on the length of δ.

Base. $len(\delta) = 0$. 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. $len(\delta) > 0$. 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 δ (and hence there are no $\mathbf{L}$-steps in δ). Assume also that $c : H \leftarrow \mathbf{B}$ is the input clause used in the first derivation step of δ and ϑ_1 is the simply local mgu of A and H used in this step. By Corollary 2.12,

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

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

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

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 wrt. $(\mathbf{B}, \mathbf{R})\vartheta_1$ and

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

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

$$(\vartheta_1\vartheta_{2|(\mathbf{B},\mathbf{R})\vartheta_1})_{|(A,\mathbf{R})} = (\vartheta_1\vartheta_2)_{|(A,\mathbf{R})} \text{ is simply local wrt. } (A, \mathbf{R}). \tag{5}$$

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

$$(\vartheta_1\vartheta_2)_{|(\mathbf{L},A,\mathbf{R})} \text{ is simply local wrt. } (\mathbf{L}, A, \mathbf{R}). \tag{6}$$

To conclude the proof it remains to shown that

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

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

In order to prove (7) note that ϑ_1 is a simply local mgu of A and H, so $(\vartheta_1)_{|H}$ is simply local wrt. $H \leftarrow$. Moreover, by Lemma 3.5, $(\vartheta_2)_{|\mathbf{B}\vartheta_1}$ is simply local wrt. $\mathbf{B}\vartheta_1$. Note also that, by standardization apart, $\vartheta_{1|H}$ and $\vartheta_{2|\mathbf{B}\vartheta_1}$ are variable compatible wrt. 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 wrt. 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 $\mathbf{A} : A_1, \ldots, A_n$. 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 θ simply local wrt. $\mathbf{A}$. Since both $\mathbf{A}$ and $\mathbf{A}\theta$ consist of simply moded atoms, and θ is a simply local substitution wrt. $\mathbf{A}$, it follows that θ is just a renaming of the output variables of $\mathbf{A}$. The thesis follows by taking ϑ to be the empty substitution and δ 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 $\mathbf{A} = A$, θ is simply local wrt. A and $A\theta \in T_P^{SL} \uparrow k(I)$. 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 ϑ such that

$$\vartheta \text{ is simply local wrt. } c \tag{8}$$

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

$$A\theta = H\vartheta. \tag{10}$$

By (8) and Lemma 3.6 there exist σ_0 and σ_1 such that $\vartheta = \sigma_0\sigma_1$, $\sigma_0 = \vartheta_{|H}$ is simply local wrt. $H \leftarrow$ and σ_1 is simply local wrt. $\mathbf{B}\sigma_0$.

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 σ_1 or ϑ_2 are those occurring also in $\mathbf{B}\sigma_0$.

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

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

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 θ_1 and θ_2 such that $\theta = \theta_1\theta_2$, $\theta_1 = \theta_{|A}$ is simply local wrt. A and θ_2 is simply local wrt. $\mathbf{R}\theta_1$, and θ_1 and θ_2 are variable compatible wrt. 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 δ_2 are standardized apart wrt. δ_1.

Then there exist δ,

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

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

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* $\delta : \mathbf{A} \overset{\vartheta_1}{\Longrightarrow} \mathbf{C}_1 \xrightarrow{\vartheta_2} \mathbf{C_2}$ *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* δ *and* ϑ_1 *be a simply local mgu employed in this step. Then* $(\vartheta_1\vartheta_2)_{|\mathbf{A}}$ *is simply local wrt.* $\mathbf{A}$ *and* $(\vartheta_1\vartheta_2)_{|c}$ *is simply local wrt.* c.

PROOF. It follows from (5) in the proof of Lemma 4.4 (the proof above). □

REFERENCES

APT, K. R. 1990. Logic Programming. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B: Formal Models and Semantics. Elsevier and The MIT Press, Amsterdam and Cambridge, MA, 495–574.

APT, K. R. 1997. *From Logic Programming to Prolog.* Prentice Hall, London.

APT, K. R. AND ETALLE, S. 1993. On the unification free Prolog programs. In *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS'93)*, A. Borzyszkowski and S. Sokolowski, Eds. Lecture Notes in Computer Science, vol. 711. Springer-Verlag, Berlin, Germany, 1–19.

APT, K. R. AND LUITJES, I. 1995. Verification of logic programs with delay declarations. In *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, (AMAST'95)*, A. Borzyszkowski and S. Sokolowski, Eds. Lecture Notes in Computer Science, vol. 936. Springer-Verlag, Berlin, Germany, 1–19.

APT, K. R. AND PEDRESCHI, D. 1994. Modular termination proofs for logic and pure Prolog programs. In *Advances in Logic Programming Theory*, G. Levi, Ed. Oxford University Press, Oxford, UK, 183–229.

BOSSI, A., ETALLE, S., AND ROSSI, S. 2000. Semantics of well-moded input-consuming logic programs. *Computer Languages 26,* 1, 1–25.

BOSSI, A., ETALLE, S., AND ROSSI, S. 2002. Properties of input-consuming derivations. *Theory and Practice of Logic Programming 2,* 2, 125–154.

BOSSI, A., ETALLE, S., ROSSI, S., AND SMAUS, J.-G. 2001. Semantics and termination of simply-moded logic programs with dynamic scheduling. In *Proceedings of the European Symposium on Programming*, D. Sands, Ed. Lecture Notes in Computer Science, vol. 2028. Springer-Verlag, Genova, Italy, 402–416.

BOYE, J. 1996. Directional types in logic programming. Ph.D. thesis, Linköpings Universitet.

DE BOER, F. AND PALAMIDESSI, C. 1991. A fully abstract model for concurrent constraint programming. In *Proc. of the International Joint Conference on Theory and Practice of Software Development, (TAPSOFT/CAAP)*, S. Abramsky and T. Maibaum, Eds. Lecture Notes in Computer Science, vol. 493. Springer-Verlag, Brighton, UK, 296–319.

ETALLE, S., BOSSI, A., AND COCCO, N. 1999. Termination of well-moded programs. *Journal of Logic Programming 38,* 2, 243–257.

ETALLE, S., GABBRIELLI, M., AND MEO, M. C. 2002. Transformations of ccp programs. *ACM Transactions on Programming Languages and Systems 23,* 3, 304–395.

FALASCHI, M., GABBRIELLI, M., MARRIOTT, K., AND PALAMIDESSI, C. 1997. Constraint logic programming with dynamic scheduling: a semantics based on closure operators. *Information and Computation 137,* 1, 41–67.

KOWALSKI, R. A. 1979. Algorithm = Logic + Control. *Communications of the ACM 22,* 7, 424–436.

LLOYD, J. W. 1987. *Foundations of Logic Programming.* Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, Berlin, Germany. Second edition.

LÜTTRINGHAUS-KAPPEL, S. 1993. Control generation for logic programs. In *Proc. Tenth International Conference on Logic Programming*, D. Warren, Ed. The MIT Press, Budapest, Hungary, 478–495.

MARCHIORI, E. AND TEUSINK, F. 1999. Termination of logic programs with delay declarations. *Journal of Logic Programming 39,* 1–3, 95–124.

MARTIN, J. C. AND KING, A. M. 1997. Generating efficient, terminating logic programs. In *Proc. of the 7th International Joint Conference on Theory and Practice of Software Development*, M. Bidoit and M. Dauchet, Eds. Lecture Notes in Computer Science, vol. 1214. Springer-Verlag, Lille, France, 273–184.

NAISH, L. 1986. *Negation and control in Prolog.* Lecture Notes in Computer Science, vol. 238. Springer-Verlag, New York.

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

NAISH, L. August 1988. Parallelizing NU-Prolog. In *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, K. A. Bowen and R. A. Kowalski, Eds. The MIT Press, Seattle, Washington, 1546–1564.

NAISH, L. March 1982 (Revised July 1983). An introduction to MU-Prolog. Tech. Rep. 82/2, Department of Computer Science, University of Melbourne, Melbourne, Australia.

PEDRESCHI, D., RUGGIERI, S., AND SMAUS, J.-G. 2002. Classes of terminating logic programs. *Theory and Practice of Logic Programming 3,* 2, 369–418.

RUGGIERI, S. 1997. Termination of Constraint Logic Programs. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP'97).* Lecture Notes in Computer Science, vol. 1256. Springer-Verlag, Bologna, Italy, 838–848.

SARASWAT, V. A. AND RINARD, M. 1990. Concurrent constraint programming. In *Proc. of the Seventeenth ACM Symposium on Principles of Programming Languages.* ACM, New York, San Francisco, California, 232–245.

SMAUS, J.-G. 1999a. Modes and types in logic programming. Ph.D. thesis, University of Kent at Canterbury. Available from `http://www.cs.ukc.ac.uk/pubs/1999/986/`.

SMAUS, J.-G. 1999b. Proving termination of input-consuming logic programs. In *Proceedings of the 16th International Conference on Logic Programming*, D. D. Schreye, Ed. The MIT Press, Las Cruces, New Mexico, USA, 335–349.

SMAUS, J.-G., HILL, P. M., AND KING, A. M. 1998. Termination of logic programs with `block` declarations running in several modes. In *Proceedings of the 10th Symposium on Programming Language Implementations and Logic Programming*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 1490. Springer-Verlag, Pisa, Italy, 73–88.

SMAUS, J.-G., HILL, P. M., AND KING, A. M. 2001. Verifying termination and error-freedom of logic programs with `block` declarations. *Theory and Practice of Logic Programming 1,* 4, 447–486.

STERLING, L. AND SHAPIRO, E. 1986. *The Art of Prolog.* The MIT Press, Cambridge, MA.

UEDA, K. AND FURUKAWA, K. 1988. Transformation rules for GHC Programs. In *Proc. of the International Conference on Fifth Generation Computer Systems.* Institute for New Generation Computer Technology, Tokyo, OHMSHA Ltd. Tokyo and Springer-Verlag, Tokyo, Japan, 582–591.

UEDA, K. AND MORITA, M. 1994. Moded flat GHC and its message-oriented implementation technique. *New Generation Computing 13,* 1, 3–43.

VAN EMDEN, M. H. AND DE LUCENA, G. J. 1982. Predicate logic as a language for parallel programming. In *Logic Programming*, K. Clark and S.-A. Tärnlund, Eds. Academic Press, London.

Received December 2001; revised October 2002 and February 2003; accepted February 2003