Semantics and Termination of Simply-Moded Logic Programs with Dynamic Scheduling

Annalisa Bossi[1], Sandro Etalle[2,3], Sabina Rossi[1], and Jan-Georg Smaus[3]

[1] Università di Venezia, {bossi,srossi}@dsi.unive.it
[2] Universiteit Maastricht, etalle@cs.unimaas.nl
[3] CWI, Amsterdam, jan.smaus@cwi.nl

Abstract. In logic programming, *dynamic scheduling* refers to a situation where the selection of the atom in each resolution (computation) step is determined at runtime, as opposed to a fixed selection rule such as the left-to-right one of Prolog. This has applications e.g. in parallel programming. A mechanism to control dynamic scheduling is provided in existing languages in the form of *delay declarations*.
Input-consuming derivations were introduced to describe dynamic scheduling while abstracting from the technical details. In this paper, we first formalise the relationship between delay declarations and input-consuming derivations, showing in many cases a one-to-one correspondence. Then, we define a model-theoretic semantics for input-consuming derivations of simply-moded programs. Finally, for this class of programs, we provide a necessary and sufficient criterion for termination.

1 Introduction

Background. Logic programming is based on giving a computational interpretation to a fragment of first order logic. Kowalski [14] advocates the separation of the *logic* and *control* aspects of a logic program and has coined the famous formula

Algorithm = Logic + Control.

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

In reality, logic programming is far from this ideal. Without the programmer being aware of the control and writing programs accordingly, logic programs would usually 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 provides appropriate control for many applications, there are situations, e.g. in the context of parallel execution or the test-and-generate paradigm, that require a more flexible control mechanism, namely, *dynamic scheduling*, where the selectable atoms are determined at runtime. Such a mechanism is provided in modern logic programming languages in the form of *delay declarations* [16].

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 (read_tree and write_list are defined elsewhere)

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

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 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). In order to avoid nontermination one should adopt appropriate delay declarations, namely

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

These declarations avoid that in_order, append and write_list are 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 in [17] and "has an important advantage over the ones proposed in the literature in that it allows us to parallelise programs written in a large subset of Prolog by merely adding to them delay declarations, so *without modifying* the original program" [4].

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 *deadlock*.

In the first place, for such programs the well-known equivalence between model-theoretic and operational semantics does not hold. For example, the query append(X,Y,Z) 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. This shows that a model-theoretic semantics in the classical sense is not achievable, in fact the problem of finding a suitable

declarative semantics is still open. Moreover, while for the left-to-right selection rule there are results that allow us to characterise when a program is terminating, these results do not apply any longer in presence of dynamic scheduling.

Contributions. This paper contains essentially four contributions tackling the above problems.

In order to provide a characterisation of dynamic scheduling that is reasonably abstract and hence amenable to semantic analysis, we consider *input-consuming derivations* [18], a formalism similar to *Moded GHC* [20]. In an input-consuming derivation, only atoms whose input arguments are not instantiated through the unification step may be selected. Moreover, we restrict our attention to the class of *simply-moded* programs, which are programs that are, in a well-defined sense, consistent wrt. the modes. As also shown by the benchmarks in Sec. 6, most practical programs are simply-moded. We analyse the relations between input-consuming derivations and programs with delay declarations. We demonstrate that under some statically verifiable conditions, input-consuming derivations are exactly the ones satisfying the (natural) delay declarations of programs.

We define a denotational semantics which enjoys a model-theoretical reading and has a bottom-up constructive definition. We show that it is compositional, correct and fully abstract wrt. the computed answer substitutions of successful derivations. E.g., it captures the fact that the query `append(X,Y,Z)` does not succeed.

Since dynamic scheduling also allows for parallelism, it is sometimes important to model the result of *partial* (i.e., incomplete) derivations. For instance, one might have queries (processes) that never terminate, which by definition may never reach the state of *success*, i.e. of successful completion of the computation. Therefore, we define a second semantics which enjoys the same properties as the one above. We demonstrate that it is correct, fully abstract and compositional wrt. the computed substitutions of partial derivations. We then have a uniform (in our opinion elegant) framework allowing us to model both successful and partial computations.

Finally, we study the problem of termination of input-consuming programs. We present a result which fully characterises termination of simply-moded input-consuming programs. This result is based on the semantics mentioned in the previous paragraph.

The rest of this paper is organised as follows. The next section introduces some preliminaries. Section 3 defines input-consuming derivations and delay declarations, and formally compares the two notions. Section 4 provides a result on denotational semantics for input-consuming derivations, first for complete derivations, then for incomplete (input-consuming) derivations. Section 5 provides a sufficient and necessary criterion for termination of programs using input-consuming derivations. Section 6 surveys some benchmark programs. Section 7 concludes. The proofs have been omitted and can be found in [8].

2 Preliminaries

The reader is assumed to be familiar with the terminology and the basic results of the semantics of logic programs [1, 2, 15]. Following [2], we use boldface characters to denote sequences of objects: $\mathbf{t}$ denotes a sequence of terms, $\mathbf{B}$ is a query (i.e., a possibly empty sequence of atoms). The empty query is denoted by $\Box$. The relation symbol of an atom A is denoted $Rel(A)$. The set of variables occurring in a syntactic object o is denoted $Var(o)$. We say that o is *linear* if every variable occurs in it at most once. Given a *substitution* $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$, we say that $\{x_1, \ldots, x_n\}$ is its *domain* (denoted by $Dom(\sigma)$), and $Var(\{t_1, \ldots, t_n\})$ is its *range* (denoted by $Ran(\sigma)$). Note that $Var(\sigma) = Dom(\sigma) \cup Ran(\sigma)$. If $t_1, \ldots, t_n$ is a permutation of $x_1, \ldots, x_n$ then we say that σ is a *renaming*. The *composition* of substitutions is denoted by juxtaposition ($x\theta\sigma = (x\theta)\sigma$). We say that a term t is an *instance* of t' iff for some σ, $t = t'\sigma$; further, t is a *variant* of t', written $t \approx t'$, iff t and t' are instances of each other. A substitution θ is a *unifier* of terms t and t' iff $t\theta = t'\theta$. We denote by $mgu(t, t')$ any *most general unifier* (*mgu*, in short) of t and t'. A query $Q : \mathbf{A}, B, \mathbf{C}$ and a clause $c : H \leftarrow \mathbf{B}$ (variable disjoint with Q) yield the resolvent $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ with $\theta = mgu(B, H)$. We say that $\mathbf{A}, B, \mathbf{C} \stackrel{\theta}{\Longrightarrow} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ is a *derivation step (using c)*, and call B the *selected atom*. A *derivation of* $P \cup \{Q\}$ is a sequence of derivation steps $Q \stackrel{\theta_1}{\Longrightarrow} Q_1 \stackrel{\theta_2}{\Longrightarrow} \cdots$ using (variants of) clauses in the program P. A finite derivation $Q \stackrel{\theta_1}{\Longrightarrow} \cdots \stackrel{\theta_n}{\Longrightarrow} Q_n$ is also denoted $Q \stackrel{\vartheta}{\longrightarrow}_P Q_n$, where $\vartheta = \theta_1 \ldots \theta_n$. The restriction of ϑ to Q is a *computed answer substitution* (*c.a.s.*). If $Q_n = \Box$, the derivation is *successful*.

Delay Declarations. Logic programs with delay declarations consist of two parts: a set of clauses and a set of delay declarations, one for each of its predicate symbols. A *delay declaration* associated with an n-ary predicate symbol p has the form

$$\texttt{delay}\ \ p(t_1, \ldots, t_n)\ \ \texttt{until}\ \ Cond(t_1, \ldots, t_n)$$

where $Cond(t_1, \ldots, t_n)$ is a formula in some assertion language [12]. A derivation is *delay-respecting* if an atom $\mathtt{p}(t_1, \ldots, t_n)$ is selected only if $Cond(t_1, \ldots, t_n)$ is satisfied. In particular, we consider delay declarations of the form

$$\texttt{delay}\ \ \mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_n)\ \ \texttt{until}\ \ \mathtt{nonvar}(\mathtt{X}_{i_1}) \wedge \ldots \wedge \mathtt{nonvar}(\mathtt{X}_{i_k}).$$

where $1 \leq i_1 < \ldots < i_k \leq n$.[1] The condition $\mathtt{nonvar}(t_{i_1}) \wedge \ldots \wedge \mathtt{nonvar}(t_{i_k})$ is satisfied if and only if $t_{i_1}, \ldots, t_{i_k}$ are non-variable terms. Such delay declarations are equivalent to the `block` declarations of SICStus Prolog [13].

Moded Programs. A *mode* indicates how a predicate should be used.

[1] For the case that $k = 0$, the empty conjunction might be denoted as `true`, or the delay declaration might simply be omitted.

Definition 2.1. A *mode* for a predicate symbol p of arity n, is a function m_p from $\{1,\ldots,n\}$ to $\{In, Out\}$. □

If $m_p(i) = In$ (resp. Out), we say that i is an *input* (resp. *output*) *position of* p. 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.

The notion of simply-moded program is due to Apt and Etalle [3].

Definition 2.2. A clause $p(\mathbf{t}_0,\mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1,\mathbf{t}_1),\ldots,p_n(\mathbf{s}_n,\mathbf{t}_n)$ is *simply-moded* iff $\mathbf{t}_1,\ldots,\mathbf{t}_n$ is a linear vector of variables and for all $i \in [1,n]$

$$Var(\mathbf{t}_i) \cap Var(\mathbf{t}_0) = \emptyset \quad \text{and} \quad Var(\mathbf{t}_i) \cap \bigcup_{j=1}^{i} Var(\mathbf{s}_j) = \emptyset.$$

A query $\mathbf{B}$ is *simply-moded* iff the clause $q \leftarrow \mathbf{B}$ is simply-moded, where q is any variable-free atom. A program is simply-moded iff 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. Notice also that programs `APPEND` and `IN_ORDER` are simply-moded wrt. the modes `append(In,In,Out)` and `in_order(In,Out)`.

3 Input-Consuming Programs

Input-consuming derivations are a formalism for describing dynamic scheduling in an abstract way [18].

Definition 3.1. A derivation step $\mathbf{A},B,\mathbf{C} \stackrel{\theta}{\Longrightarrow} (\mathbf{A},\mathbf{B},\mathbf{C})\theta$ is *input-consuming* iff $In(B)\theta = In(B)$. A derivation is *input-consuming* iff all its derivation steps are input-consuming. □

Thus, allowing only input-consuming derivations is a form of dynamic scheduling, since selectability depends on the degree of instantiation at runtime. If no atom is resolvable via an input-consuming derivation step, the query *deadlocks*.[2]

It has been shown that the input-consuming resolvent of a simply-moded query using a simply-moded clause is simply-moded [4].

Example 3.2. Consider again the delay declaration

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

[2] Notice that there is a difference between this notion of deadlock and the one used for programs with delay declarations; see [6] for a detailed discussion.

It is easy to check that *every* derivation starting in a query `append(`t`,`s`,X)`, where `X` is a variable disjoint from s and t, is input-consuming wrt. `append(In,In,Out)` *iff* it respects the delay declaration. □

To show the correspondence between delay declarations and input-consuming derivations suggested by Ex. 3.2, we need some further definitions. We call a term t *flat* if t has the form $f(x_1, \ldots, x_n)$ where the x_i are distinct variables. Note that constants are flat terms. The significance of flat term arises from the following observation: if s and t are unifiable, s is non-variable and t is flat, then s is an instance of t. Think here of s being a term in an input position of a selected atom, and t being the term in that position of a clause head.

Definition 3.3. A program P is *input-consistent* iff for each clause $H \leftarrow \mathbf{B}$ of it, the family of terms filling in the input positions of H is linear, and consists of variables and flat terms. □

We also consider here delay declarations of a restricted type.

Definition 3.4. A program with delay declarations is *simple* if every delay declaration is of the form

$$\texttt{delay}\ \ p(\texttt{X}_1, \ldots, \texttt{X}_n)\ \ \texttt{until}\ \ \texttt{nonvar}(\texttt{X}_{i_1}) \wedge \ldots \wedge \texttt{nonvar}(\texttt{X}_{i_k}).$$

where $i_1, \ldots, i_k$ are input positions of p.

Moreover, we say that the positions $i_1, \ldots, i_k$ of p are *controlled*, while the other input positions of p are *free*. □

Thus the controlled positions are those "guarded" by a delay declaration. The main result of this section shows that, under some circumstances, using delay declarations is equivalent to restricting to input-consuming derivations.

Lemma 3.5. Let P be simply-moded, input-consistent and simple. Let Q be a simply-moded query.

- If for every clause $H \leftarrow \mathbf{B}$ of P, H contains variables in its free positions, then every derivation of $P \cup \{Q\}$ respecting the delay declarations is input-consuming (modulo renaming).
- If in addition for every clause $H \leftarrow \mathbf{B}$ of P, the head H contains flat terms in its controlled positions, then every input-consuming derivation of $P \cup \{Q\}$ respects the delay declarations. □

In order to assess how realistic these conditions are, we have checked them against a number of programs from various collections. (The results can be found in Sec. 6). Concerning the statement that all delay-respecting derivations are input-consuming, we are convinced that this is the case in the overwhelming majority of practical cases. Concerning the converse, that is, that all input-consuming derivations are delay-respecting, we could find different examples in which this was not the case. In many of them this could be fixed by a simple

transformation of the programs[3], in other cases it could not (e.g., `flatten`, [19]). Nevertheless, we strongly believe that the latter form a small minority.

The delay declarations for the considered programs were either given or derived based on the presumed mode. Note that delay declarations as in Def. 3.4 can be more efficiently implemented than, e.g., delay declarations testing for groundness. Usually, the derivations permitted by the latter delay declarations are a strict subset of the input-consuming derivations.

4 A Denotational Semantics

Previous declarative semantics for logic programs cannot correctly model dynamic scheduling. E.g., none of them reflects the fact that `append(X,Y,Z)` deadlocks. We define a model-theoretic semantics that models computed answer substitutions of input-consuming derivations of simply-moded programs and queries.

We now define *simply-local* substitutions, which reflect the way clauses become instantiated in input-consuming derivations. A simply-local substitution can be decomposed into several substitutions, corresponding to the instantiation of the *output* of each *body atom*, as well as the *input* of the *head*.

Definition 4.1. Let θ be a substitution. We say that θ is *simply-local* wrt. the clause $c : p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ iff 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 for $i \in \{0, \ldots, n\}$,

- $Dom(\sigma_i) \subseteq Var(\mathbf{t}_i)$,
- $Ran(\sigma_i) \subseteq Var(\mathbf{s}_i\sigma_0\sigma_1 \cdots \sigma_{i-1}) \cup v_i$.[4]

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

Note that if $\mathbf{A}, B, \mathbf{C} \stackrel{\theta}{\Longrightarrow} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ is an input-consuming derivation step using clause $c : H \leftarrow \mathbf{B}$, , then $\theta_{|H}$ is simply-local wrt. the clause $H \leftarrow$ and $\theta_{|B}$ is simply-local wrt. the query B.

Example 4.2. Consider `APPEND` in mode `append(In,In,Out)`, and its recursive clause c : `append([H|Xs],Ys,[H|Zs])` $\leftarrow$ `append(Xs,Ys,Zs)`. The substitution $\theta =$ {`H/V,Xs/[],Ys/[W],Zs/[W]`} is simply-local wrt. c: let $\sigma_0 =$ {`H/V,Xs/[],Ys/[W]`} and $\sigma_1 =$ {`Zs/[W]`}; then $Dom(\sigma_0) \subseteq$ {`H,Xs,Ys`}, and $Ran(\sigma_0) \subseteq v_0$ where $v_0 =$ {`V,W`}, and $Dom(\sigma_1) \subseteq$ {`Zs`}, and $Ran(\sigma_1) \subseteq Var((\texttt{Xs}, \texttt{Ys})\sigma_0)$.

[3] To give an intuitive idea, the transformation would, e.g., replace the clause `even(s(s(X))):- even(X).` with `even(s(Y)):- s_decomp(Y,X), even(X).`, where we define `s_decomp(s(X),X).` and the mode is `s_decomp(In,Out).`

[4] Note that $\mathtt{s}_0$ is undefined. By abuse of notation, $Var(\mathtt{s}_0 \ldots) = \emptyset$.

4.1 Modelling Complete Derivations

In predicate logic, an interpretation states which formulas are true and which ones are not. For our purposes, it is convenient to formalise 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 4.3. Let M be an interpretation. We say that M is a *simply-local model* of $c : H \leftarrow B_1, \ldots, B_n$ iff 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 iff it is a simply-local model of each clause of it. □

Note that a simply-local model is not necessarily a model in the classical sense, since the substitution in (1) is required to be simply-local. For example, given the program $\{\mathtt{q(1)},\ \mathtt{p(X)} \leftarrow \mathtt{q(X)}\}$ with modes $\mathtt{q(In)}, \mathtt{p(Out)}$, a model must contain the atom $\mathtt{p(1)}$, whereas a simply-local model does not necessarily contain $\mathtt{p(1)}$, since $\{\mathtt{X/1}\}$ is not simply-local wrt. $\mathtt{p(X)} \leftarrow \mathtt{q(X)}$.

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: Given a program P and an interpretation I, define

$$T_P^{SL}(I) = \{H\theta \mid \exists\ c : H \leftarrow B_1, \ldots, B_n \in P \\ \exists\ \theta \text{ simply-local wrt.} c \\ B_1, \ldots, B_n\theta \in I \qquad \}.$$

Operator's powers are defined in the standard way: $T_P^{SL} \uparrow 0(I) = I$, $T_P^{SL} \uparrow (i+1)(I) = T_P^{SL}(T_P^{SL} \uparrow i(I))$, and $T_P^{SL} \uparrow \omega(I) = \bigcup_{i=0}^{\infty} T_P^{SL} \uparrow i(I)$. It is easy to show that T_P^{SL} is continuous on the lattice where interpretations are ordered by set inclusion. Hence, by well-known results, $T_P^{SL} \uparrow \omega$ exists and is the least fixpoint of T_P^{SL}. We can now state our main result.

Theorem 4.4. Let P be simply-moded. Then $T_P^{SL} \uparrow \omega(\emptyset)$ is the least simply-local model of P. □

We now prove correctness, fully abstractness and compositionality of the semantics. We denote the least simply-local model of P by M_P^{SL}.

Theorem 4.5. Let the program P and the query $\mathbf{A}$ be simply-moded. The following statements are equivalent:

(i) there exists an input-consuming successful derivation $\mathbf{A} \xrightarrow{\vartheta}_P \square$,
(ii) there exists a substitution θ, simply-local wrt. $\mathbf{A}$, such that $\mathbf{A}\theta \in M_P^{SL}$,

where $\mathbf{A}\theta$ is a variant of $\mathbf{A}\vartheta$. □

Example 4.6. Considering again `APPEND`, we have that

$$M^{SL}_{\texttt{APPEND}} = \bigcup_{n=0}^{\infty} \{\texttt{append}([t_1,\ldots,t_n], s, [t_1,\ldots,t_n|s]) \mid t_1,\ldots,t_n,s \text{ are any terms } \}.$$

Using Thm. 4.5, we can conclude that the query `append([a,b],X,Y)` succeeds with computed answer $\theta = \{\texttt{Y}/[\texttt{a},\texttt{b}|\texttt{X}]\}$. In fact, `append([a,b],X,[a,b|X])`$\in M^{SL}_{\texttt{APPEND}}$, and θ is simply-local wrt. the query above.

On the other hand, we can also say that the query `append(X,[a,b],Y)` has *no successful input-consuming derivations*. In fact, for every $A \in M^{SL}_{\texttt{APPEND}}$ we have that the first input position of A is filled in by a non-variable term. Therefore there is no simply-local θ such that `append(X,[a,b],Y)`$\theta \in M^{SL}_{\texttt{APPEND}}$. This shows that this semantics allows us to model correctly deadlocking derivations.

However, `append(X,[a,b],Y)` has instances in $M^{SL}_{\texttt{APPEND}}$, and successful derivations, if the requirement of simply-local substitutions, resp. input-consuming derivations, is ignored.

4.2 Modelling Partial Derivations

Dynamic scheduling also allows for parallelism. In this context it is important to be able to model the result of partial derivations. That is to say, instead of considering computed answer substitutions for complete derivations, we now consider computed answer substitutions for partial derivations. As we will see, this will be essential in order to prove termination of the programs.

Let SM_P be the set of all simply-moded atoms of the extended Herbrand universe of P. In analogy to Theorem 4.4, we have the following theorem.

Theorem 4.7. Let P be simply-moded. Then $T_P^{SL} \uparrow \omega(SM_P)$ is the least simply-local model of P containing SM_P. □

We denote the least simply-local model of P containing SM_P by PM_P^{SL}, for *partial model*. We now show correctness, fully abstractness and compositionality of this semantics for partial derivations.

Theorem 4.8. Let the program P and the query $\mathbf{A}$ be simply-moded. The following statements are equivalent:

(i) there exists an input-consuming derivation $\mathbf{A} \xrightarrow{\vartheta}_P \mathbf{A}'$,
(ii) there exists a substitution θ, simply-local wrt. $\mathbf{A}$, such that $\mathbf{A}\theta \in PM_P^{SL}$,

where $\mathbf{A}\theta$ is a variant of $\mathbf{A}\vartheta$. □

Note that the derivation in point (i) ends in $\mathbf{A}'$, which might be non-empty.

Example 4.9. Consider again `APPEND`. First, $PM^{SL}_{\texttt{APPEND}}$ contains $M^{SL}_{\texttt{APPEND}}$ as a subset (see Ex. 4.6). Note that $M^{SL}_{\texttt{APPEND}}$ is obtained by starting from the fact clause `append([],Ys,Ys)` and repeatedly applying the T_P^{SL} operator using the recursive clause of `APPEND`. Now to obtain the remaining atoms in $PM^{SL}_{\texttt{APPEND}}$, we must

repeatedly apply 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 does not occur in s or t. It is easy to see that we thus have to add SM_P together with

$$\{\texttt{append}([t_1, \ldots, t_n|s], t, [t_1, \ldots, t_n|x]) \mid \begin{array}{l} t_1, \ldots, t_n, s, t \text{ are arbitrary terms,} \\ x \text{ is a fresh variable}\}. \end{array}$$

Using Thm. 4.8, we can conclude that the query `append([a,b|X],Y,Z)` has a partial derivation with computed answer $\theta = \{\texttt{Z}/[\texttt{a}, \texttt{b}|\texttt{Z}']\}$, and indeed, $\texttt{append}([\texttt{a}, \texttt{b}|\texttt{X}], \texttt{Y}, [\texttt{a}, \texttt{b}|\texttt{Z}']) \in PM^{SL}_{\texttt{APPEND}}$, and θ is simply-local wrt. the query above. Notice that, following the same reasoning, one can also conclude that the query also has a partial derivation with computed answer $\theta = \{\texttt{Z}/[\texttt{a}|\texttt{Z}']\}$.

5 Termination

Input-consuming derivations were originally conceived as an abstract and "reasonably strong" assumption about the selection rule in order to prove termination [18]. 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 [6]. In this section, we only consider *simply* moded programs and queries (simply-moded and well-moded programs form two largely overlapping, but distinct classes), and we provide a criterion for termination which is sufficient and *necessary*, and hence an exact characterisation of termination. We first define our notion of termination.

Definition 5.1. A program is *input terminating* iff all its input-consuming derivations started in a simply-moded query are finite. □

In order to prove that a program is input terminating we need the concept of moded level mapping [10].

Definition 5.2. A function $|\ |$ is a *moded level mapping* iff it maps atoms into $\mathbb{N}$ and such that for any $\mathbf{s}$, $\mathbf{t}$ and $\mathbf{u}$, $|p(\mathbf{s}, \mathbf{t})| = |p(\mathbf{s}, \mathbf{u})|$. □

The condition $|p(\mathbf{s}, \mathbf{t})| = |p(\mathbf{s}, \mathbf{u})|$ states that the *level* of an atom is independent from the terms in its output positions.

Note that programs without recursion terminate trivially. In this context, we need the following standard definitions [2].

Definition 5.3. Let P be a program, p and q be relations. We say that

- p *refers to* q iff there is a clause in P with p in the head and q in the body.
- p *depends on* q iff (p, q) is in the reflexive and transitive closure of the relation *refers to*.
- p and q are *mutually recursive*, written $p \simeq q$, iff p and q depend on each other. □

We now define *simply-acceptability*, which is in analogy to acceptability [5], but defined to deal with simply-moded and input-consuming programs.

Definition 5.4. Let P be a program and M a simply-local model of P containing SM_P. A clause $H \leftarrow \mathbf{A}, B, \mathbf{C}$ is *simply-acceptable wrt. the moded level mapping* $|\,|$ *and* M iff for every substitution θ simply-local wrt. it,

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

The program P is *simply-acceptable wrt.* M iff 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. We can now show that this concept allows to characterize the class of input terminating programs.

Theorem 5.5. A simply-moded program P is simply-acceptable iff it is input terminating. In particular, if P is input terminating, then it is simply-acceptable wrt. PM_P^{SL}. □

Let us compare simply-acceptability to acceptability, used to prove left-termination [5]. Acceptability is based on a (classical) model M of the program, and for a clause $H \leftarrow A_1, \ldots, A_n$, one requires $|H\theta| > |A_i\theta|$ only if $M \models (A_1, \ldots, A_{i-1})\theta$. The reason is that for LD-derivations, $A_1, \ldots, A_{i-1}$ must be completely resolved before A_i is selected. By the correctness of LD resolution [2], it turns out that the c.a.s. θ, just before A_i is selected, is such that $M \models (A_1, \ldots, A_{i-1})\theta$. It has been argued previously that it is difficult to use a similar argument for input-consuming derivations [18]. Using the results of the previous section, we have overcome this problem. We exploited that provided that programs and queries are simply-moded, we know that even though $A_1, \ldots, A_{i-1}$ may not be resolved completely, $A_1, \ldots, A_{i-1}\theta$ will be in any "partial model" of the program.

Example 5.6. Figure 1 shows program 15.3 from [19]: `quicksort` using a form of difference lists (we permuted two body atoms for the sake of clarity). This program is simply-moded, and when used in combination with dynamic scheduling, the standard delay declarations for it are the following:

```
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)
```

The last two declarations fall out of the scope of Lemma 3.5. Nevertheless, if we think of the built-ins `>` and `=<` as being conceptually 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

```
%  quicksort(Xs, Ys)  ← Ys is an ordered permutation of  Xs.

   quicksort(Xs,Ys)  ← quicksort_dl(Xs,Ys,[]).

   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).

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

   mode quicksort(In,Out).
   mode quicksort_dl(In,Out,In).
   mode partition(In,In,Out,Out).
   mode =<(In,In).
   mode >(In,In).
```

Fig. 1. The `quicksort` program

are exactly the input-consuming ones. We can prove that the program is input terminating. Define *len* as

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

We use the following moded level mapping (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 model, the simplest solution is to use the model that expresses the dependency between the list lengths of the arguments of `partition`, i.e., M should contain all atoms of the form $\texttt{partition}(l_1, x, l_2, l_3)$ where $len(l_1) > len(l_2)$ and $len(l_1) > len(l_3)$.

6 Benchmarks

In order to assess how realistic the conditions of Lemma 3.5 are, we have looked into three collections of logic programs, and we have checked whether those programs were simply moded (**SM**), input-consistent (**IC**) and whether they satisfied both sides of Lemma 3.5 (**L**). Notice that programs which are not input-consistent do not satisfy the conditions of Lemma 3.5. For this reason, some **L** columns are left blank. The results, reported in Tables 1 to 3, show that our results apply to the majority of the programs considered. We considered in Table 1 the programs from Apt's collection [2, 5], in Table 2 those of the DPPD's collection, (http://dsse.ecs.soton.ac.uk/~mal/systems/dppd.html), and in Table 3 some programs of Lindenstrauss's collection (http://www.cs.huji.ac.il/~naomil).

	SM	IC	L		SM	IC	L
append(In,In,Out)	yes	yes	yes	mergesort(In,Out)	yes	no	
append(Out,Out,In)	yes	yes	no	mergesort(Out,In)	no		
append3(In,In,In,Out)	yes	yes	yes	mergesort_variant(In,Out,In)	yes	yes	no
color_map(In,Out)	yes	no		ordered(In)	yes	no	
color_map(Out,In)	yes	yes	yes	overlap(In,In)	yes	no	
dcsolve(In,_)	yes	yes	yes	overlap(In,Out)	yes	yes	yes
even(In)	yes	no		overlap(Out,In)	yes	yes	yes
fold(In,In,Out)	yes	yes	yes	perm_select(In,Out)	yes	yes	no
list(In)	yes	yes	yes	perm_select(Out,In)	yes	yes	no
lte(In,In)	yes	yes	no	qsort(In,Out)	yes	yes	yes
lte(In,Out)	yes	yes	yes	qsort(Out,In)	no		
lte(Out,In)	yes	yes	no	reverse(In,Out)	yes	yes	yes
map(In,In)	yes	yes	yes	reverse(Out,In)	yes	yes	yes
map(In,Out)	yes	yes	yes	select(In,In,Out)	yes	no	
map(Out,In)	yes	yes	yes	select(Out,In,Out)	yes	yes	yes
member(In,In)	yes	no		subset(In,In)	yes	no	
member(In,Out)	yes	yes	yes	subset (Out,In)	yes	yes	yes
member(Out,In)	yes	yes	yes	sum(In,In,Out)	yes	yes	yes
type(In,In,Out)	no			sum(Out,Out,In)	yes	yes	yes

Table 1. Programs from Apt's Collection

7 Conclusion

In this paper, we have proven a result that *demonstrates* – for a large class of programs – the equivalence between delay declarations and input-consuming derivations. This was only speculated in [6, 7]. In fact, even though the class of programs we are considering here (simply-moded programs) is only slightly smaller than the one of nicely-moded programs considered in [6, 7], for the latter a result such as Lemma 3.5 does not hold.

We have provided a denotational semantics for input-consuming derivations using a variant of the well-known T_P-operator. Our semantics follows the s-semantics approach [9] and thus enjoys the typical properties of semantics in this class. This semantics improves on the one introduced in [7] in two respects: The semantics of this paper models (within a uniform framework) both complete and incomplete derivations, and there is no requirement that the program must be well-moded.

Falaschi *et al.* [11] 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 modelling a possible effect of resolving the query on a program state (i.e., constraint on the program variables). However, we believe that our approach is more suited to termination proofs.

As mentioned in Sec. 4.2, in the context of parallelism and concurrency [17], one can have derivations that never *succeed*, and yet compute substitutions.

	SM	IC	L		SM	IC	L
applast(In,In,Out)	yes	yes	yes	relative (In,Out)	yes	yes	yes
depth(In,Out)	yes	no		relative (Out,In)	yes	yes	yes
flipflip(In,Out)	yes	yes	yes	rev_acc(In,In,Out)	yes	yes	yes
flipflip(Out,In)	yes	yes	yes	rotate(In,Out)	yes	yes	yes
generate(In,In,Out)	yes	no		rotate(Out,In)	yes	yes	yes
liftsolve(In,In)	yes	yes	yes	solve(In,In,Out))	yes	no	
liftsolve(In,Out)	yes	yes	yes	square_square(In,Out)	yes	yes	yes
match(In,In)	yes	no		squretr(In,Out)	yes	yes	yes
match_app(In,In)	yes	yes	no	ssupply(In,In,Out)	yes	yes	yes
match_app(In,Out)	yes	yes	no	trace(In,In,Out)	yes	no	
max_lenth(In,Out,Out)	yes	yes	yes	trace(In,Out,Out)	no		
memo_solve(In,Out)	yes	no		transpose(In,Out)	yes	no	
prune(In,Out)	yes	no		transpose(Out,In)	yes	yes	yes
prune(Out,In)	yes	no		unify(In,In,Out)	yes	no	

Table 2. Programs from DPPD's Collection

	SM	IC	L		SM	IC	L
ack(In,In,_)	yes	yes	no	huffman(In,Out)	no		
concatenate(In,In,Out)	yes	yes	yes	huffman(In,Out)	no		
credit(In,Out)	yes	yes	yes	normal_form(_,In)	yes	no	
deep(In,Out)	yes	yes	yes	queens(In,Out)	yes	yes	yes
deep(Out,In)	no			queens(Out,In)	yes	yes	no
descendant(In,Out)	yes	yes	yes	rewrite(In,Out)	yes	no	
descendant(Out,In)	yes	yes	yes	transform(In,In,In,Out)	yes	yes	yes
holds(In,Out)	yes	yes	yes	twoleast(In,Out)	no		

Table 3. Programs from Lindenstrauss's Collection

Moreover, input-consuming derivations essentially correspond to the execution mechanism of (Moded) FGHC [20]. Thus we have provided a model-theoretic semantics for such programs/programming languages, which go beyond the usual success-based SLD resolution mechanism of logic programming.

On a more practical level, our semantics for partial derivations is used in order to prove 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. For instance, we can now prove the termination of QUICKSORT, which is not possible with the tools of [18, 6] (which provided only a sufficient condition). In the termination proofs, we exploit that any selected atom in an input-consuming derivation is in a model for partial derivations, in a similar way as this is done for proving left-termination. It is only on the basis of the semantics that we could present a characterisation of input-consuming termination for simply-moded programs.

References

1. K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.
2. K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.
3. K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of MFCS '93*, LNCS, pages 1–19. Springer-Verlag, 1993.
4. K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In V. S. Alagar and M. Nivat, editors, *Proceedings of AMAST'95*, LNCS, pages 66–90. Springer-Verlag, 1995. Invited Lecture.
5. K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.
6. A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. *ENTCS*, 30(1), 1999. http://www.elsevier.nl/locate/entcs.
7. A. Bossi, S. Etalle, and S. Rossi. Semantics of input-consuming programs. In J. Lloyd, editor, *CL 2000*. Springer-Verlag, 2000.
8. A. Bossi, S. Etalle, S. Rossi, and J.-G. Smaus. Semantics and termination of simply-moded logic programs with dynamic scheduling. Available via CoRR: `http://arXiv.org/archive/cs/intro.html`, 2001.
9. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The *s*-semantics approach: theory and applications. *Journal of Logic Programming*, 19/20:149–197, 1994.
10. S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *Journal of Logic Programming*, 38(2):243–257, 1999.
11. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Constraint logic programming with dynamic scheduling: A semantics based on closure operators. *Information and Computation*, 137:41–67, 1997.
12. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.
13. Intelligent Systems Laboratory, Swedish Institute of Computer Science, PO Box 1263, S-164 29 Kista, Sweden. *SICStus Prolog User's Manual*, 1998. `http://www.sics.se/sicstus/docs/3.7.1/html/sicstus_toc.html`.
14. R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.
15. J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer-Verlag, 1987.
16. L. Naish. *Negation and Control in Prolog*, volume 238 of *LNCS*. Springer-Verlag, 1986.
17. L. Naish. Parallelizing NU-Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of ICLP/SLP '88*, pages 1546–1564. MIT Press, 1988.
18. J.-G. Smaus. Proving termination of input-consuming logic programs. In D. De Schreye, editor, *Proceedings of ICLP'99*, pages 335–349. MIT Press, 1999.
19. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.
20. K. Ueda and M. Morita. Moded Flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.