Semantics of Input-Consuming Logic Programs

Annalisa Bossi[1], Sandro Etalle[2], and Sabina Rossi[1]

[1] Dipartimento di Informatica, Università di Venezia, Italy
{bossi,srossi}@dsi.unive.it
[2] Universiteit Maastricht, The Netherlands
etalle@cs.unimaas.nl

Abstract. Input-consuming programs are logic programs with an additional restriction on the selectability (actually, on the resolvability) of atoms. This class of programs arguably allows to model logic programs employing a dynamic selection rule and constructs such as *delay declarations*: as shown also in [5], a large number of them are actually input-consuming.
In this paper we show that – under some syntactic restrictions – the $\mathcal{S}$-semantics of a program is correct and fully abstract also for input-consuming programs. This allows us to conclude that for a large class of programs employing delay declarations there exists a model-theoretic semantics which is equivalent to the operational one.
Keywords: logic programming, dynamic scheduling, semantics.

1 Introduction

Most implementations of logic programming languages allow the possibility of employing a *dynamic selection rule*: a selection rule which is not bound to the fixed left-to-right order of PROLOG. While this allows for more flexibility, it can easily yield to nontermination or to an inefficient computation. For instance, if we consider the standard program `APPEND`

```
   app([ ],Ys,Ys).
   app([H|Xs],Ys,[H|Zs]) ← app(Xs,Ys,Zs).
```

we have that the query `q1: app([1,2],[3,4],Xs), app(Xs,[5,6],Ys).` might easily loop infinitely (one just has to keep resolving the rightmost atom together with the second clause). To avoid this, most implementations use constructs such as *delay declarations*. In the case of `APPEND` when used for concatenating two lists the natural delay declaration is

```
d1: delay app(Xs,_,_) until nonvar(Xs).
```

This statement forbids the selection of an atom of the form `app(s,t,u)` unless `s` is a non-variable term, which is precisely what we need in order to run the query `q1` without overhead. Delay declarations, advocated by van Emden and de Lucena [16] and introduced explicitly in logic programming by Naish [13], provide the programmer with a better control over the computation and allow one to improve the efficiency of programs (wrt unrestricted selection rule), to prevent run-time errors, to enforce termination and to express some degree of synchronization among different processes (i.e., atoms) in a program, which allows to model parallelism (coroutining).

This extra control comes at a price: Many crucial results of logic programming do not hold in this extended setting. In particular, the equivalence between the declarative and operational semantics does not apply any longer. For instance, while the Herbrand semantics of `APPEND` is non-empty, the query `app(X,Y,Z)` has no successful derivation, as the computation starting in it *deadlocks*[1].

In this paper we address the problem of providing a model-theoretic semantics to programs using dynamic scheduling. In order to do so, we need a declarative way of modeling construct such as delay declarations: for this we restrict our attention to *input-consuming programs*. The definition of input-consuming program employs the concept of *mode*: We assume that programs are *moded*, that is, that the positions of each atom are partitioned into *input* and *output* ones. Then, *input-consuming* derivation steps are precisely those in which the input arguments of the selected atom will not be instantiated by the unification with the clause's head. For example, the standard mode for the program `APPEND` when used for concatenating two lists is `app(In,In,Out)`. Notice that in this case, for queries of the form `app(ts,us,X)` (`X` is variable disjoint from `ts` and `us`, which can be any possibly non-ground terms) the delay declaration `d1` guarantees precisely that if an atom is selectable and resolvable, then it is so via an input-consuming derivation step; conversely, in every input-consuming derivation the resolved atom satisfies the `d1`, thus it would have been selectable also in presence of the delay declaration. This reasoning applies for a large class of queries (among which `q1`), and is actually not a coincidence: In the sequel we argue that in most situations delay declarations are employed precisely for ensuring that the derivation is input-consuming (modulo renaming, i.e. modulo $\sim$, as explained later). Because of this, we are interested in providing a model-theoretic semantics for input-consuming programs. Clearly, most difficulties one has in doing this for programs with delay declarations apply to input-consuming programs as well. Intuitively speaking, the crucial problem here lies in the fact that computations may deadlock: i.e., reach a state in which no atom is resolvable (e.g., the query `app(X,Y,Z)`). Because of this the operational semantics is correct but not complete wrt the declarative one.

We prove that, if a program is well- and nicely-moded, then, for nicely-moded queries the operational semantics provided by the input-consuming resolution rule is correct and complete wrt the $\mathcal{S}$-semantics [11] for logic programs. The $\mathcal{S}$-semantics is a denotational semantics which – for programs without delay declarations – intuitively corresponds to the set of answer substitutions to the most general atomic queries, i.e., queries of the form $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are distinct variables. Moreover, the $\mathcal{S}$-semantics is compositional, it enjoys a

[1] A deadlock occurs when the current query contains no atom which can be selected for resolution.

model-theoretic reading, and it corresponds to the least fixpoint of a continuous operator.

Summarizing, we show that the $\mathcal{S}$-semantics of a program is compositional, correct and fully abstract also for input-consuming programs, provided that the programs considered are well- and nicely-moded, and that the queries are nicely-moded. It is important to notice that the queries we are considering don't have to be well-moded. Because of this, they might also deadlock. For instance, the query `app(X,Y,Z)` is nicely-moded, thus our results are applicable to it. One of the interesting aspects of the results we will present is that in some situations one can determine, purely from the declarative semantics of a program, that a query does (or does not) yield to deadlock.

This paper is organized as follows. The next section contains the preliminary notations and definitions. In the one which follows we introduce the $\mathcal{S}$-semantics together with the key concepts of moded and of input-consuming program. Section 4 contains the main results, and some examples of their applications. Section 5 concludes the paper. Some proofs are omitted for space reasons, and can be found in [7].

2 Preliminaries

The reader is assumed to be familiar with the terminology and the basic results of the semantics of logic programs [1, 2, 12]. Here we adopt the notation of [2] in the fact that we use boldface characters to denote sequences of objects; therefore $\boldsymbol{t}$ denotes a sequence of terms while $\boldsymbol{B}$ is a query (notice that – following [2] – queries are simply conjunctions of atoms, possibly empty). We denote atoms by $A, B, H, \ldots$, queries by $Q, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \ldots$, clauses by $c, d, \ldots$, and programs by P.

For any syntactic object o, we denote by $Var(o)$ the set of variables occurring in o. We also say that o is *linear* if every variable occurs in it at most once. Given a *substitution* $\sigma = \{x_1/t_1, ..., x_n/t_n\}$ we say that $\{x_1, \ldots, x_n\}$ is its *domain* (denoted by $Dom(\sigma)$) and that $Var(\{t_1, ..., t_n\})$ is its *range* (denoted by $Ran(\sigma)$). Further, we denote by $Var(\sigma) = Dom(\sigma) \cup Ran(\sigma)$. If $\{t_1, ..., t_n\}$ consists of variables then σ is called a *pure variable substitution*. If, in addition, $t_1, ..., t_n$ is a permutation of $x_1, ..., x_n$ then we say that σ is a *renaming*. The *composition* of substitutions is denoted by juxtaposition ($\theta\sigma(X) = \sigma(\theta(X))$). We say that a term t is an *instance* of t' iff for some σ, $t = t'\sigma$, further t is called 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'. An mgu θ of terms t and t' is called *relevant* iff $Var(\theta) \subseteq Var(t) \cup Var(t')$. The definitions above are extended to other syntactic objects in the obvious way.

Computations are sequences of derivation steps. The non-empty query q : $\boldsymbol{A}, B, \boldsymbol{C}$ and a clause $c : H \leftarrow \boldsymbol{B}$ (renamed apart wrt q) yield the resolvent $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})\theta$, provided that $\theta = mgu(B, H)$. A *derivation step* is denoted by $\boldsymbol{A}, B, \boldsymbol{C} \overset{\theta}{\Longrightarrow}_{P,c} (\boldsymbol{A}, \boldsymbol{B}, \mathbf{C})\theta$. c is called its *input clause*, and B is called the *selected atom* of q. A derivation is obtained by iterating derivation steps. A maximal sequence $\delta := Q_0 \overset{\theta_1}{\Longrightarrow}_{P,c_1} Q_1 \overset{\theta_2}{\Longrightarrow}_{P,c_2} \cdots Q_n \overset{\theta_{n+1}}{\Longrightarrow}_{P,c_{n+1}} Q_{n+1} \cdots$ of derivation steps is called an *SLD derivation of* $P \cup \{Q_0\}$ provided that for every step the standardization apart condition holds, i.e., the input clause employed at each step is variable disjoint from the initial query Q_0 and from the substitutions and the input clauses used at earlier steps. If the program P is clear from the context and the clauses $c_1, \ldots, c_{n+1}, \ldots$ are irrelevant, then we drop the reference to them. An SLD derivation in which at each step the leftmost atom is resolved is called a *LD derivation.* 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 $\delta := Q_0 \xrightarrow{\theta} Q_n$ with $\theta = \theta_1 \cdots \theta_n$, we say that δ is a *partial derivation* of $P \cup \{Q_0\}$. If δ is maximal and ends with the empty query then the restriction of θ to the variables of Q is called its *computed answer substitution* (*c.a.s.*, for short). The length of a (partial) derivation δ, denoted by $len(\delta)$, is the number of derivation steps in δ.

We recall the notion of *similar* SLD derivations and some related properties.

Definition 1 (Similar Derivations). *We say that two SLD derivations δ and δ' are* similar *($\delta \sim \delta'$) if (i) their initial queries are variants of each other; (ii) they have the same length; (iii) for every derivation step, atoms in the same positions are selected and the input clauses employed are variants of each other.*

Lemma 2. *Let $\delta := Q_1 \xrightarrow{\theta} Q_2$ be a partial SLD derivation of $P \cup \{Q_1\}$ and Q_1' be a variant of Q_1. Then, there exists a partial SLD derivation $\delta' := Q_1' \xrightarrow{\theta'} Q_2'$ of $P \cup \{Q_1'\}$ such that δ and δ' are similar.*

Lemma 3. *Consider two similar partial SLD derivations $Q \xrightarrow{\theta} Q'$ and $Q \xrightarrow{\theta'} Q''$. Then $Q\theta$ and $Q\theta'$ are variants of each other.*

3 Basic Definitions

In this section we introduce the basic definitions we need: The ones of input-consuming derivations and of the $\mathcal{S}$-semantics. Then we introduce the concepts of well- and nicely-moded programs.

Input-Consuming Derivations We start by recalling the notion of *mode*, which is a function that labels as *input* or *output* the positions of each predicate in order to indicate how the arguments of a predicate should be used.

Definition 4 (Mode). *Consider an n-ary predicate symbol p. By a* mode *for p we mean a function m_p from $\{1, \ldots, n\}$ to $\{In, Out\}$.*

If $m_p(i) = In$ (resp. Out), we say that i is an *input* (resp. *output*) *position of p* (with respect to m_p). We assume that each predicate symbol has a unique mode associated to it; multiple modes may be obtained by simply renaming the predicates. We denote by $In(Q)$ (resp. Out(Q)) the sequence of terms filling in the input (resp. output) positions of Q. Moreover, when writing an atom as $p(\boldsymbol{s}, \boldsymbol{t})$, we are indicating with $\boldsymbol{s}$ the sequence of terms filling in its input positions and with $\boldsymbol{t}$ the sequence of terms filling in its output positions. The notion of input-consuming derivation was introduced in [14] and is defined as follows.

Definition 5 (Input-Consuming).

- *A derivation step* $\boldsymbol{A}, B, \boldsymbol{C} \overset{\theta}{\Longrightarrow}_c (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})\theta$ *is called* input-consuming *iff* $In(B)\theta = In(B)$.
- *A derivation is called* input-consuming *iff all its derivation steps are input-consuming.*

Thus, a derivation step is input consuming if the corresponding mgu does not affect the input positions of the selected atom. Clearly, because of this additional restriction, there exist queries in which no atom is resolvable via an input-consuming derivation step. In this case we say that the query *suspends*.

Example 6. Consider the following program `REVERSE` using an accumulator.

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

When used for reversing a list, the natural mode for this program is[2] the following one: `reverse(In,Out)`, `reverse_acc(In,Out,In)`. Consider now the query `reverse([X1,X2],Zs)`. The following derivation is input-consuming.

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

As usual, □ denotes the empty query. Notice also that a natural delay declaration for this program would be

```
delay reverse(X,_) until nonvar(X).
delay reverse_acc(X,_,_) until nonvar(X).
```

Now, it is easy to see that for queries of the form `reverse(t,X)`, where `t` is *any* term and `X` any variable disjoint from `t`, the above delay declarations guarantee precisely that the resulting derivations are input-consuming (modulo $\sim$). Furthermore, for the same class of queries it holds that in any input-consuming derivation the selected atom satisfies the above delay declarations. □

Delay declarations vs. input-consuming derivations As suggested in the above example, and stated in the introduction, we believe that the concept of input-consuming program allows one to model programs employing delay declarations in a nice way: we claim that in most programs delay declarations are used to enforce that the derivations are input-consuming (modulo $\sim$). We have addressed this topic already in [5]. We now borrow a couple of arguments from it, and extend them.

[2] The other possible modes are `reverse(Out,In)` (which is symmetric and equivalent to the above one) and `reverse(In,In)` which might be used for checking if a list is a palindrome.

Generally, delay declarations are employed to guarantee that the interpreter will not use an "inappropriate" clause for resolving an atom (the other, perhaps less prominent use of delay declarations is to ensure absence of runtime errors, we don't address this issue in this paper). In fact, if the interpreter always selected the appropriate clause, by the independence from the selection rule one would not have to worry about the order of the selection of the atoms in the query. In practice, delay declarations prevent the selection of an atom until a certain degree of instantiation is reached. This degree of instantiation ensures that the atom is unifiable only with the heads of the "appropriate" clauses. In presence of modes, we can reasonably assume that this degree of instantiation is the one of the *input* positions. Now, take an atom $p(\boldsymbol{s},\boldsymbol{t})$, that it is resolvable with a clause c by means of an input-consuming derivation step. Then, for every instance $\boldsymbol{s}'$ of $\boldsymbol{s}$, we have that the atom $p(\boldsymbol{s}',\boldsymbol{t})$ is as well resolvable with c by means of an input-consuming derivation step. Thus, no further instantiation of the input positions of $p(\boldsymbol{s},\boldsymbol{t})$ can rule out c as a possible clause for resolving it, and c must then be one of the "appropriate" clauses for resolving $p(\boldsymbol{s},\boldsymbol{t})$ and we can say that $p(\boldsymbol{s},\boldsymbol{t})$ is "sufficiently instantiated" in its input positions to be resolved with c. On the other hand, following the same reasoning, if $p(\boldsymbol{s},\boldsymbol{t})$ is resolvable with c but not via an input-consuming derivation step, then there exists an instance $\boldsymbol{s}'$ of $\boldsymbol{s}$, such that $p(\boldsymbol{s}',\boldsymbol{t})$ is not resolvable with c. In this case we can say that $p(\boldsymbol{s},\boldsymbol{t})$ is not instantiated enough to know whether c is one of the "appropriate" clauses for resolving it.

We conclude this section with a result stating that also when considering input-consuming derivations, it is not restrictive to assume that all mgu's used in a derivation are relevant. The proof can be found in [7].

Lemma 7. *Let $p(\boldsymbol{s},\boldsymbol{t})$ and $p(\boldsymbol{u},\boldsymbol{v})$ be two atoms. If there exists an mgu θ of $p(\boldsymbol{s},\boldsymbol{t})$ and $p(\boldsymbol{u},\boldsymbol{v})$ such that $\boldsymbol{s}\theta = \boldsymbol{s}$ then there exists a* relevant *mgu ϑ of $p(\boldsymbol{s},\boldsymbol{t})$ and $p(\boldsymbol{u},\boldsymbol{v})$ such that $\boldsymbol{s}\vartheta = \boldsymbol{s}$.*

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

The $\mathcal{S}$-semantics The aim of the $\mathcal{S}$-semantics approach (see [8]) is modeling the observable behaviors for a variety of logic languages. The observable we consider here is the *computed answer substitutions*. The semantics is defined as follows:

$$\begin{aligned}\mathcal{S}(P) = \{\ p(x_1,\ldots,x_n)\theta \mid\ & x_1,\ldots,x_n \text{ are distinct variables and} \\ & p(x_1,\ldots,x_n) \xrightarrow{\theta}_P \Box \text{ is an SLD derivation}\}.\end{aligned}$$

This semantics enjoys all the valuable properties of the least Herbrand model. Technically, the crucial difference is that in this setting an interpretation might contain non-ground atoms. To present the main results on the $\mathcal{S}$-semantics we need to introduce two further concepts: Let P be a program, and I be a set of atoms. The immediate consequence operator for the $\mathcal{S}$-semantics is defined as:

$$\begin{aligned}T_P^{\mathcal{S}}(I) = \{\ H\theta \mid\ & \exists\ H \leftarrow \boldsymbol{B} \in P \\ & \exists\ \boldsymbol{C} \in I, \text{renamed apart}^3 \text{ wrt } H, \boldsymbol{B} \\ & \theta = mgu(\boldsymbol{B},\boldsymbol{C}) \qquad\qquad \}.\end{aligned}$$

Moreover, a set of atoms I is called an *$\mathcal{S}$-model* of P if $T_P^{\mathcal{S}}(I) \subseteq I$. Falaschi et al. [11] showed that $T_P^{\mathcal{S}}$ is continuous on the lattice of term interpretations, that is sets of possibly non-ground atoms, with the subset-ordering. They proved the following:

- $\mathcal{S}(P) =$ least $\mathcal{S}$-model of $P = T_P^{\mathcal{S}} \uparrow \omega$.

Therefore, the $\mathcal{S}$-semantics enjoys a declarative interpretation and a bottom-up construction, just like the Herbrand one. In addition, we have that the $\mathcal{S}$-semantics reflects the observable behavior in terms of computed answer substitutions, as shown by the following well-known result.

Theorem 8. *[11] Let P be a program, $\boldsymbol{A}$ be a query, and θ be a substitution. The following statements are equivalent.*

- *There exists an SLD derivation $\boldsymbol{A} \xrightarrow{\vartheta}_P \Box$, where $\boldsymbol{A}\vartheta \approx \boldsymbol{A}\theta$.*
- *There exists $\boldsymbol{A}' \in \mathcal{S}(P)$ (renamed apart wrt $\boldsymbol{A}$), such that $\sigma = mgu(\boldsymbol{A}, \boldsymbol{A}')$ and $\boldsymbol{A}\sigma \approx \boldsymbol{A}\theta$.*

Let us see this semantics applied to the programs so far encountered.

$\mathcal{S}$(APPEND) = { `app([],X,X),`
`app([X1],X,[X1|X]),`
`app([X1,X2],X,[X1,X2|X]),` ...}.

$\mathcal{S}$(REVERSE) = { `reverse([],[]),`
`reverse([X1],[X1]),`
`reverse([X1,X2],[X2,X1]),` ...
`reverse_acc([],X,X),`
`reverse_acc([X1],X,[X1|X]),`
`reverse_acc([X1,X2],X,[X2,X1|X]),` ...}.

Well and Nicely-Moded Programs Even in presence of modes, the $\mathcal{S}$-semantics does not reflect the operational behavior of input-consuming programs (and thus of programs employing delay declarations). In fact, if we extend `APPEND` by adding to it the clause `q` $\leftarrow$ `app(X,Y,Z).` we have that `q` belongs to the semantics but the query `q` will not succeed (it suspends). In order to guarantee that the semantics is fully abstract (wrt the computed answer substitutions) we need to restrict the class of allowed programs and queries. To this end we introduce the concepts of well-moded [10] and of nicely-moded programs.

Definition 9 (Well-Moded).
- *A query $p_1(\boldsymbol{s}_1, \boldsymbol{t}_1), \ldots, p_n(\boldsymbol{s}_n, \boldsymbol{t}_n)$ is* well-moded *if for all $i \in [1, n]$*

$$Var(\boldsymbol{s}_i) \subseteq \bigcup_{j=1}^{i-1} Var(\boldsymbol{t}_j).$$

[3] Here and in the sequel, when we write "$\boldsymbol{C} \in I$, renamed apart wrt some expression e", we naturally mean that I contains a set of atoms $C_1', \ldots, C_n'$, and that $\boldsymbol{C}$ is a renaming of $C_1', \ldots, C_n'$ such that $\boldsymbol{C}$ shares no variable with e and that two distinct atoms of $\boldsymbol{C}$ share no variables with each other.

– *A clause* $p(\boldsymbol{t}_0, \boldsymbol{s}_{n+1}) \leftarrow p_1(\boldsymbol{s}_1, \boldsymbol{t}_1), \ldots, p_n(\boldsymbol{s}_n, \boldsymbol{t}_n)$ *is* well-moded *if for all* $i \in [1, n+1]$

$$Var(\boldsymbol{s}_i) \subseteq \bigcup_{j=0}^{i-1} Var(\boldsymbol{t}_j).$$

– *A program is* well-moded *if all of its clauses are well-moded.*

Thus a query is well-moded if every variable occurring in an input position of an atom occurs in an output position of an earlier atom in the query. A clause is well-moded if (1) every variable occurring in an input position of a body atom occurs either in an input position of the head, or in an output position of an earlier body atom; (2) every variable occurring in an output position of the head occurs in an input position of the head, or in an output position of a body atom.

The concept of nicely-moded programs was first introduced by Chadha and Plaisted [9].

Definition 10 (Nicely-Moded).

– *A query* $p_1(\boldsymbol{s}_1, \boldsymbol{t}_1), \ldots, p_n(\boldsymbol{s}_n, \boldsymbol{t}_n)$ *is called* nicely-moded *if* $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_n$ *is a linear sequence of terms and for all* $i \in [1, n]$

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

– *A clause* $p(\boldsymbol{s}_0, \boldsymbol{t}_0) \leftarrow p_1(\boldsymbol{s}_1, \boldsymbol{t}_1), \ldots, p_n(\boldsymbol{s}_n, \boldsymbol{t}_n)$ *is* nicely-moded *if its body is nicely-moded and*

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

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

Note that an atomic query $p(\boldsymbol{s}, \boldsymbol{t})$ is nicely-moded if and only if $\boldsymbol{t}$ is linear and $Var(\boldsymbol{s}) \cap Var(\boldsymbol{t}) = \emptyset$.

Example 11. Programs `APPEND` and `REVERSE` are both well- and nicely-moded. Furthermore, Consider now the following program `PALINDROME`

```
palindrome(Xs) ← reverse(Xs,Xs).
```

Together with `REVERSE`. With the mode `palindrome(In)`, this program is well-moded but not nicely-moded (`Xs` occurs both in an input and in an output position of the same body atom). Nevertheless, it becomes both well-moded and nicely-moded if the adopted modes of `REVERSE` are the following ones: `reverse(In,In)`, `reverse_acc(In,In,In)`. □

4 Semantics of Input-Consuming Programs

In this section we are going to make the link between input-consuming programs, well- and nicely-moded programs and the $\mathcal{S}$-semantics: We show that the $\mathcal{S}$-semantics of a program is compositional, correct and fully abstract also for input-consuming programs, provided that the programs are well- and nicely-moded and that only nicely-moded queries are considered.

Properties of Well-Moded Programs We start by demonstrating some important features of well-moded programs. For this, we need additional notations: First, the following notion of *renaming for a term t* from [2] will be used.

Definition 12. *A substitution* $\theta := \{x_1/y_1, \ldots, x_n/y_n\}$ *is called a* renaming for a term t *if* $Dom(\theta) \subseteq Var(t)$, $y_1, \ldots, y_n$ *are different variables, and* $(Var(t) - \{x_1, \ldots, x_n\}) \cap \{y_1, \ldots, y_n\} = \emptyset$ *(θ does not introduce variables which occur in t but are not in the domain of θ).*

Observe that terms s and t are variants iff there exists a renaming θ for s such that $t = s\theta$. Then, we need the following: Let $Q := p_1(\boldsymbol{s}_1, \boldsymbol{t}_1), \ldots, p_n(\boldsymbol{s}_n, \boldsymbol{t}_n)$. We define

- $VIn^*(Q) = \bigcup_{i=1}^{n} \{x \mid x \in Var(\boldsymbol{s}_i) \text{ and } x \notin \bigcup_{j=1}^{i-1} Var(\boldsymbol{t}_j)\}$

Thus, $VIn^*(Q)$ denotes the set of variables occurring in an input position of an atom of Q but not occurring in an output position of an earlier atom. Note also that if Q is well-moded then $VIn^*(Q) = \emptyset$.

We now need the following technical result concerning well-moded programs. Because of lack of space, the proof is omitted, and can be found in [7].

Lemma 13. *Let P be a well-moded program, Q be a query and $\delta := Q \xrightarrow{\theta} Q'$ be a partial LD derivation of $P \cup \{Q\}$. If $\theta_{|VIn^*(Q)}$ is a renaming for Q then δ is similar to an input-consuming partial (LD) derivation.*

We can now prove our crucial result concerning well-moded programs. Basically, it states the *correctness* of the $\mathcal{S}$-semantics for well-moded, input-consuming programs. This can be regarded as "one half" of the main result we are going to propose.

Proposition 14. *Let P be a well-moded program, A be an atomic query and θ be a substitution.*

- *If there exists $A' \in \mathcal{S}(P)$ (renamed apart wrt A), and $\sigma = mgu(A, A')$ such that*
  (i) $In(A)\sigma \approx In(A)$,
  (ii) $A\sigma \approx A\theta$,
- *then there exists an input-consuming (LD) derivation $\delta := A \xrightarrow{\vartheta}_P \Box$, such that $A\vartheta \approx A\theta$.*

Proof. Let $A' \in \mathcal{S}(P)$ (renamed apart wrt A) and σ be such that the hypothesis are satisfied. By Theorem 8, there exists a successful SLD derivation of $P \cup \{A\}$ with c.a.s. ϑ' such that $A\vartheta' \approx A\theta$. By the Switching Lemma [2], there exists a successful LD derivation δ' of $P \cup \{A\}$ with c.a.s. ϑ'. From the hypothesis, it follows that $\vartheta'_{|In(A)}$ is a renaming for A. By Lemma 13, there exists an input-consuming derivation $A \xrightarrow{\vartheta}_P \Box$ similar to δ'. The thesis follows by Lemma 3. $\Box$

Properties of Nicely-Moded Programs Now, we need to establish some properties of nicely-moded programs. First, we recall the following from [5, 6].

Lemma 15. *Let the program P and the query Q be nicely moded. Let $\delta :=$ $Q \xrightarrow{\theta} Q'$ be a partial input-consuming derivation of $P \cup \{Q\}$. Then, for all $x \in Var(Q)$ and $x \notin Var(Out(Q))$, $x\theta = x$.*

Note that if Q is nicely-moded then $x \in Var(Q)$ and $x \notin Var(Out(Q))$ iff $x \in VIn^*(Q)$. Now, we can prove that the $\mathcal{S}$-semantics is *fully abstract* for input-consuming, nicely-moded programs and queries. This can be regarded as the counterpart of Proposition 14.

Proposition 16. *Let P be a nicely-moded program, A be a nicely-moded atomic query and θ be a substitution.*

- *If there exists an input-consuming SLD derivation $\delta := A \xrightarrow{\vartheta}_P \Box$, such that $A\vartheta \approx A\theta$,*
- *then there exists $A' \in \mathcal{S}(P)$ (renamed apart wrt A), and $\sigma = mgu(A, A')$ such that*
  (i) $In(A)\sigma \approx In(A)$,
  (ii) $A\sigma \approx A\theta$.

Proof. By Theorem 8, there exist $A' \in \mathcal{S}(P)$ (renamed apart wrt A) and a substitution σ such that $\sigma = mgu(A, A')$ and (ii) holds. Since δ is an input-consuming derivation, by Lemma 15, it follows that $\vartheta_{|In(A)}$ is a renaming for A. Hence (i) follows by the hypothesis and (ii). $\Box$

Semantics of Input-Consuming Derivations We now put together the above propositions and extend them compositionally to arbitrary (non-atomic) queries. For this, we need the the following simple result.

Lemma 17. *Let the program P be well and nicely-moded and the query Q be nicely-moded. Then, there exists a well- and nicely-moded program P' and a nicely-moded atomic query A such that the following statements are equivalent.*

- *There exists an input-consuming successful derivation δ of $P \cup \{Q\}$ with c.a.s. θ.*
- *There exists an input-consuming successful derivation δ' of $P' \cup \{A\}$ with c.a.s. θ.*

Proof. (sketch). This is done in a straightforward way by letting P' be the program $P \cup \{c: \ \mathit{new}(\boldsymbol{x}, \boldsymbol{y}) \leftarrow Q\}$ where $\boldsymbol{x} = \mathit{VIn}^*(Q)$, $\boldsymbol{y} = \mathit{Var}(\mathit{Out}(Q))$, *new* is a fresh predicate symbol and $A = \mathit{new}(\boldsymbol{x}, \boldsymbol{y})$. $\square$

We are now ready for the main result of this paper, which asserts that the declarative semantics $\mathcal{S}(P)$ is compositional and fully abstract for input-consuming programs, provided that programs are well- and nicely-moded and that queries are nicely-moded.

Theorem 18. *Let P be a well- and nicely-moded program, $\boldsymbol{A}$ be a nicely-moded query and θ be a substitution. The following statements are equivalent.*

(i) *There exists an input-consuming derivation $\boldsymbol{A} \xrightarrow{\vartheta}_P \square$, such that $\boldsymbol{A}\vartheta \approx \boldsymbol{A}\theta$.*
(ii) *There exists $\boldsymbol{A}' \in \mathcal{S}(P)$ (renamed apart wrt $\boldsymbol{A}$), and $\sigma = mgu(\boldsymbol{A}, \boldsymbol{A}')$ such that*
  (a) *$\sigma_{|\mathit{VIn}^*(\boldsymbol{A})}$ is a renaming for $\boldsymbol{A}$,*
  (b) *$\boldsymbol{A}\sigma \approx \boldsymbol{A}\theta$.*

Proof. It follows immediately from Propositions 14, 16 and Lemma 17. $\square$

Note that in case of an atomic query $\boldsymbol{A} := A$, we might substitute condition *(a)* above with the somewhat more attractive condition *(a') $\mathit{In}(A)\sigma \approx \mathit{In}(A)$*. Let us immediately see some examples.

Example 19.

– `app([X,b],Y,Z)` has an input-consuming successful derivation, with c.a.s. $\theta \approx \{\mathtt{Z}/[\mathtt{X},\mathtt{b}|\mathtt{Y}]\}$. This can be concluded by just looking at $\mathcal{S}(\mathtt{APPEND})$, from the fact that A = `app([X1,X2],X3,[X1,X2|X3])` $\in \mathcal{S}(P)$. Notice that `app([X,b],Y,Z)` is – in its input position – an instance of A.
– `app(Y,[X,b],Z)` has no input-consuming successful derivations. This is because there is no $A \in \mathcal{S}(P)$ such that $\mathit{In}(\mathtt{app}(\mathtt{Y}, [\mathtt{X}, \mathtt{b}], \mathtt{Z})$ is an instance of A in the input position. This actually implies that in presence of delay declarations `app(Y,[X,b],Z)` will eventually either deadlock or run into an infinite derivation; we are going to talk more about this in the next section. $\square$

Note that Theorem 18 holds also in the case that programs are *permutation well- and nicely-moded* and queries are *permutation nicely-moded* [15], i.e., programs which would be well- and nicely-moded after a permutation of the atoms in the bodies and queries which would be nicely-moded through a permutation of their atoms.

Deadlock We now consider again programs employing delay declarations. An important consequence of Theorem 18 is that when the delay declarations imply that the derivations are input-consuming (modulo $\sim$), then one can determine from the model-theoretic semantics whether a query is bound to deadlock or not. Let us establish some simple notation. In this section we assume that programs are augmented with delay declarations, and we say that a derivation *respects* the delay declarations iff every selected atom satisfies the delay declarations.

Notation 20. Let P be a program and $\boldsymbol{A}$ be a query.

– We say that $P \cup \{\boldsymbol{A}\}$ is *input-consuming correct* iff every SLD derivation of $P \cup \{\boldsymbol{A}\}$ which respects the delay declarations is similar to an input-consuming derivation.
– We say that $P \cup \{\boldsymbol{A}\}$ is *input-consuming complete* iff every input-consuming derivation of $P \cup \{\boldsymbol{A}\}$ respects the delay declarations.
– We say that $P \cup \{\boldsymbol{A}\}$ *is bound to deadlock* if
  (i) every SLD derivation of $P \cup \{\boldsymbol{A}\}$ which respects the delay declarations either fails or deadlocks[4], and
  (ii) there exists at least one non-failing SLD derivation of $P \cup \{\boldsymbol{A}\}$ which respects the delay declarations. □

For example, consider the program `REVERSE` (including delay declarations).

– `REVERSE` $\cup$ `reverse(s,Z)` is input-consuming correct and complete provided that `Z` is a variable disjoint from `s`.

Consider now the program `APPEND` augmented with the delay declaration `d1` of the introduction.

– `APPEND` $\cup$ `app(s,t,Z)` is input-consuming correct and complete provided that `Z` is a variable disjoint from the possibly non-ground terms `s` and `t`.
– Now, following up on Example 19, since `APPEND` $\cup$ `app([X,b],Y,Z)` is input-consuming complete, we can state that `APPEND` $\cup$ `app([X,b],Y,Z)` is not bound to deadlock.

In order to say something about the other query of Example 19 (`app(Y,[X,b],Z)`) we need a further reasoning: Consider for the moment the nicely-moded query `app(X,Y,Z)`. Since $\mathcal{S}$(`APPEND`) contains instances of it, by Theorem 8, `app(X,Y,Z)` has at least one successful SLD derivation. Thus, it does not fail. On the other hand, every atom in $\mathcal{S}$(`APPEND`) is in its input positions a proper instance of `app(X,Y,Z)`. Thus by Theorem 18, `app(X,Y,Z)` has no input-consuming successful derivations. Therefore, since `APPEND` $\cup$ `app(X,Y,Z)` is input-consuming correct, we can state that `app(X,Y,Z)` either has an infinite input-consuming derivation or it is bound to deadlock. This fact can be nicely combined with the fact that `APPEND` is *input-terminating* [5]: i.e., all its input-consuming derivations starting in a nicely-moded query are finite. In [5] we provided conditions which guaranteed that a program is input-terminating; these conditions easily allow one to show that `APPEND` in input-terminating. Because of this, we can conclude that the query `app(X,Y,Z)` is bound to deadlock.

By simply formalizing this reasoning, we obtain the following.

Theorem 21. *Let P be a well- and nicely-moded program, and A be nicely-moded atomic query. If*

[4] A derivation deadlocks if its last query contains no selectable atom, i.e., no atom which satisfies the delay declarations

1. $\exists\ B \in \mathcal{S}(P)$, such that A unifies with B,
2. $\forall\ B \in \mathcal{S}(P)$, if A unifies with B, then $In(A)$ is not an instance of $In(B)$,
3. $P \cup \{A\}$ is input-consuming-correct,

then A either has an infinite SLD derivation respecting the delay declarations or it is bound to deadlock.
If in addition P is input-terminating then A is bound to deadlock.

This result can be immediately generalized to non-atomic queries, as done for our main result. Let us see more examples:

– `APPEND` $\cup$ `app(Y,[X,b],Z)` either has an infinite derivation or it is bound to deadlock.
– Since `APPEND` is input terminating, we have that `APPEND` $\cup$ `app(Y,[X,b],Z)` is bound to deadlock.

One might wonder why in order to talk about deadlock we went back to programs using delay declarations. The crucial point here lies in the difference between *resolvability* - via an input-consuming derivation step - (used in input-consuming programs) and *selectability* (used in programs using delay declarations). When resolvability does not reduce to selectability, we cannot talk about (the usual definition of) deadlocking derivation. Consider the following program, where all atom's positions are moded as *input*.

```
p(X) ← q(a).    p(a).    q(b).
```

The derivation starting in `p(X)` does not succeed, does not fail, but it also does not deadlock in the usual sense: in fact, `p(X)` can be resolved with the first clause, which however yields to failure. We can say that each input-consuming SLD tree starting in `p(X)` is incomplete, as it contains a branch which cannot be followed. In the moment that the program is input-consuming correct, we can refer to the usual definition of deadlocking derivation.

Counterexamples The following examples demonstrate that the syntactic restrictions used in Theorem 18 are necessary. Consider the following program.

```
p(X,Y) ← equal_lists(X,Y), list_of_zeroes(Y).
equal_lists([ ],[ ]).
equal_lists([H|T],[H|T']) ← equal_lists(T,T').
list_of_zeroes([ ]).
list_of_zeroes([0|T]) ← list_of_zeroes(T).
```

With the modes: `p(In,Out)`, `equal_lists(In,Out)`, `list_of_zeroes(Out)`. The first clause is not nicely-moded because of the double occurrence of `Y` in the body's output positions. Here, there exists a successful input-consuming derivation starting in `p([X1],Y)`, and producing the c.a.s. $\{$`X1/0`$,$ `Y/[X1]`$\}$. Nevertheless, there exists no corresponding $A' \in \mathcal{S}(P)$ (in fact, $\mathcal{S}(P)|_{\mathtt{p}}$ contains all and only all the atoms of the form `p(list0,list0)` where `list0` is a list containing only zeroes). This shows that if the program is well-moded but not nicely-moded then the implication (i) $\Rightarrow$ (ii) in Theorem 18 does not hold. Now consider the following program:

```
p(X) ← list(Y), equal_lists(X,Y).
equal_lists([ ], [ ]).
equal_lists([H|T],[H|T']) ← equal_lists(T,T').
list([ ]).
list([HH|T]) ← list(T).
```

With the modes `p(In)`, `equal_lists(In, In)`, `list(Out)`. This program is nicely-moded, but not well-moded: The variable `HH` in the output position of the head occurs neither in an output position of the body nor in an input position of the head. It is easy to check that there does not exist any successful input-consuming derivation for the query p([a]); at the same time, p([X1]) $\in \mathcal{S}(P)$. Thus, if the program is nicely-moded but not well-moded then the implication (ii) $\Rightarrow$ (i) in Theorem 18 does not hold.

5 Concluding Remarks

We have shown that – under some syntactic restrictions – the $\mathcal{S}$-semantics reflects the operational semantics also when programs are *input-consuming.* The $\mathcal{S}$-semantics is a denotational semantics which enjoys a model-theoretical reading. The relevance of the results is due to the fact that input-consuming programs often allow to model the behavior of programs employing delay declarations; hence for a large part of programs employing dynamic scheduling there exists a declarative semantics which is equivalent to the operational one.

As related work we want to mention Apt and Luitjes [3]. The crucial difference with it is that in [3] conditions which ensure that the queries are *deadlock-free* are employed. Under these circumstances the equivalence between the operational and the Herbrand semantics follows. On the other hand, the class of queries we consider here (the nicely-moded ones) includes many which would "deadlock" (e.g., `app(X,Y,Z)`): Theorem 18 proves that in many cases one can tell by the declarative semantics for instance if a query is "sufficiently instantiated" to yield a success or if it is bound to deadlock.

Concerning the restrictiveness of the syntactic concepts we use here (well- and nicely-moded programs and nicely-moded queries) we want to mention that [4, 5] both contain mini-surveys of programs with the indication whether they are well- and nicely-moded or not. From them, it appears that most "usual" programs satisfy both definitions. It is important to stress that under this restriction one might still want to employ a dynamic selection rule. Consider for instance a query of the form `read_tokens(X),modify(X,Y),write_tokens(Y)`, where the modes are `read_tokens(Out),modify(In,Out),write_tokens(Out)`. If `read_tokens` cannot read the input stream all at once, it makes sense that `modify` and `write_tokens` be called in order to process and display the tokens that are available, even if `read_tokens` has not finished reading the input. This can be done by using dynamic scheduling, using either delay declarations or an input-consuming resolution rule in order to avoid nontermination and inefficiencies.

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

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

[5] A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. *Electronic Notes in Theoretical Computer Science*, 30(1), 1999. http://www.elsevier.nl/locate/entcs, temporarily available at http://www.cs.unimaas.nl/∼etalle/papers/index.htm.

[6] A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. Technical Report CS 99-06, Universiteit Maastricht, 1999.

[7] A. Bossi, S. Etalle, and S. Rossi. Semantics of input-consuming programs. Technical Report CS 00-01, Universiteit Maastricht, 2000.

[8] Annalisa Bossi, Maurizio Gabrielli, Giorgio Levi, and Maurizio Martelli. The S-semantics approach: Theory and applications. *The Journal of Logic Programming*, 19 & 20:149–198, May 1994.

[9] R. Chadha and D.A. Plaisted. Correctness of unification without occur check in Prolog. Technical report, Department of Computer Science, University of North Carolina, Chapel Hill, N.C., 1991.

[10] P. Dembinski and J. Maluszynski. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the International Symposium on Logic Programming*, pages 29–38, Boston, 1985.

[11] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

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

[13] L. Naish. An introduction to mu-prolog. Technical Report 82/2, The University of Melbourne, 1982.

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

[15] J.-G. Smaus, P. M. Hill, and A. M. King. Termination of logic programs with `block` declarations running in several modes. In C. Palamidessi, editor, *Proceedings of the 10th Symposium on Programming Language Implementations and Logic Programming*, LNCS. Springer-Verlag, 1998.

[16] M.H. van Emden and G.J. de Lucena. Predicate logic as a language for parallel programming. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, London, 1982. Academic Press.