Semantics of Well-Moded Input-Consuming Logic Programs

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

[1]Dipartimento di Informatica, Università Ca' Foscari di Venezia
via Torino 155, 30172 Venezia, Italy
{bossi,srossi}@dsi.unive.it

[2]Department of Computer Science, University of Maastricht
and
CWI, Amsterdam
P.O. Box 616, 6200 MD Maastricht, The Netherlands
etalle@cs.unimaas.nl

Abstract

Recent logic programming languages employ dynamic scheduling of calls to improve efficiency of programs. Dynamic scheduling is realized by allowing some calls to be dynamically "delayed" until their arguments are sufficiently instantiated. To this end, logic languages are extended with constructs such as *delay declarations*.

However, many declarative properties that hold for logic and pure Prolog programs do not apply any longer in this extended setting. In particular, the equivalence between the model-theoretic and operational semantics does not hold.

In this paper, we study the class of *input-consuming* programs. Firstly, we argue that input-consuming logic programs are suitable for modeling programs employing delay declarations. Secondly, 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. Thus, input-consuming programs are shown to be the right answer for conjugate efficiency and declarativeness.

Keywords: Logic programming, dynamic scheduling, semantics

1 Introduction

Control in Logic Programming According to the widely-known and accepted slogan *Algorithm = Logic + Control* [1], a program can be regarded as a logic specification together with a control mechanism for executing it. In this light, one of the inspiring ideas of the logic programming paradigm is to ask the programmer only for the *logic specification* and leave the control part to the interpreter.

As an example of this declarative style, in Prolog, queries (calls) can often be used in different manners. The well-known predicate `append` reported below, for example, can be used either to concatenate two lists or to split a list into two parts in a nondeterministic way: the call `append([a,b],[c,d],X)` will succeed by unifying the variable `X` to the list `[a,b,c,d]`, while the call `append(X1,X2,[a,b,c,d])` will split the list `[a,b,c,d]` into `X1` and `X2`.

Nevertheless, as one can expect, if in a logic program the control component was totally absent, programs would be hopelessly inefficient, if not divergent. To see how the control component comes back in the picture one has to look into the underneath resolution process. The execution of a logic program with respect to a given query consists in building and exploring a proof tree to validate the query. The control lies thus in the way the proof tree is built (for a query there may exist different proof trees) and is traversed. Prolog employs for instance a left-to-right selection rule together with a top-down clause selection method. These two components determine the shape of the proof tree, which is then traversed depth-first.

A programmer is *always* aware of this methodology, and writes her programs according to it. We are convinced that most practical Prolog programs would diverge if used in combination with another selection rule.

Dynamic Selection Rules While Prolog's rule has proven to be extremely efficient and effective, for many application a fixed selection rule is too limited to be practical. As an example, consider the program `APPEND` and the program `IN_ORDER` which constructs the list of the nodes of a binary tree by means of an in-order transversal

```
%  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).`

(we assume that the predicates `read_tree` and `write_list` are defined elsewhere in the program). 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 `in_order` and `write_list` on the available input. This can only be achieved if we depart from Prolog's left-to-right selection rule, which would call `in_order` only when `read_tree` had finished reading the input.

However, dropping Prolog's selection rules poses the problem of nontermination and of efficiency. In the above program, the computation of the query Q would immediately diverge when adopting any fixed selection rule different from Prolog's. The computation would instead result in an enormous waste of resources when using fair or random selection rules. What we would need here is to interleave the execution of the three "processes" in the query, in a *controlled* manner. This can be achieved using a *dynamic selection rule*, i.e., a selection rule which employs a runtime mechanism which determines which atoms *might* be selected. For instance, in the case of the above example, a correct computation can be achieved by means of the following *delay declarations:*

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

The semantics of these delay declarations is rather straightforward: they forbid the selection of an atom of the form `in_order`(s,t) (resp. `append`(s,t,u) or `write_list`(s,t)) unless s is a non-variable term. We can say that these statements avoid that predicates `in_order`, `append` and `write_list` be selected "too early". Notice that with these declarations `IN_ORDER` enjoys a parallel execution by means of interleaving.

The use of a non-fixed selection rule in combination with the above delay declarations is thus an example of a *dynamic selection rule.* Dynamic selection rules have proven to be useful in a number of applications; among other things, they allow one to model co-routining [2, 3] and parallel executions [4]. A dynamic selection rule provides the programmer with a flexible control over the computation which can be used to improve the efficiency of programs, prevent run-time errors and enforce termination [5, 3].

Dynamic selection rules are usually implemented by means of a mechanism preventing the selection of those atoms which are not sufficiently instantiated. To this end, different languages use different constructs. In GHC [6] programs are augmented with *guards* in order to control the selection of atoms dynamically. Moded flat GHC [7] uses an extra condition on the input positions, which is extremely similar to the concept of input-consuming derivation step we refer to in the sequel: The resolution of an atom with a definition might not instantiate the input arguments of the resolved atom. On the other hand, Gödel [2] and Eclipse [8] use *delay declarations* like the above ones, and SICStus Prolog [9] employs `block` declarations (which are strictly less expressive than delay declarations). Both `delay` and `block` declarations check the partial instantiation of some arguments of calls.

Limitations of the Approach The adoption of a control mechanism such as delay declarations comes at a price: Many declarative properties that have been proven for logic and pure Prolog programs do not apply any longer. In particular, the well-known equivalence between the model-theoretic and operational semantics (see [10, 11]) does not hold. For example, the query `append(X,Y,Z)` with the above delay declaration does not succeed: the atom `append(X,Y,Z)` does not satisfy its delay declaration (since the first argument is a variable) and then it cannot be selected (and resolved). In this case we say that the query `append(X,Y,Z)` *deadlocks*[1] 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 is clearly a heavy loss, since the equivalence between declarative and operational semantics is one of the strong points of the logic programming paradigm.

Contributions of the Paper In this paper we address the problem of providing a model-theoretic semantics for programs using a dynamic selection rule. In order to do so, we need a "declarative" way of modeling them, and for this we restrict our attention to *input-consuming programs* [12]. 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, when the program `APPEND` reported above is used for concatenating two lists, we assume that the first two arguments fill in *input* positions while the third argument fills in an *output* position.

In [13] we showed that, assuming the above moding, for queries of the form `append`(s, t, X) (with X being a variable disjoint from s and t), the delay declaration `delay append(Ls,_,_) until nonvar(Ls).` 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 step the resolved atom always satisfies the given delay declaration, thus it would have been selectable by any mechanism implementing delay declarations. This reasoning applies for a large class of queries and is actually not a coincidence: As shown by Smaus in [14] for `block` declarations and further discussed by the authors in [13], one can argue that in most situations delay declarations are employed precisely for ensuring that the derivation is input-consuming. Thus, we are interested in providing a model-theoretic semantics for input-consuming programs. Clearly, many of the difficulties one has in doing this for programs with delay declarations apply to input-consuming programs as well. Intuitively speaking, the crucial problem originates in the fact that input-consuming derivations may deadlock[2], i.e., reach a stadium in which no atom is resolvable (e.g., the query `append(X,Y,Z)`). Because of this, a declarative semantics for logic programs is generally not correct for input-consuming programs.

In this paper we show that, if a program is well- and nicely-moded, then, for nicely-moded queries the operational semantics provided by the input-consuming

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

[2]As we will discuss later, this notion of deadlock differs, in some way, from the usual one, which is given in the case of programs employing delay declarations.

resolution rule is correct and complete wrt. the $\mathcal{S}$-semantics [15] for logic programs. The $\mathcal{S}$-semantics is a denotational semantics which – for programs without delay declarations – correctly denotes the set of the computed answer substitutions associated with 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 and can be also viewed as a model-theoretic semantics, 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.

This paper is organized as follows. The next section contains the preliminary notations and definitions. In Section 3 we discuss the relation between input-consuming derivations and programs using delay declarations. Section 4 contains the main results and some examples. In Section 5 we show how the semantics for input-consuming derivations we present can be used for reasoning about deadlock of programs using delay declarations. Finally, Section 6 concludes the paper. Some proofs are reported in the Appendix.

A preliminary, shorter version of this paper has appeared in [16].

2 Preliminaries

In this paper we consider definite logic programs and assume the reader is familiar with the terminology and the basic results of the semantics of definite logic programs (see, for instance, [17, 11, 18]). Here we adopt the notation of [11] in the fact that we use boldface characters to denote sequences of objects; therefore $\mathbf{t}$ denotes a sequence of terms while $\mathbf{B}$ is a query (notice that – following [11] – queries are simply conjunctions of atoms, possibly empty). We denote atoms by $A, B, H, \ldots$, queries by $Q, \mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$, clauses by $c, d, \ldots$, and programs by P. The empty query is denoted by $\square$.

2.1 Substitutions and Derivations

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* 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 $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ we say that $\{x_1, \ldots, x_n\}$ is its *domain* (denoted by $Dom(\sigma)$) and that $Var(\{t_1, \ldots, t_n\})$ is its *range* (denoted by $Ran(\sigma)$) Notice that $Var(\sigma) = Dom(\sigma) \cup Ran(\sigma)$. If $\{t_1, \ldots, t_n\}$ consists of variables then σ is called a *pure variable substitution*. If, in addition, $t_1, \ldots, t_n$ 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 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 $t\theta = t'\theta$. A *most general unifier* (*mgu*, in short) of t and t' is unique, up to renaming; we denote it by $mgu(t, t')$. 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 $Q :=$ $\mathbf{A}, B, \mathbf{C}$ and a clause $c := H \leftarrow \mathbf{B}$ (renamed apart wrt. Q) yield the resolvent $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$, provided that $\theta = mgu(B, H)$. A *derivation step* is denoted by

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

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 a *SLD-derivation of* $P \cup \{Q_0\}$ provided that for every step an appropriate renaming of the input clause is used, so that to satisfy the standardization apart condition: 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 or we are not interested in the specific input clauses or mgu's used, then we drop the reference to them. A 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.

Definition 2.1 (Similar Derivations) We say that two SLD-derivations δ and δ' are called *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.

The following results hold.

Lemma 2.2 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 2.3 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.

2.2 Input-Consuming Derivations

A*mode* is a function that labels as *input* or *output* the positions of each predicate in order to indicate how the arguments of a predicate should be used.

Definition 2.4 (Mode) Consider an n-ary predicate symbol p. By a *mode* for p we mean a function m_p from $\{1, \dots, n\}$ to $\{In, Out\}$.

If $m_p(i) = In$ (resp. Out), we say that i is an *input* (resp. *output*) *position of* p (with respect to m_p). We assume that each predicate symbol has a unique mode associated to it; multiple modes may be obtained by simply renaming the predicates. If Q is a query, we denote by $In(Q)$ (resp. $Out(Q)$) the 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 with $\mathbf{s}$ the sequence of terms filling in the input positions of p and with $\mathbf{t}$ the sequence of terms filling in the output positions of p.

The notion of input-consuming derivation was introduced by Smaus in [12] and is defined as follows.

Definition 2.5 (Input-Consuming)

- A derivation step $\mathbf{A}, B, \mathbf{C} \overset{\theta}{\Longrightarrow}_c (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ 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.

Example 2.6 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 relation symbol is

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

Consider now the query `reverse([X1,X2],Zs)`. The following derivation starting in `reverse([X1,X2],Zs)` is input-consuming (as usual, $\Box$ denotes the empty query).

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

The following result states that also when considering input-consuming derivations, it is not restrictive to assume that all mgu's used in a derivation are relevant.

Lemma 2.7 Let $p(\mathbf{s},\mathbf{t})$ and $p(\mathbf{u},\mathbf{v})$ be two atoms. If there exists an mgu θ of $p(\mathbf{s},\mathbf{t})$ and $p(\mathbf{u},\mathbf{v})$ such that $\mathbf{s}\theta = \mathbf{s}$ then there exists a *relevant* mgu ϑ of $p(\mathbf{s},\mathbf{t})$ and $p(\mathbf{u},\mathbf{v})$ such that $\mathbf{s}\vartheta = \mathbf{s}$.

Proof. Since $p(\mathbf{s},\mathbf{t})$ and $p(\mathbf{u},\mathbf{v})$ are unifiable, there exists a relevant mgu θ_{rel} of them (see [11], Theorem 2.16). Now, θ_{rel} is a renaming of θ. Thus $\mathbf{s}\theta_{rel}$ is a variant of $\mathbf{s}$. Then there exists a renaming ρ such that $Dom(\rho) \subseteq Var(\mathbf{s},\mathbf{t},\mathbf{u},\mathbf{v})$ and $\mathbf{s}\theta_{rel}\rho = \mathbf{s}$. Now, take $\vartheta = \theta_{rel}\rho$. ■

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

2.3 The $\mathcal{S}$-semantics

The aim of the $\mathcal{S}$-semantics approach (see [19]) 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:

$$\begin{array}{lll} \mathcal{S}(P) = \{\ p(x_1,\dots,x_n)\theta & \mid & x_1,\dots,x_n \text{ are distinct variables and} \\ & & p(x_1,\dots,x_n) \xrightarrow{\theta}_P \Box \text{ is a SLD-derivation}\}. \end{array}$$

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:

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

- I is called an $\mathcal{S}$-*model* of P if $T_P^{\mathcal{S}}(I) \subseteq I$.

Falaschi et al. [15] 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 $P = T_P^{\mathcal{S}} \uparrow \omega$.

[3] Here and in the sequel, when we write "$\mathbf{C} \in I$, renamed apart wrt. some expression e", we naturally mean that I contains a set of atoms $C'_1,\dots,C'_n$, and that $\mathbf{C}$ is a renaming of $C'_1,\dots,C'_n$ such that $\mathbf{C}$ shares no variable with e and that two distinct atoms of $\mathbf{C}$ share no variables with each other.

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 2.8 [15] Let P be a program, $\mathbf{A}$ be a query, and θ be a substitution. The following statements are equivalent.

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

Example 2.9 Let us see this semantics applied to the programs `APPEND` and `REVERSE` so far encountered.

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

2.4 Well- and Nicely-Moded Programs

Clearly, also 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 consider the extension of `APPEND` obtained by adding the following clause to it

```
q ← append(X,Y,Z).
```

we have that `q` belongs to the semantics but the query `q` will not succeed (since the atom `append(X,Y,Z)` is not resolvable via an input-consuming derivation step). 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 and of nicely-moded programs.

The concept of well-moded program is due to Dembinski and Maluszynski [20].

Definition 2.10 (Well-Moded)

- A query $p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is *well-moded* if for all $i \in [1, n]$

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

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

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

- 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 [21].

Definition 2.11 (Nicely-Moded)

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

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

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

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

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

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

Example 2.12 Programs `APPEND` and `REVERSE` are both well- and nicely-moded. Furthermore, consider the following program `PALINDROME`

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

together with the program REVERSE with the modes reverse(In,Out) of Example 2.6. This program is well-moded but not nicely-moded (since Xs occurs both in an input and in an output position of the same body atom). However, since the program REVERSE is used here for checking whether a list is a palindrome, its natural modes are reverse(In,In) and reverse_acc(In,In,In). With these modes, the program PALINDROME is both well-moded and nicely-moded.

3 Input-Consuming vs. Delay Declarations

There is a main difference between the concept of delay declaration and the one of input-consuming derivation: While in the first case only the atom selectability is controlled, in the second one both the atom and the clause selectability are affected. In fact, in presence of delay declarations, if an atom is selectable then it can be resolved with respect to any program clause (provided it unifies with its head); on the contrary, in an input-consuming derivation, if an atom is selectable then it is input-consuming resolvable with respect to some, but not necessarily all, program clauses, i.e, only a restricted class of clauses can be used for resolution.

Example 3.1 Consider the following piece of program where the predicate generate generates a list formed by the constant a and variables, arbitrarily mixed.

```
generate-select ← generate(Xs), select(Xs).
generate([a|Xs]) ← generate(Xs).
generate([_|Xs]) ← generate(Xs).
generate([ ]).

mode generate(Out).
mode select(In).
```

Then, suppose we would like to define the predicate select, used in the body of generate-select, with the following behavior: It non-deterministically chooses to call the program first-choice or the program second-choice, if the generated list starts with the constant a; it deterministically calls the program second-choice, if the first element of the list is a variable, and fails on the empty list.

We can obtain this behavior with an input-consuming program, defined as follows.

```
select([a|Xs]) ← first-choice.
select([X|Xs]) ← second-choice.
```

In fact, for input-consuming derivations the first clause is selected only if the first element of the list is not a variable. Note that we cannot obtain such a behavior by means of delay declarations.

Also the concept of *deadlock* has to be understood in two different ways. For programs using delay declarations a deadlock situation occurs when no atom in a query satisfies the delay declarations (i.e., no atom is selectable), while for input-consuming derivations a deadlock occurs when no atom in a query is resolvable via an input-consuming derivation step and the derivation does not fail, i.e., there is some atom in the query which unifies with a clause head but the unification is not input-consuming.

Example 3.2 Consider again the predicate `select` defined above.

- The query `select`(X) unifies with both clause heads but it is not resolvable via an input-consuming derivation step. This is a deadlock situation for input-consuming programs.
- Consider now the delay declaration

  ```
  delay select([X|_]) until nonvar(X).
  ```

  With this delay declaration the query `select`$([X|Xs])$ is not selectable and so it immediately deadlocks. However, there is an input-consuming derivation obtained by unifying the query with the head of the second clause.

In spite of these differences, we believe that in the majority of practical situations there is a strict relation between programs using delay declarations and input-consuming derivations.

Example 3.3 Consider again the program `REVERSE` of Example 2.6 for reversing a list

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

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

with modes

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

A natural delay declaration for this program is

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

One can easily get convinced 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. Furthermore, for the same class of queries it holds that in any input-consuming derivation the selected atom satisfies the above delay declarations.

The relation between programs using delay declarations and input-consuming derivations is studied by Smaus in his PhD thesis [14]. More precisely, Smaus proves a result that relates `block` declarations and input-consuming derivations. A `block` declaration is a special case of delay declaration and it is used to declare that certain arguments of an atom must be *non-variable* when the atom is selected for resolution. In Chapter 7 of [14], Smaus shows that `block` declarations can be used to ensure that derivations are input-consuming.

In force of this result and of practical experience, in the introduction we have stated the claim that in most "usual" moded programs using them, delay declarations are employed precisely for ensuring the input-consumedness of the derivations. As we have already mentioned, this thesis is also substantiated by the fact that the concept of input-consuming resolution is very similar to the selection mechanism employed in moded flat GHC [7], and by the arguments in [14]. Nevertheless, since this claim is of crucial importance for the relevance of our results, now that we have formalized the notion of input-consuming derivation we can add another argument sustaining it.

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, but we do not address this issue in this paper). 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, which are the ones carrying the information.

Now, take an atom $p(\mathbf{s}, \mathbf{t})$ that it is resolvable with a clause c by means of an input-consuming derivation step. Then, for every instance $\mathbf{s}'$ of $\mathbf{s}$, we have that the atom $p(\mathbf{s}', \mathbf{t})$ is also resolvable with c by means of an input-consuming derivation step. In other words, no further instantiation of the input positions of $p(\mathbf{s}, \mathbf{t})$ can rule out c as a possible clause for resolving it. Thus c must be one of the "appropriate" clauses for resolving $p(\mathbf{s}, \mathbf{t})$ and we can say that $p(\mathbf{s}, \mathbf{t})$ is "sufficiently instantiated' in its input positions to be resolved with c. On the other hand, following the same reasoning, if $p(\mathbf{s}, \mathbf{t})$ is resolvable with c but not via an input-consuming derivation step, then there exists an instance $\mathbf{s}'$ of $\mathbf{s}$, such that $p(\mathbf{s}', \mathbf{t})$ is not resolvable with c. In this case we can say that $p(\mathbf{s}, \mathbf{t})$ is not instantiated enough to know whether c is one of the "appropriate" clauses for resolving it.

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.

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 [11] will be used.

Definition 4.1 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 distinct variables, and $(Var(t) - \{x_1, \ldots, x_n\}) \cap \{y_1, \ldots, y_n\} = \emptyset$. (Note that $\{x_1, \ldots, x_n, y_1, \ldots, y_n\}$ is a set of distinct variables and 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.

Notation 4.2 Let $Q := p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$. We define

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

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 $VIn^*(Q) = \emptyset$.

Now we can state the following technical result concerning well-moded programs. The proof is in the Appendix.

Lemma 4.3 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 first result concerning well-moded programs. Basically, it states the *correctness* of the $\mathcal{S}$-semantics for well-moded, input-consuming programs. It can be regarded as "one half" of the main result we are going to propose.

Proposition 4.4 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 a substitution such that the hypotheses are satisfied. By Theorem 2.8, 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 [11], there exists a successful LD-derivation δ' of $P \cup \{A\}$ with c.a.s. ϑ'. From the hypotheses, it follows that $\vartheta'_{|In(A)}$ is a renaming for A. By Lemma 4.3, there exists an input-consuming successful derivation $\delta := A \xrightarrow{\vartheta}_P \Box$ of $P \cup \{A\}$ such that δ and δ' are similar. The assertion follows from Lemma 2.3. ■

Unfortunately, the reverse implication of Proposition 4.4 does not hold in general. However, it holds for a particular class of programs and queries: the nicely-moded ones. To prove that, we need to recall some properties of nicely-moded programs from [13].

Lemma 4.5 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 4.4.

Proposition 4.6 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 2.8, 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, it follows by Lemma 4.5 that $\vartheta_{|In(A)}$ is a renaming for A. Hence (i) follows by the hypotheses and (ii). ■

We now put together the pieces provided in the previous sections and extend the results to arbitrary (non-atomic) queries. The following simple result allows us to generalize results concerning atomic queries.

Lemma 4.7 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. Let *new* be a predicate symbol not occurring in P. Let $\mathbf{x}$ be a sequence of distinct variables containing precisely $VIn^*(Q)$ and $\mathbf{y}$ be a sequence of distinct variables containing precisely $Var(Out(Q))$. Consider now the atom $A := new(\mathbf{x}, \mathbf{y})$, the clause $c := A \leftarrow Q$, and the program

$$P' = P \cup \{c\}.$$

By construction, $In(A) = VIn^*(Q)$ and $Var(Out(A)) = Var(Out(Q))$. It is straightforward to check that, by the nicely-modedness of Q, both A and c are nicely-moded.

Moreover, by construction, each variable of c occurring in an input position of a body atom but not occurring in an output position of an earlier atom belongs to $VIn^*(Q)$, i.e., occurs in an input position of the head, and each variable occurring in an output position of a body atom also occurs in an output position of the head. Thus, c is well-moded. The thesis follows easily. ■

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

(i) There exists an input-consuming derivation $\mathbf{A} \xrightarrow{\vartheta}_P \Box$, such that $\mathbf{A}\vartheta \approx \mathbf{A}\theta$.

(ii) There exists $\mathbf{A}' \in \mathcal{S}(P)$ (renamed apart wrt. $\mathbf{A}$), and $\sigma = mgu(\mathbf{A}, \mathbf{A}')$ such that

(a) $\sigma_{| VIn^*(\mathbf{A})}$ is a renaming for $\mathbf{A}$,

(b) $\mathbf{A}\sigma \approx \mathbf{A}\theta$.

Proof. It follows immediately from Propositions 4.4, 4.6 and Lemma 4.7. ■

Note that in case of an atomic query $\mathbf{A} := A$, we might substitute condition (a) above with the somewhat more attractive condition

(a$'$) $In(A)\sigma \approx In(A)$.

Note also that, given a well- and nicely-moded program P, the above Theorem 4.8 allows us to identify the subset $\mathcal{S}_{ic}(P)$ of $\mathcal{S}(P)$, defined by

$$\begin{array}{lll} \mathcal{S}_{ic}(P) = \{ \; A' \in \mathcal{S}(P) & | & \exists A \text{ nicely-moded an renamed apart wrt. } A' \\ & & \exists \sigma = mgu(A, A') \\ & & In(A)\sigma \approx In(A)\}, \end{array}$$

which fully characterizes the behavior of P on nicely-moded queries. Therefore, given two well- and nicely-moded programs P_1 and P_2, they compute the same answer substitutions for any nicely-moded query iff $\mathcal{S}_{ic}(P_1) = \mathcal{S}_{ic}(P_2)$.

Let us immediately see some examples. The first example demonstrates that the syntactic restrictions used in Theorem 4.8 are necessary.

Example 4.9 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).

mode p(In,Out).
mode equal_lists(In,Out).
mode list_of_zeroes(Out).
```

Note that the first clause is not nicely-moded since the sequence of terms filling in the output positions of the body atoms is not linear. The $\mathcal{S}$-semantics of this program restricted to the predicate p contains all and only all the atoms of the form p(*list*, *list*) where *list* is a list containing only zeroes. Consider now the atomic query A := p([X1], Y). There exists an input-consuming derivation starting in it, namely,

$$\begin{array}{llll} \texttt{p([X1],Y)} & \xRightarrow{\theta_1} & \texttt{equal\_lists([X1],Y), list\_of\_zeroes(Y)} & \xRightarrow{\theta_2} \\ & \xRightarrow{\theta_2} & \texttt{equal\_lists([ ],T'), list\_of\_zeroes([X1|T'])} & \xRightarrow{\theta_3} \\ & \xRightarrow{\theta_3} & \texttt{list\_of\_zeroes([X}_1\texttt{])} \xRightarrow{\theta_4} \texttt{list\_of\_zeroes([ ])} & \xRightarrow{\theta_5} \quad \square \end{array}$$

with $\theta_1 = \{\texttt{X/[X1]}\}$, $\theta_2 = \{\texttt{H/X1, T/[ ], Y/[X1|T']}\}$, $\theta_3 = \{\texttt{T'/[ ]}\}$, $\theta_4 = \{\texttt{X1/0, T1/[ ]}\}$, $\theta_5 = \epsilon$. The computed answer substitution is $\theta = \{\texttt{X1/0, Y/[0]}\}$. Nevertheless, there does not exist any atom $A' \in \mathcal{S}(P)$ (renamed apart wrt. A) such that A and A' unify with a most general unifier σ such that $\sigma_{|In(A)}$ is a renaming for A. This is clear from the fact that the atoms belonging to $\mathcal{S}(P)$ are all ground.

This shows that if the program is well-moded but not nicely-moded then the implication (i) $\Rightarrow$ (ii) in Theorem 4.8 does not hold.

Consider now 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([H|T]) ← list(T).

mode p(In).
mode equal_lists(In, In).
mode list(Out)
```

This program is nicely-moded, but not well-moded: In the last clause the variable `H` occurring in the output position of the head does occur neither in an output position of the body nor in an input position of the head. The $\mathcal{S}$-semantics of this program restricted to the predicate `p` contains all and only all the atoms of the form `p`(*list*) where *list* is any list containing only distinct variables. It is easy to see that there does not exist any input-consuming derivation for a query `p`(*list*) with *list* being a ground list. Indeed, consider the execution of the atom $A = \mathtt{p([0])}$. A call `equal_lists([0],[H])` is reached. However, it does not exist any input-consuming derivation for the atomic query `equal_lists([0],[H])` with its arguments filling in both the input positions. Nevertheless, there exists an atom $A' \in \mathcal{S}(P)$ (renamed apart wrt. A), e.g., $A' = \mathtt{p([X1])}$, such that A and A' unify with a most general unifier σ such that $\sigma_{|\mathit{In}(A)}$ is a renaming for A (obvious, since A is ground).

This shows that if the program is nicely-moded but not well-moded then the implication (ii) $\Rightarrow$ (i) in Theorem 4.8 does not hold.

The next example reports two applications of Theorem 4.8.

Example 4.10 Consider the program `APPEND` of the introduction with the moding `append(In,In,Out)`.

- `append([X,b],Y,Z)` has an input-consuming successful derivation.

  In particular, it has an input-consuming derivation with c.a.s. $\{\mathtt{Z/[X,b|Y]}\}$. This can be concluded by just looking at $\mathcal{S}(\mathtt{APPEND})$, from the fact that $A = \mathtt{append([X1,X2],X3,[X1,X2|X3])} \in \mathcal{S}(P)$.
  Note that `append([X,b],Y,Z)` is - in its input position - an instance of A.

- `append(Y,[X,b],Z)` has no input-consuming successful derivations.

  This is because there is no $A \in \mathcal{S}(P)$ such that $\mathtt{append(Y,[X,b],Z)}$ is an instance of A in the input position. This actually implies that – in presence of delay declarations – `append(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.

Note that the results we have provided hold also in the case that programs are *permutation well- and nicely-moded* and queries are *permutation nicely-moded* [22], that is 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.

5 An Application: Reasoning about Deadlock

In this section we consider again programs employing delay declarations.

An important consequence of Theorem 4.8 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 first 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 if and only if every selected atom satisfies the corresponding delay declaration. As we have already stated in the introduction, we say that a derivation *deadlocks* if its last query contains no selectable atom, i.e., no atom which satisfies the corresponding delay declarations.

Notation 5.1 Let P be a program and $\mathbf{A}$ be a query.

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

Example 5.2 Consider the program `REVERSE` of Example 3.3

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

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

with modes

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

and delay declarations

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

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

Example 5.3 Consider now the program `APPEND` augmented with its delay declaration of the introduction.

- `APPEND` $\cup$ {`append(`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.

- `APPEND` $\cup$ {`append([X,b],Y,Z)`} has an input-consuming successful derivation (see Example 4.10) and is input-consuming complete. Then, we can state that `APPEND` $\cup$ {`append([X,b],Y,Z)`} is not bound to deadlock.

Consider now the nicely-moded query `append(X,Y,Z)`. Since $\mathcal{S}$(`APPEND`) contains instances of it, by Theorem 2.8, `append(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 `append(X,Y,Z)`. Thus by Theorem 4.8, `append(X,Y,Z)` has no input-consuming successful derivations. Therefore, since `APPEND` $\cup$ {`append(Y,X,Z)`} is input-consuming correct, we can state that `append(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* [13], i.e., all its input-consuming derivations starting in a nicely-moded query are finite. In [13] we provided conditions which guaranteed that a program is input terminating; these conditions easily allow one to show that `APPEND` is input-terminating. Because of this, we can conclude that

- `APPEND` $\cup$ {`append(Y,X,Z)`} is bound to deadlock.

By simply formalizing this reasoning, we obtain the following result.

Theorem 5.4 Let P be a well- and nicely-moded program, and $\mathbf{A}$ be nicely-moded query. If

1. $\exists\ \mathbf{B} \in \mathcal{S}(P)$, such that $\mathbf{A}$ unifies with $\mathbf{B}$,
2. $\forall\ \mathbf{B} \in \mathcal{S}(P)$, if $\sigma = mgu(\mathbf{A},\mathbf{B})$ then $\sigma_{|\,VIn^*(\mathbf{A})}$ is not a renaming for $\mathbf{A}$,
3. $P \cup \{\mathbf{A}\}$ is input-consuming-correct,

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

Proof. By 1. and Theorem 2.8, there exists at least one successful SLD-derivation of $P \cup \{\mathbf{A}\}$. By 2. and Theorem 4.8 there is no successful input-consuming derivation of $P \cup \{\mathbf{A}\}$. Thus, by 3., there is no successful SLD-derivation of $P \cup \{\mathbf{A}\}$ which respects the delay declarations. Hence, $\mathbf{A}$ either has an infinite SLD-derivation respecting the delay declarations or it is bound to deadlock.
Moreover, if P is input-terminating then there cannot exist an infinite SLD-derivation respecting the delay declarations for $P \cup \{\mathbf{A}\}$; hence $\mathbf{A}$ must be bound to deadlock. ■

Let us see more examples.

Example 5.5 Let us continue to discuss the program `APPEND` above. Using Theorem 5.4, we can state that

- `APPEND` ∪ {`append(Y,[X,b],Z)`} either has an infinite derivation or it is bound to deadlock.

Since `APPEND` is input terminating [13], we can also say that

- `APPEND` ∪ {`append(Y,[X,b],Z)`} is bound to deadlock.

Example 5.6 Let us now consider program 15.3 from [23]: `QUICKSORT` using a form of difference-lists.

```
%  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(Littles,Ys,[X|Ys1]),          % atom a1
      quicksort_dl(Bigs,Ys1,Zs).                 % atom a2
   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,[],[]).
```

with the modes

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

This program is permutation well- and nicely-moded (it becomes well-moded by permuting atoms `a1` and `a2` in the body of the second clause). When used in combination with dynamic scheduling, the standard delay declarations for it are the following ones:

```
   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 meant to avoid nontermination and to increase efficiency, the last two are needed to avoid runtime errors: in fact comparison predicates have to be called with both arguments ground, otherwise an exception occurs. One can naturally assume that the semantics of the built-ins `>` and `=<` is given by the set of *ground* atoms {`>(a,b)` | `a` larger than `b`} together with {`=<(a,b)` | `a` smaller or equal than `b`}. The fact that this semantics is ground and that both arguments of both predicates are input reflects that these predicates have to be called with ground arguments. Under these assumptions, the $\mathcal{S}$-semantics of the program restricted to the predicates `quicksort` and `quicksort_dl`, contains all and only all the atoms of the form

- `quicksort(`s`,`t`)`, where s is a ground list and t is an ordered permutation of s;
- `quicksort_dl(`s`,`t`,`u`)`, where s is a ground list and t is an ordered permutation of s with u appended to t.

Observe that, if the terms filling in the input positions of an atom are variable disjoint from those filling in the output positions of the same atom, then the input cannot become instantiated as a "side effect" of the instantiation of the output. Hence, we can prove that

- if s and t are variable disjoint terms then `QUICKSORT` $\cup$ {`quicksort`(s,t)} is input-consuming correct;
- if t is variable disjoint from s and u then `QUICKSORT` $\cup$ {`quicksort_dl`(s,t,u)} is input-consuming correct.

By applying Theorem 4.8 it follows that

- the query `quicksort(`s`,`t`)` is not bound to deadlock provided that s is a list of ground terms;
- `quicksort_dl(`s`,`X`,`t`)` is not bound to deadlock provided that s is a list of ground terms and X is a variable disjoint from t.

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 positions are moded as *input*.

```
p(X) ← q(a).
p(a).
q(b).
```

There are no delay declarations with respect to which this program is input-consuming complete. In fact, there are two input-consuming derivations starting in `p(X)`: one fails while the other one deadlocks. This does not correspond to the usual notion of deadlock: an atom cannot simultaneously be selectable and deadlocked.

6 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 number of programs employing dynamic scheduling there exists a declarative semantics which is equivalent to the operational one.

A related work is the one of Apt and Luitjes [5]. The crucial difference between this approach and our is that in [5] 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., `append(X,Y,Z)`). In many cases we capture this behavior by using Theorem 4.8 which can tell us 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 queries) we want to mention that [24, 13] both contain mini-surveys of programs indicating whether they are well- and nicely-moded or not. From them, it appears that most "usual" programs satisfy both definitions.

A Appendix

In this appendix we report the proof of Lemma 4.3. Let us first introduce some preliminaries.

Definition A.1 Let $\theta = \{x_1/y_1, \ldots, x_n/y_n\}$ be a renaming for a term t. We define $\overleftarrow{\theta}$ as the pure variable 1-1 substitution $\{y_1/x_1, \ldots, y_n/x_n\}$.

Observe that:

- The substitution $\{x_1/y_1, \ldots, x_n/y_n, y_1/x_1, \ldots, y_n/x_n\}$ is a renaming.
- If θ is a renaming for a term t then $\overleftarrow{\theta}$ is a renaming for the term $t\theta$.
- $(\theta\overleftarrow{\theta})_{|Dom(\theta)} = \epsilon$.

The following properties hold.

Lemma A.2 Let Q be a query.

(i) If Q is an atomic query then $\mathit{VIn}^*(Q) = \mathit{Var}(\mathit{In}(Q))$.

(ii) For any prefix Q' of Q, $\mathit{VIn}^*(Q') \subseteq \mathit{VIn}^*(Q)$.

(iii) For any substitution θ, $\mathit{VIn}^*(Q\theta) \subseteq \mathit{Var}(\mathit{VIn}^*(Q)\theta)$.

(iv) For any substitution θ, $\mathit{Var}(\mathit{Out}(Q\theta)) = \mathit{Var}(\mathit{Out}(Q)\theta)$.

We can now prove Lemma 4.3.

Lemma 4.3 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 $\theta_{|\mathit{VIn}^*(Q)}$ is a renaming for Q then δ is similar to an input-consuming partial (LD-) derivation.
Proof. We first state the following facts.

Claim 1 Let θ be a substitution, S be a set of variables and t be a term such that $\theta_{|S}$ is a renaming for t. Suppose that $\theta := \theta_1\theta_2$. Then, $\theta_{1|S}$ is a renaming for t.

Claim 2 Let θ be a substitution, S be a set of variables and t be a term such that $\theta_{|S}$ is a renaming for t. Suppose that $\theta := \theta_1\theta_2$. Let $S' = \{x \in S|\ x \notin \mathit{Dom}(\theta_1)\}$. Then, $\theta_{2|\mathit{Ran}(\theta_{1|S}) \cup S'}$ is a renaming for $t\theta_1$.

The proof proceeds by induction on $\mathit{len}(\delta)$.

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

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

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

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

By the Lemma's hypotheses and Claim 1, it follows that $\theta_{1|\,VIn^*(Q)}$ is a renaming for Q. Observe that

$$\begin{array}{lcll} Var(\mathbf{s}) & = & Var(In(p(\mathbf{s},\mathbf{t}))) & \\ & = & VIn^*(p(\mathbf{s},\mathbf{t})) & \text{by Lemma A.2 (i)} \\ & \subseteq & VIn^*(Q) & \text{by Lemma A.2 (ii).} \end{array}$$

By relevance of θ_1, $\theta_{1|\,VIn^*(Q)} = \theta_{1|\mathbf{s}}$. Let $\theta_{1|\mathbf{s}} = \{x_1/y_1,\dots,x_n/y_n\}$. Note that $\{x_1,\dots,x_n,y_1,\dots,y_n\}$ is a set of distinct variables. Consider the renaming $\rho = \{x_1/y_1,\dots,x_n/y_n,y_1/x_1,\dots,y_n/x_n\}$. Since $Var(\rho) \subseteq Var(\theta_1)$, the substitution $\theta_1\rho$ is a relevant mgu of $p(\mathbf{s},\mathbf{t})$ and $p(\mathbf{u},\mathbf{v})$ (see [11], Lemma 2.23).
It is easy to see that $\theta_1\rho = \theta_1(\overleftarrow{\theta_{1|\mathbf{s}}})$. Let $\theta_1' = \theta_1(\overleftarrow{\theta_{1|\mathbf{s}}})$. We have that $\mathbf{s}\theta_1' = \mathbf{s}\theta_1(\overleftarrow{\theta_{1|\mathbf{s}}}) = \mathbf{s}$. Therefore,

(1) $\qquad p(\mathbf{s},\mathbf{t}),\mathbf{C} \overset{\theta_1'}{\Longrightarrow} (\mathbf{B},\mathbf{C})\theta_1'$

is an input-consuming LD-derivation step.

Since $\theta_{1|\mathbf{s}}$ is a renaming for Q and, by standardization apart, $Var(\mathbf{B}) \cap Var(Q) = \emptyset$, we have that $\theta_{1|\mathbf{s}}$ is a renaming for $(\mathbf{B},\mathbf{C})$. Hence, $(\overleftarrow{\theta_1})_{|S}$ where S is the set of variables $\{x|\ x \in Var((\mathbf{B},\mathbf{C})\theta_1)$ and $x \in Ran(\theta_{1|\mathbf{s}})\}$, is a renaming for $(\mathbf{B},\mathbf{C})\theta_1$. Now observe that $(\mathbf{B},\mathbf{C})\theta_1' = (\mathbf{B},\mathbf{C})\theta_1(\overleftarrow{\theta_{1|\mathbf{s}}}) = (\mathbf{B},\mathbf{C})\theta_1(\overleftarrow{\theta_1})_{|S}$.

Therefore, $(\mathbf{B},\mathbf{C})\theta_1'$ is a variant of $(\mathbf{B},\mathbf{C})\theta_1$. By Lemma 2.2, there exists a partial LD-derivation $(\mathbf{B},\mathbf{C})\theta_1' \xrightarrow{\theta_2'} Q''$ similar to $(\mathbf{B},\mathbf{C})\theta_1 \xrightarrow{\theta_2} Q'$. It follows that

(2) $\qquad p(\mathbf{s},\mathbf{t}),\mathbf{C} \overset{\theta_1'}{\Longrightarrow} (\mathbf{B},\mathbf{C})\theta_1' \xrightarrow{\theta_2'} Q''$

is an LD-derivation of $P \cup \{Q\}$ that is similar to δ.

Let $\theta' = \theta_1'\theta_2'$. By Lemma 2.3, $Q\theta$ and $Q\theta'$ are variants of each other.

Consider now the set $S' = VIn^*((\mathbf{B},\mathbf{C})\theta_1')$. We prove that $\theta'_{2|S'}$ is a renaming for $(\mathbf{B},\mathbf{C})\theta_1'$. Let $x \in S'$. There are two cases.

(a) $x \in VIn^*(\mathbf{B}\theta_1')$. By Lemma A.2 (iii), $VIn^*(\mathbf{B}\theta_1') \subseteq Var(VIn^*(\mathbf{B})\theta_1')$. Then, there exists $z \in VIn^*(\mathbf{B})$ such that $x \in Var(z\theta_1')$. By well-modedness of $c := p(\mathbf{u},\mathbf{v}) \leftarrow \mathbf{B}$, $z \in Var(\mathbf{u})$. Since $\mathbf{u}\theta_1' = \mathbf{s}\theta_1'$, we have that there exists $y \in Var(\mathbf{s})$, i.e., $y \in VIn^*(Q)$, such that $x \in Var(y\theta_1')$.

(b) $x \in VIn^*(\mathbf{C}\theta_1')$ and $x \notin Var(Out(\mathbf{B}\theta_1'))$. We distinguish two cases.

(b1) $x \in Var(\mathbf{t}\theta_1')$. From the fact that $\mathbf{t}\theta_1' = \mathbf{v}\theta_1'$, we have that $x \in Var(\mathbf{v}\theta_1')$, i.e., there exists $z \in Var(\mathbf{v})$ such that $x \in Var(z\theta_1')$. Since z occurs in an output

position of the head of $c := p(\mathbf{u}, \mathbf{v}) \leftarrow \mathbf{B}$ and c is well-moded, we have that either $z \in \mathit{Var}(\mathit{Out}(\mathbf{B}))$ or $z \in \mathit{Var}(\mathbf{u})$. Let us distinguish these two cases.

$(b11)$ $z \in \mathit{Var}(\mathit{Out}(\mathbf{B}))$. In this case $x \in \mathit{Var}(z\theta'_1) \subseteq \mathit{Var}(\mathit{Out}(\mathbf{B}\theta'_1))$. However, this contradicts the hypothesis that $x \notin \mathit{Var}(\mathit{Out}(\mathbf{B}\theta'_1))$.

$(b12)$ $z \in \mathit{Var}(\mathbf{u})$. In this case, since $\mathbf{u}\theta'_1 = \mathbf{s}\theta'_1$, we have that $x \in \mathit{Var}(\mathbf{s}\theta'_1)$. Hence, there exists $y \in \mathit{Var}(\mathbf{s})$, i.e., $y \in \mathit{VIn}^*(Q)$, such that $x \in \mathit{Var}(y\theta'_1)$.

$(b2)$ $x \notin \mathit{Var}(\mathbf{t}\theta'_1)$. By Lemma A.2 (iii), $\mathit{VIn}^*(\mathbf{C}\theta'_1) \subseteq \mathit{Var}(\mathit{VIn}^*(\mathbf{C})\theta'_1)$. Thus, there exists $y \in \mathit{VIn}^*(\mathbf{C})$ such that $x \in \mathit{Var}(y\theta'_1)$. Note that $y \notin \mathit{Var}(\mathbf{t})$, otherwise we would have $x \in \mathit{Var}(\mathbf{t}\theta'_1)$ contradicting the hypothesis. Hence, $y \in \mathit{VIn}^*(Q)$.

We have proved that

(3) for all $x \in S'$, there exists $y \in \mathit{VIn}^*(Q)$ such that $x \in \mathit{Var}(y\theta'_1)$.

From the fact that $Q\theta$ and $Q\theta'$ are variants of each other and $\theta_{|\mathit{VIn}^*(Q)}$ is a renaming for Q, it follows that also $\theta'_{|\mathit{VIn}^*(Q)}$ is a renaming for Q.
Let $S'' = \{x \in \mathit{VIn}^*(Q) |\ x \notin \mathit{Dom}(\theta'_1)\}$. By Claim 2, $\theta'_{2|\mathit{Ran}(\theta'_{1|\mathit{VIn}^*(Q)}) \cup S''}$ is a renaming for $Q\theta'_1$. By (3), $S' \subseteq \mathit{Ran}(\theta'_{1|\mathit{VIn}^*(Q)}) \cup S''$. Hence, by standardization apart, $\theta'_{2|S'}$ is a renaming for $(\mathbf{B}, \mathbf{C})\theta'_1$. By the induction hypothesis, there exists a partial LD-derivation

(4) $\qquad (\mathbf{B}, \mathbf{C})\theta'_1 \xrightarrow{\theta''_2} Q'''$

which is similar to $(\mathbf{B}, \mathbf{C})\theta'_1 \xrightarrow{\theta'_2} Q''$ and it is input-consuming.

Hence, by (1), (2) and (4),

$$\delta' := p(\mathbf{s}, \mathbf{t}), \mathbf{C} \overset{\theta'_1}{\Longrightarrow} (\mathbf{B}, \mathbf{C})\theta'_1 \xrightarrow{\theta''_2} Q'''$$

is an input-consuming partial LD-derivation of $P \cup \{Q\}$ such that δ and δ' are similar. ∎

References

[1] R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

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

[3] L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, Department of Computer Science, University of Melbourne, 1992.

[4] L. Naish. Parallelizing NU-Prolog. In K. A. Bowen and R. A. Kowalski, editors, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, pages 1546–1564, Seattle, Washington, August 1988.

[5] 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 936, Springer-Verlag, Berlin, pages 66–90, 1995.

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

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

[8] M. G. Wallace, S. Novello and J. Schimpf. ECLiPSe : A platform for constraint logic programming. *ICL Systems Journal*, 12 (1), 1997.

[9] Intelligent Systems Laboratory, Swedish Institute of Computer Science, PO Box 1263, S-164 29 Kista, Sweden. *SICStus Prolog User's Manual*, 1997. `http://www.sics.se/isl/sicstus/sicstus_toc.html`.

[10] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[11] K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[12] J. G. Smaus. Proving termination of input-consuming logic programs. In D. De Schreye, editor, *16th International Conference on Logic Programming*, Las Cruces, New Mexico, USA, The MIT Press, pages 335–349, 1999.

[13] 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. Also available on CoRR: http://arXiv.org/abs/cs/0101022.

[14] J.-G. Smaus. *Modes and Types in Logic Programming.* PhD thesis, University of Kent at Canterbury, October 1999. Draft available from `www.cs.ukc.ac.uk/people/staff/jgs5/thesis.ps`.

[15] 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.

[16] A. Bossi, S. Etalle, and S. Rossi. Semantics of input-consuming programs. In J. Lloyd et. al., editor, *First International Conf. on Computational Logic (CL2000)*, Lecture Notes in Artificial Intelligence 1861, pages 194–208, Springer-Verlag, Berlin, 2000.

[17] 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.

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

[19] 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.

[20] 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, IEEE-CS, Boston, 1985.

[21] 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.

[22] 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*, Lecture Notes in Computer Science 1490. Springer-Verlag, Berlin, pages 73–88, 1998.

[23] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

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