Declarative Semantics of Input Consuming Logic Programs

Annalisa Bossi[1], Nicoletta Cocco[1], Sandro Etalle[2,3], and Sabina Rossi[1]

[1] Università di Venezia, {bossi,cocco,srossi}@dsi.unive.it
[2] University of Twente s.etalle@utwente.nl
[3] CWI, Amsterdam,

Abstract. Most logic programming languages actually provide some kind of *dynamic scheduling* to increase the expressive power and to control execution. *Input consuming* derivations have been introduced to describe dynamic scheduling while abstracting from the technical details. In this paper we review and compare the different proposals given for denotational semantics of programs with input consuming derivations. We also show how they can be applied to termination analysis.

1 Introduction

1.1 Dynamic Scheduling in Logic Programming

In logic programming the *selection rule* determines which atom in a query is selected at each derivation step. The standard selection rule is the left-to-right rule of Prolog, simple to implement, but which can cause problems both with termination and with negation when selected atoms are not fully instantiated. Moreover there are situations, e.g., in the context of parallel executions or the test-and-generate paradigm, that require a more flexible control mechanism, where the selectable atoms are determined at runtime.

Most logic programming languages actually provide some kind of *dynamic scheduling* in order to increase the expressive power and to control execution. In practical systems, dynamic selection rules are implemented by means of constructs such as *delay declarations* and *block declarations*. Delay declarations, advocated by van Emden and de Lucena [46], were introduced explicitly in logic programming by Naish [37, 34]. Delay declarations associate conditions to atoms, indicating when their evaluation can proceed. Such conditions are based on instantiation. Typical delay declarations are `ground(X)` or `nonvar(X)` which specify that the associated atom can be selected for evaluation when its argument `X` is respectively a ground term or a non-variable term. Delay declarations can be also conjoined or disjoined to allow more complex control.

Gödel [26] and ECLiPSe [27] use *delay declarations*, while SICStus Prolog [28] employs *block declarations* (which are a special kind of delay declarations).

Also in concurrent logic languages, such as GHC [43], programs are augmented with *guards* in order to control the selection of atoms dynamically. For example Moded Flat GHC [45] use conditions based on modes and instantiation constraints imposed on individual clauses.

To see how dynamic scheduling can be controlled by delay declarations, consider the following programs APPEND and IN_ORDER:

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

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

together with the query

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

where `read_tree` and `write_list` are defined elsewhere. If `read_tree` cannot read the whole tree at once – say, it receives the input from a stream – it would be nice to be able to run the "processes" `in_order` and `write_list` on the available input. This can be done properly if one uses a dynamic selection rule. Prolog's rule would call `in_order` only after `read_tree` has finished, while other fixed rules would immediately diverge and/or have an unwanted behavior. For instance, the fixed rule that selects always the second atom in a clause body, and that selects the first one only when the body contains only one atom can lead to nontermination, as the query `in_order(Tree,List)` can easily diverge. In the above program, in order to avoid nontermination one can declare that predicates `in_order`, `append` and `write_list` can be selected only if their first argument is not just a variable. Formally,

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

These declarations prevent `in_order`, `append` and `write_list` from being selected "too early", i.e., when their arguments are not "sufficiently instantiated". Note that instead of having interleaving "processes", one can also select several atoms in *parallel*, as long as the delay declarations are respected. This approach to parallelism has been first proposed by Naish [36] and – as observed by Apt and Luitjes [5] – "has an important advantage over the ones proposed in the literature in that it allows us to parallelize programs written in a large subset of Prolog by merely adding to them delay declarations, so *without modifying* the original program".

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

1.2 Semantics of Logic Programs with Dynamic Scheduling

By introducing dynamic scheduling we obtain more powerful and flexible programs but we are faced with the problem of finding new techniques for ensuring correctness and termination of such programs and more generally for analyzing them. The standard semantics and properties are no longer valid when an atom can be delayed under some condition, moreover it is not easy to extend such semantics also because of the possibility of floundering when no atom in the goal can be selected. Hence it is not surprizing that not so many proposals have been given for a semantics for logic programs with dynamic scheduling despite of their practical importance.

The first proposal of an *operational semantics for dynamic scheduling* in the form of coroutining was given by Naish [35]. He defined *SLDF resolution*, which is a straightforward generalization of SLD resolution, where execution of atoms may be suspended indefinitely. He also considered termination of such programs and observed that, if the set of callable atoms is closed under instantiation, termination properties are simplified. Moreover Naish stressed the importance of mode information for reasoning about termination of such programs. An operational semantics for constraint logic programs (CLP) with dynamic scheduling have been given also by Debray *et al.* [19].

Falaschi *et al.* [24, 33, 23] have defined *a denotational semantics for CLP programs with dynamic scheduling* where the semantics of a query is given by a set of closure operators (each operator corresponds to a sequence of rule choices). They start from an operational semantics for constraint logic programs with dynamic scheduling given in terms of derivations from the goals, which is similar to the one in [19] and in [32]. Then they give a semantics in terms of and-trees, which capture the structure of a derivation in a compositional way. An and-tree can be seen as a function mapping an initial constraint to its answer. The denotation of a sequence of atoms is then a set of closure operators, corresponding to the and-trees which have this sequence as root. Their denotational semantics is the analogue of the bottom-up $\mathcal{S}$-semantics [13] for usual logic programs, where atoms are mapped to their set of answers.

Such a denotational semantics can be used as a basis for the *analysis of logic programs with dynamic scheduling*, since closure operators can be abstracted by descriptions which capture their behaviour. This idea was followed by Marriott *et al.* in [32] where a framework for global dataflow analysis for logic programming languages with dynamic scheduling is developed. Its main use is to give information on calling patterns. In [17] the analysis is further improved both in precision and in efficiency. From such proposals also optimization techniques for logic programs with dynamic scheduling have been derived, such as in [38].

A very elegant definition of *an algebraic and logical semantics for constraint logic languages with dynamic scheduling* have been given by Marriott in [31]. It corresponds to an operational semantics based on the one given by Naish in [35] generalized to arbitrary constraints. Delayed atoms are considered as constraints, then the soundness and completeness results for success and finite failure for CLP are extended to CLP with dynamic scheduling. The completeness result for finite failure is necessarily weaker.

In spite of these proposals some problems remain. Dynamic scheduling is often introduced to ensure the termination of the program, preventing possible diverging derivations. Nevertheless, while for pure Prolog programs (i.e., logic programs employing the fixed leftmost selection rule) there exist results characterizing when a program is terminating such as in [7, 18, 14] no such a characterization was derived for programs with dynamic scheduling from these semantics.

1.3 Semantics of Input Consuming Derivations

In order to provide a characterization of dynamic scheduling that is reasonably abstract and amenable to termination analysis, Smaus [40] introduced *input consuming derivations*. The definition of input consuming program relies on the concept of *mode*. A *moded program* is a program in which each atom's arguments are partitioned into *input* and *output* ones. Output arguments are those which can be produced by the computation process, while input arguments should be only consumed. Roughly speaking, in an input consuming program only atoms whose input arguments are not instantiated through the unification step are allowed to be selected.

We believe that – in many cases – the adoption of "natural" delay declarations is equivalent to considering only input consuming derivations [11]. This is the case, for instance, of the programs mentioned in the example above together with their natural mode where the first position of `in_order` is considered in input, while the second one is in output. In fact under normal circumstances, the adoption of the stated delay declarations enforces nothing but a restriction to input consuming derivations. Moreover also other control mechanisms, such as the one in Moded Flat GHC, are similar to requiring an input consuming derivation step: the resolution of an atom with a definition must not instantiate the input arguments of the resolved atom.

Input consuming programs allow for simpler definitions of denotational semantics and have nice properties regarding termination. Henceforth they seem to be a resonable and safe approximation to programs with general dynamic scheduling. In this paper we review and compare the different proposals given for denotational semantics of programs with input consuming derivations. We also show how they can be applied to termination analysis.

1.4 Structure of the Paper

The paper is organized as follows. Section 2 contains some preliminary notations and definitions including input consuming programs. Section 3 introduces a first denotational semantics capturing computed answer substitutions of successful derivations. This semantics applies to well and nicely moded input consuming programs. In Section 4 a second denotational semantics for simply moded input consuming programs is presented which is able to model also intermediate results of partial derivations. Section 5 shows how these semantics have been used to characterize termination properties of input consuming programs. Section 6 concludes the paper.

2 Preliminaries

The reader is assumed to be familiar with the terminology and the basic results of logic programs and their semantics [1, 2, 29]. In this Section we introduce few notions that will be used in the sequel.

2.1 Terms and Substitutions

Let $\mathcal{T}$ be the set of terms built on a finite set of *data constructors* $\mathcal{C}$ and a denumerable set of *variable symbols* $\mathcal{V}$. For any syntactic object o, we denote by $Var(o)$ the set of variables occurring in o. A syntactic object is linear if every variable occurs in it at most once. A *substitution* θ is a mapping from $\mathcal{V}$ to $\mathcal{T}$. Given a *substitution* $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$, we say that $\{x_1, \ldots, x_n\}$ is its *domain* (denoted by $Dom(\sigma)$), and $Var(\{t_1, \ldots, t_n\})$ is its *range* (denoted by $Ran(\sigma)$). Note that $Var(\sigma) = Dom(\sigma) \cup Ran(\sigma)$. We denote by ϵ the empty substitution: $Dom(\epsilon) = Ran(\epsilon) = \emptyset$. Given a *substitution* σ and a syntactic object E, we denote by $\sigma_{|E}$ the restriction of σ to the variables in $Var(E)$, i.e., $\sigma_{|E}(x) = \sigma(x)$ if $x \in Var(E)$, otherwise $\sigma_{|E}(x) = x$. If $t_1, \ldots, t_n$ 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, i.e., $\theta\sigma(x) = \sigma(\theta(x))$. The result of the application of a substitution θ to a term t is said an *instance* of t and it is denoted by $t\theta$. We say that t is a *variant* of t', written $t \approx t'$, if t and t' are instances of each other. In this case there exists a renaming θ such that $t' = t\theta$. A substitution θ is a *unifier* of terms t and t' if $t\theta = t'\theta$. We denote by $mgu(t, t')$ any *most general unifier* (*mgu*, in short) of t and t'.

2.2 Programs and Derivations

Let $\mathcal{P}$ be a finite set of *predicate symbols*. An *atom* is an object of the form $p(t_1, \ldots, t_n)$ where $p \in \mathcal{P}$ is an n-ary predicate symbol and $t_1, \ldots, t_n \in \mathcal{T}$. Given an atom A, we denote by $Rel(A)$ the predicate symbol of A. A *query* is a finite, possibly empty, sequence of atoms $A_1, \ldots, A_m$. The empty query is denoted by $\square$. Following the convention adopted in [2], we use bold characters to denote sequences of objects: so, for instance, $\mathbf{t}$ denotes a sequence of terms, while $\mathbf{B}$ is a query (i.e., a possibly empty sequence of atoms). A *(definite) clause* is a formula $H \leftarrow \mathbf{B}$ where H is an atom (the *head*) and $\mathbf{B}$ is a query (the *body*). When $\mathbf{B}$ is empty, $H \leftarrow \mathbf{B}$ is written $H \leftarrow$ and is called a *unit clause*. A *(definite) program* is a finite set of clauses. 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.

Computations are constructed as sequences of "basic" steps. Consider a non-empty query $\mathbf{A}, B, \mathbf{C}$ and a clause c. Let $H \leftarrow \mathbf{B}$ be a variant of c variable disjoint from $\mathbf{A}, B, \mathbf{C}$. Let B and H unify with mgu θ. The query $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ is called a *resolvent of* $\mathbf{A}, B, \mathbf{C}$ *and* c *with selected atom* B *and mgu* θ. A *derivation step* is denoted by $\mathbf{A}, B, \mathbf{C} \overset{\theta}{\Longrightarrow}_{P,c} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$. The clause $H \leftarrow \mathbf{B}$ is called its *input clause*. The atom B is called the *selected atom* of $\mathbf{A}, B, \mathbf{C}$.

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

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

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

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

2.3 Modes & Input Consuming Programs

Modes are a common tool for verification. A *mode* is a function that labels as *input* or *output* the positions of each predicate in order to indicate how the arguments of such a predicate should be used.

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

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

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

The notion of input consuming derivation was introduced in [40] as a formalism for describing dynamic scheduling in an abstract way.

Definition 2 (Input Consuming Derivation).

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

In the following sometimes we use *ic-derivation* for input consuming derivation and we call *input consuming program* (*ic-program*) a program when considered with respect to input consuming derivations only.

Example 3. Consider the program `REVERSE` with accumulator and the following modes: `reverse`(I, O) and `reverse_acc`(I, O, I).

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

The following derivation δ of `REVERSE` $\cup$ {`reverse([X1,X2],Zs)`} is input consuming.

δ: `reverse([X1,X2],Zs)` $\Rightarrow$ `reverse_acc([X1,X2],Zs,[])` $\Rightarrow$
`reverse_acc([X2],Zs,[X1])` $\Rightarrow$ `reverse_acc([],Zs,[X2,X1])` $\Rightarrow \Box$.

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

2.4 Classes of Moded Programs

In the sequel we are going to refer to classes of programs that in some way behave well with respect to the given mode. In particular, we are going to use the concepts of well moded program (Dembinski and Maluszynski [20]), of nicely moded program (Chadha and Plaisted [15]) and of simply moded program (Apt and Etalle [4]).

Definition 4 (Well, Nicely and Simply Moded Program).

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

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

If we call *producing* positions the input positions of the head and the output positions of the body and *consuming* positions the other ones, then we can intuitively say that a clause is well moded if every variable in a consuming position occurs also in an *earlier* producing position (notice that the consuming positions in the head are the "last" ones in this particular order).

– ***Nicely Moded***. *A clause* $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ *is* nicely moded *if*
  - $\mathbf{t}_1, \ldots, \mathbf{t_n}$ *is a linear sequence of terms*
  - $\mathrm{Var}(\mathbf{t}_0) \cap \mathrm{Var}(\mathbf{t}_1, \ldots, \mathbf{t_n}) = \emptyset$.
  - *and for all* $i \in [1, n]$

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

Intuitively a clause is nicely moded if there are no conflicts among producing positions, (a variable may appear in at most one producing position with one exception: a variable may appear twice in a producing position of the head), and a variable may not be consumed before it is produced.

– ***Simply Moded***. *A clause* $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ *is* simply moded *if it is nicely moded and* $\mathbf{t}_1, \ldots, \mathbf{t_n}$ *is a linear sequence of variables.*

– *A query* $\mathbf{Q}$ *is well (resp. nicely, simply) moded, if the clause* $q \leftarrow \mathbf{Q}$ *is well (resp. nicely, simply) moded, where* q *is a variable-free atom.*
Note that an atomic query $p(\mathbf{s}, \mathbf{t})$ is well moded if $\mathbf{s}$ is a sequence of ground terms and it is nicely moded if $\mathbf{t}$ is linear and $Var(\mathbf{s}) \cap Var(\mathbf{t}) = \emptyset$.

– *A program is well (resp. nicely, simply) moded, if all of its clauses are well (resp. nicely, simply) moded.*

Hence the class of simply moded programs is a subclass of nicely moded ones and it includes both some well moded and some non-well moded programs.

In [42] *permutation well (nicely) moded* programs and queries are also defined, i.e., programs and queries which would be well (nicely) moded after a permutation of the atoms respectively in the bodies and in the queries.

Example 5.

– The program `APPEND` of the introduction in the mode `append`(I, I, O) is well nicely and simply moded.
– `REVERSE` with accumulator of Example 3 is well nicely and simply moded.
– Furthermore, consider the following program `PALINDROME` in the mode `palindrome`(I)

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

together with the program REVERSE with the modes `reverse`(I,O). 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`(I,I) and `reverse_acc`(I,I,I). With these modes, the program PALINDROME is both well moded, nicely and simply moded.

Most programs are simply moded (see the mini-survey at the end of [4]) and often non simply moded programs can naturally be transformed into simply moded ones (see [10]).

The above notions of well, nicely and simply moded are "persistent" with respect to input consuming derivations. The following Lemma is a straightforward extension of [5, Lemma 30].

Lemma 6. *In a input consuming derivation, every resolvent of a well (resp. nicely, simply) moded query and a well (resp. nicely, simply) moded clause is well (resp. nicely, simply) moded.*

Notice that in the case of nicely and simply moded programs the above Lemma depends on the fact that only input consuming derivations are considered. Indeed, when "normal" SLD derivations are considered, it is easy to find an example in which the SLD resolvent of a nicely moded query and a nicely moded clause is not nicely moded. On the other hand, for well moded programs, any SLD resolvent of a well moded query with a well moded clause is well moded ([2]).

Finally, it is worth reminding that, when considering nicely (respectively simply) moded, input consuming programs, half of the famous switching Lemma still applies. The following Left-Switching Lemma that has been proven in [10].

Lemma 7. (Left-Switching) *Let the program P and the query Q_0 be nicely moded. Let δ be a (partial) input consuming derivation of $P \cup \{Q_0\}$ of the form*

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

where

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

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

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

and

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

where δ' is input consuming and

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

2.5 The $\mathcal{S}$-semantics

The aim of the $\mathcal{S}$-semantics approach (see [13]) 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:

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

This semantics enjoys all the valuable properties of the least Herbrand model. 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 closed under variance.

– The immediate consequence operator for the $\mathcal{S}$-semantics is defined as:

$$\begin{aligned}T_P^{\mathcal{S}}(I) = \{\ H\theta \mid {} & \exists\ H\ \leftarrow\ \mathbf{B} \text{ variant of a clause of } P \\ & \exists\ \mathbf{C} \in I, \text{renamed apart}^4 \text{ wrt. } H, \mathbf{B} \\ & \theta = mgu(\mathbf{B}, \mathbf{C})\}.\end{aligned}$$

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

Falaschi et al. [25] 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 ([25]). *Let P be a program, $\mathbf{A}$ be a query. The following statements are equivalent.*

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

[4] 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.

where $\mathbf{A}\sigma \approx \mathbf{A}\vartheta$.

Example 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]), ... }.

2.6 Semantics of Input Consuming Programs

In the following two Sections we present two semantics for input consuming programs which are related to $\mathcal{S}$-semantics. To define such semantics, the observables we focus on are the *computed answer substitutions*. First, we consider a semantics given by the computed answer substitutions of *successful* derivations. This corresponds to the $\mathcal{S}$-semantics of logic programming [13] when restricted to a particular set of queries. Given a program P and a set of queries C, this semantics can be defined formally as

$$\mathcal{O}_s^{ic}(P,C) = \{\mathbf{A}\theta |\ \mathbf{A} \in C \text{ and there exists an ic-derivation } \mathbf{A} \xrightarrow{\theta}_P \ \Box\}.$$

While this semantics appears very natural, it can be unsuitable for modelling the reactive nature of input consuming programs. In fact, as we mentioned in the introduction, input consuming derivations can be used to model dynamic scheduling and parallelism, and in this context it is very important to model the results of partial computations. Indeed, standard semantics for concurrent logic languages such as ccp [39, 22] and GHC [44] often capture such intermediate results, or in any case, also the results of non-successful computations [16]. In fact, the (partial) result of a computation may trigger another computation by instantiating sufficiently the input positions of another atom so that it becomes resolvable. Because of this, when one wants to characterize for instance termination, the adoption of a semantics which is able to model intermediate results becomes essential, as shown in Section 5. Thus we also consider a semantics capturing the results of partial input consuming derivations. Given a program P and a set of queries C, this semantics can be defined formally as

$$\mathcal{O}_p^{ic}(P,C) = \{\mathbf{A}\theta |\ \mathbf{A} \in C \text{ and there exists an ic-derivation } \mathbf{A} \xrightarrow{\theta}_P \ \mathbf{B}\}.$$

where $\mathbf{B}$ is any query.

3 Semantics of Well Moded Input Consuming Programs

To characterize our semantics for ic-programs, we start from the simplest case: when one is interested only in the successful derivations. Then the observables (given by successful derivations) can be reconducted to the $\mathcal{S}$-semantics of classical logic programs.

We show that *the standard $\mathcal{S}$-semantics is compositional and correct also for input consuming programs, provided that the programs are well and nicely moded and that only nicely moded queries are considered.* The results reported in this Section are proved in [9].

Proposition 10. *Let P be a well and nicely moded program, A be a nicely moded atomic query. The following statements are equivalent.*

(i) There exists an input consuming derivation $A \xrightarrow{\vartheta}_P \Box$,
(ii) There exists $A' \in \mathcal{S}(P)$ (renamed apart wrt. A), and $\sigma = mgu(A, A')$ such that $In(A)\sigma \approx In(A)$,

where $A\sigma \approx A\vartheta$.

To extend Proposition 10 to arbitrary (non-atomic) queries we need the following definition.

Definition 11. *Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$. We define*

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

$VIn^*(Q)$ 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 that if Q is well moded then $VIn^*(Q) = \emptyset$.

Theorem 12. *Let P be a well and nicely moded program, $\mathbf{A}$ be a nicely moded query and NM be the class of nicely moded queries. The following statements are equivalent.*

(i) There exists $\mathbf{A}\vartheta \in \mathcal{O}_s^{ic}(P, NM)$,
(ii) There exists $\mathbf{A}' \in \mathcal{S}(P)$ (renamed apart wrt. $\mathbf{A}$), and $\sigma = mgu(\mathbf{A}, \mathbf{A}')$ such that $\mathbf{A}\sigma_{|VIn^*(\mathbf{A})} \approx \mathbf{A}$,

where $\mathbf{A}\sigma \approx \mathbf{A}\vartheta$.

Condition $\mathbf{A}\sigma_{|VIn^*(\mathbf{A})} \approx \mathbf{A}$ above says that the substitution σ just renames the variables occurring in an input position of A but not occurring in an output position of an earlier atom. In case of an atomic query $\mathbf{A} := A$, we might substitute this condition with the somewhat more attractive condition $In(A)\sigma \approx In(A)$ of Proposition 10.

Hence $\mathcal{S}(P)$ is *compositional* and *correct* for input consuming programs, provided that programs are well and nicely moded and that queries are nicely moded. In other words, given the restrictions on programs and queries, the $\mathcal{S}$-semantics is correct with respect to the observables given by the computed answer substitutions of successful ic-derivations.

Example 13. Consider the program APPEND of the Introduction with the moding append(I,I,O). $\mathcal{S}$(APPEND), given in Example 9, allows us to draw a number of conclusions:

- append([X,b],Y,Z) has an input consuming successful derivation.
  In particular, it has an input consuming derivation with c.a.s. {Z/[X,b|Y]}. This can be derived by just looking at $\mathcal{S}$(APPEND), from the fact that $A =$ append([X1,X2],X3,[X1,X2|X3]) $\in \mathcal{S}(P)$ and that append([X,b],Y,Z) is - in its input positions - 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 append(Y,[X,b],Z) is an instance of A in the input positions.
- Observe that the query append(Y,[X,b],Z) has infinite successful SLD derivations and no failures. Therefore it does not fail also when we consider ic-derivations. Since, as noted above, the query has no input consuming successful derivations, this implies that – in presence of input consuming derivations – append(Y,[X,b],Z) will eventually either deadlock or run into an infinite derivation.

The previous results hold also in case the programs are *permutation well and nicely moded* and queries are *permutation nicely moded* (see [42]).

While in the context of SLD (not input consuming) derivations the $\mathcal{S}$-semantics is also *fully abstract*, when considering input consuming program this is not so. Consider the following two trivial programs:

```
P1  = { c1:  p(X).
        c2:  p(a).   }
P2  = {      p(X).   }
```

In both programs the mode is p(I). These two programs, despite being different, yield exactly the same computed answer substitutions for all queries when ic-derivations are considered. In fact the extra clause c2 in P1 can resolve an atom A only if A contains the term a in its input position, but in this case c2 behaves exactly as c1 does[5]. Nevertheless, the $\mathcal{S}$(P1) = {p(X), p(a)} $\neq$ {p(X)} = $\mathcal{S}$(P2), demonstrating that the $\mathcal{S}$-semantics is not fully abstract when considering ic-derivations. In the next Section we present a more complex semantics, which is also fully abstract for ic-derivations.

[5] The only observable difference between P1 and P2 lies in the *multiplicity* of the answers: the query q(a) succeeds twice in P1 and only once in P2, but answer multiplicity is not an observable we consider here.

4 Semantics of Simply Moded Input Consuming Programs

The semantics presented in the previous Section applies only when we are interested in the computed answer substitutions of *successful* derivations. As we discussed before, there are many situations in which we also want to model the (intermediate) results of partial derivations. For instance, this will be the case when – in the next Section – we study the termination of input consuming programs.

In this Section we define a somewhat more complex denotational semantics which has the advantage to model the observables given by both successful and partial derivations in a rather symmetric way. In addition, in exchange for a moderate syntactic restriction (instead of nicely moded programs and queries we have to consider *simply moded* ones) it allows us to drop the requirement that programs have to be well moded. The two semantics we are going to introduce are *compositional, correct and fully abstract* with respect to the operational semantics of input consuming simply moded programs and queries, i.e., $\mathcal{O}_s^{ic}(P, SM)$ and $\mathcal{O}_p^{ic}(P, SM)$, where SM is the class of simply moded queries. As in the standard $\mathcal{S}$-semantics, we build a denotational semantics by means of a bottom-up construction.

4.1 Simply Local Substitutions and Simply Local Models

When input consuming derivations are applied to simply moded programs and queries, important properties follow from the way clauses become instantiated along the derivations. The notion of *simply local* substitution is introduced in [12] to reflect this instantiation mechanism. A clause $c = H \leftarrow B_1, \ldots, B_n$ becomes instantiated by its "caller" (the atom that is resolved using c) and its "callees" (the clauses used to resolve the body atoms of c). Thus, a simply local substitution is defined as the composition of several substitutions, $\sigma_0, \sigma_1 \ldots, \sigma_n$, one for each atom in the given clause, such that σ_0 binds the input variables of the head of the clause, and each σ_i ($i > 0$) creates a binding from the output variables to input terms of $B_i\sigma_0, \ldots, \sigma_{i-1}$.

Definition 14 (Simply Local Substitution). *Let θ be a substitution. We say that θ is* simply local *with respect to the clause $H \leftarrow B_1, \ldots, B_n$ if there exist substitutions $\sigma_0, \sigma_1 \ldots, \sigma_n$ and disjoint sets of fresh (with respect to c) variables $v_0, v_1, \ldots, v_n$ such that $\theta = \sigma_0\sigma_1 \cdots \sigma_n$ where*

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

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

Example 15. Consider the program APPEND together with the mode append(I, I, O) and its recursive clause

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

The substitution $\theta = \{\texttt{Xs/[]}, \texttt{Ys/W}, \texttt{Zs/W}\}$ is simply local with respect to c. In fact $\theta = \sigma_0\sigma_1$ where $\sigma_0 = \{\texttt{Xs/[]}, \texttt{Ys/W}\}$ and $\sigma_1 = \{\texttt{Zs/W}\}$. Consider now the query

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

The substitution $\theta = \{\texttt{Zs/[a,X,c|Ys]}\}$ is simply local with respect to Q. In fact $\theta = \sigma_1\sigma_2$ where $\sigma_1 = \{\texttt{Zs/[a,X,c|Ys]}\}$ and σ_2 is the empty substitution.

The denotational semantics defined in [12] is based on a restricted notion of model. Here and in the sequel we consider *sets of moded atoms* closed under variance.

Definition 16 (Simply Local Model). *Let M be a set of moded atoms. We say that M is a* simply local model *of a clause $c : H \leftarrow B_1, \ldots, B_n$ if for every substitution θ simply local with respect to c,*

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

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

Clearly a simply local model is not necessarily a model in the classical sense, since the substitution θ in (1) is required to be simply local. For example, given the program $\{\texttt{q(1).}, \ \texttt{p(X)} \leftarrow \texttt{q(X).}\}$ with modes $\texttt{q}(I)$, $\texttt{p}(O)$, a model must contain the atom $\texttt{p(1)}$, whereas a simply local model does not necessarily contain $\texttt{p(1)}$, since $\{\texttt{X/1}\}$ is not simply local with respect to $\texttt{p(X)} \leftarrow \texttt{q(X)}$.

A minimal simply local model exists and it is bottom-up computable by applying the following operator [12].

Definition 17. *Given a program P and a set of moded atoms I, we define*

$$\begin{aligned} T_P^{SL}(I) = I \cup \{H\theta \mid\ & \exists\, c : H \leftarrow \mathbf{B} \textit{ variant of a clause of } P,\\ & \theta \textit{ is simply local with respect to } c,\\ & \mathbf{B}\theta \in I\} \end{aligned}$$

T_P^{SL} is monotonic and continuous on the lattice where set of moded atoms are ordered by set inclusion. Powers of an operator T are defined in the standard way as follows: $T \uparrow 0(I) = I$, $T \uparrow (i+1)(I) = T(T \uparrow i(I))$, and $T \uparrow \omega(I) = \bigcup_{i=0}^{\infty} T \uparrow i(I)$.

In the following we denote by SM_P the set of all simply moded atoms of the extended Herbrand universe of P. In [12] it is proven that if P is simply moded and $I \subseteq SM_P$ then

$$T_P^{SL} \uparrow \omega(I) \text{ is the least simply local model of } P \text{ containing } I \tag{2}$$

This allows us to define our models.

Definition 18. *Let P be a program, we define*

- M_P^{SL} *is the least simply local model of* P*,*
- PM_P^{SL} *is the least simply local model of* P *containing* SM_P*.*

The existence of these models is guaranteed by (2), in fact (2) also shows how to construct them, as it implies that

$$M_P^{SL} = T_P^{SL} \uparrow \omega(\emptyset), \text{ and } PM_P^{SL} = T_P^{SL} \uparrow \omega(SM_P) \tag{3}$$

4.2 Relation among Denotational and Operational Semantics

To relate the M_P^{SL} and PM_P^{SL} to $\mathcal{O}_s^{ic}(P, SM)$ and $\mathcal{O}_p^{ic}(P, SM)$ we need to relate T_P^{SL} to the results of input consuming derivations; this is achieved in the following Lemma, proved in [12].

Lemma 19. *Let the program P and the query $\mathbf{A}$ be simply moded and $I \subseteq SM_P$ be a set of moded atoms. The following statements are equivalent.*

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

where $\mathbf{A}\vartheta \approx \mathbf{A}\theta$*.*

We can now prove that M_P^{SL} and PM_P^{SL} fully characterize the semantics of ic-derivations for simply moded programs and queries, namely they are equal to $\mathcal{O}_s^{ic}(P, SM)$ and $\mathcal{O}_p^{ic}(P, SM)$, respectively.

Theorem 20. *Let P be simply moded. Then*

(i) $M_P^{SL} = \mathcal{O}_s^{ic}(P, SM)$*.*
(ii) $PM_P^{SL} = \mathcal{O}_p^{ic}(P, SM)$*.*

Proof. *Immediate by (3), Lemma 19 and the definitions of* $\mathcal{O}_s^{ic}(P, SM)$ *and* $\mathcal{O}_p^{ic}(P, SM)$*.*

An example follows.

Example 21. Let us consider again the program `APPEND`.

1. First let us consider its *successful* ic-derivations. Hence we have to build $M_{\texttt{APPEND}}^{SL}$

   $$\begin{aligned} M_{\texttt{APPEND}}^{SL} = \{&\texttt{append}(\texttt{[}t_1, \ldots, t_n\texttt{]}, s, \texttt{[}t_1, \ldots, t_n\texttt{|}s\texttt{]}) \mid n \in [0..\infty], \\ &\text{and } t_1, \ldots, t_n, s \text{ are any terms}\}. \end{aligned}$$

   Notice that this model is different from $\mathcal{S}(\texttt{APPEND})$, reported in Example 9. We are going to relate $\mathcal{S}(P)$ and M_P^{SL} later in this Section.

2. Now let us consider the results of *partial* derivations. Recall that $PM^{SL}_{\mathtt{APPEND}}$ is obtained by repeatedly applying T^{SL}_P to each simply moded atom. Simply moded atoms are $\mathtt{append}(s,t,x)$ where s and t are arbitrary terms but x is a variable not occurring in s or in t. We obtain

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

where $s,t,t_1,\dots,t_m$ are arbitrary terms.
Consider now the query $\mathtt{append([a,b,c|X],Y,Z)}$. It is straightforward to check that the substitution $\theta = \{\mathtt{Z/[a,b|Z']}\}$ is simply local with respect to it, and that $\mathtt{append([a,b,c|X],Y,Z)}\theta \in PM^{SL}_{\mathtt{APPEND}}$. Therefore, by using Theorem 20, we can conclude that there exists a partial derivation starting in $\mathtt{append([a,b,c|X],Y,Z)}$, with computed answer θ. Following the same reasoning, one can also conclude that the query has a partial derivation with computed answer $\theta' = \{\mathtt{Z/[a|Z']}\}$.

4.3 Relation among $\mathcal{S}$-semantics and Denotational Semantics for IC-programs

In this section we compare the denotational semantics M^{SL}_P with the $\mathcal{S}$-semantics $\mathcal{S}(P)$ of simply moded programs.

First, we need a new definition: let I be a set of moded atoms, the *input closure* of I is defined as:

$$InCl(I) = \{A\theta \mid A \in I \text{ and } Var(A) \cap Var(\theta) \subseteq Var(In(A))\}$$

So the input closure of an atom is obtained by instantiating its input positions in all possible ways, provided that no new links are created between the input and the output positions.

Theorem 22. *Let P be a well and simply moded program, then*

$$M^{\mathrm{SL}}_P = InCl(\mathcal{S}(P))$$

Proof. First observe that the class of simply moded programs is contained in the class of nicely moded programs, hence Theorem 12 holds also when we consider well and simply moded programs and simply moded queries.

- $M^{SL}_P \subseteq InCl(\mathcal{S}(P))$. Let A be simply moded and $A\vartheta \in M^{SL}_P$ then, by Theorem 20, $A\vartheta \in \mathcal{O}^{ic}_s(P,SM)$. By Theorem 12 there exists $A' \in \mathcal{S}(P)$ (renamed apart wrt. A), and $\sigma = mgu(A,A')$ such that $In(A)\sigma \approx In(A)$ and $A\sigma \approx A\vartheta$. Since A is simply moded, we can choose σ such that $Dom(\sigma) \cap Var(A') \subseteq Var(In(A'))$. Therefore $A\vartheta \approx A\sigma = A'\sigma \in InCl(\mathcal{S}(P))$.

- $M^{SL}_P \supseteq InCl(\mathcal{S}(P))$. Let $A'\theta \in InCl(\mathcal{S}(P))$ and $A' = p(\mathbf{s},\mathbf{t}) \in \mathcal{S}(P)$. There exist a simply moded atom $A = p(\mathbf{s}',\mathbf{z})$, renamed apart wrt. A', and a substitution σ such that $\sigma = mgu(A,A')$, $In(A)\sigma = In(A)$ and $A\sigma = A'\sigma \approx A'\theta$. By Theorem 12 there exists ϑ such that $A\vartheta \in \mathcal{O}^{ic}_s(P,SM)$ and $A\vartheta \approx A\sigma \approx A'\theta$. Hence, by Theorem 20, $A'\theta \in M^{SL}_P$.

5 Semantic-Based Verification of Termination

There have been only few proposals which tackled the specific problem of verifying the termination of logic programs with dynamic scheduling. Namely by Apt and Luitjes [5], Marchiori and Teusink [30] and Smaus. Input consuming derivations were indeed introduced by Smaus in [40] to simplify the study of program properties which depend on selection rules and in [41] he started to study in particular the problem of termination of input consuming derivations.

In [10] and [12] we study two classes of programs terminating with respect to input consuming derivations and well-formed queries. The two classes differ in various aspects. First of all, two different classes of well-formed queries are considered: nicely moded queries in [10], simply moded queries in [12]. To give an uniform presentation, in [12] we consider a parametric class of programs in which all input consuming derivations terminate. The parameter is a given class C of queries.

Definition 23 (Input Termination wrt. a class C of queries). *Let C be a class of queries. A program is called* input terminating with respect to C *if all its input consuming derivations started in a query in C are finite.*

The second difference among the two classes of terminating programs in [10] and [12] is in the termination proof style. The first class follows the style of [3, 8] and it uses a simple (syntactic) termination condition, but it is also a rather restrictive class. The second class follows the style of [6, 7], that is based on a more complex model theoretic approach, and it uses the semantics introduced in Section 4; this is a significantly larger class of programs.

Let us consider first the more restrictive and simple class introduced in [10]: The class of nicely moded *quasi recurrent* programs. Its definition is based on the notion of well moded level mapping, first introduced in [21]. Here we use its extension ([10]) to all the terms on $\mathcal{B}_P^{\mathcal{E}}$, the extended Herbrand base of P, that is the set of equivalence classes of all (possibly non-ground) atoms, modulo renaming, whose predicate symbols appear in P.

Definition 24 (Moded Level Mapping). *Let P be a program and $\mathcal{B}_P^{\mathcal{E}}$ be the extended Herbrand base for the language associated with P. A function $|\ |$ is a* moded level mapping for P *if:*

- *it is a function $|\ | : \mathcal{B}_P^{\mathcal{E}} \to \mathbf{N}$ from atoms to natural numbers;*
- *for any $\mathbf{t}$ and $\mathbf{u}$, $|p(\mathbf{s},\mathbf{t})| = |p(\mathbf{s},\mathbf{u})|$.*

For $A \in \mathcal{B}_P^{\mathcal{E}}$, $|A|$ is the level *of A.*

Definition 25 (Quasi Recurrency). *Let P be a program.*

- *A clause of P is called* quasi recurrent with respect to a moded level mapping $|\ |$ *if for every instance of it, $H \leftarrow \mathbf{A}, B, \mathbf{C}$*

$$\textit{if } Rel(H) \simeq {}^{6} Rel(B) \textit{ then } |H| > |B|.$$

[6] Given two predicate symbols defined in a program P we denote by $p \simeq q$ the fact that the definitions of the two predicates are mutually recursive.

– *P is called* quasi recurrent with respect to $| \; |$ *if all its clauses are. P is called* quasi recurrent *if it is quasi recurrent with respect to some moded level mapping* $| \; | : \mathcal{B}_P^{\mathcal{E}} \to \mathbf{N}$.

Theorem 26. *Let P be a nicely moded program. If P is quasi recurrent then P is input terminating with respect to the class of nicely moded queries.*

The proof of this Theorem can be found in [10].

Thus, the quasi recurrent condition is a sufficient condition for input termination of nicely moded programs and nicely moded queries. But it is not a necessary condition: there are nicely moded programs input terminating on all nicely moded queries which are not quasi recurrent as shown by the following simple example taken from [10].

Example 27. Consider the following program with moding `p(I,O)`.

```
p(X,a) ← p(X,b).
p(X,b).
```

This program is clearly input terminating, however it is not quasi recurrent. If it was, we would have that $|\mathtt{p(X,a)}| > |\mathtt{p(X,b)}|$, for some moded level mapping $| \; |$ (otherwise the first clause would not be quasi recurrent). On the other hand, since `p(X,a)` and `p(X,b)` differ only for the terms filling in their output positions, by definition of moded level mapping, $|\mathtt{p(X,a)}| = |\mathtt{p(X,b)}|$. Hence, we have a contradiction.

A full characterization can be obtained only by further restricting the class of programs, passing from nicely moded to simply moded and *input-recursive* programs.

Definition 28 (Input-Recursive Program). *Let P be a program.*

– *A clause $H \leftarrow \mathbf{A}, B, \mathbf{C}$ of P is called* input-recursive *if*

$$\textit{if } Rel(H) \simeq Rel(B) \textit{ then } Var(In(B)) \subseteq Var(In(H)).$$

– *A program P is called* input-recursive *if all its clauses are.*

Input-recursive is a syntactic condition on a clause requiring that the set of variables occurring in the arguments filling in the input positions of each recursive call in the clause body is a subset of the set of variables occurring in the arguments filling in the input positions of the clause head. The class of input-recursive programs has strong similarities with the class of primitive recursive functions. It does not include programs such that `quicksort`, `permute`, `transpose` and we can compare it with the class of recurrent logic programs, that is programs whose termination does not depend on the so-called *inter-argument relations*.

Quasi recurrency fully characterizes input termination of simply moded and input-recursive programs with respect to nicely moded queries.

Theorem 29. *Let P be a simply moded and input-recursive program. P is quasi recurrent if and only if P is input terminating with respect to the class of nicely moded queries.*

The proof of this Theorem can be found in [10].

To consider a larger class of input terminating programs we can follow the same approach pursued by Apt and Pedreschi in defining acceptable programs and use a model to capture the inter-argument relations between the atoms in a query. Intuitively, the model represents all the possible contexts in which a specific atom in a query can be called. Standard models suffice when standard left-to-right derivations are considered, that is when the contexts depends only on the computed answers of the atoms occurring on the left of the considered one. When input consuming derivations are considered, the description of all the possible contexts is much more complex since there may be atoms in the query which are only partially computed when the considered atom is selected. Hence a computed answer semantics does not provide enough information, that is why we need to capture partial computed answers of input consuming derivations.

The semantics defined in [12] and the concept of simply local model give us the right tools and allow us to identify a large class of input terminating programs which includes also programs employing a non-trivial recursion scheme such as `quicksort`, `permute`, `transpose`. In fact, based on the notion of simply local models, in [12] we introduced the notion of simply acceptable programs which corresponds to the notion of acceptable programs introduced in [6].

Definition 30 (Simply Acceptable Program). *Let P be a program and M a simply local model of P containing SM_P.*

- *A clause c of P is* simply acceptable with respect to a moded level mapping $|\ |$ and M *if for every variant $H \leftarrow \mathbf{A}, B, \mathbf{C}$ of c and every substitution θ simply local with respect to c,*

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

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

Simple acceptability fully characterizes input termination of simply moded programs with respect to simply moded queries.

Theorem 31. *Let P be a simply moded program. P is simply acceptable if and only if it is input terminating with respect to simply moded queries.*

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

Example 32. Consider the following `PERMUTE` program

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

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

We consider it with two different modes.

1. First, consider the mode $\texttt{permute}(O, I)$, $\texttt{insert}(O, O, I)$.
   Notice that the program is not input terminating in this mode: by repeatedly selecting the rightmost atom, the query `permute(Xs,Ys)` generates an infinite input consuming derivation. By Theorem 31, we can prove it by showing that `PERMUTE` in this mode cannot be simply acceptable with respect to $PM^{SL}_{\texttt{PERMUTE}}$ and a moded level mapping which is invariant under renaming.
   First note that $PM^{SL}_{\texttt{PERMUTE}}$ contains every atom of the form $\texttt{insert}(\texttt{Us}, \texttt{U}, t)$ where `Us` and `U` are disjoint from t, i.e., every simply moded atom whose predicate is `insert`. Therefore, in particular, $\texttt{insert}(\texttt{Us}, \texttt{U}, \texttt{Vs}) \in PM^{SL}_{\texttt{PERMUTE}}$. The substitution $\theta = \{\texttt{Ys}/\texttt{Vs}, \texttt{Zs}/\texttt{Us}, \texttt{X}/\texttt{U}\}$ is simply local with respect to the first clause. Therefore, for this clause to be simply acceptable, by Theorem 31, there would have to be a moded level mapping, invariant under renaming, such that $|\texttt{permute}(\texttt{[U|Xs]}, \texttt{Vs})| > |\texttt{permute}(\texttt{Xs}, \texttt{Us})|$. This is a contradiction since a *moded* level mapping depends only on the input arguments (the second argument of `permute`) and we are considering a level mapping invariant under renaming.
   Thus Theorem 31 can be used to diagnose a program, in that we can pinpoint why it does not input terminate.
2. Now consider the program `PERMUTE` together with the mode $\texttt{permute}(I, O)$, $\texttt{insert}(I, I, O)$.
   In this case, in order to make the program simply moded we have to permute the two body atoms of the first `permute` clause[7]. I.e., `permute` is redefined as

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

   Notice that the program is now input terminating with respect to simply moded queries. This is in fact the *natural* mode of the `PERMUTE` program. To demonstrate the termination one can apply Theorem 31 using *any* simply local model containing SM_P together with the following moded level mapping:

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

[7] Actually, everything we state applies to the class of *permutation* simply moded programs, i.e., those programs and queries that are simply moded possibly after a permutation of body atoms. For the sake of notation simplicity, we avoid to refer to this in a structural way.

6 Conclusion

In this paper, we have illustrated two denotational semantics proposed in [9] and in [12] for input consuming derivation in logic programs and we have shown how these semantics have been used for studying termination properties of such programs.

The two semantics are quite orthogonal to each other: while the first one (introduced in [9]) models exclusively the results of successful derivations and requires programs to be *well moded* and *nicely moded*, the second one (introduced in [12]) models also the results of incomplete derivations and requires programs and queries to be simply moded.

As mentioned in the Introduction, in the context of parallel and concurrent programs, one can have derivations that never *succeed*, and yet compute substitutions [36]. Thus we have provided a denotational semantics also for such programs, which goes beyond the usual success-based SLD resolution mechanism of logic programming.

Input consuming derivations bear a certain resemblance with derivations in the language of *Moded (Flat) GHC* [45]. Actually, input consuming programs can be seen as a simplified version of moded (F)GHC. We want to note however that Moded (F)GHC is a full-fledged programming paradigm, while input consuming programs are meant for abstraction purposes.

As a concluding remark, we want to stress the relation between ic-programs and programs that use delay declarations. A significant class of programs with delay declarations whose derivations are input consuming derivations has been identified in [11].

References

1. K. R. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier and The MIT Press, Amsterdam and Cambridge, MA, 1990.
2. K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, London, 1997.
3. K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3&4):335–363, 1991.
4. K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS'93)*, volume 711 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, Germany, 1993. Springer-Verlag.
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)*, volume 936 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, Germany, 1995. Springer-Verlag.
6. K. R. Apt and D. Pedreschi. Proving termination of general Prolog programs. In T. Ito and A. Meyer, editors, *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science 526, pages 265–289, Berlin, Germany, 1991. Springer-Verlag.

7. K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.
8. M. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1&2):79–97, 1993.
9. A. Bossi, S. Etalle, and S. Rossi. Semantics of well-moded input-consuming logic programs. *Computer Languages*, 26(1):1–25, 2000.
10. A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. *Theory and Practice of Logic Programming*, 2(2):125–154, 2002.
11. A. Bossi, S. Etalle, S. Rossi, and J.-G. Smaus. Semantics and termination of simply-moded logic programs with dynamic scheduling. In D. Sands, editor, *Proceedings of the European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 402–416, Genova, Italy, 2001. Springer-Verlag.
12. A. Bossi, S. Etalle, S. Rossi, and J.-G. Smaus. Termination of simply-moded logic programs with dynamic scheduling. *ACM Transactions on Computational Logic (TOCL)*, 2004. To appear.
13. A. Bossi, M. Gabrielli, G. Levi, and M. Martelli. The S-semantics approach: Theory and applications. *The Journal of Logic Programming*, 19 & 20:149–198, May 1994.
14. A. Bossi, S. Etalle N. Cocco, and S. Rossi. On Modular Termination Proofs of General Logic Programs. *Theory and Practice of Logic Programming*, 2(3):263–291, 2002.
15. 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.
16. F.S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of the International Joint Conference on Theory and Practice of Software Development, (TAPSOFT/CAAP)*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319, Brighton, UK, 1991. Springer-Verlag.
17. M. Garcia de la Banda, K. Marriott, and P. Stuckey. Efficient analysis of logic programs with dynamic scheduling. In J. Lloyd, editor, *Proc. Twelfth International Logic Programming Symposium*, pages 417–431. The MIT Press, 1995.
18. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.
19. S. Debray, D. Gudemann, and P. Bigot. Detection and optimization of suspension-free logic programs. In M. Bruynooghe, editor, *Proc. Eleventh International Logic Programming Symposium*, pages 487–504. The MIT Press, 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, Boston, 1985.
21. S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *Journal of Logic Programming*, 38(2):243–257, 1999.
22. S. Etalle, M. Gabbrielli, and M. C. Meo. Transformations of ccp programs. *ACM Transactions on Programming Languages and Systems*, 23(3):304–395, 2002.
23. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Constraint logic programming with dynamic scheduling: a semantics based on closure operators. *Information and Computation*, 137(1):41–67, 1997.
24. M. Falaschi, M. Gabrielli, K. Marriott, and C. Palamidessi. Compositional analysis for concurrent constraint programming. In *Proceedings of the IEEE Symposium on Logic in Computer Science*. IEEE, 1993.

25. 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.
26. P. M. Hill and J. W. Lloyd. *The Gödel programming language*. The MIT Press, 1994.
27. IC Parck, Imperial College London. *The ECLiPSe Constraint Logic Programming System*, 2003. http://www-icparc.doc.ic.ac.uk/eclipse/.
28. Intelligent Systems Laboratory, Swedish Institute of Computer Science, PO Box 1263, S-164 29 Kista, Sweden. *SICStus Prolog Page*, 2003. http://www.sics.se/sicstus/.
29. J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, Berlin, Germany, 1987. Second edition.
30. E. Marchiori and F. Teusink. Termination of logic programs with delay declarations. *Journal of Logic Programming*, 39(1–3):95–124, 1999.
31. K. Marriott. Algebraic and logical semantics for CLP languages with dynamic scheduling. *Journal of Logic Programming*, 32(1):71–84, 1997.
32. K. Marriott, M. Garcia de la Banda, and M. Hermenegildo. Analyzing logic programs with dynamic scheduling. In *Proc. 21st Annual ACM Symp. on Principles of Programming Languages*, pages 240–253. ACM Press, 1994.
33. K. Marriott, M. Falaschi, M. Gabrielli, and C. Palamidessi. A simple semantics for logic programming languages with delay. In *Proceedings of the Eighteenth Australian Computer Science Conference*, 1995.
34. L. Naish. *Negation and control in Prolog*, volume 238 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1986.
35. L. Naish. Coroutining and the construction of terminating logic programs. *Australian Computer Science Communications*, 15(1):181–190, 1993.
36. 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. The MIT Press.
37. L. Naish. An introduction to MU-Prolog. Technical Report 82/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, March 1982 (Revised July 1983).
38. G. Puebla, M. Garcia de la Banda, K. Marriott, and P. Stuckey. Optimization of logic programs with dynamic scheduling. In *ICLP 1997*, pages 93–107, 1997.
39. V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, California, 1990. ACM, New York.
40. J.-G. Smaus. *Modes and Types in Logic Programming*. PhD thesis, University of Kent at Canterbury, 1999. Available from `http://www.cs.ukc.ac.uk/pubs/1999/986/`.
41. J.-G. Smaus. Proving termination of input-consuming logic programs. In D. De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming*, pages 335–349, Las Cruces, New Mexico, USA, 1999. The MIT Press.
42. J.-G. Smaus, P. M. Hill, and A. M. King. Termination of logic programs with `block` declarations running in several modes. In C. Palamidessi, editor, *Proceedings of the 10th Symposium on Programming Language Implementations and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 73–88, Pisa, Italy, 1998. Springer-Verlag.
43. 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.

44. K. Ueda and K. Furukawa. Transformation rules for GHC Programs. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 582–591, Tokyo, Japan, 1988. Institute for New Generation Computer Technology, Tokyo, OHMSHA Ltd. Tokyo and Springer-Verlag.
45. K. Ueda and M. Morita. Moded flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.
46. 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.