Declarative Interpretations Reconsidered

Krzysztof R. Apt
Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
and
Faculty of Mathematics and Computer Science
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands
apt@cwi.nl

Maurizio Gabbrielli
Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
gabbri@cwi.nl

Abstract

Three semantics have been proposed as the most promising candidates for a declarative interpretation for logic programs and pure Prolog programs: the least Herbrand model, the least term model, i.e. the $\mathcal{C}$-semantics, and the $\mathcal{S}$-semantics. Previous results show that a strictly increasing information ordering between these semantics exists for the class of all programs. In particular, the $\mathcal{S}$-semantics allows us to model computed answer substitutions, which is not the case for the other two.

We study here the relationship between these three semantics for specific classes of programs. We show that for a large class of programs (which is Turing complete) these three semantics are isomorphic. As a consequence, given a query, we can extract from the least Herbrand model of the program all computed answer substitutions. This result is applied to propose a method for proving partial correctness of programs based on the least Herbrand model.

1 Introduction

1.1 Motivation

The basic question we are trying to answer in this paper is: can one reason about partial correctness (that is about the computed answer substitutions) of "natural" pure Prolog programs using the least Herbrand semantics? We claim that the answer to this question is affirmative by showing that many logic and pure Prolog programs satisfy a property which implies that various declarative semantics of them are isomorphic.

Usually the declarative semantics is identified with the least Herbrand model. When considering the class of all logic programs there are a number of problems associated with this choice. First, this model depends on the underlying first-order language. For certain choices of this language this model is equivalent with the least

term model, and for others not. Secondly, in general it matches the procedural interpretation of logic programs only for ground queries. So the procedural behaviour of the program cannot be completely "retrieved" from this model.

The least term model of Clark [6] (or $\mathcal{C}$-semantics of Falaschi et al. [8]) is another natural candidate for the declarative semantics, and in fact it has been successfuly used in the probably most elegant and compact proof of the strong completeness of the SLD-resolution due to Stärk [12]. However, it shares with the least Herbrand model the same deficiencies.

The last choice is the $\mathcal{S}$-semantics proposed by Falaschi et al. in [7]. This semantics provides a precise match with the procedural interpretation of logic programs. So it captures completely the procedural behaviour of the program. However, for specific programs it is rather laborious to construct and difficult to reason about.

We show here that for a large class of programs, called subsumption free programs, these three semantics are in fact isomorphic. This allows us to reason about partial correctness of subsumption free programs using the least Herbrand model. To prove that a program is subsumption free we apply a result of Maher and Ramakrishnan [10]. Using it we checked that several standard pure Prolog programs are subsumption free.

1.2 A Word on Terminology

In principle, we use the standard notation of logic programming. We consider here finite programs and queries w.r.t. a first-order language defined by a signature Σ. Given two expressions E_1, E_2, we say that E_1 *is more general than* E_2, and write $E_1 \leq E_2$, if there exist a substitution θ such that $E_1\theta = E_2$. $\leq$ is called the subsumption ordering. If $E_1 \leq E_2$ but not $E_2 \leq E_1$, we write $E_1 < E_2$, and when both $E_1 \leq E_2$ and $E_2 \leq E_1$, we say that E_1 and E_2 are *variants*. Finally we denote by $Var(E)$ the set of all variables occurring in the expression E.

A substitution if called *grounding* if all terms in its range are ground. A substitution is called a *renaming* if it is a permutation of the variables in its domain. We say that substitutions θ_1 and θ_2 are *variants* if for some renaming η we have $\theta_1 = \theta_2\eta$. Below we shall freely use the well-known result that all mgu's of two expressions are variants and that E_1 and E_2 are variants iff for some renaming η we have $E_1 = E_2\eta$. Further, we denote by $\mathcal{B}$ the set of all atoms (the *base* of the language) and by $\mathcal{B}_\mathcal{H}$ the set of all ground atoms.

For a number of reasons, we found it more convenient to work here with the concept of a query, correct and computed instance, and most general instance, instead of, respectively, the concepts of a goal, correct and computed answer substitution, and most general unifier.

In short, a query is a finite sequence of atoms, denoted by letters $Q, \mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$. Given a program P, Q' is a *correct instance* of Q, if $P \models Q'$ and $Q' = Q\theta$ for a substitution θ; Q' is a *computed instance* of Q, if there exists a successful SLD-derivation of Q with a computed answer substitution θ such that $Q' = Q\theta$.

Our interest here is in finding for a given program P the set of computed instances of a query. In analogy to the case of imperative programs, we write $\{Q\}\ P\ \mathcal{Q}$ to denote the fact that $\mathcal{Q}$ is the set of computed instances of the query Q, and denote the set of computed instances of the query Q by $sp(Q, P)$ (for *strongest postcondition* of Q w.r.t. P). Given two queries Q and Q' we write

$$mgi(Q, Q') = \{Q\theta \mid \theta \text{ is an mgu of } Q \text{ and } Q'\}.$$

So $mgi(Q, Q')$ is the set of most general instances of Q and Q'.

A query is called *separated* if the atoms forming it are pairwise variable disjoint. Given a set of atoms I we denote by I^* the set of separated queries formed from the atoms of I. Given a query Q and a set of atoms I we write

$$mgi(Q, I) = \{Q\theta \mid \exists Q' \in I^*(\, Var(Q) \cap Var(Q') = \emptyset \text{ and } \theta \text{ is an mgu of } Q \text{ and } Q' \,)\}.$$

So $mgi(Q, I)$ is the set of most general instances of Q and any query from I^* variable disjoint with Q. Finally, an atom is called *pure* if it is of the form $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are different variables.

2 Background - Three Declarative Semantics

Three semantics of logic programs, each yielding a single model, were introduced in the literature and presented as "declarative". We review them now briefly and discuss their positive and problematic aspects.

2.1 The Least Herbrand Model, or $\mathcal{M}$-semantics

This semantics was introduced by van Emden and Kowalski [15]. It associates with each program its least Herbrand model. Identifying each Herbrand model with the set of ground atoms true in it, we can equivalently define this semantics as

$$\mathcal{M}(P) \;=\; \{A \in \mathcal{B}_H \mid P \models A\}.$$

As is well-known this semantics completely characterizes the operational behaviour of a program on ground queries because (see Apt and van Emden [4]), for a ground Q a successful SLD-derivation of Q exists iff $Q \in \mathcal{M}(P)^*$. However, for non-ground queries the situation changes as the following example of Falaschi et al. in [7] shows.

Example 2.1 Consider the two programs $P_1 = \{\texttt{p(X).}\}$ and $P_2 = \{\texttt{p(a).}\ ,\ \texttt{p(X).}\}$. Then $\mathcal{M}(P_1) = \mathcal{M}(P_2)$ but the query $\texttt{p(X)}$ yields different computed answer substitutions w.r.t. to each program. □

So in general, the $\mathcal{M}$-semantics is not a function of the operational behaviour of a program.

2.2 The Least Term Model, or $\mathcal{C}$-semantics

This semantics was introduced by Clark [6] and more extensively studied in Falaschi et al. [8]. It associates with each program its least term model. Identifying each term model with the set of atoms true in it, we can equivalently define this semantics as

$$\mathcal{C}(P) \;=\; \{A \in \mathcal{B} \mid P \models A\}.$$

As we shall see in Section 4, when the signature contains infinitely many constants, this semantics is equivalent to $\mathcal{M}$-semantics, so it cannot model the operational behaviour of a program either.

2.3 $\mathcal{S}$-semantics

This semantics was introduced in Falaschi et al. [7]. Its aim is to provide a *precise match* between the procedural and declarative interpretation of logic programs. Ideally, we would like to be able to "reconstruct" the procedural interpretation from the declarative one. Now, a procedural interpretation of a program P can be identified with the set of all pairs (Q, θ) where θ is a computed answer substitution for Q, or, equivalently with the set of all statements of the form $\{Q\}\ P\ Q$.

The $\mathcal{S}$-semantics assigns to a program P the set of atoms [1]

$$\mathcal{S}(P) = \{A \in \mathcal{B} \mid A \text{ is a computed instance of a pure atom}\}.$$

It seems at first sight that the restriction to pure atoms results in a "loss of information" and as a result the operational interpretation cannot be reconstructed from $\mathcal{S}(P)$. But it is not so, as the following theorem of Falaschi et al. [7] shows.

Theorem 2.2 (Strong Completeness) *For a program P and a query Q*

$$\{Q\}\ P\ mgi(Q, \mathcal{S}(P)).$$

□

Consequently, by the form of $\mathcal{S}(P)$ we have

Corollary 2.3 (Full abstraction) *For all programs P_1, P_2*

$$\mathcal{S}(P_1) = \mathcal{S}(P_2) \text{ iff } sp(Q, P_1) = sp(Q, P_2) \text{ for all queries } Q.$$

□

An important property of the $\mathcal{S}$-semantics is that it can be defined by means of a fixpoint construction. More precisely, Falaschi et al. [7] introduced the following operator on term interpretations

$$T_P^{\mathcal{S}}(I) = \{H\theta \mid \exists\ \mathbf{B}, \mathbf{C}\ (H \leftarrow \mathbf{B} \in P,\ \mathbf{C} \in I^*,\ Var(H \leftarrow \mathbf{B}) \cap Var(\mathbf{C}) = \emptyset,\ \theta \text{ is an mgu of } \mathbf{B} \text{ and } \mathbf{C})\}$$

and proved the following.

Theorem 2.4

(i) $T_P^{\mathcal{S}}$ *is continuous on the complete lattice of term interpretations ordered with* $\subseteq$.

(ii) $\mathcal{S}(\mathcal{P})$ *is the least fixpoint and the least pre-fixpoint of* $T_P^{\mathcal{S}}$.

(iii) $\mathcal{S}(\mathcal{P}) = T_P^{\mathcal{S}} \uparrow \omega$. □

3 Relating Them

In what follows we wish to clarify the relationship between these three semantics for various classes of programs. To this end we introduce the following definition, where we view semantics as a function from the considered class of programs to some further unspecified semantic domain $\mathcal{D}$.

[1] In the original proposal actually the sets of equivalence classes of atoms w.r.t. to the "variant of" relation are considered. We found it more convenient to work with the above definition.

Definition 3.1 Consider a class of programs $\mathbf{C}$. We say that two semantics $\mathcal{S}_1 : \mathbf{C} \rightarrow \mathcal{D}_1$ and $\mathcal{S}_2 : \mathbf{C} \rightarrow \mathcal{D}_2$ are *isomorphic* on $\mathbf{C}$ iff there exist two functions, $\phi_1 : Range(\mathcal{S}_1) \rightarrow Range(\mathcal{S}_2)$ and $\phi_2 : Range(\mathcal{S}_2) \rightarrow Range(\mathcal{S}_1)$ such that, for any program $P \in \mathbf{C}$

$$\mathcal{S}_1(P) = \phi_2(\mathcal{S}_2(P)) \text{ and } \mathcal{S}_2(P) = \phi_1(\mathcal{S}_1(P)).$$

□

Alternatively, two semantics $\mathcal{S}_1 : \mathbf{C} \rightarrow \mathcal{D}_1$ and $\mathcal{S}_2 : \mathbf{C} \rightarrow \mathcal{D}_2$ are isomorphic on $\mathbf{C}$ iff there exists a bijection $\phi : Range(\mathcal{S}_1) \rightarrow Range(\mathcal{S}_2)$ such that, for any program $P \in \mathbf{C}$, $\mathcal{S}_2(P) = \phi(\mathcal{S}_1(P))$.

Every semantics $\mathcal{T}$ for $\mathbf{C}$ induces an equivalence relation $\approx_{\mathcal{T}}$ on programs from $\mathbf{C}$ defined by $P_1 \approx_{\mathcal{T}} P_2$ iff $\mathcal{T}(P_1) = \mathcal{T}(P_2)$. Note that the notion of isomorphism can be also equivalently given in terms of equivalences, by defining two semantics isomorphic on $\mathbf{C}$ if they induce the same equivalence relation on $\mathbf{C}$. When constructing isomorphisms between the semantics the following operators will be useful.

Definition 3.2 Let I be a set of atoms. We define

(i) $Variant(I) = \{A \in \mathcal{B} \mid \exists B \in I \text{ s.t. } B \leq A \text{ and } A \leq B\}$, the set of variants,

(ii) $Up(I) = \{A \in \mathcal{B} \mid \exists B \in I \text{ s.t. } B \leq A\}$, the set of instances,

(iii) $Ground(I) = \{A \in \mathcal{B}_H \mid \exists B \in I \text{ s.t. } B \leq A\}$, the set of ground instances,

(iv) $Min(I) = \{A \in I \mid \neg\exists B \in I \text{ s.t. } B < A\}$, the set of minimal (i.e. most general) elements,

(v) for I a set of ground atoms
$True(I) = \{A \in \mathcal{B} \mid I \models A\}$, the set of atoms true in the Herbrand interpretation I.

□

Note that $Variant, Up, Ground$ and Min are all idempotent. Moreover, the following clearly holds.

Note 3.3 *For all I, $Min(Up(I)) = Min(I)$.* □

4 Relating $\mathcal{M}$-semantics and $\mathcal{C}$-semantics

We begin by clarifying the relationship between $\mathcal{M}(P)$ and $\mathcal{C}(P)$. The following result is an immediate consequence of the definitions.

Note 4.1 $\mathcal{M}(P) = Ground(\mathcal{C}(P))$. □

So the $\mathcal{M}$-semantics can be reconstructed from the $\mathcal{C}$-semantics. The converse does not hold in general as the following argument due to Falaschi et al. [8] shows.

Example 4.2 Consider the two programs $P_1 = \{\texttt{p(X).}\}$ and $P_2 = \{\texttt{p(a).}, \texttt{p(b).}\}$ defined w.r.t. the language with the signature $\Sigma = \{\texttt{a/0}, \texttt{b/0}\}$. Then $\mathcal{M}(P_1) = \mathcal{M}(P_2) = \{\texttt{p(a)}, \texttt{p(b)}\}$, while $\mathcal{C}(P_1) = \{\texttt{p(X)}, \texttt{p(a)}, \texttt{p(b)}\}$ and $\mathcal{C}(P_2) = \{\texttt{p(a)}, \texttt{p(b)}\}$.

□

In case the signature contains infinitely many constants, the situation changes, as the following result due to Maher [9] shows.

Theorem 4.3 *Assume that* Σ *contains infinitely many constants. Then* $\mathcal{C}(P) = True(\mathcal{M}(P))$.

Proof. We provide here an alternative, direct proof based on the theory of SLD-resolution. The implication $\mathcal{C}(P) \subseteq True(\mathcal{M}(P))$ always holds, since $\mathcal{M}(P)$ is a model of P. Take now $A \in True(\mathcal{M}(P))$. Let $x_1, \ldots, x_n$ be the variables of A and $c_1, \ldots, c_n$ distinct constants which do not appear in P or A. Let $\theta = \{x_1/c_1, \ldots, x_n/c_n\}$. Then $A\theta \in \mathcal{M}(P)$. By the completeness of SLD-resolution there exists a successful SLD-derivation of $A\theta$ with the empty computed answer substitution. By replacing in it c_i by x_i for $i \in [1, n]$ we get a successful SLD-derivation of A with the empty computed answer substitution. Now by the soundness of SLD-resolution $A \in \mathcal{C}(P)$.

□

Consequently, when the signature contains infinitely many constants, the semantics $\mathcal{M}(P)$ and $\mathcal{C}(P)$ are isomorphic. We shall exploit this fact later.

5 Relating $\mathcal{C}$-semantics and $\mathcal{S}$-semantics

Next, we clarify the relationship between $\mathcal{C}(P)$ and $\mathcal{S}(P)$. First, we have the following result of Falaschi et al. [8].

Theorem 5.1 $\mathcal{C}(P) = Up(\mathcal{S}(P))$. □

So the $\mathcal{C}$-semantics can be reconstructed from the $\mathcal{S}$-semantics. The converse does not hold in general as the following argument due to Falaschi et al. [7] shows.

Example 5.2 Consider the programs P_1 and P_2 of Example 2.1.
Then $\mathcal{C}(P_1) = \mathcal{C}(P_2) = Up(\{\mathtt{p(X)}\})$, while $\mathcal{S}(P_1) = Variant(\{\mathtt{p(X)}\})$ and $\mathcal{S}(P_2) = Variant(\{\mathtt{p(X)}, \mathtt{p(a)}\})$. Note that the signature of the language was immaterial here. □

Thus on the class of all programs the $\mathcal{C}$-semantics and the $\mathcal{S}$-semantics. are not isomorphic. In what follows we show that for a large class of programs they are in fact isomorphic. First, we have the following result.

Lemma 5.3 $Min(\mathcal{C}(P)) \subseteq \mathcal{S}(P)$.

Intuitively, it states that all most general atoms true in $\mathcal{C}(P)$ belong to $\mathcal{S}(P)$.

Proof. By Theorem 5.1 $Min(\mathcal{C}(P)) = Min(Up(\mathcal{S}(P)))$ and the claim follows by Note 3.3, since for all I we have $Min(I) \subseteq I$.

□

In general, the converse inclusion does not hold.

Example 5.4 Consider the following program $P = \{\mathtt{p(a).},\ \mathtt{p(X).}\}$ defined w.r.t. the language with the signature $\Sigma = \{\mathtt{a}/0\}$. Then $\mathcal{S}(P) = Variant(\{\mathtt{p(Y)}\}) \cup \{\mathtt{p(a)}\}$, whereas $Min(\mathcal{C}(P)) = Variant(\{\mathtt{p(Y)}\})$. □

A closer examination of the situation reveals the following. By the soundness of SLD-resolution we always have $\mathcal{S}(P) \subseteq \mathcal{C}(P)$. The above example shows that the stronger inclusion $\mathcal{S}(P) \subseteq Min(\mathcal{C}(P))$ does not need to hold. The reason is that

$\mathcal{S}(P)$ can contain a pair A, B such that A strictly subsumes B (i.e. $A < B$). This cannot happen when $\mathcal{S}(P)$ contains only minimal elements. So we are brought to the following definition due to Maher and Ramakrishnan [10].

Definition 5.5 A set of atoms I is called *subsumption free* if $Min(I) = I$. A program P is called *subsumption free* if $\mathcal{S}(P)$ is. □

We now show that that the notion of a subsumption free program is a key for establishing the converse of Lemma 5.3.

Theorem 5.6 *$\mathcal{S}(P) = Min(\mathcal{C}(P))$ iff P is subsumption free.*

Proof. ($\Rightarrow$) We have

$$\begin{array}{rcl} Min(\mathcal{S}(P)) & = & \{\text{assumption}\} \\ Min(Min(\mathcal{C}(P))) & = & \{\text{idempotence of } Min\} \\ Min(\mathcal{C}(P)) & = & \{\text{assumption}\} \\ \mathcal{S}(P). & & \end{array}$$

($\Leftarrow$) We have

$$\begin{array}{rcl} \mathcal{S}(P) & = & \{\text{assumption}\} \\ Min(\mathcal{S}(P)) & = & \{\text{Note 3.3}\} \\ Min(Up(\mathcal{S}(P))) & = & \{\text{Theorem 5.1}\} \\ Min(\mathcal{C}(P)). & & \end{array}$$

□

Consequently, the $\mathcal{C}$-semantics and $\mathcal{S}$-semantics are isomorphic on subsumption free programs. Additionally, when the signature contains infinitely many constants, all three semantics are isomorphic. Combining Theorems 2.2, 4.3 and 5.6 we thus obtain.

Corollary 5.7 *Assume that Σ contains infinitely many constants. Then for a subsumption free program P and a query Q*

$$\{Q\}\ P\ mgi(Q, Min(True(\mathcal{M}(P)))).$$

□

It shows that partial correctness of subsumption free programs can be fully reconstructed from the least Herbrand model, using unification. In the next section we shall identify a smaller class of programs for which this characterization of partial correctness does not involve unification.

Of course, if we do not make any assumption on the class of programs $\mathbf{C}$, subsumption freedom is only a sufficient condition for the isomorphism of the $\mathcal{C}$-semantics and $\mathcal{S}$-semantics. Indeed, when the class of programs consists of just the program from Example 5.4, which is not subsumption free, then the $\mathcal{C}$-semantics and $\mathcal{S}$-semantics are obviously isomorphic. However, for a "reasonably large" class of programs subsumption freedom turns out to be also a necessary condition for isomorphism of programs.

Definition 5.8 A class of programs $\mathbf{C}$ is *$\mathcal{S}$-closed* if for every program P in $\mathbf{C}$ every finite subset of $\mathcal{S}(P)$ is in $\mathbf{C}$. □

Indeed, we have the following result.

Note 5.9 *For an $\mathcal{S}$-closed class $\mathbf{C}$ of programs, the $\mathcal{C}$-semantics and $\mathcal{S}$-semantics are isomorphic on $\mathbf{C}$ iff $\mathbf{C}$ is a class of subsumption free programs.*

Proof. ($\Rightarrow$) Suppose that some $P \in \mathbf{C}$ is not subsumption free. Then for some atoms $A, B \in \mathcal{S}(P)$ we have $A < B$. By the definition of $\mathcal{S}$-closedness both $P_1 = \{A, B\}$ and $P_2 = \{A\}$ are in $\mathbf{C}$. Now $\mathcal{C}(P_1) = Up(\{A, B\}) = Up(\{A\}) = \mathcal{C}(P_2)$, whereas $\mathcal{S}(P_1) = Variant(\{A, B\}) \neq Variant(\{A\}) = \mathcal{S}(P_2)$. Contradiction.
($\Leftarrow$) This is the contents of Theorems 5.1 and 5.6.

□

This shows that the notion of subsumption freedom is crucial for our considerations. In what follows we provide some means of establishing that a program is subsumption free.

6 $\mathcal{S}$-Unification Free Programs

We begin by studying a subclass of subsumption free programs.

Definition 6.1 A program P is called *$\mathcal{S}$-unification free* iff $\mathcal{S}(P)$ does not contain a pair of non-variant unifiable atoms. □

We prefer to use the qualification "$\mathcal{S}$-" in order to avoid confusion with the class of unification free programs studied in Apt and Etalle [2]. Clearly, $\mathcal{S}$-unification freedom implies subsumption freedom, since $\mathcal{S}(\mathcal{P})$ is closed under renaming and $A < B$ implies that A and a variant B' of B are non-variant and unifiable. The converse does not hold.

Example 6.2 Consider the following program $P = \{\mathtt{p(X,a).},\ \mathtt{p(a,X).}\}$ defined w.r.t. the language with the signature $\Sigma = \{\mathtt{a}/0\}$.
Then $\mathcal{S}(P) = Variant(\{\mathtt{p(X,a)}, \mathtt{p(a,X)}\})$, so P is not $\mathcal{S}$-unification free. However, it is clearly subsumption free, because the atoms $\mathtt{p(X,a)}$ and $\mathtt{p(a,X)}$ are not comparable in the subsumption ordering. □

The following theorem summarizes the difference between the subsumption free and $\mathcal{S}$-unification free programs in a succinct way. Let us extend the Min operator in an obvious way to sets of queries.

Theorem 6.3

(i) P is subsumption free iff for all pure atoms A, $Min(sp(A, P)) = sp(A, P)$.

(ii) P is $\mathcal{S}$-unification free iff for all queries Q, $Min(sp(Q, P)) = sp(Q, P)$.

Proof.
(i) Note that for some variables $x_1, x_2, \ldots$, $\mathcal{S}(P)$ is a disjoint union of sets of the form $sp(p(x_1, \ldots, x_{arity(p)}), P)$ and that atoms belonging to different such sets are incomparable in the $\leq$ ordering. Thus $Min(\mathcal{S}(P))$ is a disjoint union of sets of the form $Min(sp(p(x_1, \ldots, x_{arity(p)}), P))$.

(ii) ($\Rightarrow$) Consider two computed instances Q_1 and Q_2 of Q. By Theorem 2.2 there exist $\mathbf{C}_1$ and $\mathbf{C}_2$ in $\mathcal{S}(P)^*$ such that for $i \in [1, 2]$ Q and $\mathbf{C}_i$ are variable disjoint and

$$Q_i \in mgi(Q, \mathbf{C}_i). \tag{1}$$

In particular $\mathbf{C}_1 \leq Q_1$ and $\mathbf{C}_2 \leq Q_2$.

Suppose now that $Q_1 < Q_2$. Then $\mathbf{C}_1 \leq Q_2$, so Q_2 is an instance of both $\mathbf{C}_1$ and $\mathbf{C}_2$. Since we may assume that $\mathbf{C}_1$ and $\mathbf{C}_2$ are variable disjoint, we conclude that $\mathbf{C}_1$ and $\mathbf{C}_2$ are unifiable. By assumption about P and the fact that $\mathbf{C}_1$ and $\mathbf{C}_2$ are separated queries, $\mathbf{C}_1$ and $\mathbf{C}_2$ are variants. This implies by (1) that Q_1 and Q_2 are variants, as well. Contradiction.

($\Leftarrow$) Suppose that $\mathcal{S}(P)$ does contain a pair A, B of non-variant unifiable atoms. Let $C \in mgi(A, B)$. Then $A \leq C$ and $B \leq C$ and at least one of these subsumption relations, say the first one, is strict. So $A < C$. Take now a variant A' of A variable disjoint with A and B. By Theorem 2.2 $A, C \in sp(A', P)$. So $Min(sp(A', P)) \neq sp(A', P)$. Contradiction.

□

For $\mathcal{S}$-unification free programs we can simplify the formulation of Corollary 5.7.

Corollary 6.4 *For a $\mathcal{S}$-unification free program P and a query Q*

$$\{Q\}\ P\ Min(\{Q\theta \mid P \models Q\theta\}).$$

Proof. This result follows from Theorem 6.3(ii) and the following two claims.

Claim 1 *For an arbitrary program P and a query Q*

$$Min(\{Q\theta \mid P \models Q\theta\}) \subseteq sp(Q, P) \subseteq \{Q\theta \mid P \models Q\theta\}.$$

Proof. Take $Q_1 \in Min(\{Q\theta \mid P \models Q\theta\})$. By the Strong Completeness of SLD-resolution there exists a computed instance Q_2 of Q_1 such that $Q_2 \leq Q_1$. By the choice of Q_1, $P \models Q_2$, so by the minimality of Q_1, Q_1 and Q_2 are variants. Thus Q_1 is also a computed instance of Q, i.e. $Q_1 \in sp(Q, P)$. □

Claim 2 *For sets of queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$, if $Min(\mathcal{Q}_1) \subseteq \mathcal{Q}_2 \subseteq \mathcal{Q}_1$ and $Min(\mathcal{Q}_2) = \mathcal{Q}_2$, then $\mathcal{Q}_2 = Min(\mathcal{Q}_1)$.*

Proof. Immediate.

□

So for $\mathcal{S}$-unification free programs the sets of computed instances can be defined without the use of unification. In Corollary 6.4 we can always replace "$P \models$" by "$\mathcal{C}(P) \models$", and also by "$\mathcal{M}(P) \models$" if Σ contains infinitely many constants.

Maher and Ramakrishnan [10] studied subsumption free programs in the context of the bottom up computation in deductive databases and showed that for these programs this computation can be performed more efficiently. They also provided a method allowing us to conclude that a program is $\mathcal{S}$-unification free, so a fortiori subsumption free. Using this method they proved that the class of $\mathcal{S}$-unification free programs is Turing complete.

Their method is equally applicable in our situation. To formulate it the following notation is useful. By $hground(P)$ we denote the set of instances of clauses of P whose head is ground. Given a Herbrand interpretation I and a query $\mathbf{B}$ we write

$$I \models \exists \leq 1 \mathbf{B}$$

if at most one sequence of ground $t_1, \ldots, t_n$ exists such that $I \models \mathbf{B}\{x_1/t_1, \ldots, x_n/t_n\}$, where $x_1, \ldots, x_n$ are the variables occurring in $\mathbf{B}$.

Theorem 6.5 (Maher and Ramakrishnan [10]) *Suppose that the following conditions hold for a program P:*

SEM1. *If c, d are different clauses in P, then no pair $A \in T^{\mathcal{S}}_{\{c\}}(\mathcal{S}(P))$ and $B \in T^{\mathcal{S}}_{\{d\}}(\mathcal{S}(P))$ is unifiable.*

SEM2. *For every clause $H \leftarrow \mathbf{B}$ in hground(P)*

$$\text{if } \mathcal{M}(P) \models H \text{ then } \mathcal{M}(P) \models \exists \leq 1 \mathbf{B}.$$

Then P is S-unification free.

Proof. To keep the paper self-contained we provide here a direct proof. By Theorem 2.4(iii) it suffices to show that, for $n \geq 0$, $T^{\mathcal{S}}_P \uparrow n$ does not contain a pair of non-variant unifiable atoms. The proof is by induction on n.
($n = 0$) Obvious.
($n > 0$). Denote $T^{\mathcal{S}}_P \uparrow n$ by I and consider $A, B \in T^{\mathcal{S}}_P(I)$. Two cases arise.

Case 1 A and B are generated by the different clauses, say c and d. Then $A \in T^{\mathcal{S}}_{\{c\}}(I)$ and $B \in T^{\mathcal{S}}_{\{d\}}(I)$ and the claim follows by condition SEM1 since by Theorem 2.4 $I \subseteq \mathcal{S}(P)$ and $T^{\mathcal{S}}_P$ is monotonic.

Case 2 A and B are generated by the same clause, say $H \leftarrow \mathbf{B}$. Then for some $\mathbf{C}_1 \in I^*$, $\mathbf{C}_2 \in I^*$ and ϑ_1, ϑ_2

$$A = H\vartheta_1, \ \mathit{Var}(H \leftarrow \mathbf{B}) \cap \mathit{Var}(\mathbf{C}_1) = \emptyset, \ \vartheta_1 \text{ is an mgu of } \mathbf{B} \text{ and } \mathbf{C}_1, \tag{2}$$

$$B = H\vartheta_2, \ \mathit{Var}(H \leftarrow \mathbf{B}) \cap \mathit{Var}(\mathbf{C}_2) = \emptyset, \ \vartheta_2 \text{ is an mgu of } \mathbf{B} \text{ and } \mathbf{C}_2 \tag{3}$$

Suppose A and B are unifiable. Then there exists a grounding η whose domain includes the variables of $(H \leftarrow \mathbf{B})\vartheta_1$ and $(H \leftarrow \mathbf{B})\vartheta_2$, such that both $\vartheta_1\eta$ and $\vartheta_2\eta$ are grounding and $H\vartheta_1\eta = H\vartheta_2\eta$. So $\vartheta_1\eta$ and $\vartheta_2\eta$ coincide on the variables of H. Denote their common restriction to $\mathit{Var}(H)$ by δ. Then $(H \leftarrow \mathbf{B})\delta$ is in hground(P).

By the soundness of SLD-resolution $\mathcal{M}(P) \models H\delta$, since $A \in \mathcal{S}(P)$ and $H\delta = A\vartheta_1\eta$. Thus by condition SEM2

$$\mathcal{M}(P) \models \exists \leq 1 \ \mathbf{B}\delta. \tag{4}$$

Now, for some grounding δ_1 and δ_2 we have $\vartheta_1\eta = \delta \cup \delta_1$ and $\vartheta_2\eta = \delta \cup \delta_2$, so $\vartheta_1\eta = \delta\delta_1$ and $\vartheta_2\eta = \delta\delta_2$. This implies by (2) that

$$\mathbf{B}\delta\delta_1 = \mathbf{B}\vartheta_1\eta = \mathbf{C}_1\vartheta_1\eta = \mathbf{C}_1\delta\delta_1, \tag{5}$$

and similarly by (3)

$$\mathbf{B}\delta\delta_2 = \mathbf{C}_2\delta\delta_2. \tag{6}$$

Again by the soundness of SLD-resolution $\mathcal{M}(P) \models \mathbf{C}_1\delta\delta_1$ and $\mathcal{M}(P) \models \mathbf{C}_2\delta\delta_2$, since $\mathbf{C}_1 \in \mathcal{S}(P)^*$ and $\mathbf{C}_2 \in \mathcal{S}(P)^*$. By (4) $\mathbf{B}\delta\delta_1 = \mathbf{B}\delta\delta_2$, so by (5) and (6) $\mathbf{C}_1\delta\delta_1 = \mathbf{C}_2\delta\delta_2$. Thus $\mathbf{C}_1$ and $\mathbf{C}_2$ are unifiable, since we may assume that they are variable disjoint. By the induction hypothesis and the fact that $\mathbf{C}_1$ and $\mathbf{C}_2$ are separated, $\mathbf{C}_1$ and $\mathbf{C}_2$ are variants. Thus by (2) and (3) $H\vartheta_1$ and $H\vartheta_2$, i.e. A and B are variants. This concludes the proof of the induction step.

□

In certain situations the conditions of the above theorem can be ensured by means of syntactic restrictions. Namely, condition SEM1 is obviously implied by condition

SYN1. If $H_1 \leftarrow \mathbf{B}_1$ and $H_2 \leftarrow \mathbf{B}_2$ are different clauses in P, then H_1 and H_2 do not unify,

and condition SEM2 is automatically satisfied when condition

SYN2. If $H \leftarrow \mathbf{B} \in P$, then $Var(\mathbf{B}) \subseteq Var(H)$

holds.

It is worth noting that an immediate proof of Turing completeness for $\mathcal{S}$-unification free programs can be obtained by using the encoding of two register machines into pure logic programs given in Shepherdson [11]. In fact, conditions SYN1 and SYN2 readily apply to programs obtained by such an encoding. In the next section we assess the applicability of Theorem 6.5.

7 Applications

We first provide 4 illustrative uses of Theorem 6.5.

Example 7.1
(i) Consider the APPEND program:

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

Here the syntactic conditions SYN1 and SYN2 readily apply.

(ii) Consider the SUFFIX program:

```
suffix(Xs, Xs).
suffix(Xs,[Y | Ys]) ← suffix(Xs,Ys).
```

Note that the heads of the clauses unify, so we cannot use condition SYN1. To prove condition SEM1 we reason as follows. Denote by OCC the set of atoms of the form `suffix(Z,`t_Z`)` where `Z` is a variable and t_Z a term containing `Z`. By definition of $T_P^{\mathcal{S}}$, $T^{\mathcal{S}}_{\texttt{SUFFIX}}(OCC) \subseteq OCC$, i.e. OCC is a pre-fixpoint of $T^{\mathcal{S}}_{\texttt{SUFFIX}}$. By Theorem 2.4 $\mathcal{S}(\texttt{SUFFIX}) \subseteq OCC$. So because of the occur-check `suffix(Xs, Xs)` does not unify with any $A \in OCC$ of the form `suffix(Z,t)` with `t` a proper term. Thus SEM1 holds.

The clauses of SUFFIX do not contain local variables, so condition SYN2 applies.

(iii) Consider now the naive REVERSE program:

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

augmented by the "well-typed" APPEND_T program:

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

list([H | Ts]) ← list(Ts).
list([].
```

The heads of different clauses do not unify, so condition SYN1 applies. However, due to presence of the local variable `Ys` in the second clause, condition SYN2 does not apply. To prove condition SEM2 we analyze the least Herbrand model

$\mathcal{M}$(REVERSE). Using the techniques of Apt and Pedreschi [3] it is straightforward to check that

$$\begin{aligned}\mathcal{M}(\texttt{REVERSE}) &= \{\texttt{reverse(s,t)} \mid \texttt{s,t} \text{ are ground lists and } \texttt{t=rev(s)}\} \\ &\cup\ \mathcal{M}(\texttt{APPEND\_T}) \qquad \text{where}\end{aligned}$$

$$\begin{aligned}\mathcal{M}(\texttt{APPEND\_T}) &= \{\texttt{append\_t(s,t,u)} \mid \texttt{s,t,u} \text{ are ground lists and } \texttt{s*t=u}\} \\ &\cup\ \{\texttt{list(s)} \mid \texttt{s} \text{ is a ground list }\}\end{aligned}$$

where given a list s, rev(s) denotes its reverse, and $*$ denotes the operation of concatenating two lists. Take now an instance
reverse([x | xs], zs) ← reverse(xs,ys), append_t(ys,[x],zs)
of the second clause with reverse([x|xs], zs) ground and in $\mathcal{M}$(REVERSE). Then reverse(xs, ys) $\in \mathcal{M}$(REVERSE) implies ys = rev(xs), so condition SEM2 holds for this clause. For other clauses condition SYN2 applies. We conclude that REVERSE is $\mathcal{S}$-unification free.

(iv) Finally, consider the following program HANOI from Sterling and Shapiro [13] which, for the query hanoi(n,a,b,c,Moves), solves the "Towers of Hanoi" problem with n disks and three pegs a, b and c giving the sequence of moves forming the solution in Moves:

```
hanoi(s(0),A,B,C,[A to B]).
hanoi(s(N),A,B,C,[A to B]) ←
   hanoi(N,A,C,B,Ms1)
   hanoi(N,C,B,A ,Ms2)
   append_t(Ms1,[A to B|Ms2],Moves).
```

augmented by the APPEND_T program.

Note that conditions SYN1 and SYN2 do not apply here. First observe that for any I, if hanoi(t1,t2,t3,t4,t5) $\in T^{\mathcal{S}}_{\texttt{HANOI}}(I)$ then t1 $\neq 0$. So by Theorem 2.4, hanoi(t1,t2,t3,t4,t5) $\in \mathcal{S}$(HANOI) implies t1 $\neq 0$ and consequently condition SEM1 holds.

To prove SEM2 we use the methodology of Maher and Ramakrishnan [10] based on functional dependencies. First we need a definition.

Definition 7.2 Let p be an n-ary relation symbol. A functional dependency is a construct of the form $p[I \rightarrow J]$ where $I, J \subseteq \{1, \ldots, n\}$. Let M be a set of ground atoms. We say that $p[I \rightarrow J]$ *holds over* M if for all $p(s_1, \ldots, s_n), p(t_1, \ldots, t_n) \in M$, the following implication holds:

$$(\forall i \in I.\ s_i = t_i) \Rightarrow (\forall j \in J.\ s_j = t_j).$$

A set F of functional dependencies holds over M iff each of them holds over M. □

We now show that the set of functional dependencies

$$F = \{\texttt{hanoi}[\{1,2,3,4\} \rightarrow \{5\}], \texttt{append\_t}[\{1,2\} \rightarrow \{3\}]\}$$

holds over $\mathcal{M}$(HANOI). By the fixpoint definition of $\mathcal{M}(P)$, if $A \in \mathcal{M}(P)$ then A is a ground instance of the head of a clause in P. Then a simple syntactic check on the heads of the clauses in HANOI reveals that hanoi$[\{1,2,3,4\} \rightarrow \{5\}]$ holds over $\mathcal{M}$(HANOI). The other functional dependency can be directly established by considering the explicit definition of $\mathcal{M}$(APPEND_T) previously given.

Using the information given by F it is now straightforward to prove the implication required by SEM2. The only clause that we have to consider is the non unit clause for `hanoi`. Consider an instance

```
hanoi(s(n),a,b,c,[a to b]) ← hanoi(n,a,c,b,ms1),hanoi(n,c,b,a,ms2),
                             append_t(ms1,[a to b|ms2],moves)
```

of such a clause with `hanoi(s(n),a,b,c,[a to b])` ground and in $\mathcal{M}$(`HANOI`).

Since `hanoi`$[\{1,2,3,4\} \rightarrow \{5\}]$ holds over $\mathcal{M}$(`HANOI`), if `hanoi(n,a,c,b,ms1)` $\in \mathcal{M}$(`HANOI`) then there exists no `hanoi(n,a,c,b,ms1')` $\in \mathcal{M}$(`HANOI`) such that `ms1` $\neq$ `ms1'`. Analogously for `ms2` and, using the dependency `append_t`$[\{1,2\} \rightarrow \{3\}]$, for `moves`. Consequently, SEM2 holds and `HANOI` is $\mathcal{S}$-unification free.

A general method for establishing functional dependencies on $\mathcal{M}(P)$, based on an extended version of Amstrong axioms (see Ullman [14]), is given in Maher and Ramakrishnan [10]. □

Note that Theorem 6.5 only provides sufficient conditions for $\mathcal{S}$-unification freedom. Indeed, the program {`p(X) ← q(X,Y)., q(a,b)., q(a,c).`} is easily seen to be $\mathcal{S}$-unification free but condition SEM2 does not hold. Moreover, for certain natural programs Theorem 6.5 cannot be used to establish their subsumption freedom, simply because they are not $\mathcal{S}$-unification free. An example is of course the program considered in Example 6.2. But more natural programs exist. In such situations we still can use a direct reasoning.

Example 7.3 Consider the `MEMBER` program:

```
member(X,[X | Xs]).
member(X,[Y | Xs]) ← member(X,Xs).
```

We now prove that `MEMBER` is subsumption free. By Theorem 2.4 it suffices to show that if I is subsumption free then $T^{\mathcal{S}}_{\texttt{MEMBER}}(I)$ is subsumption free. Denote the first clause by c_1 and the second one by c_2. Consider a pair $A_1, A_2 \in T^{\mathcal{S}}_{\texttt{MEMBER}}(I)$. The following two cases arise.

Case 1 $A_1 \in T^s_{\{c_1\}}(I)$ and $A_2 \in T^s_{\{c_2\}}(I)$.

By definition of $T^{\mathcal{S}}_P$, A_1 = `member(X,[X|Xs])`ρ for a renaming ρ and A_2 = `member(X,[Y|Xs])`ϑ where ϑ is an *mgu* of `member(X,Xs)` and B for a B such that `Y`$\notin Var(B)$. This implies `X`$\vartheta \not\equiv$ `Y`ϑ and hence $A_1 \not\leq A_2$ and $A_2 \not\leq A_1$.

Case 2 $A_1, A_2 \in T^s_{\{c_2\}}(I)$.

By definition, A_i = `member(X,[Y|Xs])`ϑ_i where ϑ_i is an *mgu* of `member(X,Xs)` and B_i for $i = 1,2$. Assuming B_i= `member(`t_i`,`l_i`)` we have $\vartheta_i = \{\texttt{X}/t_i, \texttt{Xs}/l_i\}$ (up to renaming). Then the assumption $B_1 \not\leq B_2$ implies `member(X,Xs)` $\vartheta_1 \not\leq$ `member(X,Xs)`ϑ_2 and hence $A_1 \not\leq A_2$. Analogously for the symmetric case.

Note that `MEMBER` is not $\mathcal{S}$-unification free. □

The results contained in the previous sections can be applied to prove partial correctness of logic programs by using the least Herbrand model. Given a program P and a query Q, we wish to prove assertions of the form $\{Q\}\ P\ \mathcal{Q}$. This can be done by performing the steps listed below, which extend a methodology introduced in Apt [1] to the case of "non-ground" inputs (or more precisely to queries with "non-ground" computed instances). We illustrate our technique by means of an example. Consider the program `REVERSE` of Example 7.1 and the query Q = `reverse(s,X)`, where s is a (possibly non-ground) list and X is a variable. In the following, we assume an infinite signature.

1. *Construct* $\mathcal{M}(P)$.

   Usually, the "specification" of the program limited to its ground queries coincides with $\mathcal{M}(P)$. The techniques of Apt and Pedreschi [3] are useful for verifying validity of such a guess.

2. *Prove that* P *is* $\mathcal{S}$*-unification (subsumption) free* (see Example 7.1).

3. *Find a correct instance* Q' *of* Q, i.e. such that $\mathcal{M}(P) \models Q'$. Note that by definition

$$\mathcal{M}(P) \models Q' \text{ iff } Ground(Q') \subseteq \mathcal{M}(P)^*. \tag{7}$$

   In our case, by definition of $\mathcal{M}$(REVERSE), if Q'' is a ground instance of `reverse(s,rev(s))` then $Q'' \in \mathcal{M}$(REVERSE) holds. Therefore by (7)

$$\mathcal{M}(\text{REVERSE}) \models \texttt{reverse(s,rev(s))}.$$

4. *By suitably generalizing from 3. find a minimal correct instance* Q' *of* Q, i.e. such that $\mathcal{M}(P) \models Q\gamma$ implies $Q' \leq Q\gamma$. (In general, find the set of minimal correct instances of Q). Here the following implication which holds for any pair of expressions E_1, E_2 can be useful

$$(\forall \eta \text{ s.t. } (E_1 = E_2)\eta \text{ is ground } . E_1\eta = E_2\eta) \Rightarrow E_1 = E_2. \tag{8}$$

   Assume in our case that

$$\mathcal{M}(\text{REVERSE}) \models \texttt{reverse(s,X)}\gamma.$$

   By (7), for any η s.t. `reverse(s,X)`$\gamma\eta$ is ground, `X`$\gamma\eta = rev($`s`$\gamma\eta) =$ (by definition of rev) $rev($`s`$)\gamma\eta$. Then by (8) `X`$\gamma = rev($`s`$)\gamma$ and hence

$$\texttt{reverse(s,rev(s))} \leq \texttt{reverse(s,X)}\gamma$$

   holds.

5. *Apply Corollary 6.4 (or Corollary 5.7 for programs which are not S-unification free).* For REVERSE we obtain

$$\{\texttt{reverse(s,X)}\} \text{ REVERSE } Variant(\{\texttt{reverse(s,rev(s))}\}).$$

8 Conclusions

We now present a list of example programs from the book of Sterling and Shapiro [13] for which we proved that $\mathcal{S}$-semantics and $\mathcal{M}$-semantics are isomorphic. For each program it is indicated by what method the result was established. For example SEM1-SYN2 means that condition SEM1 of Theorem 6.5 and condition SYN2 following it were used. DP stands for a "direct proof". In all cases condition SEM2 was established by means of the functional dependency analysis.

To deal with programs which use arithmetic relations we assumed that each such relation is defined by infinitely many ground unit clauses which form its true ground instances. Note that such ground unit clauses obviously satisfy the conditions SYN1 and SYN2. It should be noted that the results of this paper hold for programs with infinitely many clauses provided we modify the assumption "the signature has

infinitely many constants" to "the signature has infinitely many constants which do not occur in the program".

Finally, it should be made clear that none of the considered semantics deals with the problem of errors which can arise in presence of arithmetic relations. To handle properly this issue, the results concerning partial correctness (so Corollaries 5.7 and 6.4) have to be restricted to the queries whose evaluation cannot yield an error. Apt [1] provides a method for proving absence of errors for pure Prolog programs augmented by arithmetic relations.

program	page	sub.free	$\mathcal{S}$-unif. free	method
member	45	yes	no	DP
prefix	45	yes	yes	SYN1-SYN2
suffix	45	yes	yes	SEM1-SYN2
naive reverse	48	yes	yes	SYN1-SEM2
reverse_accum.	48	yes	yes	SYN1-SYN2
delete	53	yes	yes	SEM1-SYN2
select	53	yes	no	DP
permutation	55	yes	no	DP
permutation sort	55	yes	no	DP
insertion sort	55	yes	yes	SEM1-SEM2
partition	56	yes	yes	SEM1-SYN2
quicksort	56	yes	yes	SEM1-SEM2
tree_member	58	yes	no	DP
iso_tree	58	yes	yes	SEM1-SYN2
substitute	60	yes	yes	SEM1-SYN2
pre_order	60	yes	yes	SYN1-SEM2
in_order	60	yes	yes	SYN1-SEM2
post_order	60	yes	yes	SYN1-SEM2
polynomial	62	yes	no	DP

This provides a strong indication that for most "natural" pure Prolog programs the $\mathcal{S}$-semantics is isomorphic to the $\mathcal{M}$-semantics. For such programs it is possible to reason about their partial correctness using the least Herbrand model only. This might suggest that $\mathcal{S}$-semantics is not needed. This would be, however, a too hastily drawn conclusion. First of all, $\mathcal{S}$-semantics has other uses than the ones investigated in this paper – for example in the area of abstract interpretations (see e.g. Bossi et al. [5] for an overview). Secondly, to formulate and prove the key results, namely Corollary 5.7 and Theorem 6.5, we did use the $\mathcal{S}$-semantics. It would be interesting to find proofs of these results by means of the $\mathcal{M}$-semantics.

Acknowledgments

We thank the referees of the paper for useful comments. The research of the first author was partly supported by the ESPRIT Basic Research Action 6810 (Compulog 2). The research of the second author was supported by the Italian National Research Council (CNR).

References

[1] K. R. Apt. Declarative programming in Prolog. In Dale Miller, editor, *Proc. Int'l Symposium on Logic Programming*, pages 12–35. The MIT Press, Cambridge, Mass., 1993.

[2] K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proc. of the Conference on Mathematical Foundations of Computer Science (MFCS 93)*, *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, Berlin, 1993.

[3] K. R. Apt and D. Pedreschi. Reasoning about Termination of Pure Prolog Programs. *Information and Computation*, 106(1):109–157, 1993.

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

[5] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. Technical Report TR 9/93, Dipartimento di Informatica, Università di Pisa, 1993. To appear in the *Journal of Logic Programming*.

[6] K. L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

[7] 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.

[8] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102(1):86–113, 1993.

[9] M. J. Maher. Equivalences of Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, Los Altos, Ca., 1988.

[10] M. J. Maher and R. Ramakrishnan. Déjà Vu in Fixpoints of Logic Programs. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming*, pages 963–980. The MIT Press, Cambridge, Mass., 1989.

[11] J.C. Shepherdson. Unsolvable problems for SLDNF resolution. *Journal of Logic Programming*, 10(1):19–22, 1991.

[12] R. Stärk. A direct proof for the completeness of SLD-resolution. In Börger, H. Kleine Büning, and M.M. Richter, editors, *Computation Theory and Logic 89*, volume 440 of *Lecture Notes in Computer Science*, pages 382–383. Springer-Verlag, 1990.

[13] L. Sterling and E. Y. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., 1986.

[14] J. D. Ullman. *Principles of Database and Knowledge-base Systems*, volume I. Computer Science Press, 1988.

[15] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.