A Closer Look at Declarative Interpretations

Krzysztof R. Apt
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Faculty of Mathematics and Computer Science
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

Maurizio Gabbrielli
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy

Dino Pedreschi
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy

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 the 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 a program in this class all computed answer substitutions.

However, for specific programs the least Herbrand model is tedious to construct and reason about because it contains "ill-typed" facts. Therefore we propose a fourth semantics which associates with a "correctly typed" program the "well-typed" subset of its least Herbrand model. This semantics is used to reason about partial correctness of correctly typed programs. The results are extended to programs with arithmetic.

Note. A preliminary, shorter, version of this paper appeared as Apt and Gabbrielli [2].

AMS Subject Classification (1991): 68Q40, 68T15.
CR Subject Classification (1991): F.3.2., F.4.1, H.3.3, I.2.3.
Keywords and Phrases: logic programs, declarative semantics, isomorphism, partial correctness.

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 of a logic program 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 [7] (or $\mathcal{C}$-semantics of Falaschi et al. [9]) is another natural candidate for the declarative semantics, and in fact it has been successfully used in the probably most elegant and compact proof of the strong completeness of the SLD-resolution due to Stärk [16]. 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 [8]. 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 propose a semantic method based on the least Herbrand model. We also prove its equivalence with the method of Maher and Ramakrishnan [13] which is based on the $\mathcal{S}$-semantics. Using it we checked that several standard pure Prolog programs are subsumption-free.

However, for several natural programs, including `APPEND`, `MEMBER` and other classical logic programs, the least Herbrand model is "overdefined" because it also includes facts with "ill-typed" arguments. As a result the least Herbrand models are usually tedious to construct and to reason about. This problem has to do with the fact that logic and Prolog programs are untyped whereas in usual applications one uses these programs only with "well-typed" queries.

To remedy this problem we introduce yet another semantics, which consists of a "well-typed" fragment of the least Herbrand model. To define it we use types. We prove that this semantics, like the other three, admits a simple characterization in terms of fixpoints. Then we show how this semantics can be naturally used to reason about partial correctness of logic programs.

These results are extended to pure Prolog with arithmetic.

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 and 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. Moreover, we allow arbitrary mgu's when forming resolvents in SLD-derivations and use the notion of standardization apart as in Lloyd [11].

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). So by definition $\{Q\}\ P\ sp(Q,P)$ for any Q and 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 [19]. 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 van Emden and Kowalski [19] showed, this semantics can be characterized by means of the following immediate consequence operator defined on Herbrand interpretations:

$$T_P(I) = \{H \mid \exists\ \mathbf{B}\ (H \leftarrow \mathbf{B} \in Ground(P),\ I \models \mathbf{B})\}.$$

More precisely, they established the following theorem.

Theorem 2.1 ($\mathcal{M}$-Characterization)

(i) T_P *is continuous on the complete lattice of Herbrand interpretations ordered with* $\subseteq$.

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

(iii) $\mathcal{M}(P) = T_P \uparrow \omega$.

□

In Section 8 we shall use an obvious generalization of this Theorem to infinite programs.

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 [8] shows.

Example 2.2 Consider two programs, P_1:

```
p(X).
```

and P_2:

```
p(a).
p(X).
```

Then $\mathcal{M}(P_1) = \mathcal{M}(P_2)$ but the query 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 [7] and more extensively studied in Falaschi et al. [9]. 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

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

Falaschi et al. [9] showed that also this semantics can be characterized by means of an operator, this time the following one defined on term interpretations:

$$U_P(I) = \{H \mid \exists B_1 \ldots \exists B_n\ (H \leftarrow B_1, \ldots, B_n \in inst(P),\ \{B_1, \ldots, B_n\} \subseteq I)\}.$$

where $inst(P)$ denotes the set of all the instances of clauses in P.

Then they established the following theorem analogous to the $\mathcal{M}$-characterization Theorem 2.1.

Theorem 2.3 ($\mathcal{C}$-Characterization)

(i) U_P *is continuous on the complete lattice of term interpretations ordered with* $\subseteq$.

(ii) $\mathcal{C}(P)$ *is the least pre-fixpoint and the least fixpoint of* U_P.

(iii) $\mathcal{C}(P) = U_P \uparrow \omega$. □

However, the $\mathcal{C}$-semantics cannot model the operational behaviour of a program either, since for Example 2.2 we have also $\mathcal{C}(P_1) = \mathcal{C}(P_2)$.

2.3 $\mathcal{S}$-semantics

This semantics was introduced in Falaschi et al. [8]. For a survey on the $\mathcal{S}$-semantics and its uses see Bossi et al. [6]. The aim of this semantics 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 $\{Q\}\ P\ \mathcal{Q}$.

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. [8] shows.

Theorem 2.4 (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 2.5 (Full abstraction) *For all programs* P_1, P_2

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

□

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. [8] introduced the following operator on term interpretations

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

and proved the following.

Theorem 2.6 ($\mathcal{S}$-Characterization)

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

[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.

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 $\mathcal{D}$.

Definition 3.1 Consider a class of programs $\mathbf{C}$. We say that two semantics $\mathcal{S}_1 : \mathbf{C} \to \mathcal{D}_1$ and $\mathcal{S}_2 : \mathbf{C} \to \mathcal{D}_2$ are *isomorphic* on $\mathbf{C}$ iff there exist two functions, $\phi_1 : \mathcal{D}_1 \to \mathcal{D}_2$ and $\phi_2 : \mathcal{D}_2 \to \mathcal{D}_1$ 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 $\mathcal{S}_1 : \mathbf{C} \to \mathcal{D}_1$ and $\mathcal{S}_2 : \mathbf{C} \to \mathcal{D}_2$ are isomorphic on $\mathbf{C}$ iff there exists a bijection $\phi : \mathit{Range}(\mathcal{S}_1) \to \mathit{Range}(\mathcal{S}_2)$ 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) $\mathit{Variant}(I) = \{A \in \mathcal{B} \mid \exists B \in I \text{ s.t. } B \leq A \text{ and } A \leq B\}$, the set of variants,

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

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

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

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

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

Note 3.3 *For all I, $\mathit{Min}(\mathit{Up}(I)) = \mathit{Min}(I)$.* □

3.1 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 3.4 $\mathcal{M}(P) = \mathit{Ground}(\mathcal{C}(P))$. □

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. [9] shows.

Example 3.5 Consider two programs, P_1:

```
p(X).
```

and P_2:

```
p(a).
p(b).
```

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

$\square$

In case the signature contains infinitely many constants, the situation changes, as the following result due to Maher [12] shows.

Theorem 3.6 *Assume that the signature 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 $A \in \mathcal{C}(P)$.

$\square$

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.

3.2 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. [9].

Theorem 3.7 $\mathcal{C}(P) = Up(\mathcal{S}(P))$. $\square$

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. [8] shows.

Example 3.8 Consider the programs P_1:

```
p(X).
```

and P_2:

```
p(a).
p(X).
```

of Example 2.2. 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.

$\square$

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 3.9 $Min(\mathcal{C}(P)) \subseteq \mathcal{S}(P)$.

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

Proof. By Theorem 3.7 $Min(\mathcal{C}(P)) = Min(Up(\mathcal{S}(P)))$ 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 3.10 Consider the following program P:

```
p(a).
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 [13].

Definition 3.11 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 3.9.

Theorem 3.12 *$\mathcal{S}(P) = Min(\mathcal{C}(P))$ iff P is subsumption-free.*

Proof. ($\Rightarrow$) We have

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

($\Leftarrow$) We have

$$\begin{array}{cl} & \mathcal{S}(P) \\ = & \quad \{\text{assumption}\} \\ & Min(\mathcal{S}(P)) \\ = & \quad \{\text{Note 3.3}\} \\ & Min(Up(\mathcal{S}(P))) \\ = & \quad \{\text{Theorem 3.7}\} \\ & Min(\mathcal{C}(P)). \end{array}$$

□

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 2.4, 3.6 and 3.12 we thus obtain:

Corollary 3.13 *Assume that the signature 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 3.10, 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 3.14 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 3.15 *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 3.7 and 3.12.

□

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.

4 Redundance-free Programs

We begin by studying a subclass of subsumption-free programs.

Definition 4.1 A program P is called *redundancy-free* iff $\mathcal{S}(P)$ does not contain a pair of non-variant unifiable atoms. □

Clearly, redundancy-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 4.2 Consider the following program P defined w.r.t. the language with the signature $\Sigma = \{\mathtt{a}/0\}$:

```
p(X,a).
p(a,X).
```

Then $\mathcal{S}(P) = Variant(\{\mathtt{p(X,a)}, \mathtt{p(a,X)}\})$, so P is not redundancy-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 redundancy-free programs in a succinct way. Let us extend the Min operator in an obvious way to sets of queries.

Theorem 4.3

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

(ii) P is redundancy-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 2.4 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 2.4 $A, C \in sp(A', P)$. So $Min(sp(A', P)) \neq sp(A', P)$. Contradiction.

□

For redundancy-free programs we can simplify the formulation of Corollary 3.13.

Corollary 4.4 *Consider a redundancy-free program P and a query Q. Then*

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

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

(iii) If the signature contains infinitely many constant symbols

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

Proof. (i) follows from Theorem 4.3 (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 two 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. □

Now (ii) is a straightforward consequence of (i) and the definition of the $\mathcal{C}$-semantics. Finally, (iii) follows from (ii) and Theorem 3.6. □

So for redundancy-free programs the sets of computed instances can be defined without the use of unification.

The following result provides a method based on the least Herbrand model which allows us to conclude that a program is redundancy-free, so a fortiori subsumption-free.

Theorem 4.5 *Suppose that the following conditions hold for a program P:*

SEM1. *If $H \leftarrow \mathbf{B}_1$ and $H \leftarrow \mathbf{B}_2$ are ground instances of two different clauses in P, then*

$$\mathcal{M}(P) \not\models \mathbf{B}_1 \wedge \mathbf{B}_2.$$

SEM2. *If $H \leftarrow \mathbf{B}_1$ and $H \leftarrow \mathbf{B}_2$ are distinct ground instances of the same clause in P, then*

$$\mathcal{M}(P) \not\models \mathbf{B}_1 \wedge \mathbf{B}_2.$$

Then P is redundancy-free.

Proof. We shall need the following observation.

Claim 1 *Let ξ be an SLD-refutation of a query and a program P and let ϑ be the composition of the mgu's used in ξ. If $H \leftarrow \mathbf{B}$ is an input clause used in ξ, then*

$$\mathcal{M}(P) \models \mathbf{B}\vartheta.$$

Proof. We have $\vartheta = \vartheta_1\vartheta_2$ where ϑ_1 is the composition of the mgu's used in ξ until $H \leftarrow \mathbf{B}$ is used, and ϑ_2 is the composition of the mgu's used in ξ from that moment on. By the Soundness Theorem for SLD resolution

$$\mathcal{M}(P) \models \mathbf{B}\vartheta_2.$$

But by the standardization apart $\mathbf{B}\vartheta_1 = \mathbf{B}$, so in fact

$$\mathcal{M}(P) \models \mathbf{B}\vartheta$$

which concludes the proof. □

We prove now the contrapositive. Assume that the program P is not redundancy-free. By Theorem 4.3 there exists a query Q which admits two computed instances Q' and Q'' such that $Q' < Q''$. Consider then two SLD-refutations ξ' and ξ'' for Q which use the same selection rule, yielding the computed instances $Q' = Q\gamma$ and $Q'' = Q\delta$ where γ and δ are the compositions of the mgu's used in ξ' and ξ'', respectively. Note that, by a suitable choice of the variants of the clauses used in ξ' and ξ'', we can assume without loss of generality that Q' and Q'' are variable disjoint and thus unifiable.

Let $c_1, \ldots, c_n$ $(n \geq 1)$ be the sequence of clauses of P used in ξ', and $d_1, \ldots, d_m$ $(m \geq 1)$ the sequence of clauses of P used in ξ''. Next, consider k $(1 \leq k \leq min(n, m))$ such that

$$\begin{array}{rcll} c_i & = & d_i & \quad \text{for } i \in [1, k-1] \\ c_k & \neq & d_k. & \end{array}$$

Observe that k exists, since Q' and Q'' are not variants. Assume that $H' \leftarrow \mathbf{B}'$ is the variant of c_k used as input clause in ξ' and $H'' \leftarrow \mathbf{B}''$ is the variant of d_k used as input clause in ξ''. The following two cases arise.

Case 1 *($H'\gamma$ and $H''\delta$ unify).*

By the definition of a unifier there exists a ground instance $H \leftarrow \mathbf{B}_1$ of $(H' \leftarrow \mathbf{B}')\gamma$ and a ground instance $H \leftarrow \mathbf{B}_2$ of $(H'' \leftarrow \mathbf{B}'')\delta$, where H is a common ground instance of $H'\gamma$ and $H''\delta$. From Claim 1 it follows $\mathcal{M}(P) \models \mathbf{B}_1 \wedge \mathbf{B}_2$ and consequently P does not satisfy condition SEM1.

Case 2 *($H'\gamma$ and $H''\delta$ do not unify).*

In this case let $R_1 \ldots, R_k$ be the first k resolvents of both SLD-refutations, so $R_1 = Q$ and, for $i \in [2, k]$, R_i is obtained from R_{i-1} by using the clause c_{i-1} $(= d_{i-1})$. Let A be the selected atom in R_k.

From the definition of γ, δ, c_k and d_k it follows that $A\gamma = H'\gamma$ and $A\delta = H''\delta$. Therefore the non-unifiability of $H'\gamma$ and $H''\delta$ implies that $R_k\gamma$ and $R_k\delta$ are not unifiable. On the other hand, by the previous assumption, $R_1\gamma$ $(= Q')$ and $R_1\delta$ $(= Q'')$ are unifiable.

Thus there exists an index $j \in [2, k]$ such that

$$\begin{array}{lll} R_i\gamma \quad \text{and} \quad R_i\delta \text{ unify} & \qquad & \text{for } i \in [1, j-1] \\ R_j\gamma \quad \text{and} \quad R_j\delta \text{ do not unify .} & & \end{array} \tag{2}$$

Let c_j be of the form $K \leftarrow \mathbf{B}$. Since non-relevant mgu's can be used in the SLD derivation, we can assume without loss of generality that

$$Var((K \leftarrow \mathbf{B})\gamma) \cap Var((K \leftarrow \mathbf{B})\delta) = \emptyset. \tag{3}$$

From the definition of the R_i's and from (2) it follows that $K\gamma$ and $K\delta$ unify, while $\mathbf{B}\gamma$ and $\mathbf{B}\delta$ are not unifiable. This, together with (3), implies that there exist two different ground instances $H \leftarrow \mathbf{B}_1$ and $H \leftarrow \mathbf{B}_2$ of the clauses $(K \leftarrow \mathbf{B})\gamma$ and $(K \leftarrow \mathbf{B})\delta$, and hence of the clause $K \leftarrow \mathbf{B}$, such that H is a common ground instance of $K\gamma$ and $K\delta$. Again from Claim 1 it follows $\mathcal{M}(P) \models \mathbf{B}_1 \wedge \mathbf{B}_2$. Consequently, P does not satisfy condition SEM2 and this completes the proof.

□

If $H \leftarrow \mathbf{B}_1$ and $H \leftarrow \mathbf{B}_2$ are ground instances of clauses in P, then clearly $\mathcal{M}(P) \not\models \mathbf{B}_1 \wedge \mathbf{B}_2$ iff $\mathcal{M}(P) \not\models H \wedge \mathbf{B}_1 \wedge \mathbf{B}_2$. Therefore, in some cases we shall consider the formulation of SEM1 and SEM2 which uses $\mathcal{M}(P) \not\models H \wedge \mathbf{B}_1 \wedge \mathbf{B}_2$, since this will simplify the reasoning. It is also easy to see that SEM1 and SEM2 are respectively implied by the following two conditions:

SYN. No variable disjoint variants of two clauses of P unify.

SEM. If $H \leftarrow \mathbf{B}_1, H \leftarrow \mathbf{B}_2 \in Ground(P)$ and $\mathbf{B}_1 \neq \mathbf{B}_2$ then $\mathcal{M}(P) \not\models \mathbf{B}_1 \wedge \mathbf{B}_2$.

Note that condition SEM alone does not ensure subsumption-freedom (and hence, a fortiori, redundancy-freedom), as the program {p(X)., p(a).} shows.

Maher and Ramakrishnan [13] 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 proved that the class of redundancy-free programs is Turing complete. They also provided two conditions ensuring redundancy-freedom. One was based on $\mathcal{M}(P)$ and, using our terminology, is exactly condition SEM2 used above. The other condition was based on the $\mathcal{S}$-semantics and can be expressed as follows:

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.

Interestingly, the simpler condition SEM1 turns out to be equivalent to SEM1'. This is the content of the following Lemma.

Lemma 4.6 *For a program P, SEM1' holds iff SEM1 holds.*

Proof. We prove the contrapositive for both implications.

($\Rightarrow$) Assume that SEM1 does not hold. Then there exist two ground instances $(H_1 \leftarrow \mathbf{B}_2)\eta_1$ and $(H_2 \leftarrow \mathbf{B}_2)\eta_2$ of two different clauses $c: \ H_1 \leftarrow \mathbf{B}_1$ and $d: \ H_2 \leftarrow \mathbf{B}_2$ in P, such that $\mathcal{M}(P) \models \mathbf{B}_1\eta_1 \wedge \mathbf{B}_2\eta_2$ and $H_1\eta_1 = H_2\eta_2$.

But $\mathcal{M}(P) = Ground(\mathcal{S}(P))$, so there exist some $\mathbf{C}_1 \in \mathcal{S}(P)^*$, $\mathbf{C}_2 \in \mathcal{S}(P)^*$, γ_1 and γ_2 such that

$$\mathbf{B}_1\eta_1 = \mathbf{C}_1\gamma_1, \tag{4}$$

$$\mathbf{B}_1\eta_2 = \mathbf{C}_2\gamma_2. \tag{5}$$

We can assume without loss of generality that $H_i \leftarrow \mathbf{B}_i$ and $\mathbf{C}_i$ do not share variables, for $i \in [1, 2]$. Therefore (4) and (5) imply that there exists ϑ_1, ϑ_2, β_1 and β_2 such that

$$\vartheta_1 \text{ is a relevant mgu of } \mathbf{B}_1 \text{ and } \mathbf{C}_1,\ H_1\eta_1 = H_1\vartheta\beta_1, \tag{6}$$

$$\vartheta_2 \text{ is a relevant mgu of } \mathbf{B}_2 \text{ and } \mathbf{C}_2,\ H_2\eta_2 = H_2\vartheta\beta_2. \tag{7}$$

Consider now $A = H_1\vartheta_1$ and $B = H_2\vartheta_2$. From (6) and (7) it follows that $A \in T^{\mathcal{S}}_{\{c\}}(\mathcal{S}(P))$ and $B \in T^{\mathcal{S}}_{\{d\}}(\mathcal{S}(P))$. In order to show that A and B are unifiable note that, again without loss of generality, we can assume $Var(H_i) \cap Var(H_j \leftarrow \mathbf{B}_j) = \emptyset$ and $Var(H_i) \cap Var(\mathbf{C}_j) = \emptyset$, for $i, j \in [1, 2], i \neq j$. From the fact that the mgu's ϑ_i are relevant it follows that also $H_1\vartheta_1$ and $H_2\vartheta_2$ do not share variables. Therefore, from the assumption $H_1\eta_1 = H_2\eta_2$, (6) and (7) it follows that $H_1\vartheta$ and $H_2\vartheta$ are unifiable. Thus condition SEM1' does not hold.

($\Leftarrow$) Assume that SEM1' does not hold. Then there exists a pair $A \in T^{\mathcal{S}}_{\{c\}}(\mathcal{S}(P))$ and $B \in T^{\mathcal{S}}_{\{d\}}(\mathcal{S}(P))$ which is unifiable, where $c: \ H_1 \leftarrow \mathbf{B}_2$ and $d: \ H_2 \leftarrow \mathbf{B}_2$ are two different clauses in P. Then for some $\mathbf{C}_1 \in \mathcal{S}(P)^*$, $\mathbf{C}_2 \in \mathcal{S}(P)^*$ and ϑ_1, ϑ_2

$$A = H_1\vartheta_1,\ Var(H \leftarrow \mathbf{B}_1) \cap Var(\mathbf{C}_1) = \emptyset,\ \vartheta_1 \text{ is an mgu of } \mathbf{B}_1 \text{ and } \mathbf{C}_1,$$

$$B = H_2\vartheta_2,\ \ Var(H \leftarrow \mathbf{B}_2) \cap Var(\mathbf{C}_2) = \emptyset,\ \vartheta_2 \text{ is an mgu of } \mathbf{B}_2 \text{ and } \mathbf{C}_2.$$

Since A and B are unifiable there exists an η such that $H_1\vartheta_1\eta = H_2\vartheta_2\eta$ and $(H_1 \leftarrow \mathbf{B}_1)\vartheta_1\eta$, $(H_2 \leftarrow \mathbf{B}_2)\vartheta_2\eta$ are ground instances of c and d, respectively. Note 3.4 and Theorem 3.7 imply $\mathcal{M}(P) = Ground(\mathcal{S}(P))$. Therefore

$$\mathcal{M}(P) \models \mathbf{B}_1\vartheta_1\eta \wedge \mathbf{B}_2\vartheta_2\eta,$$

since $\mathbf{C}_i \in \mathcal{S}(P)^*$ and $\mathbf{B}_i\vartheta_i\eta = \mathbf{C}_i\vartheta_i\eta$ for $i \in [1,2]$. Consequently $SEM1$ does not hold and this completes the proof. □

Let us discuss now the conditions of Theorem 4.5. It is obvious that conditions SEM1 and SEM2 are only sufficient for proving that a program is redundancy-free. Indeed, adding to a program a variant of its clause does not change any of its semantics, so a fortiori its redundancy-freedom status, but it invalidates SEM1 condition.

To deal with such problems consider the following strengthening of the equivalent condition SEM1':

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, unless A and B are variants.

Theorem 4.5 remains valid when SEM1 is replaced by SEM1", since essentially the same proof as in [13] holds. This strengthening of SEM1 is of use not only for "artificial" programs. Namely, consider the following program ISO_TREE:

```
iso(void, void).
iso(tree(X,Left1,Right1), tree(X,Left2,Right2)) ←
iso(Left1,Left2),iso(Right1,Right2).
iso(tree(X,Left1,Right1), tree(X,Left2,Right2)) ←
iso(Left1,Right2),iso(Right1,Left2).
```

from Sterling and Shapiro [17, page 58], which tests whether two binary trees are isomorphic. Clearly, condition SEM2 is satisfied by ISO_TREE, since actually its stronger version SYN2 holds, but SEM1 does not hold since

```
iso(tree(void,void,void), tree(void,void,void)) ← iso(void,void),iso(void,void).
```

is a ground instance of both the second and the third clause of ISO_TREE and clearly

$$\mathcal{M}(\texttt{ISO\_TREE}) \models \texttt{iso(void,void),iso(void,void)}$$

holds. However, condition SEM1" does hold. Indeed, define by induction a most general tree (mgt) as follows. `void` is a mgt. If $\texttt{t}_1$ and $\texttt{t}_2$ are variable disjoint mgt's and `X` is a variable which appears neither in $\texttt{t}_1$ nor in $\texttt{t}_2$, then $\texttt{tree(X,}\texttt{t}_1\texttt{,}\texttt{t}_2\texttt{)}$ is an mgt.

The following observations follow from the definitions by a straightforward inductive argument:

(i) if $\texttt{iso(}\texttt{t}_1\texttt{,}\texttt{t}_2\texttt{)} \in \mathcal{S}(\texttt{ISO\_TREE})$ then $\texttt{t}_1$ and $\texttt{t}_2$ are mgt's,

(ii) if $\texttt{t}_1$ and $\texttt{t}_2$ are unifiable mgt's, then they are variants.

In order to show that SEM1" holds for the program ISO_TREE, let us consider two atoms $A \in T^{\mathcal{S}}_{\{c\}}(\mathcal{S}(\texttt{ISO\_TREE}))$ and $B \in T^{\mathcal{S}}_{\{d\}}(\mathcal{S}(\texttt{ISO\_TREE}))$ where $c: \ H_2 \leftarrow \mathbf{B}_2$ is the second clause and $d: \ H_3 \leftarrow \mathbf{B}_3$ is the third clause. Assume that $\texttt{iso(t}_1\texttt{,t}_2\texttt{)}$, $\texttt{iso(t}_3\texttt{,t}_4\texttt{)}$, $\texttt{iso(l}_1\texttt{,l}_2\texttt{)}$ and $\texttt{iso(l}_3\texttt{,l}_4\texttt{)}$ are pairwise variable disjoint atoms in $\mathcal{S}(\texttt{ISO\_TREE})$ such that

$$\begin{array}{l} \vartheta_2 \text{ is an mgu of } \mathbf{B}_2 \text{ and } \texttt{iso(t}_1\texttt{,t}_2\texttt{),iso(t}_3\texttt{,t}_4\texttt{)}, \\ \vartheta_3 \text{ is an mgu of } \mathbf{B}_3 \text{ and } \texttt{iso(l}_1\texttt{,l}_2\texttt{),iso(l}_3\texttt{,l}_4\texttt{)}. \end{array}$$

and $A = H\vartheta_2$, $B = H\vartheta_3$. Then

$$\begin{array}{l} A = \texttt{iso(tree(X,t}_1\texttt{,t}_3\texttt{), tree(X,t}_2\texttt{,t}_4\texttt{))}, \\ B = \texttt{iso(tree(Y,l}_1\texttt{,l}_3\texttt{), tree(Y,l}_4\texttt{,l}_2\texttt{))}. \end{array}$$

If A and B unify, from (i) and (ii) above and an easy inspection of the unification algorithm it follows that A and B are variants. So SEM1" holds and ISO_TREE is redundancy-free.

In certain situations the conditions of Theorem 4.5 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 variable disjoint variants of 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. Note that the qualification "variable disjoint variants" cannot be dropped from SYN1. Indeed, consider the program P

```
p(X).
p(f(X)).
```

Then for P this modification of SYN1 holds, but SEM1 does not hold.

It is worth mentioning that an immediate proof of Turing completeness for redundancy-free programs can be obtained by using the encoding of two register machines into pure logic programs given in Shepherdson [15]. 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 4.5.

5 Applications to Program Semantics

We provide here four illustrative uses of Theorem 4.5.

Example 5.1
(i) Consider first the proverbial 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 now 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 ground atoms of the form $\mathtt{suffix(s,t_s)}$ where $\mathtt{t_s}$ a term containing the term s. By definition of T_P, $T_{\mathtt{SUFFIX}}(OCC) \subseteq OCC$, i.e. OCC is a pre-fixpoint of $T_{\mathtt{SUFFIX}}$. By the $\mathcal{M}$-characterization Theorem 2.1 $\mathcal{M}(\mathtt{SUFFIX}) \subseteq OCC$. So, for any ground instance

$$\mathtt{suffix(t_1,[t_2\ |\ t_3])} \leftarrow \mathtt{suffix(t_1,t_3)}$$

of the second clause, if $\mathcal{M}(\mathtt{SUFFIX}) \models \mathtt{suffix(t_1,t_3)}$ then $\mathtt{t_1}$ and $\mathtt{[t_2\ |\ t_3]}$ are different terms. Thus $\mathtt{suffix(t_1,[t_2\ |\ t_3])}$ is not an instance of the first clause and consequently 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(Ys,[X],Zs)
```

augmented by the APPEND program.

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 $\mathcal{M}(\mathtt{REVERSE})$. Using the list constructor binary function [.|..] let us define the notation $[t_1|t_2|\ldots|t_n]$ for $n \geq 2$ by induction as follows. For $n = 2$, $[t_1|t_2]$ is the induction base. For $n > 2$ we define by induction

$$[t_1|t_2|\ldots|t_n] = [t_1|\ [t_2|\ldots|t_n]\ ]$$

A *list* is then defined as either the constant symbol [ ] (the empty list), or a construct of the form $[t_1|t_2|\ldots|t_n]$ where $n \geq 2$ and $t_n = [\ ]$. Finally, given a list s and a term t, we define their concatenation $s * t$ as follows:

$$\begin{array}{l} \text{if } s = [\ ] \text{ then } s * t = t, \\ \text{if } s = [t_1|\ldots|t_{n-1}|[\ ]] \text{ then } s * t = [t_1|\ldots|t_{n-1}|t]. \end{array}$$

Then it can be shown that

$$\mathcal{M}(\mathtt{APPEND}) = \{\mathtt{append(s,t,u)} \mid \text{ s is a ground list, t is a ground term and } \mathtt{s*t=u}\ \}$$

and

$$\begin{array}{rcl} \mathcal{M}(\mathtt{REVERSE}) & = & \{\mathtt{reverse(s,t)} \mid \mathtt{s,t} \text{ are ground lists and } \mathtt{t=rev(s)}\} \\ & & \cup\ \mathcal{M}(\mathtt{APPEND}) \end{array}$$

where given a list s, rev(s) denotes its reverse.

Take now a ground instance

```
reverse([x | xs], zs) ← reverse(xs,ys), append(ys,[x],zs)
```

of the second clause with `reverse([x|xs], zs)` 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 redundancy-free.

(iv) Finally, consider the following program HANOI from Sterling and Shapiro [17] 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,Moves) ←
   hanoi(N,A,C,B,Ms1)
   hanoi(N,C,B,A ,Ms2)
   append(Ms1,[A to B|Ms2],Moves).
```

augmented by the APPEND program.

Note that conditions SYN1 and SYN2 do not apply here. To prove condition SEM1 first note that $\mathcal{M}$(HANOI) $\models$ `hanoi(t1,t2,t3,t4,t5)` implies t1 $\neq$ 0. Hence for any ground instance `hanoi(t1,t2,t3,t4,t5)` $\leftarrow \mathbf{B}$ of the second clause, if t1 = s(0) then $\mathcal{M}$(HANOI) $\not\models \mathbf{B}$. This implies SEM1.

To prove condition SEM2 we use the methodology of Maher and Ramakrishnan [13] based on functional dependencies. First we need a definition.

Definition 5.2 Let p be an n-ary relation symbol. A functional dependency is a construct of the form $p[I \to J]$ where $I, J \subseteq \{1, \ldots, n\}$. Let M be a set of ground atoms. We say that $p[I \to J]$ *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

$$F = \{\mathtt{hanoi}[\{1,2,3,4\} \to \{5\}], \mathtt{append}[\{1,2\} \to \{3\}]\}$$

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 $\mathtt{hanoi}[\{1,2,3,4\} \to \{5\}]$ holds over $\mathcal{M}$(HANOI). The other functional dependency can be directly established by considering the explicit definition of $\mathcal{M}$(APPEND) 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

$$\mathtt{hanoi(s(n),a,b,c,moves)} \leftarrow \quad \mathtt{hanoi(n,a,c,b,ms1)}, \mathtt{hanoi(n,c,b,a,ms2)}, \mathtt{append(ms1,[a\ to\ b|ms2],moves)}$$

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

Since $\mathtt{hanoi}[\{1,2,3,4\} \to \{5\}]$ 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 $\mathtt{append}[\{1,2\} \to \{3\}]$, for moves. Consequently, SEM2 holds and HANOI is redundancy-free.

A general method for establishing functional dependencies on $\mathcal{M}(P)$, based on an extended version of Amstrong axioms (see Ullman [18]), is given in Maher and Ramakrishnan [13]. □

Note that Theorem 4.5 only provides sufficient conditions for redundancy-freedom. Indeed, the program {`p(X)` $\leftarrow$ `q(X,Y).`, `q(a,b).`, `q(a,c).`} is easily seen to be redundancy-free but condition SEM2 does not hold. Moreover, for certain natural programs Theorem 4.5 cannot be used to establish their subsumption-freedom, simply because they are not redundancy-free. An example is of course the program considered in Example 4.2. But more natural programs exist. In such situations we still can use a direct reasoning to prove subsumption-freedom.

Example 5.3 Consider the `MEMBER` program:

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

We now prove that `MEMBER` is subsumption-free. By the $\mathcal{S}$-characterization Theorem 2.6 it suffices to show that if I is subsumption-free, then $T^{\mathcal{S}}_{\mathtt{MEMBER}}(I)$ is subsumption-free. Denote the first clause by c_1 and the second one by c_2. Consider a pair $A_1, A_2 \in T^{\mathcal{S}}_{\mathtt{MEMBER}}(I)$. 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 $T^{\mathcal{S}}_P$, $A_1 = \mathtt{member(X,[X|Xs])}\rho$ for a renaming ρ and $A_2 = \mathtt{member(X,[Y|Xs])}\vartheta$ where ϑ is an mgu of `member(X,Xs)` and B for a B such that $\mathtt{Y} \notin Var(B)$. This implies $\mathtt{X}\vartheta \neq \mathtt{Y}\vartheta$ 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, $A_i = \mathtt{member(X,[Y|Xs])}\vartheta_i$ where ϑ_i is an mgu of `member(X,Xs)` and B_i for $i = 1, 2$. Assuming $B_i = \mathtt{member(t}_i\mathtt{,l}_i\mathtt{)}$ we have $\vartheta_i = \{\mathtt{X}/t_i, \mathtt{Xs}/l_i\}$ (up to renaming). Then the assumption $B_1 \not\leq B_2$ implies $\mathtt{member(X,Xs)}\ \vartheta_1 \not\leq \mathtt{member(X,Xs)}\vartheta_2$ and hence $A_1 \not\leq A_2$. Analogously for the symmetric case.

Note that `MEMBER` is not redundancy-free. □

6 Fourth Semantics – $\mathcal{M}_{(pre,post)}$

The results of the previous sections indicate that the $\mathcal{M}$-semantics precisely captures the procedural interpretation for the subsumption-free programs. However, it should be noticed that for many programs it is quite cumbersome to construct their least Herbrand model. Note for example that $\mathcal{M}$(`APPEND`) contains elements of the form `append(s,t,u)` where neither `t` nor `u` is a list, and analogously for $\mathcal{M}$(`MEMBER`), since it can be shown that

$$\mathcal{M}(\mathtt{MEMBER}) = \{\mathtt{member(t,[t_1|t_2|\ldots|t_n])} \mid n \geq 2,\ \mathtt{t},\ \mathtt{t_1},\ \ldots,\mathtt{t}_n \text{ are ground terms and } \mathtt{t} = \mathtt{t}_j \text{ for some } j \in [1, n-1]\}.$$

Clearly, it is quite clumsy to reason about programs when even in so simple cases their semantics is defined in such a laborious way. Preferably, one would rather like to associate with the `APPEND` program the following, more natural meaning:

$$\{\mathtt{append(s,t,u)} \mid \mathtt{s,t,u} \text{ are ground lists and } \mathtt{s*t=u}\} \tag{8}$$

and with the `MEMBER` program the following meaning:

$$\{\mathtt{member(s,t)} \mid \mathtt{t} \text{ is a ground list and } \mathtt{s} \text{ is an element in } \mathtt{t}\}.$$

To be able to do this we have to find a systematic way of associating with the APPEND program the set (8) etc. Note that the set (8), when viewed as a Herbrand interpretation, is *not* a model of APPEND, because the first clause does not hold in it.

The solution proposed here involves the use of types. We use the notion of a well-typed query and clause as in Apt [1] (which from the semantics point of view coincides with the method of Bossi and Cocco [5] for proving partial correctness), but follow the equivalent presentation of Ruggieri [14] which is more convenient for our purposes.

Definition 6.1 Consider a pair *pre*, *post* of Herbrand interpretations.

- A query is called (*pre, post*)*-correct* if for every ground instance $A_1, \ldots, A_n$ of it, for $j \in [1, n]$

$$A_1, \ldots, A_{j-1} \in post \;\Rightarrow\; A_j \in pre.$$

- A clause is called (*pre, post*)*-correct* if for every ground instance $H \leftarrow B_1, \ldots, B_n$ of it

  – $H \in pre \wedge B_1, \ldots, B_{j-1} \in post \;\Rightarrow\; B_j \in pre,$ for $j \in [1, n]$, and
  – $H \in pre \wedge B_1, \ldots, B_n \in post \;\Rightarrow\; H \in post.$

- A program is called (*pre, post*)*-correct* if every clause of it is. □

Note that every instance and every prefix of a (*pre*, *post*)-correct query is (*pre*, *post*)-correct. To see the equivalence with the notion of well-typedness call an atom a *p-atom* if its relation symbol is p. In Apt [1] with each relation symbol p a pair pre_p, $post_p$ of two sets of p-atoms closed under substitution is associated. Consider now a program P. Let *pre* be the union of all sets $Ground(pre_p)$ where p ranges over the relation symbols of P, and similarly for *post*. Then given the type assignment pre_p, $post_p$, a program P is well-typed in the sense of Apt [1] iff it is (*pre*, *post*)-correct in the above sense.

Conversely, given a Herbrand interpretation I and a relation symbol p define I_p to be the set of p-atoms belonging to I. Then P is (*pre*, *post*)-correct iff it is well-typed in the sense of Apt [1] given the type assignment pre_p, $post_p$.

Given a pair of Herbrand interpretations *pre*, *post* and a (*pre*, *post*)-correct program P we now define its "well-typed" semantics as

$$\mathcal{M}_{(pre,post)}(P) = \mathcal{M}(P) \cap pre.$$

Intuitively, $\mathcal{M}_{(pre,post)}(P)$ is the "well-typed" fragment of the least Herbrand model of a program P. We call it $\mathcal{M}_{(pre,post)}$-semantics. Note that the $\mathcal{M}_{(pre,post)}$-semantics does not depend on *post*, but as the following result of Ruggieri [14] shows, for (*pre*, *post*)-correct programs $\mathcal{M}_{(pre,post)}(P)$ can be equivalently defined as $\mathcal{M}(P) \cap pre \cap post$.

Lemma 6.2 *For a (pre, post)-correct program P we have $\mathcal{M}_{(pre,post)}(P) \subseteq post$.* □

In general, the $\mathcal{M}_{(pre,post)}$-semantics is not a model of the program. But for the (*pre*, *post*)-correct queries it turns out to be equivalent to the $\mathcal{M}$-semantics. This is the content of the following result.

Lemma 6.3 *For a (pre, post)-correct program P and a (pre, post)-correct query Q*

$$\mathcal{M}(P) \models Q \textit{ iff } \mathcal{M}_{(pre,post)}(P) \models Q.$$

Proof. ($\Rightarrow$) Consider a ground instance $A_1, \ldots, A_n$ of the query Q such that $A_1, \ldots, A_n \in \mathcal{M}(P)$. We show, by induction on n, that $A_j \in pre$ for $j \in [1, n]$. For the base case ($n = 0$), the claim holds vacuously. For the induction step ($n > 0$), we have $A_1, \ldots, A_{n-1} \in pre$ by the induction hypothesis. Together with the assumption $A_1, \ldots, A_{n-1} \in \mathcal{M}(P)$ this implies $A_1, \ldots, A_{n-1} \in post$ by Lemma 6.2. By the fact that $A_1, \ldots, A_n$ is (pre, $post$)-correct, we conclude that $A_n \in pre$, which completes the proof of the first implication.

($\Leftarrow$) Obvious, as by definition $\mathcal{M}_{(pre,post)}(P) \subseteq \mathcal{M}(P)$. □

The following example should clarify the idea behind this approach to types. Here and in other natural cases $post \subseteq pre$. Then by the above lemma we have $\mathcal{M}_{(pre,post)}(P) = \mathcal{M}(P) \cap post$ which makes the $\mathcal{M}_{(pre,post)}$-semantics somewhat easier to construct.

Example 6.4 Consider the program `APPEND`. In general, `APPEND` is used either to concatenate two lists or to split a list. This use is reflected in the following choice of pre:

$$\begin{aligned} pre = \ & \{\texttt{append(s,t,u)} \mid \texttt{s,t} \text{ are ground lists and } \texttt{u} \text{ is a ground term}\} \\ & \cup \\ & \{\texttt{append(s,t,u)} \mid \texttt{s,t} \text{ are ground terms and } \texttt{u} \text{ is a ground list}\}. \end{aligned}$$

Intuitively, pre is the set of all ground instances of the intended one atom queries. It is readily checked that `APPEND` is (pre, $post$)-correct, where

$$post = \{\texttt{append(s,t,u)} \mid \texttt{s,t,u} \text{ are ground lists}\}.$$

Now, using the previously obtained characterization of $\mathcal{M}(\texttt{APPEND})$ we obtain

$$\mathcal{M}_{(pre,post)}(\texttt{APPEND}) = \{\texttt{append(s,t,u)} \mid \texttt{s,t,u} \text{ are ground lists and } \texttt{s*t=u}\}.$$

□

The above example shows how to construct the set $\mathcal{M}_{(pre,post)}(P)$ by using the least Hebrand model $\mathcal{M}(P)$. But, as we already noticed, the construction of $\mathcal{M}(P)$ can be quite cumbersome, so we would prefer to define $\mathcal{M}_{(pre,post)}(P)$ directly, without constructing $\mathcal{M}(P)$ first. To this end we introduce the notion of a *reduced* program w.r.t. a Herbrand interpretation.

Definition 6.5 Consider a program P and a Herbrand interpretation J. Then the *reduced program w.r.t. J* , denoted by $J(P)$, is the (possibly infinite) program consisting of the ground instances of clauses from P the head of which is in J. That is:

$$J(P) \quad = \quad \{A \leftarrow \mathbf{B} \in Ground(P) \mid A \in J\}.$$

□

As a direct consequence of the definition, observe that

$$T_{J(P)}(I) \quad = \quad T_P(I) \cap J \tag{9}$$

and that $T_{J(P)}$ is continuous on the complete lattice of Herbrand interpretations ordered with $\subseteq$.

We now prove that for a (pre, $post$)-correct program P, the $\mathcal{M}_{(pre,post)}$-semantics coincides with the $\mathcal{M}$-semantics of $pre(P)$. This result provides us with a method for removing the "ill-typed" atoms from the $\mathcal{M}$-semantics by using the reduced program $pre(P)$.

Theorem 6.6 *For a (pre, post)-correct program P*

$$\mathcal{M}_{(pre,post)}(P) = \mathcal{M}(pre(P)).$$

Proof. By the $\mathcal{M}$-characterization Theorem 2.1 $\mathcal{M}_{(pre,post)}(P) = T_P \uparrow \omega \cap pre$ and $\mathcal{M}(pre(P)) = T_{pre(P)} \uparrow \omega$. Now, on the account of (9), we have $T_{J(P)} \uparrow \omega \subseteq T_P \uparrow \omega \cap J$, for all J, so for pre in particular. Thus $\mathcal{M}(pre(P)) \subseteq \mathcal{M}_{(pre,post)}(P)$.

To prove the other inclusion we show by induction that for $n \geq 0$

$$T_P \uparrow n \cap pre \subseteq T_{pre(P)} \uparrow n.$$

The induction base ($n = 0$) is obvious. For the induction step ($n > 0$) assume $H \in T_P \uparrow n \cap pre$. Then there exists a ground instance $H \leftarrow B_1 \ldots B_m$ of a clause in P such that

$$\{B_1 \ldots B_m\} \subseteq T_P \uparrow (n-1). \tag{10}$$

Since the program P is $(pre, post)$-correct, it is easy to prove by induction on m, that also the inclusion

$$\{B_1 \ldots B_m\} \subseteq pre. \tag{11}$$

holds. Indeed, for the base case ($m = 0$) the claim holds vacuously. For the induction step ($m > 0$) assume that $\{B_1 \ldots B_{m-1}\} \subseteq pre$. This together with (10) implies $\{B_1 \ldots B_{m-1}\} \subseteq \mathcal{M}_{(pre,post)}(P)$ and hence, by Lemma 6.2, $\{B_1 \ldots B_{m-1}\} \subseteq post$ holds. Since by assumption $H \in pre$, it follows from Definition 6.1 that $B_m \in pre$.

Now the induction hypothesis, (10) and (11) imply $\{B_1 \ldots B_m\} \subseteq T_{pre(P)} \uparrow (n-1)$ and consequently $H \in T_{pre(P)} \uparrow n$ which concludes the proof. □

This allows us to conclude that the $\mathcal{M}_{(pre,post)}$-semantics admits the characterizations analogous to those of the other three semantics so far considered. Namely we have the following analogue of the Characterization Theorems 2.1, 2.3 and 2.6.

Theorem 6.7 ($\mathcal{M}_{(pre,post)}$-Characterization 1) *For a (pre, post)-correct program P*

(i) $T_{pre(P)}$ *is continuous on the complete lattice of Herbrand interpretations ordered with* $\subseteq$.

(ii) $\mathcal{M}_{(pre,post)}(P)$ *is the least fixpoint and the least pre-fixpoint of* $T_{pre(P)}$.

(iii) $\mathcal{M}_{(pre,post)}(P) = T_{pre(P)} \uparrow \omega$.

Proof. We already noticed that (i) is a consequence of (9). (ii) and (iii) follow directly from Theorem 6.6 and Theorem 2.1 applied to $pre(P)$. □

As already mentioned, in specific applications it is often the case that for a $(pre, post)$-correct program we have $post \subseteq pre$. In this case an alternative characterization of the $\mathcal{M}_{(pre,post)}$-semantics in terms of $post(P)$ can be given. Namely we have the following analogue of the above theorem.

Theorem 6.8 ($\mathcal{M}_{(pre,post)}$-Characterization 2) *Suppose that* $post \subseteq pre$. *Then for a (pre, post)-correct program P*

(i) $T_{post(P)}$ *is continuous on the complete lattice of Herbrand interpretations ordered with* $\subseteq$.

(ii) $\mathcal{M}_{(pre,post)}(P)$ *is the least fixpoint and the least pre-fixpoint of* $T_{post(P)}$.

(iii) $\mathcal{M}_{(pre,post)}(P) = T_{post(P)} \uparrow \omega$.

Proof. By Lemma 6.2 $\mathcal{M}_{(pre,post)}(P) \subseteq post$. Thus to prove (ii) and (iii) it suffices to prove by the $\mathcal{M}_{(pre,post)}$-Characterization 1 Theorem 6.7 that $post \subseteq pre$ implies that for $n \geq 0$

$$T_{pre(P)} \uparrow n \cap post = T_{post(P)} \uparrow n.$$

The proof of the $\subseteq$ inclusion does not use the assumption $post \subseteq pre$ and is by induction on n. The induction base ($n = 0$) is obvious. For the induction step ($n > 0$) assume $H \in T_{pre(P)} \uparrow n \cap post$. Then there exists a ground instance $H \leftarrow B_1 \ldots B_m$ of a clause in $pre(P)$ such that

$$\{B_1 \ldots B_m\} \subseteq T_{pre(P)} \uparrow (n-1).$$

By Lemma 6.2 and the $\mathcal{M}_{(pre,post)}$-Characterization 1 Theorem 6.7 we also have

$$\{B_1 \ldots B_m\} \subseteq post,$$

so by the induction hypothesis $\{B_1 \ldots B_m\} \subseteq T_{post(P)} \uparrow (n-1)$ and consequently $H \in T_{post(P)} \uparrow n \cap post$.

For the other inclusion note that $T_{post(P)} \uparrow n \subseteq T_{post(P)} \uparrow n \cap post$ and $post \subseteq pre$ now implies $T_{post(P)} \uparrow n \cap post \subseteq T_{pre(P)} \uparrow n \cap post$.

This concludes the proof. □

Returning to Example 6.4 note that using the above theorem it is now easy to construct $\mathcal{M}_{(pre,post)}(\texttt{APPEND})$ by proving by induction on $n > 0$ that

$T_{post(\texttt{APPEND})} \uparrow n = \{$`append(s,t,u)` $\mid$ `s,t,u` are ground lists, `s` is of length $n-1$ and `s*t=u`$\}$.

Finally, let us remark that for a large class of programs it is possible to verify that a Herbrand interpretation coincides with the $\mathcal{M}_{(pre,post)}$-semantics in a simple way. Call a program *left terminating* if all its SLD-derivations w.r.t. the leftmost selection rule, starting with a ground query are finite. Call a model I of a program P *supported* if for every ground atom A such that $I \models A$ there exists $\mathbf{B}$ such that $A \leftarrow \mathbf{B} \in Ground(P)$ and $I \models \mathbf{B}$.

In Apt and Pedreschi [3] it is argued that most natural pure Prolog programs are left terminating and a natural method is proposed to prove that a program is left terminating. A result of Apt and Pedreschi [3] states that for a left terminating program P the least Herbrand model $\mathcal{M}(P)$ of P is the unique supported Herbrand model of P. Now, if P is left terminating, then so is $Ground(P)$ and a fortiori $pre(P)$ and $post(P)$. Thus, for a left terminating program, by the $\mathcal{M}_{(pre,post)}$-Characterization Theorems 6.7 and 6.8 we have that $\mathcal{M}_{(pre,post)}(P)$ is the unique supported Herbrand model of $pre(P)$ and, if $post \subseteq pre$, the unique supported Herbrand model of $post(P)$. Usually, checking that a given Herbrand interpretation is a supported model is straightforward.

7 Applications to Program Verification

The results of 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 5.1 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 $\mathcal{M}(P)$.*

(2) *Prove that P is redundant or subsumption-free.*

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

$$\mathcal{M}(P) \models Q' \text{ iff } Ground(Q') \subseteq \mathcal{M}(P)^*. \tag{12}$$

In our case, by the form of $\mathcal{M}(\texttt{REVERSE})$, if Q'' is a ground instance of `reverse(s,rev(s))` then $Q'' \in \mathcal{M}(\texttt{REVERSE})$ holds. Therefore by (12)

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

(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

$$(\forall \eta.(E_1 = E_2)\eta \text{ is ground } \Rightarrow \ E_1\eta = E_2\eta) \Rightarrow E_1 = E_2. \tag{13}$$

In our case assume that

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

we have $\texttt{X}\gamma\eta = rev(\texttt{s}\gamma\eta) =$ (by definition of rev) $rev(\texttt{s})\gamma\eta$. Then by (13) $\texttt{X}\gamma = rev(\texttt{s})\gamma$ and hence

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

holds.

(5) *Apply Corollary 4.4 (or Corollary 3.13 for programs which are not redundancy-free).* For `REVERSE` we obtain

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

In view of our comments of Section 6, the drawback of this approach to proving partial correctness is point (1), so the construction of the $\mathcal{M}$-semantics. We also argued that for (pre, $post$)-correct programs it is usually easier to construct their $\mathcal{M}_{(pre,post)}$-semantics. So it is legitimate to rephrase the above methodology for partial correctness by using $\mathcal{M}_{(pre,post)}(P)$ instead of $\mathcal{M}(P)$. To this end, we introduce the following notion of (pre, $post$)-redundancy-freedom.

Definition 7.1 A program P is said to be *(pre, post)-redundancy-free* if it is (pre, $post$)-correct and, for any (pre, $post$)-correct query Q, $Min(sp(Q,P)) = sp(Q,P)$, that is the set of computed instances of Q is subsumption-free. □

Observe that, on the account of Theorem 4.3 (ii), for a (*pre*, *post*)-correct program P, if P is redundancy-free then it is (*pre*, *post*)-redundancy-free. Later we shall exhibit Herbrand interpretations *pre*, *post* and a natural program which is (*pre*, *post*)-redundancy-free but not redundancy-free. The next result is a relativized version of Corollary 4.4. It shows that, for (*pre*, *post*)-redundancy-free programs, the computed instances of the (*pre*, *post*)-correct queries can be retrieved from $\mathcal{M}_{(pre,post)}(P)$, thus motivating the previous definition.

Corollary 7.2 *Consider a (pre, post)-redundancy-free program P and a (pre, post)-correct query Q. Then*

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

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

(iii) If the signature contains infinitely many constant symbols

$$\{Q\}\ P\ Min(\{Q\theta \mid \mathcal{M}_{(pre,post)}(P) \models Q\theta\}).$$

Proof. From Claims 1 and 2 of the proof of Corollary 4.4 we obtain (i), (ii) and also

$$\{Q\}\ P\ Min(\{Q\theta \mid \mathcal{M}(P) \models Q\theta\}),$$

provided that the signature contains infinitely many constant symbols. Then (iii) follows from Lemma 6.3. □

Thus for (*pre*, *post*)-redundancy-free programs the set of computed instances of a (*pre*, *post*)-correct query coincides with the set of its most general instances that are true in $\mathcal{M}_{(pre,post)}(P)$. We are now faced with the problem of proving that a (*pre*, *post*)-correct program P is (*pre*, *post*)-redundancy-free. Clearly, redundancy freedom is a sufficient condition for (*pre*, *post*)-redundancy-freedom. However, the proof method for redundancy freedom, namely Theorem 4.5, is based on $\mathcal{M}(P)$, whereas for (*pre*, *post*)-correct programs, we would like to use $\mathcal{M}_{(pre,post)}(P)$.

To solve this problem, we provide an analogue of Theorem 4.5 which employs a modification of the conditions SEM1 and SEM2. The new conditions refer to $\mathcal{M}_{(pre,post)}(P)$ instead of $\mathcal{M}(P)$, and allow us to prove that a program is (*pre*, *post*)-redundancy-free.

In the proof of Theorem 7.4 we use LD-resolution, that is SLD-resolution with the leftmost selection rule, as adopted in Prolog. The following Persistence Lemma, due to Ruggieri [14], will be needed.

Lemma 7.3 (Persistence) *Let P and Q be (pre, post)-correct and let ξ be an LD-derivation of $P \cup \{Q\}$. Then all resolvents in ξ are (pre, post)-correct.* □

Theorem 7.4 *Suppose that the following conditions hold for a (pre, post)-correct program P:*

SEM1. *If $H \leftarrow \mathbf{B}_1$ and $H \leftarrow \mathbf{B}_2$ are ground instances of two different clauses in P then*

$$\mathcal{M}_{(pre,post)}(P) \not\models H \wedge \mathbf{B}_1 \wedge \mathbf{B}_2.$$

SEM2. *If $H \leftarrow \mathbf{B}_1$ and $H \leftarrow \mathbf{B}_2$ are distinct ground instances of the same clause in P then*

$$\mathcal{M}_{(pre,post)}(P) \not\models H \wedge \mathbf{B}_1 \wedge \mathbf{B}_2.$$

Then P is (pre, post)-redundancy-free.

Proof. The proof follows closely that of Theorem 4.5. First, we shall need the following observation.

Claim 1 *Let ξ be an LD-refutation of a (pre, post)-correct query and a (pre, post)-correct program P, and let θ be the composition of the mgu's used in ξ. If $H \leftarrow \mathbf{B}$ is an input clause used in ξ, then*

$$\mathcal{M}_{(pre,post)}(P) \models (H \wedge \mathbf{B})\vartheta.$$

Proof. From Claim 1 of the proof of Theorem 4.5 it follows $\mathcal{M}(P) \models \mathbf{B}\theta$, which implies that also $\mathcal{M}(P) \models H\theta$. Further both H and $\mathbf{B}$ are instances of a prefix of a resolvent in ξ, so by the Persistence Lemma 7.3 both H and $\mathbf{B}$ are (*pre*, *post*)-correct. It suffices now to apply Lemma 6.3. □

We now prove the contrapositive. Assume that the program P is not (*pre*, *post*)-redundancy-free that is, there exists a (*pre*, *post*)-correct query Q which admits two computed instances Q' and Q'' such that $Q' < Q''$. By virtue of the Strong Completeness of SLD-resolution we can consider then two LD-refutations ξ' and ξ'' for Q which yield its computed instances Q' and Q''. The rest of the proof is from now on the same as that of Theorem 4.5, using Claim 1 above instead of Claim 1 of the proof of Theorem 4.5. □

Example 7.5 Reconsider the `MEMBER` program of Example 5.3:

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

We showed that `MEMBER` is subsumption-free, although it is not redundancy-free. We now prove in a straightforward manner that it is (*pre*, *post*)-redundancy-free w.r.t. a class of natural queries. Consider

$$pre = post = \{\texttt{member(x,t)} \mid \texttt{x} \text{ is a ground term and } \texttt{t} \text{ is a ground list of distinct elements}\}.$$

It is readily checked that `MEMBER` is (*pre*, *post*)-correct, and that

$$\begin{aligned}\mathcal{M}_{(pre,post)}(\texttt{MEMBER}) \; = \; & \{\texttt{member(x,t)} \mid \texttt{x} \text{ is a ground term,} \\ & \quad \texttt{t} \text{ is a ground list of distinct elements, and } \texttt{x} \text{ is in } \texttt{t}\}.\end{aligned}$$

Condition SYN2 of Section 4 obviously applies to the `MEMBER` program. To check condition SEM1 of Theorem 7.4, consider two ground instances with a common head of the two clauses of the program, `member(x,[x | xs])` and `member(x,[x | xs]) ← member(x, xs)`. If

$$\mathcal{M}_{(pre,post)}(\texttt{MEMBER}) \models \texttt{member(x,[x|xs])}$$

then all elements in `xs` are different from `x`, and therefore

$$\mathcal{M}_{(pre,post)}(\texttt{MEMBER}) \not\models \texttt{member(x,xs)}$$

which implies that SEM1 holds for the `MEMBER` program. By Theorem 7.4 we have that `MEMBER` is (*pre*, *post*)-redundancy-free. Now Corollary 7.2 can be applied to any query of the form `member(s,t)` where `t` is a list of pairwise non-unifiable elements, as such a query is (*pre*, *post*)-correct. □

We can now summarize our methodology for proving partial correctness on the basis of $\mathcal{M}_{(pre,post)}$-semantics.

(1) *Construct pre and post such that the program P and the query Q are $(pre,\ post)$-correct.*

Intuitively, pre is the set of ground instances of the intended atomic queries.

(2) *Construct $\mathcal{M}_{(pre,post)}(P)$.*

Usually, the "specification" of the program limited to its ground queries coincides with $\mathcal{M}(P)$. As explained at the end of Section 6, the techniques of Apt and Pedreschi [3] are useful for verifying validity of such a guess.

(3) *Prove that P is $(pre,\ post)$-redundancy-free.*

(4) *Find a correct instance Q' of Q*, i.e. such that $\mathcal{M}_{(pre,post)}(P) \models Q'$.

(5) *By suitably generalizing from* (4) *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).

(6) *Apply Corollary 7.2.*

8 Programs with Arithmetic

We now apply the results of the previous sections to an extension of logic programming with arithmetic. Since we wish to apply these results to reason about Prolog programs we follow here Prolog's approach to arithmetic. So we extend the syntax by allowing in the bodies of the program clauses the arithmetic comparison operators $<, \leq, =:=, \neq, \geq, >$ and the "`is`" relation of Prolog. We also assume that, conforming to the status of built-in's, in the original program these arithmetic relations are not used in the heads of the clauses.

To model adequately the semantics and the computation process of programs with arithmetic we follow here the approach of Kunen [10] and first add to each program infinitely many unit clauses which define the ground instances of the used arithmetic relations.

To this end we use the shorthand gae to denote a ground arithmetic expression. Given a gae `n` we denote by `val(n)` its value. For example, `val(3+4)` equals `7`. So for $<$ we add the following set of unit clauses:

$$M_{<} = \{\mathtt{m} < \mathtt{n} \mid \mathtt{m},\ \mathtt{n} \text{ are gae's and } \mathtt{val(m)} < \mathtt{val(n)}\},$$

for "`is`" we add the set

$$M_{\mathtt{is}} = \{\mathtt{val(n)\ is\ n} \mid \mathtt{n} \text{ is a gae}\},$$

etc. So for example $\mathtt{7\ is\ 3+4} \in M_{\mathtt{is}}$.

Now we can apply the previous results on all four semantics to logic programs with arithmetic. However, to deal with partial correctness of these programs we have to exercise some care because Prolog uses the leftmost selection rule and moreover in the case of programs with arithmetic run-time errors can arise.

From now on all proof-theoretic notions, such as the computed instance refer to the LD-resolution. We extend the LD-resolution by stipulating that an LD-derivation *ends in an error* when the last selected atom is with an arithmetic relation and

- either it is of the form `p(s,t)` where `p` is a comparison operator and either `s` or `t` are not gae, or
- it is of the form `s is t` and `t` is not gae.

This together with the extension of the programs by the definitions of the arithmetic relations appropriately models Prolog's computation process. For example the query `X is 3+4` yields as desired the computed answer substitution $\{x/7\}$ and the query `X is Y` yields an error.

Now, the previously established results concerning partial correctness (so Corollaries 3.13, 4.4 and 7.2) hold for all queries such that their LD-derivations do not end in error. This is a consequence of the fact that by the strong completeness of the SLD-resolution the set of computed instances does not depend on the selection rule and that for such queries the stipulated extension of the LD-resolution coincides with the LD-resolution.

This brings us to the problem of proving absence of errors. This has been taken care of in Apt [1]. To make the paper self-contained we review this method in the setting of (*pre*, *post*)-correct programs. We need the following immediate consequence of the Persistence Lemma 7.3.

Lemma 8.1 *Let P and Q be (pre, post)-correct and let ξ be an LD-derivation of $P \cup \{Q\}$. Then $pre \models A$ for every atom A selected in ξ.*

Proof. The first atom of every (*pre*, *post*)-correct query is true in *pre*. □

To apply it to a program P and a query Q that use arithmetic relations it suffices to find a pair *pre*, *post* of Herbrand interpretations such that

- P and Q are (*pre*, *post*)-correct,
- for arithmetic comparison operators `p`, $pre \models$ `p(s,t)` implies `s,t` are gae,
- for the `is` relation $pre \models$ `s is t` implies `t` is gae.

Then the LD-derivations of $P \cup \{Q\}$ do not end in error. The following two examples show an application of this methodology.

Example 8.2
Consider the following program `LENGTH`:

```
length([], 0).
length([X | Ts], N)  ←  length(Ts, M), N is M+1.
```

Let

$$pre = \{\texttt{length(s,t)} \mid \texttt{s,t} \text{ are ground}\} \cup \{\texttt{s is t} \mid \texttt{t} \text{ is gae}\},$$

$$post = \{\texttt{length(s,t)} \mid \texttt{s,t} \text{ are ground, } \texttt{t} \text{ is gae}\} \cup \{\texttt{s is t} \mid \texttt{s,t} \text{ are gae}\}.$$

It is easy to see that then `LENGTH` and all queries of the form `length(s,t)` are (*pre*, *post*)-correct. Thus for all `s,t` the LD-derivations of `LENGTH` $\cup$ {`length(s,t)`} do not end in error.

Moreover, it is easy to check that the conditions SYN1 and SEM2 of Section 4 apply to the `LENGTH` program, so by Theorem 4.5 `LENGTH` is redundancy-free. So following the procedure explained in Section 7 we conclude that for a list `s` and a variable `N`

$$\{\texttt{length(s,N)}\}\ \texttt{LENGTH}\ Variant(\{\texttt{length(s,|s|)}\}),$$

where `|s|` is the length of the list `s`.

Example 8.3
Consider the following program DICTIONARY for retrieving a pair $(key, value)$ in a dictionary organized as a binary search tree (in short, a bst):

```
lookup(X, V, tree((Y,V), L, R)) ← X =:= Y.
lookup(X, V, tree((Y,_), L, R)) ← X < Y, lookup(X, V, L).
lookup(X, V, tree((Y,_), L, R)) ← X > Y, lookup(X, V, R).
```

This program is a simplified version of program 15.9 from Sterling and Shapiro [17]. Here, a bst is represented by either the constant `void`, denoting the empty bst, or by the term `tree((x, v), l, r)`, where `x` is a gae, `v` is a term, `l` and `r` are bst's, `x` is greater than the keys occurring in the left subtree, and smaller than the keys occurring in the right subtree. The program uses the arithmetic equality built-in `=:=`, which, similarly to `>`, `<`, etc., evaluates both arguments before comparison.

This program has been designed to be queried with bst's in the third argument of `lookup`. As a result the construction of $\mathcal{M}(\mathtt{DICTIONARY})$ is particularly awkward. Recall that by the Soundness and Completeness of the SLD-resolution, $\mathcal{M}(\mathtt{DICTIONARY})$ coincides with the set of successful ground atomic queries. However, a ground query `lookup(x,v,t)` with an unordered binary tree `t`, can either succeed or not, depending on the distribution of the keys in the tree. Take now

$$\begin{aligned} pre = post \;\; = \;\; & \{\mathtt{lookup(x,v,b)} \mid \mathtt{x} \text{ is a gae, } \mathtt{v} \text{ is a ground term and } \mathtt{b} \text{ is a ground bst}\} \\ & \cup \{\mathtt{s} \;=:=\; \mathtt{t} \mid \mathtt{s},\; \mathtt{t} \text{ are gae}\} \\ & \cup \{\mathtt{s} \;<\; \mathtt{t} \mid \mathtt{s},\; \mathtt{t} \text{ are gae}\} \\ & \cup \{\mathtt{s} \;>\; \mathtt{t} \mid \mathtt{s},\; \mathtt{t} \text{ are gae}\}. \end{aligned}$$

It is easy to see that DICTIONARY is then $(pre, post)$-correct, and that on virtue of Theorem 6.7 the following natural interpretation is the well–typed fragment of its least Herbrand model:

$$\begin{aligned} \mathcal{M}_{(pre,post)}(\mathtt{DICTIONARY}) \;\; = \;\; & \{\mathtt{lookup(x,v,b)} \mid \mathtt{x} \text{ is a gae, } \mathtt{b} \text{ is a ground bst} \\ & \quad \text{and } (\mathtt{x},\; \mathtt{v}) \text{ is an element in } \mathtt{b}\} \;\cup\; M_{=:=} \;\cup\; M_{<} \;\cup\; M_{>}. \end{aligned}$$

Also, from Lemma 8.1 it follows that, for any gae `x`, term `v` and bst `b`, the LD-derivations of $\mathtt{DICTIONARY} \cup \{\mathtt{lookup(x,v,b)}\}$ do not end in error. Condition SYN2 of Section 4 readily applies to the DICTIONARY program. To check condition SEM1 of Theorem 7.4, it suffices to consider three ground instances of the three clauses of the program with a common head, namely

$$\begin{aligned} \mathtt{lookup(x,v,tree((y,v),l,r))} \;\; & \leftarrow \;\; \mathtt{x} =:= \mathtt{y}. \\ \mathtt{lookup(x,v,tree((y,v),l,r))} \;\; & \leftarrow \;\; \mathtt{x} < \mathtt{y}, \mathtt{lookup(x,v,l)}. \\ \mathtt{lookup(x,v,tree((y,v),l,r))} \;\; & \leftarrow \;\; \mathtt{x} > \mathtt{y}, \mathtt{lookup(x,v,r)}. \end{aligned}$$

and observe that, for any two gae `x` and `y`, exactly one among `x =:= y`, `x < y` and `x > y` holds in $\mathcal{M}_{(pre,post)}(\mathtt{DICTIONARY})$. This implies that SEM1 applies to the DICTIONARY program, which is therefore $(pre, post)$-redundancy-free. As a conclusion, following the procedure explained in Section 7 we have that for a gae `x`, a variable `V` and a bst `b`

$$\{\mathtt{lookup(x,V,b)}\} \; \mathtt{DICTIONARY} \; Variant(\{\mathtt{lookup(x,v,b)} \;\mid\; (\mathtt{x},\mathtt{v}) \text{ is an element of } \mathtt{b}\}).$$

□

9 Conclusions

We now present a list of example programs from the book of Sterling and Shapiro [17] 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 4.5 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 followed the approach of Section 8 and 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 here 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".

program	page	subsumption free	redundant 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
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

Thus, for many "natural" 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. Moreover, the listed programs are $(pre, post)$-correct with a natural choice of pre and $post$, which implies that it is possible to reason about the computed instances of the "well-typed" queries using the $\mathcal{M}_{(pre,post)}$-semantics only. This fact is relevant, since, according to our experience, the $\mathcal{M}_{(pre,post)}$-semantics usually coincides with the specification of the program, limited to the ground instances of the intended atomic queries and consequently is relatively easy to construct.

This provides a strong indication that, for most "natural" Prolog programs, it is possible to fully reconstruct the procedural behavior of a program from its declarative specification, a feature that accounts for the unique nature of logic programming.

Acknowledgments

We thank the referees of Apt and Gabbrielli [2] for useful comments. The research of the first and the third 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) and by the HCM grant nr. ERBCHBGCT930496 in the context of the EUROFOCS project.

References

[1] K. R. Apt. Program verification and Prolog. In E. Börger, editor, *Specification and Validation methods for Programming languages and systems.* Oxford University Press, 1994. To appear.

[2] K. R. Apt and M. Gabbrielli. Declarative interpretations reconsidered. In P. van Hentenryck, editor, *Proc. of the Eleventh Int'l Conf. on Logic Programming*, pages 74–89. MIT Press, 1994.

[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 and N. Cocco. Verifying correctness of logic programs. In *Proceedings of TAPSOFT '89*, Lecture Notes in Computer Science, pages 96–110. Springer-Verlag, 1989.

[6] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: theory and applications. *Journal of Logic Programming*, 19-20:149–197, 1994.

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

[8] 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.

[9] 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.

[10] K. Kunen. Some remarks on the completed database. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 978–992. The MIT Press, 1988.

[11] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.

[12] 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.

[13] 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.

[14] S. Ruggieri. Metodi formali per lo sviluppo di programmi logici. Technical report, Dipartimento di Informatica, Università di Pisa, 1994. Tesi di Laurea, in Italian.

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

[16] 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*, Lecture Notes in Computer Science 440, pages 382–383. Springer-Verlag, 1990.

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

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

[19] 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.