NORTH-HOLLAND

A CLOSER LOOK AT DECLARATIVE INTERPRETATIONS*

KRZYSZTOF R. APT, MAURIZIO GABBRIELLI, AND DINO PEDRESCHI

▷ 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 $\mathscr{C}$-semantics, and the $\mathscr{S}$-semantics. Previous results show that a strictly increasing information ordering between these semantics exists for the class of all programs. In particular, the $\mathscr{S}$-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 that associates with a "correctly typed" program the "well-typed" subset of its least Herbrand model. This semantics is used to reason about partial correctness and absence of failures of correctly typed programs. The results are extended to programs with arithmetic. ◁

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

Address correspondence to Krzysztof R. Apt, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, apt@cwi.nl, Maurizio Gabbrielli, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy, gabbri@di.unipi.it, or Dino Pedreschi, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy, pedre@di.unipi.it.

Received December 1994; accepted October 1995.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010

0743-1066/96/$15.00
SSDI 0743-1066(95)00150-6

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 model semantics? We claim that the answer to this question is affirmative by showing that many logic programs and pure Prolog programs (i.e., logic programs using the leftmost selection rule) satisfy a property that 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 to the least term model, and for others not. Second, 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 [8] (or $\mathscr{C}$-semantics of Falaschi et al. [12]) 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 [19]. However, it shares the same deficiencies with the least Herbrand model.

The last choice is the $\mathscr{S}$-semantics proposed by Falaschi et al. [11]. 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 and absence of failures 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 [16] which is based on the $\mathscr{S}$-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, whereas the program usually will be used only with "well-typed" arguments. As a result, the least Herbrand models are often 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 and absence of failures of logic programs.

Finally, we extend these results to pure Prolog with arithmetic built-in's.

1.2. *A Word on Terminology*

Unless otherwise specified, 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 and E_2, we say that E_1 *is more general than* E_2 and write $E_1 \leq E_2$ if there exists a substitution θ such that $E_1\theta = E_2$. The relation $\leq$ is called the subsumption pre-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 $\mathrm{Var}(E)$ the set of all variables occurring in the expression E.

A substitution is 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 $\mathscr{B}$ the set of all atoms (the *base* of the language) and by $\mathscr{B}_\mathscr{L}$ 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 [14].

In short, a query is a finite sequence of atoms, denoted by letters Q, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, ... 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 $\{Q\}P\mathscr{C}$ to denote the fact that $\mathscr{C}$ is the set of computed instances of the query Q, and we denote the set of computed instances of the query Q by $\mathrm{sp}(Q, P)$ (for *strongest postcondition* of Q w.r.t. P). So by definition $\{Q\}P\,\mathrm{sp}(Q, P)$ for any Q and P. Given two queries Q and Q', we write

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

So $\mathrm{mgi}(Q, Q')$ 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

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

So $\mathrm{mgi}(Q, I)$ 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 ($\mathcal{M}$-Semantics)*

This semantics was introduced by van Emden and Kowalski [22]. 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 [22] showed, this semantics can be characterized by means of the following immediate consequence operator defined on Herbrand interpretations:

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

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-fixed point 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 [5]), for a ground Q a successful SLD derivation of Q exists iff $Q \in \mathcal{M}(P)^*$. However, for nonground queries, the situation changes as the following example of Drabent and Maluszynski [10] shows.

Example 2.1. 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 does not characterize precisely the computed answers. This is an undesirable situation for program verification and analysis, because in these cases usually one needs to reason on the operational behaviour of programs in terms of their computed answers.

2.2. *The Least Term Model ($\mathscr{C}$-Semantics)*

This semantics was introduced by Clark [8] and more extensively studied in Falaschi et al. [12]. 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

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

Falaschi et al. [12] showed that this semantics also can be characterized by means of an operator defined on term interpretations:

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

where inst(P) denotes the set of all the instances of clauses in P. Then they established the following theorem analogous to the $\mathscr{M}$-characterization of Theorem 2.1.

Theorem 2.2 ($\mathscr{C}$-characterization)

(i) *U_P is continuous on the complete lattice of term interpretations ordered with $\subseteq$.*
(ii) *$\mathscr{C}(P)$ is the least pre-fixpoint and the least fixpoint of U_P.*
(iii) $\mathscr{C}(P) = U_P \uparrow \omega$.

However, the $\mathscr{C}$-semantics cannot model the operational behaviour of a program either, because for Example 2.1 we have also $\mathscr{C}(P_1) = \mathscr{C}(P_2)$.

2.3. *$\mathscr{S}$-Semantics*

This semantics was introduced in Falaschi et al. [11]. For a survey on the $\mathscr{S}$-semantics and its uses, see Bossi et al. [7]. 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 program P can be identified with the set of all pairs (Q, θ), where θ is a compute answer substitution for Q, or, equivalently, with the set of all statements of the form $\{Q\}P\mathscr{C}$.

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

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

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 $\mathscr{S}(P)$. However, this is not so, as the following theorem of Falaschi et al. [11] shows.

Theorem 2.3 (Strong completeness). For a program P and a query Q,

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

Consequently, by the form of $\mathscr{S}(P)$ we have the following corollary.

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

Corollary 2.1 (Full abstraction). For all programs P_1, P_2,

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

An important property of the $\mathscr{S}$-semantics is that it can be defined by means of a fixpoint construction. More precisely, Falaschi et al. [11] introduced the following operator on term interpretations,

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

and proved the following theorem.

Theorem 2.4 ($\mathscr{S}$-characterization)

(i) $T_P^{\mathscr{S}}$ *is continuous on the complete lattice of term interpretations ordered with* $\subseteq$.
(ii) $\mathscr{S}(\mathscr{P})$ *is the least fixpoint and the least pre-fixpoint of* $T_P^{\mathscr{S}}$.
(iii) $\mathscr{S}(\mathscr{P}) = T_P^{\mathscr{S}} \uparrow \omega$.

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 $\mathscr{D}$.

Definition 3.1. Consider a class of program $\mathbf{C}$. We say that two semantics $\mathscr{S}_1$: $\mathbf{C} \to \mathscr{D}_1$ and $\mathscr{S}_2$: $\mathbf{C} \to \mathscr{D}_2$ are *isomorphic* on $\mathbf{C}$ iff there exist two functions ϕ_1: $\mathscr{D}_1 \to \mathscr{D}_2$ and ϕ_2: $\mathscr{D}_2 \to \mathscr{D}_1$ such that, for any program $P \in \mathbf{C}$,

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

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

Every semantics $\mathscr{T}$, for $\mathbf{C}$ induces an equivalence relation $\approx_{\mathscr{T}}$ on programs from $\mathbf{C}$ defined by $P_1 \approx_{\mathscr{T}} P_2$ iff $\mathscr{T}(P_1) = \mathscr{T}(P_2)$. Note that the notion of isomorphism also can be 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) $\mathrm{Variant}(I) = \{A \in \mathscr{B} \mid \exists B \in I \text{ s.t. } B \leq A \text{ and } A \leq B\}$, the set of variants.
(ii) $\mathrm{Up}(I) = \{A \in \mathscr{B} \mid \exists B \in I \text{ s.t. } B \leq A\}$, the set of instances.
(iii) $\mathrm{Ground}(I) = \{A \in \mathscr{B}_H \mid \exists B \in I \text{ s.t. } B \leq A\}$, the set of ground instances.
(iv) $\mathrm{Min}(I) = \{A \in I \mid \neg \exists B \in I \text{ s.t. } B < A\}$, the set of minimal (i.e., most general) elements.
(v) For I a set of ground atoms, $\mathrm{True}(I) = \{A \in \mathscr{B} \mid I \models A\}$, the set of atoms true in the Herbrand interpretation I.

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

NOTE 3.1. For all I, $\mathrm{Min}(\mathrm{Up}(I)) = \mathrm{Min}(I)$.

3.1. Relating $\mathscr{M}$-Semantics and $\mathscr{C}$-Semantics

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

NOTE 3.2. $\mathscr{M}(P) = \mathrm{Ground}(\mathscr{C}(P))$.

Therefore, the $\mathscr{M}$-semantics can be reconstructed from the $\mathscr{C}$-semantics. The converse does not hold, in general, as the following argument due to Falaschi et al. [12] shows.

Example 3.1. Consider two programs: P_1,

```
p(X).
```

and P_2,

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

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

In case the signature contains infinitely many constants, the situation changes, as the following result due to Maher [15] shows.

Theorem 3.3. Assume that the signature contains infinitely many constants. Then

$$\mathscr{C}(P) = \mathrm{True}(\mathscr{M}(P)).$$

PROOF. We provide here an alternative, direct proof based on the theory of SLD resolution. The implication $\mathscr{C}(P) \subseteq \mathrm{True}(\mathscr{M}(P))$ always holds, because $\mathscr{M}(P)$ is a model of P. Take now $A \in \mathrm{True}(\mathscr{M}(P))$. Let $x_1, \ldots, x_n$ be the variables of A and let $c_1, \ldots, c_n$ be distinct constants that do not appear in P or A. Let $\theta = \{x_1/c_1, \ldots, x_n/c_n\}$. Then $A\theta \in \mathscr{M}(P)$. 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, $A \in \mathscr{C}(P)$. □

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

3.2. Relating $\mathscr{C}$-Semantics and $\mathscr{S}$-Semantics

Next, we clarify the relationship between $\mathscr{C}(P)$ and $\mathscr{S}(P)$. First, we have the following result of Falaschi et al. [12].

Theorem 3.4. $\mathscr{C}(P) = \mathrm{Up}(\mathscr{S}(P))$.

Therefore, the $\mathscr{C}$-semantics can be reconstructed from the $\mathscr{S}$-semantics. The converse does not hold, in general, as the following argument due to Falaschi et al. [11] shows.

Example 3.2. Consider the programs P_1,

```
p(X).
```

and P_2,

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

of Example 2.1. Then $\mathscr{C}(P_1) = \mathscr{C}(P_2) = \mathrm{Up}(\{p(X)\})$, whereas $\mathscr{S}(P_1) = \mathrm{Variant}(\{p(X)\})$ and $\mathscr{S}(P_2) = \mathrm{Variant}(\{p(X), p(a)\})$. Note that the signature of the language was immaterial here.

Thus on the class of all programs, the $\mathscr{C}$-semantics and the $\mathscr{S}$-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 3.1. $\mathrm{Min}(\mathscr{C}(P)) \subseteq \mathscr{S}(P)$.

Intuitively, Lemma 3.1 states that all most general atoms true in $\mathscr{C}(P)$ belong to $\mathscr{S}(P)$.

PROOF. By Theorem 3.4, $\mathrm{Min}(\mathscr{C}(P)) = \mathrm{Min}(\mathrm{Up}(\mathscr{S}(P)))$ and the claim follows by Note 3.1, because for all I, we have $\mathrm{Min}(I) \subseteq I$. □

In general, the converse inclusion does not hold.

Example 3.3. Consider the following program P,

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

defined w.r.t. the language with the signature $\Sigma = \{a/0\}$. Then $\mathscr{S}(P) = \mathrm{Variant}(\{p(Y)\}) \cup \{p(a)\}$, whereas $\mathrm{Min}(\mathscr{C}(P)) = \mathrm{Variant}(\{p(Y)\})$.

A closer examination of the situation reveals the following information: By the soundness of the SLD resolution we always have $\mathscr{S}(P) \subseteq \mathscr{C}(P)$. Example 3.3 shows that the stronger inclusion $\mathscr{S}(P) \subseteq \mathrm{Min}(\mathscr{C}(P))$ does not need to hold. The reason is that $\mathscr{S}(P)$ can contain a pair A, B such that A strictly subsumes B (i.e., $A < B$). This cannot happen when $\mathscr{S}(P)$ contains only minimal elements, so we are brought to the following definition due to Maher and Ramakrishnan [16].

Definition 3.3. A set of atoms I is called *subsumption-free* if $\mathrm{Min}(I) = I$. A program P is called *subsumption-free* if $\mathscr{S}(P)$ is.

We now show that the notion of a subsumption-free program is a key for establishing the converse of Lemma 3.1.

Theorem 3.5. $\mathscr{S}(P) = \text{Min}(\mathscr{C}(P))$ *iff* P *is subsumption-free.*

PROOF. ($\Rightarrow$) We have

$$\begin{aligned}
&\text{Min}(\mathscr{S}(P))\\
=\ &\{\text{assumption}\}\\
&\text{Min}(\text{Min}(\mathscr{C}(P)))\\
=\ &\{\text{idempotence of Min}\}\\
&\text{Min}(\mathscr{C}(P))\\
=\ &\{\text{assumption}\}\\
&\mathscr{S}(P).
\end{aligned}$$

($\Leftarrow$) We have

$$\begin{aligned}
&\mathscr{S}(P)\\
=\ &\{\text{assumption}\}\\
&\text{Min}(\text{S}(P))\\
=\ &\{\text{Note 3.1}\}\\
&\text{Min}(\text{Up}(\mathscr{S}(P)))\\
=\ &\{\text{Theorem 3.4}\}\\
&\text{Min}(\mathscr{C}(P)).
\end{aligned}$$

□

Consequently, the $\mathscr{C}$-semantics and $\mathscr{S}$-semantics are isomorphic on subsumption-free programs. Additionally, when the signature contains infinitely many constants, all three semantics are isomorphic. Combining Theorems 2.3, 3.3 and 3.5 we thus obtain the following corollary.

Corollary 3.1. Assume that the signature contains infinitely many constants. Then for a subsumption-free program P *and a query* Q,

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

Corollary 3.1 shows that computed answers of subsumption-free programs can be fully reconstructed from the $\mathscr{M}$-semantics using unification and, therefore, the least Herbrand model can be used to reason about partial correctness.

The point of view taken in this paper is that the $\mathscr{M}$-semantics is handier than the $\mathscr{S}$-semantics for reasoning about partial correctness. By the $\mathscr{M}$-characterization of Theorem 2.1 and the $\mathscr{S}$-characterization of Theorem 2.4, both $\mathscr{M}(P)$ and $\mathscr{S}(P)$ are obtained as the ω power of a suitable monotonic operator associated with the program under consideration and, therefore, a natural form of inductive reasoning can be adopted to construct either semantics. However, in the case of $\mathscr{S}$-semantics, it is necessary to deal with sets of nonground atoms, which entails dealing with general substitutions of variables with nonground terms—a process that may be very elaborate in practice. Moreover, for a large class of programs it is possible to show that a given interpretation coincides with the $\mathscr{M}$-semantics in a straightforward way, without using inductive arguments. This technique is briefly discussed in Section 6.

On the other hand, the $\mathscr{S}$-semantics is in general more compact than the $\mathscr{M}$-semantics. For instance, the $\mathscr{S}$-semantics of both programs given in Example 2.1 is a finite set (modulo variable renaming), whereas the $\mathscr{M}$-semantics of both such programs is infinite (albeit easy to describe in an infinitary language) when the signature contains infinitely many constants. However, this advantage of the $\mathscr{S}$-semantics is hardly useful when conducting "paper and pencil" proofs of partial correctness, due to the complications arising from manipulation of arbitrary substitutions. 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 **C**, subsumption-freedom is only a sufficient condition for the isomorphism of the $\mathscr{C}$-semantics and $\mathscr{S}$-semantics. Indeed, when the class of programs consists of just the program from Example 3.3, which is not subsumption-free, then the $\mathscr{C}$-semantics and $\mathscr{S}$-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 3.4. A class of programs **C** is *$\mathscr{S}$-closed* if, for every program P in $\mathscr{C}$, every finite subset of $\mathscr{S}(P)$ is in **C**.

Indeed, we have the following result.

NOTE 3.6. For an $\mathscr{S}$-closed class **C** of programs, the $\mathscr{C}$-semantics and $\mathscr{S}$-semantics are isomorphic on **C** iff **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 $A, B \in \mathscr{S}(P)$ we have $A < B$. By the definition of $\mathscr{S}$-closedness, both $P_1 = \{A, B\}$ and $P_2 = \{A\}$ are in **C**. Now $\mathscr{C}(P_1) = \mathrm{Up}(\{A, B\}) = \mathrm{Up}(\{A\}) = \mathscr{C}(P_2)$, whereas $\mathscr{S}(P_1) = \mathrm{Variant}(\{A, B\}) \neq \mathrm{Variant}(\{A\}) = \mathscr{S}(P_2)$. Contradiction.

($\Leftarrow$) This is the contents of Theorems 3.4 and 3.5. □

The foregoing proof 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. REDUNDANCY-FREE PROGRAMS

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

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

Clearly, redundancy-freedom implies subsumption-freedom, because $\mathscr{S}(\mathscr{P})$ is closed under renaming and $A < B$ implies that A and a variant B' of B are nonvariant and unifiable. The converse does not hold.

Example 4.1. Consider the following program P defined w.r.t. the language with the signature $\Sigma =$ {a/0}:

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

Then $\mathscr{S}(P) = \text{Variant}(\{\texttt{p(X, a)}, \texttt{p(a, X)}\})$, so P is not redundancy-free. However, it is clearly subsumption-free because the atoms `p(X, a)` and `p(a, X)` are not comparable in the subsumption pre-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 4.1

(i) *P is subsumption-free iff for all pure atoms A,* $\text{Min}(\text{sp}(A, P)) = \text{sp}(A, P)$.
(ii) *P is redundancy-free iff for all queries Q,* $\text{Min}(\text{sp}(Q, P)) = \text{sp}(Q, P)$.

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

(ii) ($\Rightarrow$) Consider two computed instances Q_1 and Q_2 of Q. By Theorem 2.3 there exist $\mathbf{C}_1$ and $\mathbf{C}_2$ in $\mathscr{S}(P)^*$ such that, for $i \in [1,2]$, Q and $\mathbf{C}_i$ are variable disjoint and

$$Q_i \in \text{mgi}(Q, \mathbf{C}_i). \tag{4.1}$$

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 $\mathbf{C}_2$. Because 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 (4.1) that Q_1 and Q_2 are variants, as well. Contradiction.

($\Leftarrow$) Suppose that $\mathscr{S}(P)$ does contain a pair A, B of nonvariant unifiable atoms. Let $C \in \text{mgi}(A, B)$. 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 2.3 A, $C \in \text{sp}(A', P)$, so $\text{Min}(\text{sp}(A', P)) \neq \text{sp}(A', P)$. Contradiction. □

For redundancy-free programs we can simplify the formulation of Corollary 3.1.

Corollary 4.1. Consider a redundancy-free program P and a query Q. Then:

(i) $\{Q\}P\ \text{Min}(\{Q\theta \mid P \models Q\theta\})$.
(ii) $\{Q\}P\ \text{Min}(\{Q\theta \mid \mathscr{C}(P) \models Q\theta\})$.
(iii) *If the signature contains infinitely many constant symbols,*

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

PROOF. (i) follows from Theorem 4.1(ii) and the following two claims.

Claim 1. For an arbitrary program P and a query Q,

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

PROOF. Take $Q_1 \in \text{Min}(\{Q\theta \mid P \models Q\theta\})$. 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., $Q_1 \in \mathrm{sp}(Q, P)$. □

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

The proof is immediate.

Now (ii) is a straightforward consequence of (i) and the definition of the $\mathscr{C}$-semantics. Finally, (iii) follows from (ii) and Theorem 3.3. □

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 it is subsumption-free.

Theorem 4.2. 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*

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

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

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

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*

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

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,

$$\mathscr{M}(P) \models \mathbf{B}\vartheta_2,$$

but by the standardization part $\mathbf{B}\vartheta_1 = \mathbf{B}$, so in fact

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

which concludes the proof. □

We prove now the contrapositive. Assume that the program P is not redundancy-free. By Theorem 4.1 there exists a query Q that 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 let $d_1, \ldots, d_m$ $(m \geq 1)$ be the sequence of clauses of P used in ξ''. Next, consider k $(1 \leq k \leq \min(n, m))$ such that

$$c_i = d_i, \quad \text{for } i \in [1, k-1],$$

$$c_k \neq d_k.$$

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 nonunifiability 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

$$R_i\gamma \text{ and } R_i\delta \text{ unify} \quad \text{for } i \in [1, j-1],$$

$$R_j\gamma \text{ and } R_j\delta \text{ do not unify.} \tag{4.2}$$

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

$$\mathrm{Var}((K \leftarrow \mathbf{B})\gamma) \cap \mathrm{Var}((K \leftarrow \mathbf{B})\delta) = \varnothing. \tag{4.3}$$

From the definition of the R_is and from (4.2) it follows that $K\gamma$ and $K\delta$ unify, whereas $\mathbf{B}\gamma$ and $\mathbf{B}\delta$ are not unifiable. This, together with (4.3), 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 that uses $\mathcal{M}(P) \not\models H \wedge \mathbf{B}_1 \wedge \mathbf{B}_2$, because 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 clause heads of P unify.

SEM. If $H \leftarrow \mathbf{B}_1, H \leftarrow \mathbf{B}_2 \in \mathrm{Ground}(P)$ 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 [16] 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 and 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 4.1. 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 $H_1\eta_1 = H_2\eta_2$. However, $\mathcal{M}(P) = \text{Ground}(\mathcal{S}(P))$, 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

$$\mathbf{B}_1\eta_1 = \mathbf{C}_1\gamma_1, \tag{4.4}$$

$$\mathbf{B}_1\eta_2 = \mathbf{C}_2\gamma_2. \tag{4.5}$$

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, (4.4) and (4.5) imply that there exists ϑ_1, ϑ_2, β_1, and β_2 such that

$$\vartheta_1 \text{ is a relevant mgu of } \mathbf{B}_1 \text{ and } \mathbf{C}_1, \qquad H_1\eta_1 = H_1\vartheta_1\beta_1, \tag{4.6}$$

$$\vartheta_2 \text{ is a relevant mgu of } \mathbf{B}_2 \text{ and } \mathbf{C}_2, \qquad H_2\eta_2 = H_2\vartheta_2\beta_2. \tag{4.7}$$

Consider now $A = H_1\vartheta_1$ and $B = H_2\vartheta_2$. From (4.6) and (4.7) 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 $\text{Var}(H_i) \cap \text{Var}(H_j \leftarrow \mathbf{B}_j) = \varnothing$ and $\text{Var}(H_i) \cap \text{Var}(\mathbf{C}_j) = \varnothing$, 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 $H_1\eta_1 = H_2\eta_2$, (4.6), and (4.7) 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,

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

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

Because 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 3.2 and Theorem 3.4 imply $\mathcal{M}(P) = \text{Ground}(\mathcal{S}(P))$. Therefore,

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

because $\mathbf{C}_i \in \mathscr{S}(P)^*$ 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 4.2. 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 the SEM1 condition.

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

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

Theorem 4.2 remains valid when SEM1 is replaced by SEM1″, because essentially the same proof as in [16] 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 ([20], p. 58), which tests whether two binary trees are isomorphic. Clearly, condition SEM2 is satisfied by `ISO_TREE`, because actually its stronger version SYN2 defined at the end of this section holds, but SEM1 does not hold because

```
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

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

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 mgts and `X` is a variable that appears neither in $\texttt{t}_1$ nor in $\texttt{t}_2$, then $\texttt{tree(X, t}_1\texttt{, t}_2\texttt{)}$ is a mgt.

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

(i) If $\texttt{iso(t}_1\texttt{, t}_2\texttt{)} \in \mathscr{S}(\texttt{ISO\_TREE})$, then $\texttt{t}_1$ and $\texttt{t}_2$ are mgts.
(ii) If $\texttt{t}_1$ and $\texttt{t}_2$ are unifiable mgts, then they are variants.

In order to show that SEM1″ holds for the program `ISO_TREE`, let us consider two atoms $A \in T^{\mathscr{S}}_{\{c\}}(\mathscr{S}(\texttt{ISO\_TREE}))$ and $B \in T^{\mathscr{S}}_{\{d\}}(\mathscr{S}(\texttt{ISO\_TREE}))$, 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

iso(t_1, t_2), iso(t_3, t_4), iso(1_1, 1_2), and iso(1_3, 1_4) are pairwise variable disjoint atoms in $\mathscr{S}$(ISO_TREE) such that

ϑ_2 is an mgu of $\mathbf{B}_2$ and iso(t_1, t_2), iso(t_3, t_4),

ϑ_3 is an mgu of $\mathbf{B}_3$ and iso(1_1, 1_2), iso(1_3, 1_4),

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

A = iso(tree(X, t_1, t_3), tree(X, t_2, t_4)),

B = iso(tree(Y, 1_1, 1_3), tree(Y, 1_4, 1_2)).

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 4.2 can be ensured by means of syntactic restrictions, namely, condition SEM1 is obviously implied by the following 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,

In addition, condition SEM2 is automatically satisfied when the following condition holds:

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

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 Turning completeness for redundancy-free programs can be obtained by using the encoding of two register machines into pure logic programs given in Shepherdson [18]. 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 4.2.

5. CHECKING REDUNDANCY-FREEDOM

We provide here four illustrative uses of Theorem 4.2.

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 `suffix(s, t_s)` where t_s is a term containing the term `s`. By definition of T_P, $T_{SUFFIX}(OCC) \subseteq OCC$, i.e., *OCC* is a pre-fixpoint of T_{SUFFIX}. By the $\mathcal{M}$-characterization Theorem 2.1, $\mathcal{M}$(SUFFIX) $\subseteq OCC$, so, for any ground instance

```
suffix(t1, [t2|t3]) ← suffix(t1, t3)
```

of the second clause, if $\mathcal{M}$(SUFFIX) $\models$ `suffix(t1, t3)`, then t_1 and `[t2|t3]` are different terms. Thus `suffix(t1, [t2|t3])` is not an instance of the first clause and consequently SEM1 holds.

The clauses of SUFFIX do not contain 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 $\mathcal{M}$(REVERSE). Using the list constructor binary function $[\cdot|\cdot]$ let us define the notation $[t_1|t_2 \cdots |t_n]$ 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,

$$[t_1|t_2|\cdots|t_n] = [t_1|[t_2|\cdots|t_n]].$$

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

If $s = [\ ]$, then $s * t = t$.
If $s = [t_1|\cdots|t_{n-1}|[\ ]]$, then $s * t = [t_1|\cdots t_{n-1}|t]$.

Then it can be shown that

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

and

$$\mathcal{M}(\text{REVERSE}) = \{\texttt{reverse(s, t)} \mid \texttt{s, t} \text{ are ground lists and } \texttt{t=rev(s)}\} \cup \mathcal{M}(\text{APPEND}),$$

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 [20], 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 $\mathcal{M}(\text{HANOI}) \models$ `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 $\mathcal{M}(\text{HANOI}) \not\models \mathbf{B}$. This implies SEM1.

To prove condition SEM2 we use the methodology of Maher and Ramakrishnan [16] based on functional dependencies. First we need a definition.

Definition 5.1. 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

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

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) 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 nonunit clause for `hanoi`. Consider an instance

```
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)
```

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

Because hanoi[{1, 2, 3, 4} → {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 [(1, 2} → {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 Armstrong axioms (see Ullman [21]), is given in Maher and Ramakrishnan [16].

Note that Theorem 4.2 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 4.2 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 4.1, but more natural programs exist. In such situations we still can use a direct reasoning to prove subsumption-freedom.

Example 5.2. 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 of Theorem 2.4 it suffices to show that if I is subsumption-free, then $T_{\text{MEMBER}}{}^{\mathcal{S}}(I)$ is subsumption-free. Denote the first clause by c_1 and the second one by c_2. Consider a pair $A_1, A_2 \in T_{\text{MEMBER}}{}^{\mathcal{S}}(I)$. The following two cases arise.

Case 1 [$A_1 \in T^{\mathcal{S}}_{\{c_1\}}(I)$ and $A_2 \in T^{\mathcal{S}}_{\{c_2\}}(I)$]. By definition of $T_P^{\mathcal{S}}$, 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 $Y \notin \text{Var}(B)$. This implies X$\vartheta \neq$ Yϑ and hence $A_1 \not\leq A_2$ and $A_2 \not\leq A_1$.

Case 2 [$A_1, A_2 \in T^{\mathcal{S}}_{\{c_2\}}(I)$]. 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 $\vartheta_i = \{X/t_i, Xs/l_i\}$ (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 redundancy-free. In fact, the query member(X, Y) has the computed instances member(X, [X|XS]) and member(X, [Y|[X|XS]]) which are unifiable.

6. FOURTH SEMANTICS—$\mathcal{M}_{(\text{pre, post})}$

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) con-

tains elements of the form append(s,t,u), where neither t nor u is a list, and analogously for $\mathcal{M}$(MEMBER), because it can be shown that

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

Clearly, it is quite clumsy to reason about programs when even in such 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:

$$\{\text{append}(s,t,u) \mid s,\ t,\ u \text{ are ground lists and } s * t = u\} \tag{6.1}$$

and with the MEMBER program the following meaning:

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

To be able to do this we have to find a systematic way of associating with the APPEND program the set (6.1), etc. Note that the set (6.1), 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 [6] for proving partial correctness), but follow the equivalent presentation of Ruggieri [17], 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 $j \in [1, n]$,

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

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

$$H \in \text{pre} \wedge B_1, \ldots, B_{j-1} \in \text{post} \quad \Rightarrow \quad B_j \in \text{pre}, \quad \text{for } j \in [1, n],$$

$$H \in \text{pre} \wedge B_1, \ldots, B_n \in \text{post} \quad \Rightarrow \quad H \in \text{post}.$$

- 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.

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

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

Intuitively, $\mathcal{M}_{(\text{pre,post})}(P)$ is the "well-typed" fragment of the least Herbrand model of a program P. We call it $\mathcal{M}_{(\text{pre,post})}$-semantics. Note that the $\mathcal{M}_{(\text{pre,post})}$-semantics does not depend on post, but as the following result of Ruggieri [17]

shows, for (pre, post)-correct programs $\mathcal{M}_{(\text{pre, post})}(P)$ can be equivalently defined as $\mathcal{M}(P) \cap \text{pre} \cap \text{post}$.

Lemma 6.1. For a (pre, post)*-correct program P we have* $\mathcal{M}_{(\text{pre, post})}(P) \subseteq \text{post}$.

In general, the $\mathcal{M}_{(\text{pre, post})}$-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 6.2. For a (pre, post)*-correct program P and a* (pre, post)*-correct query Q,*

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

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 $A_j \in \text{pre}$ for $j \in [1, n]$. For the base case ($n = 0$), the claim holds vacuously. For the induction step ($n > 0$), we have $A_1, \ldots, A_{n-1} \in \text{pre}$ by the induction hypothesis. Together with the assumption $A_1, \ldots, A_{n-1} \in \mathcal{M}(P)$ this implies $A_1, \ldots, A_{n-1} \in \text{post}$ by Lemma 6.1. By the fact that $A_1, \ldots, A_n$ is (pre, post)-correct, we conclude that $A_n \in \text{pre}$, which completes the proof of the first implication.

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

The following example should clarify the idea behind this approach to types. Here and in other natural cases $\text{post} \subseteq \text{pre}$. Then by Lemma 6.2 we have $\mathcal{M}_{(\text{pre, post})}(P) = \mathcal{M}(P) \cap \text{post}$, which makes the $\mathcal{M}_{(\text{pre, post})}$-semantics somewhat easier to construct.

Example 6.1. 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:

pre = {`append(s,t,u)` | `s, t` are ground lists and `u` is a ground term}

$\cup$ {`append(s,t,u)` | `s, t` are ground terms and `u` is a ground list}.

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 = {`append(s,t,u)` | `s,t,u` are ground lists}.

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

$\mathcal{M}_{(\text{pre, post})}$(`APPEND`)

= {`append(s,t,u)` | `s, t, u` are ground lists and `s*t=u`}.

Example 6.1 shows how to construct the set $\mathcal{M}_{(\text{pre, post})}(P)$ by using the least Herbrand model $\mathcal{M}(P)$. However, as we already noticed, the construction of $\mathcal{M}(P)$ can be quite cumbersome, so we would prefer to define $\mathcal{M}_{(\text{pre, post})}(P)$ 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 6.2. 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,

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

As a direct consequence of the definition, observe that

$$T_{J(P)}(I) = T_P(I) \cap J \tag{6.2}$$

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 $\mathcal{M}_{(\text{pre,post})}$-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 6.1. For a (pre, post)*-correct program* P,

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

PROOF. By the $\mathcal{M}$-characterization of Theorem 2.1, $\mathcal{M}_{(\text{pre,post})}(P) = T_P \uparrow \omega \cap \text{pre}$ and $\mathcal{M}(\text{pre}(P)) = T_{\text{pre}(P)} \uparrow \omega$. Now, on the account of (6.2), we have $T_{J(P)} \uparrow \omega \subseteq T_P \uparrow \omega \cap J$, for all J, so for pre in particular. Thus $\mathcal{M}(\text{pre}(P)) \subseteq \mathcal{M}_{(\text{pre,post})}(P)$.

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

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

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

$$\{B_1 \cdots B_m\} \subseteq T_P \uparrow (n-1). \tag{6.3}$$

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

$$\{B_1 \cdots B_m\} \subseteq \text{pre} \tag{6.4}$$

holds. Indeed, for the base case ($m = 0$), the claim holds vacuously. For the induction step ($m > 0$), assume that $\{B_1 \cdots B_{m-1}\} \subseteq \text{pre}$. This together with (6.3) implies $\{B_1 \cdots B_{m-1}\} \subseteq \mathcal{M}_{(\text{pre,post})}(P)$ and hence, by Lemma 6.1, $\{B_1 \cdots B_{m-1}\} \subseteq \text{post}$ holds. Because by assumption $H \in \text{pre}$, it follows from Definition 1 that $B_m \in \text{pre}$.

Now the induction hypothesis, (6.3), and (6.4) imply $\{B_1 \cdots B_m\} \subseteq T_{\text{pre}(P)} \uparrow (n-1)$ and, consequently, $H \in T_{\text{pre}(P)} \uparrow n$, which concludes the proof. □

This allows us to conclude that the $\mathcal{M}_{(\text{pre,post})}$-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, 2.2, and 2.4.

Theorem 6.2 ($\mathcal{M}_{(\text{pre,post})}$-characterization 1). For a (pre, post)*-correct program* P:

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

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

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

PROOF. We already noticed that (i) is a consequence of (6.2). (ii) and (iii) follow directly from Theorem 6.1 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 $\mathcal{M}_{(\text{pre, post})}$-semantics in terms of post(P) can be given, namely, we have the following analogue of Theorem 6.2.

Theorem 6.3 ($\mathcal{M}_{(\text{pre, post})}$-characterization 2). *Suppose that* post $\subseteq$ pre. *Then, for a* (pre, post)*-correct program* P:

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

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

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

PROOF. By Lemma 6.1, $\mathcal{M}_{(\text{pre, post})}(P) \subseteq$ post. Thus to prove (ii) and (iii) it suffices to prove by the $\mathcal{M}_{(\text{pre, post})}$-characterization 1 of Theorem 6.2 that post $\subseteq$ pre implies that, for $n \geq 0$,

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

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 $H \in T_{\text{pre}(P)} \uparrow n \cap$ post. Then there exists a ground instance $H \leftarrow B_1 \cdots B_m$ of a clause in pre(P) such that

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

By Lemma 6.1 and the $\mathcal{M}_{(\text{pre, post})}$-characterization 1 of Theorem 6.2, we also have

$$\{B_1 \cdots B_m\} \subseteq \text{post},$$

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

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

This concludes the proof. □

Returning to Example 6.1, note that using the above theorem it is now easy to construct $\mathcal{M}_{(\text{pre, post})}$(APPEND) by proving by induction on $n > 0$ that

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

Finally, let us remark that for a large class of programs it is possible to verify that a Herbrand interpretation coincides with the $\mathcal{M}_{(\text{pre, post})}$-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 $A \leftarrow \mathbf{B} \in \text{Ground}(P)$ and $I \models \mathbf{B}$.

In Apt and Pedreschi [4] 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 [4] 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 $\mathcal{M}_{(\text{pre},\text{post})}$-characterization of Theorems 6.2 and 6.3 we have that $\mathcal{M}_{(\text{pre},\text{post})}(P)$ 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

When dealing with correctness of logic programs, one needs to prove the following properties for a given program and a "relevant" query:

- All its SLD derivations terminate.
- All successful SLD derivations yield the desired results.
- Absence of failure, that is an existence of a successful SLD derivation.

The first property has been dealt with in numerous papers and is not discussed here. The second property is usually referred to as partial correctness. Partial correctness of logic programs has been studied for a long time (see, for example, Deransart [9], where various approaches are discussed and compared). Among them the most powerful one is the inductive assertion method of Drabent and Małuszyński [10] that allows us to prove various program properties that can be expressed only using nonmonotonic assertions (like `var(X)`). Various other, simpler cases of this method were presented in the literature. Apt and Marchiori [3] provided a systematic, comparative study of the relative strength and expressive power of these versions of the inductive assertion method and showed that they can be arranged in a natural hierarchy.

In contrast, we are not familiar with any approaches to prove the third property —absence of failures. In what follows we show how the results of the previous sections can be applied to prove this property *together* with the proof of partial correctness.

The point of departure in our approach is the observation that logic and pure Prolog programs can yield several answers and, consequently, partial correctness could be interpreted in two ways.

Take as an example the APPEND program. It is natural that for the query `append([1,2], [3,4], Zs)` we would like to prove that upon successful termination, the variable `Zs` is instantiated to `[1,2,3,4]`, that is, that `{Zs/[1,2,3,4]}` is the computed answer substitution.

On the other hand, for the query `append(Xs, Ys, [1,2,3,4])` we would like to prove that *all* possible splittings of the list `[1,2,3,4])` can be produced. This means that for this query we would like to prove that each of the substitutions

```
{Xs/[ ], Ys/[1, 2, 3, 4]},
{Xs/[1], Ys/[2, 3, 4]},
{Xs/[1,2], Ys/[3,4]},
{Xs/[1, 2, 3], Ys/[4]},
{Xs/[1, 2, 3, 4], Ys/[ ]}
```

is a possible computed answer substitution to the query `append(Xs, Ys, [1,2,3,4])`.

Moreover, we should also prove that *no* other answer can be produced. This boils down to the claim that the above set of substitutions coincides with the set of all computed answer substitutions. Of course, a similar strengthening is possible in the case of the first query. We would prove that the query `append([1,2], [3,4], Zs)` admits *precisely* one computed answer substitution, namely, `{Zs/[1,2,3,4]}`.

Note that such a stronger formulation of partial correctness automatically takes care of the proof of the last property—absence of failure. Indeed, this property reduces to the statement that the set of computed answer substitutions is nonempty. This explains why this formulation of partial correctness is beyond the scope of other methods.

In the terminology introduced in Section 1.2 for a given program P and a query Q, we thus wish to prove assertions of the form $\{Q\}P\mathscr{Q}$. In particular, we would like to prove that

```
{append([1,2], [3,4], Zs)} APPEND {append([1,2],
                                        [3,4], [1,2,3,4])}
```

and

```
{append(Xs, Ys, [1,2,3,4])} APPEND 𝒬,
```

where

$Q =$

```
{
append([ ], [1,2,3,4], [1,2,3,4],
append([1], [2,3,4], [1,2,3,4]),
append([1,2], [3,4], [1,2,3,4]),
append([1,2,3], [4], [1,2,3,4]),
append([1,2,3,4], [ ], [1,2,3,4])
}.
```

In Apt [1] it was shown how these properties can be established using the least Herbrand model. However, this approach was limited only to the case of "ground" inputs (or more precisely, to the queries with only "ground" computed instances). We now show that in the case of subsumption-free programs this approach can be generalized to arbitrary queries.

To this end one should perform the steps listed below. We illustrate this technique by means of an example that shows that this method can also deal with programs that use logical variables. Consider the following program REVERSE_DL, which computes the reverse of a list using difference lists:

```
reverse(Xs, Ys) ← reverse_dl(Xs, Ys-[]).

reverse_dl([X|Xs], Ys-Zs) ← reverse_dl(Xs, Ys-[X|Zs]).

reverse-dl([], Xs-Xs).
```

Take the query $Q = \texttt{reverse(s, X)}$, where s is a (possibly nonground) list and X is a variable. In the following, we assume an infinite signature.

(1) *Construct* $\mathcal{M}(P)$. Recall from Example 5.1 that for a list `s`, `rev(s)` denotes its reverse. Using the $\mathcal{M}$-characterization of Theorem 2.1, one can show that

$$\mathcal{M}(\texttt{REVERSE\_DL}) = \{\texttt{reverse\_dl(s,t-u)} \mid \texttt{s} \text{ is a ground list, } \texttt{t, u} \text{ are ground terms and } \texttt{rev(s)*u=t}\}$$
$$\cup \mathcal{M}(\texttt{reverse(s,t)} \mid \texttt{s, t} \text{ are ground lists and } \texttt{t=rev(s)})$$

(2) *Prove that P is redundancy or subsumption-free.* In the case of REVERSE_DL it suffices to note that it satisfies the conditions SYN1 and SYN2 and apply Theorem 4.2.

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

$$\mathcal{M}(P) \models Q' \quad \text{iff Ground}(Q') \subseteq \mathcal{M}(P)^*. \tag{7.1}$$

In our case, by the form of $\mathcal{M}$(REVERSE_DL), if Q'' is a ground instance of `reverse(s, rev(s))`, then $Q'' \in \mathcal{M}$(REVERSE_DL) holds. Therefore, by (7.1),

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

(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:

$$(\forall\eta.(E_1 = E_2)\eta \text{ is ground} \Rightarrow E_1\eta = E_2\eta) \Rightarrow E_1 = E_2. \tag{7.2}$$

In our case, assume that

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

We have $\texttt{X}\gamma\eta = \text{rev}(\texttt{s}\gamma\eta) = $ (by definition of rev) $\text{rev}(\texttt{s})\gamma\eta$. Then, by (7.2), $\texttt{X}\gamma = \text{rev}(\texttt{s})\gamma$ and hence

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

holds.

(5) *Apply Corollary* 4.1 (*or Corollary* 3.1 *for programs that are not redundancy-free*). *For* REVERSE_DL we obtain

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

In view of our comments in 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 $\mathcal{M}_{(\text{pre,post})}$-semantics. Therefore, it is legitimate to rephrase the above methodology for partial correctness by using $\mathcal{M}_{(\text{pre,post})}(P)$ 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, $\text{Min}(\text{sp}(Q, P)) = \text{sp}(Q, P)$, that is, the set of computed instances of Q is subsumption-free.

Observe that, because of Theorem 4.1(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 that is (pre, post)-redundancy-free, but not redundancy-free. The next result is a relativized version of Corollary 4.1. It shows that, for (pre, post)-redundancy-free programs, the computed instances of the (pre, post)-correct queries can be retrieved from $\mathscr{M}_{(\text{pre, post})}(P)$, thus motivating the previous definition.

Corollary 7.1. Consider a (pre, post)*-redundancy-free program P and a* (pre, post)*-correct query Q. Then*

(i) P Min($\{Q\theta \mid P \models Q\theta\}$).
(ii) $\{Q\}P$ Min($\{Q\theta \mid \mathscr{C}(P) \models Q\theta\}$).
(iii) *If the signature contains infinitely many constant symbols,*

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

PROOF. From Claims 1 and 2 of the proof of Corollary 4.1 we obtain (i), (ii), and also

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

provided that the signature contains infinitely many constant symbols. Then (iii) follows from Lemma 6.2. □

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 $\mathscr{M}_{(\text{pre, post})}(P)$. 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 4.2, is based on $\mathscr{M}(P)$, whereas for (pre, post)-correct programs, we would like to use $\mathscr{M}_{(\text{pre, post})}(P)$.

To solve this problem, we provide an analogue of Theorem 4.2 that employs a modification of the conditions SEM1 and SEM2. The new conditions refer to $\mathscr{M}_{(\text{pre, post})}(P)$ instead of $\mathscr{M}(P)$ and allow us to prove that a program is (pre, post)-redundancy-free.

In the proof of Theorem 7.1 we use LD resolution, that is, SLD resolution with the leftmost selection rule, as adopted in Prolog. The following lemma, due to Ruggieri [17], will be needed.

Lemma 7.1 (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 7.1. 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*

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

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

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

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

PROOF. The proof follows closely that of Theorem 4.2. 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*

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

PROOF. From Claim 1 of the proof of Theorem 4.2 it follows that $\mathcal{M}(P) \models \mathbf{B}\theta$, which implies that also $\mathcal{M}(P) \models H\theta$. Furthermore, both H and $\mathbf{B}$ are instances of a prefix of a resolvent in ξ, so by the persistence lemma (Lemma 7.1), both H and $\mathbf{B}$ are (pre, post)-correct. It suffices now to apply Lemma 6.2. □

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 that 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 that yield its computed instances Q' and Q''. The rest of the proof is from now on the same as that of Theorem 4.2, using Claim 1 above instead of Claim 1 of the proof of Theorem 4.2. □

Example 7.1. Reconsider the MEMBER program of Example 5.2:

```
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 = {member(x, t) | x is a ground term and
t is a ground list of distinct elements}.

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

$\mathcal{M}_{(\text{pre},\text{post})}$(MEMBER) = {member(x, t) | x is a ground term,
t is a ground list of distinct elements, and x is in t}.

Condition SYN2 of Section 4 obviously applies to the MEMBER program. To check condition SEM1 of Theorem 7.1, 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

$\mathcal{M}_{(\text{pre},\text{post})}$(BER) ⊨ member(x, [x|xs]),M

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

$\mathcal{M}_{(\text{pre},\text{post})}$(MEMBER) ⊭ member(x, xs),

which implies that SEM1 holds for the MEMBER program. By Theorem 7.1 we have that MEMBER is (pre, post)-redundancy-free. Now Corollary 7.1 can be applied to

any query of the form `member(s, t)`, where `t` is a list of pairwise nonunifiable elements, because such a query is (pre, post)-correct.

We can now summarize our methodology for proving partial correctness on the basis of $\mathcal{M}_{(\text{pre,post})}$-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* $\mathcal{M}_{(\text{pre,post})}(P)$. 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 [4] 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 $\mathcal{M}_{(\text{pre,post})}(P) \models Q'$.
(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* 7.1.

8. PROGRAMS WITH ARITHMETIC

We now apply the results of the previous sections to an extension of logic programming with arithmetic. Because we wish to apply these results to reason about Prolog programs, we follow here Prolog's approach to arithmetic. We extend the syntax by allowing, in the bodies of the program clauses, the arithmetic comparison operators $<$, $\leq$, $=:=$, $\neq$, $\geq$, and $>$ are the `is` relation of Prolog. We also assume that, conforming to the status of built-ins, 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 [13] and first add to each program infinitely many unit clauses that define the ground instances of the arithmtic relations used.

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 set of unit clauses

$$M_{<} = \{\texttt{m} < \texttt{n} \mid \texttt{m}, \texttt{n} \text{ are gaes and } \texttt{val(m)} < \texttt{val(n)}\},$$

for `is` we add the set

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

and so forth, so, for example, `7 is 3+4` $\in M_{\texttt{is}}$.

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 of the following statements holds:

- It is of the form `p(s, t)`, where `p` is a comparison operator and either `s` or `t` are not gae.
- It is of the form `s is t` and `t` is not a 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 3.1, 4.1, and 7.1) 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 Lemma 7.1.

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 a 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 8.1. Consider the following program `LENGTH`:

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

Let

pre = {`length(s, t)` | `s,t` are ground} ∪ {`s is t` | `t` is a gae},

post = {`length(s, t)` | `s, t` are ground, `t` is agae}
∪ {`s is t` | `s, t` are gae}.

It is easy to see that then `LENGTH` and all the queries of the form `length(s, t)` are (pre, post)-correct. Thus for all `s,t` the LD derivations of `LENGTH` ∪ {`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 4.2, `LENGTH` is redundancy-free. So

following the procedure explained in Section 7, we conclude that for a list s and a variable N,

{length(s, N)} LENGTHVariant({length(s,|s)})

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

Example 8.2. 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 [20]. Here, a bst is represented by either the constant void, denoting the empty bst, or by the term tree((x, v), 1, r), where x is a gae, v is a term, 1 and r are bsts, and 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, similar to > and <, etc., evaluates both arguments before comparison.

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

pre = post
= {lookup(x,v,b) | x is a gae, v is a ground term and b is a ground bst}
∪{s =:= t | s, t are gae}
∪{s < t | s, t are gae}
∪{s < t | s, t are gae}.

It is easy to see that DICTIONARY is then (*pre*, *post*)-correct, and that on virtue of Theorem 6.2 the following natural interpretation is the well-typed fragment of its least Herbrand model:

$$\mathscr{M}_{(pre,post)}(\text{DICTIONARY}) = \{\text{lookup}(x,v,b) \mid x \text{ is a gae, } b \text{ is a ground bst, and } (x,v) \text{ is an element in } b\}, \cup M_{=:=} \cup M_{<} \cup M_{>}.$$

Also, from Lemma 8.1 it follows that, for any gae x, term v and bst b, the LD derivations of DICTIONARY∪{lookup(x, v, b)} do not end in error. Conditions SYN2 of Section 4 readily applies to the DICTIONARY program. To check condition SEM1 of Theorem 7.1, it suffices to consider three ground instances of the three clauses of the program with a common head, namely,

```
lookup(x, v, tree((y, v), 1, r)) ← x=:=y.
lookup(x, v, tree((y, v), 1, r)) ← x<y, lookup(x, v, 1).
lookup(x, v, tree((y, v), 1, r)) ← x>y, lookup(x, v, r).
```

and observe that, for any two gae x and y, exactly one among x = : = y, x<y, and x>y holds in $\mathcal{M}_{(pre,post)}$(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,

{lookup(v, X, b)}DICTIONARYVariant({lookup(x,v,b) | (x,v) is an element of b}).

9. CONCLUSIONS

Table 1 presents a list of example programs from the book of Sterling and Shapiro [20] 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 4.2 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 that 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 that do not occur in the program."

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

TABLE 1. Example programs from Sterling and Shapiro [20].

Program	Page	Subsum.-Free	Redund.-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

possible to reason about the computed instances of the "well-typed" queries using the $\mathcal{M}_{(pre,post)}$-semantics only. This fact is relevant, because 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.

We thank both the referees of this paper and 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). This work was carried out while the second author was visiting CWI, Amsterdam. The stay was supported by the Italian National Research Council (CNR) and by HCM grant ERBCH-BGCT930496 in the context of the EUROFOCS project.

REFERENCES

1. Apt, K. R., Program Verification and Prolog, in: E. Börger (ed.), *Specification and Validation Methods for Programming Languages and Systems*, Oxford University Press, 1995, pp. 55–95.
2. Apt, K. R. and Gabbrielli, M., Declarative Interpretations Reconsidered, in: P Hentenryck (ed.), *Proceedings of Eleventh International Conference on Logic P ming*, MIT Press, Cambridge, MA, 1994, pp. 74–89.
3. Apt, K. R. and Marchiori, E., Reasoning about Prolog Programs: From Modes tl Types to Assertions, *Formal Aspects of Computing* 6(6A):743–765 (1994).
4. Apt, K. R. and Pedreschi, D., Reasoning about Termination of Pure Prolog Prograr *Inform. and Comput.* 106(1):109–157 (1993).
5. Apt, K. R. and van Emden, M. H., Contributions to the Theory of Logic Programming *J. ACM* 29(3):841–862 (1982).
6. Bossi, A. and Cocco, N., Verifying Correctness of Logic Programs, in: *Proceedings of TAPSOFT '89*, *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1989, pp. 96–110.
7. Bossi, A., Gabbrielli, M., Levi, G., and Martelli, M., The s-Semantics Approach: Theory and Applications, *J. Logic Programming* 19-20:149–197 (1994).
8. Clark, K. L., Predicate Logic as a Computational Formalism, Research Report DOC 79/59, Dept. of Computing, Imperial College, London, 1979.
9. Deransart, P., Proof Methods of Declarative Properties of Definite Programs, *Theoret. Comput. Sci.* 118:99–166 (1993).
10. Drabent, W. and Małuszynski, J., Inductive Assertion Method for Logic Programs, *Theoret. Comput. Sci.* 59(1):133–155 (1988).
11. Falaschi, M., Levi, G., Martelli, M., and Palamidessi, C., Declarative Modeling of the Operational Behavior of Logic Languages, *Theoret. Comput. Sci.* 69(3):289–318 (1989).
12. Falaschi, M., Levi, G., Martelli, M., and Palamidessi, C., A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs, *Inform. and Comput.* 102(1):86–113 (1993).
13. Kunen, K., Some remarks on the Completed Database, in: R. A. Kowaski and K. A. Bowen (eds.), *Proceedings of the Fifth International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1988, pp. 978–992.
14. Lloyd, J. W., *Foundations of Logic Programming*, 2nd ed., Springer-Verlag, Berlin, 1987.

15. Maher, M. J., Equivalences of Logic Programs, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 1988, pp. 627–658.
16. Maher, M. J. and Ramakrishnan, R., Déjà Vu in Fixpoints of Logic Programs, in: E. Lusk and R. Overbeek (Eds.), *Proceedings of the North American Conference on Logic Programming*, MIT Press, Cambridge, MA, 1989, pp. 963–980.
17. Ruggieri, S., Metodi Formali per lo Sviluppo di Programmi Logic, Tesi di Laurea, Technical Report, Dipartimento di Informatica, Università di Pisa, 1994 (in Italian).
18. Shepherdson, J. C., Unsolvable Problems for SLDNF Resolution, *J. Logic Programming* 10(1):19–22 (1991).
19. Stärk, R., A Direct Proof for the Completeness of SLD-Resolution, in: E. Börger, H. Kleine Büning, and M. M. Richter (eds.), *Computation Theory and Logic 89*, *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1990, Vol. 440, pp. 382–383.
20. Sterling, L. and Shapiro, E. Y., *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.
21. Ullman, J. D., *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Rockville, MD, 1988, Vol. 1.
22. van Emden, M. H. and Kowalski, R. A., The Semantics of Predicate Logic as a Programming Language, *J. ACM* 23(4):733–742 (1976).