A Complete Declarative Debugger of Missing Answers

Salvatore Ruggieri

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
e-mail: ruggieri@di.unipi.it

Abstract. We propose two declarative debuggers of missing answers with respect to $\mathcal{C}$- and $\mathcal{S}$-semantics. The debuggers are proved correct for every logic program. Moreover, they are complete and terminating with respect to a large class of programs, namely acceptable logic programs. The debuggers enhance existing proposals, which suffer from a problem due to the implementation of negation as failure. The proposed solution exploits decision procedures for $\mathcal{C}$- and $\mathcal{S}$-semantics introduced in [9].

Keywords. Logic programming, Declarative Debugging, Error Diagnosis, Missing Answers, Acceptable Logic Programs.

1 Introduction

Declarative debugging is concerned with finding errors that cause anomalies during the execution of a program starting from some information on the intended semantics of the program. In logic programming systems, a query which is valid in the intended meaning of a program but that is not in its actual semantics is an anomaly called *missing answer*. A missing answer originates from a "failure" in the construction of a proof tree for a valid query. The reason of such a failure is the presence of *uncovered atoms*, i.e. of atoms A in the intended interpretation of the program, for which there is no clause instance whose head is A and whose body is true in the intended interpretation. In other words, there is no immediate justification in the program in order to deduce A.

The role of a declarative debugger is to find out uncovered atoms starting from missing answers, which are usually detected during the testing phase, and from the intended semantics of the program. In this paper, we concentrate on the $\mathcal{C}$-semantics of Falaschi et al. [5] (or least term model semantics of Clark [3]) and on the $\mathcal{S}$-semantics of Falaschi et al. [6].

Many debuggers in the literature find uncovered atoms starting from missing answers that have a *finitely failed SLD-tree*. As we will point out, the assumption that missing answers have finitely failed SLD-trees is restrictive in some cases, and it is due to a well-known limitation of the *negation as failure* rule. We show that restriction in the case of Shapiro's debugger [10] [8, Debugger S.I].

In this paper, we propose two declarative debuggers of missing answers for $\mathcal{C}$- and $\mathcal{S}$-semantics that are correct for any program, and complete and terminating for

a large class of logic programs, namely acceptable programs [2]. The implementations of the debuggers rely on decidability procedures for $\mathcal{C}$- and $\mathcal{S}$-semantics which are adapted from [9].

Compared with Shapiro's approach, the debugger for $\mathcal{C}$-semantics relaxes the assumption that the missing answers in input have finitely failed SLD-trees. In addition, we show that a smaller search space is considered.

The debugger for $\mathcal{S}$-semantics is derived by applying the insights underlying the construction of that for $\mathcal{C}$-semantics to the theory of $\mathcal{S}$-semantics. The only approach on debugging of missing answers with respect to $\mathcal{S}$-semantics is due to Comini et al. [4]. They introduce a method for finding all uncovered atoms starting from the intended interpretation of an acceptable program. However, their approach is effective iff the intended interpretation is a finite set, whilst we make a weaker assumption.

Preliminaries We use in this paper the standard notation of Apt [1], when not specified otherwise. In particular, we use queries instead of goals. We denote by L the underlying language a program is defined on. $Atom_L$ denotes the set of atoms on L, B_L the Herbrand base on L. Usually, one considers $L = L_P$. $ground_L(P)$ denotes the set of ground instances of clauses from P. LD-resolution is SLD-resolution together the leftmost selection rule. 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. N is the set of natural numbers. For a ground term t, $ll(t) = ll(t1)+1$ if $t = [t2|t1]$ and $ll(t) = 0$ otherwise, i.e. ll is the list-length function.

2 Program Semantics and Missing Answers

Several declarative semantics have been considered as alternatives to the standard least Herbrand model. We focus on two of them, namely $\mathcal{C}$-semantics of Falaschi et al. [5] (also known as the least term model of Clark [3]) and $\mathcal{S}$-semantics of Falaschi et al. [6].

Definition 1. For a logic program P we define

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

By correctness of SLD-resolution, we observe that $\mathcal{S}(P) \subseteq \mathcal{C}(P)$. To each semantics is associated a continuous *immediate consequence operator*. For a program P, the least fixpoint of $T_P^{\mathcal{C}}$, $\mathcal{C}(P)$, and the upward ordinal closure $T_P^{\mathcal{C}} \uparrow \omega$ coincide [5]. Similarly, the least fixpoint of $T_P^{\mathcal{S}}$, $\mathcal{S}(P)$, and the upward ordinal closure $T_P^{\mathcal{S}} \uparrow \omega$ coincide [6].

Definition 2. For a logic program P we define the following functions from sets of atoms into set of atoms:

$$T_P^{\mathcal{C}}(I) = \{\ A\theta \in Atom_L \mid \exists\ A \leftarrow B_1, \ldots, B_n \in P, \{B_1\theta, \ldots, B_n\theta\} \subseteq I\ \}$$

$$T_P^{\mathcal{S}}(I) = \{\ A\theta \in Atom_L \mid \exists\ A \leftarrow B_1, \ldots, B_n \in P, B'_1, \ldots, B'_n \text{ variants of atoms in } I \text{ and renamed apart } \exists\ \theta = mgu((B_1, \ldots, B_n), (B'_1, \ldots, B'_n))\ \}$$

□

An intended interpretation of a program w.r.t. a semantics is a set of atoms which, in the intentions of the programmer, is supposed to be the actual semantics of the program. Starting points of the debugging analysis are *missing answers*.

Definition 3. We say that a query is in a set of atoms if every atom of the query is in the set. Let $\mathcal{F}$ be the $\mathcal{C}$- or $\mathcal{S}$-semantics, and $\mathcal{I}$ be the intended interpretation of a program P w.r.t. $\mathcal{F}$. A *missing answer* w.r.t. $\mathcal{F}$ is any query which is in $\mathcal{I}$ but that is not in $\mathcal{F}(P)$. □

Missing answer are caused by *uncovered atoms*, i.e. atoms valid in the intended meaning of a program that have no immediate justification in the program.

Definition 4. An atom A is uncovered if $A \in \mathcal{I}$ and $A \notin T_P^{\mathcal{F}}(\mathcal{I})$. □

3 Shapiro's Debugger

Consider the semantics of correct instances of logic programs, i.e. $\mathcal{C}$-semantics. A query Q which is supposed to be a logical consequence of the program, "fails" if it is not. However, by saying that a query Q "fails" it is often meant Q *finitely fails*. This stronger assumptions is due a well-known limitation of the *negation as failure* rule, and it affects several declarative debuggers in the literature. Let us consider the Shapiro's debugger [10] [8, Debugger S.I] as an example. Let P be the program under analysis.

```
miss([A | B], Goal) ←
   not( call(A) ),
   miss(A, Goal).
miss([A | B], Goal) ←
   call(A),
   miss(B, Goal).
miss(A, Goal) ←
   user_pred(A),
   clause(A, B),
   valid(B),
   miss(B, Goal).
miss(A, A) ←
```

```
user_pred(A),
not(
  (clause(A, B),
  valid(B))
).
```

`clause(`A`, [`B_1`, ..., `B_n`]).` *for every* $A \leftarrow B_1, \ldots, B_n \in P$ augmented by P.

Program 1

`user_pred` characterizes user-defined predicates, and it is a collection of facts `user_pred(p(X1, ..., Xn))` for every predicate symbol `p` of arity n. `valid` is an oracle defining the intended meaning of P. It may be implemented either by queries to the programmer or by using some specification of the program. Consider now the program:

```
s.
p(X) ←q(Y), r(Y,X).
q(a).
% q(b). % missing
r(a, c).
r(b, X).
```

Program 2

Shapiro's debugger correctly works when the missing answer in input has a finitely failed tree. On the other hand, the query `p(X),s` is a missing answer, since `p(X)` is not a logical consequence of the program, but there is no finitely failed tree, since there is a successful derivation that instantiates `X` to `c`. A call `miss( [p(X),s], A)` to the Shapiro's debugger fails to return that `q(b)` is uncovered. The need for the hypothesis of finite failure lies in the use of negation in clauses such as:

```
miss([A | B], Goal ) ←not( call(A) ), miss(A, Goal).
```

where the debugger tries to prove `not( call(A) )`. Due to well-known limitations of the negation as failure rule, `not( call(A) )` succeeds if $P \models \neg\exists A$, i.e. if there is a finitely-failed SLD-tree. Instead, the intended use of `not( call(A) )` is to prove $P \models \neg\forall A$, i.e. that A is not a logical consequence of P. A form of completeness has been shown for Ferrand's debugger [7], [8, Debugger F.1], which is obtained from *Program 1* by removing the literals `not( call(A) )` and `call(A)`. As an example, the uncovered atom `q(b)` of *Program 2* is detected by Ferrand's debugger. However, Ferrand's approach differs from ours in the fact that it considers *impossible* atoms instead of uncovered atoms. A is impossible if no instance of A is uncovered. Let us consider the following incorrect `EAppend` version of the `Append` program:

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

Program 3

The query Q = `append([X], Ys, [X|Ys])` is a missing answer, since it is in the intended interpretation, and it is not a logical consequence of `EAppend`. However, Shapiro's debugger is not able to find out that Q is an uncovered atom, since Q has not a finitely failed tree. On the other hand, Ferrand's debugger does not show that Q is uncovered, due to the fact that there exists an instance of Q which is covered, namely `append([X], [], [X])`, and then Q is not an impossible atom. Finally, it is worth noting that both debuggers ask the oracle for a valid instance of `append(Ys, Ys, Ys)`. Consequently, the call `miss(append([], [], []), Goal)` is made, where `append([], [], [])` is not a missing answer. In general, unnecessary questions are addressed to the oracle, in the sense that the search space includes queries that are not missing answers, and then cannot lead to an uncovered atom.

4 Acceptable programs

In this section, we introduce acceptable logic programs, a well-known large class for which we provide a decision procedure for $\mathcal{C}$- and $\mathcal{S}$-semantics.

4.1 The Framework

First, we recall the definition of level mappings [2].

Definition 5. A *level mapping* is a function $|\ |\ : B_L \to N$ of ground atoms to natural numbers. For $A \in B_L$, $|\ A\ |$ is called the level of A. □

We are now in the position to recall the definition of acceptable logic programs, introduced by Apt and Pedreschi [2]. Intuitively, the definition of acceptability requires that for every clause, the level of the head of any of its ground instances is greater than the level of each atom in the body which might be selected further in a LD-derivation.

Definition 6. A program P is *acceptable by* $|\ |: B_L \to N$ *and a Herbrand interpretation* I iff I is a model of P, and for every $A \leftarrow B_1, \ldots, B_n$ in $ground_L(P)$:

$$for\ i \in [1,n] \quad I \models B_1, \ldots, B_{i-1} \quad implies \quad |\ A\ | \ > |\ B_i\ |\ .$$

P is *acceptable* if it is acceptable by some $|\ |$ and I. □

Consider the following program `PREORDER` for preorder traversals of binary trees.

```
(p1)  preorder(nil, [] ).
(p2)  preorder(leaf(X), [X] ).
(p3)  preorder(tree(X, Left, Right), Ls) ←
          preorder(Left, As),
          preorder(Right, Bs),
          append([X|As], Bs, Ls).
```

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

Program 4

PREORDER is acceptable by $|\ |$ and I, where

$$|\texttt{preorder}(t,\ ls)| = nodes(t) + 1$$
$$|\texttt{append}(xs,\ ys,\ zs)| = ll(xs)$$
$$I = \{\ \texttt{append}(xs,\ ys,\ zs)\ |\ ll(zs) = ll(xs) + ll(ys)\ \} \cup$$
$$\cup\ \{\ \texttt{preorder}(t,\ ls)\ |\ ll(ls) = nodes(t)\ \}$$

where $nodes(t)$ is 1 if $t = \texttt{leaf}(s)$; it is $1 + nodes(l) + nodes(r)$ if $t = \texttt{tree}(s,\ l,\ r)$; and it is 0 otherwise.

Suppose now that the variable X in clause *(p2)* has been erroneously typed in lower case, and let PREORDER′ be PREORDER where *(p2)* is replaced by:

(p2′) `preorder(leaf(x), [x] ).`

PREORDER′ is still acceptable by the same $|\ |$ and I. Note that `preorder(tree(E, leaf(X), leaf(Y)), [E, X, Y])` is a missing answer w.r.t. $\mathcal{C}$ and $\mathcal{S}$-semantics. However, the query has no finitely failed SLD-tree. In particular, Shapiro's debugger is not able to find an uncovered atom starting from it. Other examples of acceptable programs are *Program 2* and *Program 3*.

4.2 Decision Procedures

We sum up the decidability properties of acceptable programs we are interested in by means of the following Lemma reported from [9].

Theorem 7. *Let P be an acceptable program.*
Every LD-derivation for P and any ground query (in any language) is finite. Moreover, $\mathcal{C}(P)$ and $\mathcal{S}(P)$ are decidable sets. □

The decision procedure for $\mathcal{C}$- and $\mathcal{S}$-semantics will be the crucial in the debugging approach of this paper. Interestingly, they have a natural implementation in the logic programming paradigm itself as Prolog meta-programs. The procedures are provided in [9] for observable decidability. Here we specialize them for semantics decidability.

We assume that the predicate `freeze` introduced in [12, Section 10.3] is available. A call to `freeze(A,B)` replaces every variable of A with new distinct constants to obtain B. Unfortunately, as described by Sterling and Shapiro, `freeze` is not present in existing Prolog implementations. In fact, in [9] `freeze` is approximated by a predicate `constants` that replaces every variable in Q by new distinct constants. The approximation works until a fixed maximum number of new

constants is reached, and the results are parametric to that number. Here, we abstract away from such a parameterization and assume that `freeze` and its dual `melt` are available. Given a term B, `melt(`B`, A)` replaces every constants of B introduced by freezing some term by the original variable to obtain A. For instance `freeze(p(X), Y)` succeeds by instantiating `Y` to $\mathtt{p}(a_{\mathtt{X}})$, where $a_{\mathtt{X}}$ is a fresh constant representing the frozen variable `X`. The following is the decision procedure for $\mathcal{S}$-semantics.

```
in_s(A) ←
   freeze(A, A1),
   pure(A, B),
   demo([A1],[B]),
   variants(A, B).

demo([], []).

demo([A|As], [B|Bs]) ←
   clause(A, Ls, Id),
   demo(Ls, L1s),
   demo(As, Bs),
   clause(B, L1s, Id).

pure(p(X1, ..., Xn), p(Y1, ..., Yn)).
     for every predicate symbol p of arity n

clause(A, [B1, ..., Bn], k).   for every Ck = A ← B1,...,Bn ∈ P
```

augmented by the definition of `variants` [12, Program 11.7].

Program 5

`variants(A, A1)` is an extra-logical predicate that succeeds iff A and $A1$ are variants. `pure(A, B)` computes a pure atom for the predicate symbol of a given atom. `clause(`A`, [B], `k`)` models the clause $C_k = A \leftarrow \mathbf{B}$ of P. A distinct identifier k is assigned to every clause of P. Finally, as a corollary of the results reported in [9], given a program P, we have that for an atom A, `in_s(`A`)` succeeds iff $A \in \mathcal{S}(P)$. Moreover, if P is acceptable then every LD-derivation of `in_s(`A`)` is finite.

Next we present the procedure for $\mathcal{C}$-semantics. Given a program P and an atom A, `in_c(`A`)` succeeds iff $A \in \mathcal{C}(P)$. Moreover, if P is acceptable then every LD-derivation of `in_c(`A`)` is finite.

```
in_c(A) ←
   freeze(A, A1),
   call(A1).
```

augmented by P.

Program 6

5 Declarative Debuggers

5.1 $\mathcal{C}$-semantics

We revise the Shapiro's debugger, by integrating the decision procedure `in_c` within it.

```
(o)    missing_answers_c(Q, Goal) ←
          miss(Q, Goal1),
          melt(Goal1, Goal).
(i)    miss([A | B], Goal) ←
          not( in_c(A) ),
          miss(A, Goal).
(ii)   miss([A | B], Goal) ←
          in_c(A),
          miss(B, Goal).
(iii)  miss(A, Goal) ←
          user_pred(A),
          freeze(A, A1),
          clause(A1, B),
          valid_c(B),
          miss(B, Goal).
(iv)   miss(A, A1) ←
          user_pred(A),
          freeze(A, A1),
          not(
            (clause(A1, B),
            valid_c(B))
          ).

       clause(A, [B1, ..., Bn]).    for every A ← B1,...,Bn ∈ P
```

augmented by *Program 6.*

Program 7

`valid_c` is an oracle describing the queries in the intended interpretation $\mathcal{I}$. Formally, called V the definition of `valid_c`, an atom `valid_c([B])` is in $\mathcal{C}(V)$ iff $\mathbf{B}$ is in $\mathcal{I}$. Consider now *Program 2.* We have the following definitions of `user_pred` and `valid_c`:

```
user_pred(s).
user_pred(p(X)).
user_pred(q(X)).
user_pred(r(X, Y)).

valid_c(s).
valid_c(p(X)).
valid_c(q(a)).
valid_c(q(b)).
valid_c(r(a,c)).
valid_c(r(b,X)).
```

```
valid_c([A|B]) ←
   valid_c(A), valid_c(B).
valid_c([]).
```

A call `missing_answers_c( [p(X),s], A)` has a finite LD-tree, and computes the uncovered atom A = `q(b)`. The computation progresses as follows. First `p(X)` is found to be not a logical consequence of the program. Then the clause `p(X)` ← `q(Y), r(Y,X)` is instantiated by `Y = b` in order to find a valid body. Note that the instance `Y = a, X = c` cannot be considered as `X` is frozen. Finally, `q(b)` is found to be an uncovered atom.

Consider PREORDER′ and the missing answer

$$Q = \texttt{preorder(tree(E, leaf(X), leaf(Y)), [E, X, Y])}.$$

A call `missing_answers_c( [`Q`], A)` computes A = `preorder(leaf(X),[X])`, which is indeed the correctly typed clause *(p2)*.

Consider now *Program 3*, i.e. EAppend, and the missing answer Q = `append([X], Ys, [X|Ys])`. The proposed debugger shows that Q is an uncovered atom. The only query to the oracle during the computation is `valid_c(append(Ys, Ys, Ys))` with `Ys` frozen. In other words, the oracle is asked whether `append(Ys, Ys, Ys)` is valid in the intended meaning of EAppend, which is obviously false.

The debugger is correct for every logic program.

Theorem 8 ($\mathcal{C}$-Correctness). *Let P be a program, and Q a missing answer w.r.t. $\mathcal{C}$-semantics. If* `missing_answers_c([Q], Goal)` *has a LD-computed instance* `missing_answers_c([Q], Goal)` *then Goal is an uncovered atom.*

Proof. First of all, we consider a language L' obtained by adding to L sufficiently many new constants, which are employed by the predicate `freeze`. Let us show that we are in the hypotheses of the Theorem considering L' instead of L, and

$$\mathcal{I}' = \{\ A\theta \in Atom_{L'} \mid A \in Atom_L \wedge A \in \mathcal{I}\ \}$$

instead of $\mathcal{I}$. Since Q is a missing answer w.r.t. $\mathcal{I}$, then Q is in $\mathcal{I}'$ and $P \not\models Q$, i.e. Q is a missing answer w.r.t. $\mathcal{I}'$. Moreover the definition V of `valid_c` is an oracle w.r.t. $\mathcal{I}'$. In fact, consider any query Q in L'. Q can be written as $Q'\theta$, where Q' is in L and θ replaces some variables of Q' with distinct constants not in L. Then, we have that Q is in $\mathcal{I}'$ iff Q' is in $\mathcal{I}$, and then, since V is an oracle, iff $V \models$ `valid_c([`Q'`])`. By the Theorem on Constants (see e.g. [11]), $V \models$ `valid_c([`Q'`])` iff $V \models$ `valid_c([`Q'`])`θ, when θ is of the considered form. This implies, that Q is in $\mathcal{I}'$ iff $V \models$ `valid_c([`Q`])`, i.e. that V is an oracle w.r.t. $\mathcal{I}'$. We now show that any computed instance of `miss([`Q`], Goal)` returns an uncovered atom on L'.

The proof proceeds by induction on the number n of calls to `miss` in a refutation.

($n = 1$). `Goal` can be only instantiated by applying rule *(iv)*. Then there is no clause instance $A1 \leftarrow \mathbf{B}$ whose body is in $\mathcal{I}'$, i.e. $A1 \notin T_P^{\mathcal{C}}(\mathcal{I}')$, and $A1$ is obtained by freezing A, hence $A1$ in $\mathcal{I}'$. Then `Goal` is instantiated by an uncovered atom.

($n > 1$). We show that the hypothesis of the theorem holds for calls to `miss` in clauses *(i, ii, iii)*.

(i) Since `not( in_c(A) )` succeeds, A is not in $\mathcal{C}(P)$ albeit by hypothesis it is in $\mathcal{I}'$. Therefore, A is a missing answer.

(ii) Since `in_c(A)` succeeds, A is in $\mathcal{C}(P)$. Therefore, B must be a missing answer.

(iii) $A1 \leftarrow \mathbf{B}$ is a clause instance such that $A1$ is obtained by freezing A and $\mathbf{B}$ is in $\mathcal{I}'$. By the Theorem on Constants, A is not in $\mathcal{C}(P)$ implies $A1$ not in $\mathcal{C}(P)$. As a consequence $\mathbf{B}$ is not in $\mathcal{C}(P)$. Otherwise, by Definition 2, $A1$ would be in $T_P^{\mathcal{C}}(\mathcal{C}(P)) = \mathcal{C}(P)$. Therefore, the call `miss(B, Goal)` satisfies the inductive hypothesis, i.e. $\mathbf{B}$ is a missing answer.

In conclusion, the call `miss(`Q`, Goal1)` in *(o)* instantiates `Goal1` with an uncovered atom $Goal1$ on L'. By melting the frozen variables of $Goal1$, we obtain an atom $Goal$ on L such that $Goal$ is in $\mathcal{I}$ (since $Goal1$ is in $\mathcal{I}'$) but not in $T_P^{\mathcal{C}}(\mathcal{I})$ (otherwise $Goal1$ would be in $T_P^{\mathcal{C}}(\mathcal{I}')$, i.e. $Goal$ is uncovered. □

Restricting the attention to acceptable programs, we are in the position to show completeness of the debugger.
We make the further hypothesis that there are finitely many oracle's answers for Q, i.e that there are finitely many LD-derivations for every call to `valid_c` during a LD-derivation for `missing_answers_c([`Q`], Goal)`.

Theorem 9 ($\mathcal{C}$-Completeness). *Let P be an acceptable program, and Q a missing answer w.r.t. $\mathcal{C}$-semantics such that there are finitely many oracle's answers for Q.*
Then there exists a LD-computed instance of `missing_answers_c([`Q`], Goal)`.

Proof. Reasoning as in the proof of Theorem 8, we can assume a language with infinitely many constants.

We observe that every prefix ξ of a LD-derivation for Q is finite if the variables of Q are never instantiated along ξ. In fact, let θ be a substitution of the variables of Q with new distinct constants. If there is an infinite prefix ξ of a LD-derivation for Q such that the variables of Q are never instantiated along ξ, then $\xi\theta$ would be an infinite prefix of a LD-derivation for $Q\theta$. This is impossible by Theorem 7, since $Q\theta$ is ground. We denote by d_Q the maximum length of a prefix of a LD-derivation for Q that does not instantiate any variable of Q.

The proof proceeds by induction on d_Q.

($d_Q = 1$). Let A be the leftmost atom in Q. We claim that $A \notin \mathcal{C}(P)$. Otherwise, by strong completeness of SLD-resolution, there exists a LD-refutation for A that does not instantiate any variable of A. As a consequence, $d_Q > 1$.

Therefore, A is a missing answer. By applying clause *(i)* the query `miss(`A`, Goal)` is resolved. We now distinguish two cases: either A is or not a variant of the head of a clause instance $A1 \leftarrow \mathbf{B}$ such that $\mathbf{B}$ is in the intended interpretation $\mathcal{I}$. In the latter case, by resolving `miss(`A`, Goal)` with clause *(iv)* we get a refutation, since there are finitely many oracle answers.

In the former case, A unifies with a clause head without instantiating its variables, and then $d_A > 1$ and $d_Q > 1$. In conclusion, the former case is impossible.

($d_Q > 1$). Let A be the leftmost atom in $Q = \mathbf{D}, A, \mathbf{E}$ such that $A \notin \mathcal{C}(P)$. By repeatedly applying clauses *(i, ii)* the query `miss(A, Goal)` is eventually resolved, since for acceptable programs the calls to `in_c` terminate. Moreover, since $\mathbf{D}$ is in $\mathcal{C}(P)$ then by strong completeness of SLD-resolution, there exists a LD-refutation for $\mathbf{D}$ that does not instantiate any variable of $\mathbf{D}$, hence $d_Q \geq d_A$.

Again, we distinguish two cases: either A is or not a variant of the head a clause instance of P such that the body is in the intended interpretation $\mathcal{I}$. In the latter case, by resolving `miss(A, Goal)` with clause *(iv)* we get a refutation, since there are finitely many oracle answers.

In the former case, clause *(iii)* is applicable and a query `miss(B, Goal)` is eventually resolved where $A1 \leftarrow \mathbf{B}$ is an instance of a clause c from P such that $A1$ is obtained by freezing A (i.e., $A1 = A\mu$ for μ substituting variables with fresh constants) and $\mathbf{B}$ is in $\mathcal{I}$. As shown in the proof of Theorem 8, $\mathbf{B}$ cannot be in $\mathcal{C}(P)$, otherwise A would be. Therefore, $\mathbf{B}$ is a missing answer. It is readily checked that $d_A = d_{A1}$. We claim that $d_{A1} > d_{\mathbf{B}}$. In fact, let ξ be the prefix of a LD-derivation for P and $\mathbf{B}$ that does not instantiate any variable of $\mathbf{B}$. We observe that the LD-resolvent of $A1$ and c is more general than $\mathbf{B}$. Therefore, there exists ξ' prefix of a LD-derivation for $A1$ longer than ξ. Summarizing, $d_{A1} > d_{\mathbf{B}}$. As a consequence, $\mathbf{B}$ is a missing answer and $d_Q \geq d_A = d_{A1} > d_{\mathbf{B}}$. Therefore we can apply the inductive hypothesis on $\mathbf{B}$ to obtain the conclusion of the Theorem. □

Finally, we have termination of the debugger.

Theorem 10 ($\mathcal{C}$-Termination). *Let P be an acceptable program, and Q a query such that there are finitely many oracle's answers for Q. Then the LD-tree of* `missing_answers_c([Q], Goal)` *is finite.*

Proof. Suppose there is an infinite LD-derivation. Since calls to `in_c` and `valid_c` terminate, then it necessarily happens that clause *(iii)* is called infinitely many times: `miss(`A_1`, Goal)`, ..., `miss(`A_n`, Goal)`, ... By reasoning as in the proof of Theorem 9, we have that $d_{A_1}, \ldots, d_{A_n}, \ldots$ is an infinite decreasing chain of naturals. This is impossible since naturals are well-founded. □

5.2 $\mathcal{S}$-semantics

We observe that clauses *(iii, iv)* of *Program 7* followed directly from the definition of uncovered atoms (Definition 4) and the definition of $T_P^{\mathcal{C}}$ (Definition 2). We derive the debugger for $\mathcal{S}$-semantics similarly, but considering now $T_P^{\mathcal{S}}$.

```
(o)   missing_answers_s(Q, Goal) ←
          miss(Q, Goal).
(i)   miss([A | B], Goal) ←
          not( in_s(A) ),
          miss(A, Goal).
```

(ii)
```
miss([A | B], Goal) ←
   in_s(A),
   miss(B, Goal).
```
(iii)
```
miss(A, Goal) ←
   user_pred(A),
   pure(A, A1),
   clause(A1, B),
   valid_s(B, C),
   variants(A, A1),
   miss(C, Goal).
```
(iv)
```
miss(A, A) ←
   user_pred(A),
   pure(A, A1),
   not(
     (clause(A1, B),
     valid_s(B, _),
     variants(A, A1))
   ).
```

`clause(`A`, [`B_1`, ..., `B_n`]).` *for every* $A \leftarrow B_1, \ldots, B_n \in P$

augmented by *Program 5.*

Program 8

`valid_s` is an oracle describing the queries in the intended interpretation $\mathcal{I}$. Formally, called V the definition of `valid_s`, an atom `valid_s([B],[C])` is in $\mathcal{S}(V)$ iff $\mathbf{B}$ and $\mathbf{C}$ are queries whose atoms are in $\mathcal{I}$ and variable disjoint, and $\mathbf{C}$ is a variant of $\mathbf{B}$. By [6, Theorems 7.1 and 7.7], a call `valid_s(B, C)` has a computed instance `valid_s(B, C)`θ iff for some renamed apart atom B in $\mathcal{S}(V)$, $\mu = mgu($`valid_s(B, C)`$, B)$ and `valid_s(B, C)`$\theta =$ `valid_s(B, C)`μ. Therefore, for an atom A the query

`pure(`A`, A1), clause(A1, B), valid_s(B, C), variants(A, A1)`

has a LD-refutation iff there exists a renamed apart clause $A1 \leftarrow \mathbf{B}$ such that $\theta = mgu(\mathbf{B}, \mathbf{C})$ for some $\mathbf{C}$ in $\mathcal{I}$ and $A1\theta$ is a variant of A, i.e. iff $A \in T_P^{\mathcal{S}}(\mathcal{I})$.

Consider now, as an example, the following variant of *Program 2.*

```
s.
p(X) ←q(Y), r(Y,X).
% q(a). % missing
q(b).
r(a, c).
r(b, X).
```

We have the following definition of `valid_s`:

```
valid_s(s, s).
valid_s(p(X), p(Y)).
valid_s(p(c), p(c)).
```

```
valid_s(q(a), q(a)).
valid_s(q(b), q(b)).
valid_s(r(a,c), r(a,c)).
valid_s(r(b,X), r(b,Y)).
valid_s([A|As], [B|Bs]) ←
   valid_s(A, B),
   valid_s(As, Bs).
valid_s([], []).
```

A call `missing_answers_s( [p(c),s], A)` has a finite LD-tree, and returns the uncovered atom `q(a)`. The debugger is correct for every logic program.

Theorem 11 ($\mathcal{S}$-Correctness). *Let P be a program, and Q a missing answer w.r.t. $\mathcal{S}$-semantics. If* `missing_answers_s([Q], Goal)` *has a LD-computed instance* `missing_answers_s([Q], Goal)` *then Goal is an uncovered atom.*

Proof. The proof is by induction on the number n of calls to `miss` in a refutation.

($n = 1$). `Goal` can be only instantiated by applying rule *(iv)*, i.e. if Q is an atom and there is no clause $A1 \leftarrow \mathbf{B}$ whose body unifies with a query in the intended interpretation $\mathcal{I}$ with mgu θ and $A1\theta$ is a variant of Q, namely if $Q \notin T_P^S(\mathcal{I})$.

($n > 1$). We show that the hypothesis of the theorem holds for calls to `miss` in clauses *(i, ii, iii)*.

(i) Since `not( in_s(A) )` succeeds, A is not in $\mathcal{S}(P)$ albeit by hypothesis it is in $\mathcal{I}$. Therefore, A is a missing answer.

(ii) Since `in_s(A)` succeeds, A is in $\mathcal{S}(P)$. Therefore, B must be a missing answer.

(iii) Assume that

```
pure(A, A1), clause(A1, B), valid_s(B, C), variants(A, A1)
```

succeeds. Then there exists a renamed apart clause $A1 \leftarrow \mathbf{B}$ such that $\theta = mgu(\mathbf{B}, \mathbf{C})$ for some $\mathbf{C}$ in I and $A1\theta$ is a variant of A. Since A is not in $\mathcal{S}(P)$, then $\mathbf{C}$ is not in $\mathcal{S}(P)$. Otherwise, by Definition 2, A would be in $T_P^S(\mathcal{S}(P)) = \mathcal{S}(P)$. Summarizing, by definition of `valid_s` $\mathbf{C}$ is in $\mathcal{I}$, and we showed that $\mathbf{C}$ is not in $\mathcal{S}(P)$. Therefore, the call `miss(C, Goal)` satisfies the inductive hypothesis, i.e. $\mathbf{C}$ is a missing answer. □

Restricting the attention to acceptable programs, we are in the position to show completeness of the debugger. Also, we assume that there are finitely many oracle's answers. Formally, we say that *there are finitely many oracle's answers for Q* iff there are finitely many LD-derivations for every call to `valid_s` during a LD-derivation for `missing_answers_s([Q], Goal)`.

Theorem 12 ($\mathcal{S}$-Completeness). *Let P be an acceptable program, and Q a missing answer w.r.t. $\mathcal{S}$-semantics such that there are finitely many oracle's answers.*
Then there exists a LD-computed instance of `missing_answers_s([Q], Goal)`.

Proof. As shown in the proof of Theorem 9, every prefix ξ of a LD-derivation for Q is finite if the variables of Q are never instantiated along ξ. We denote by d_Q the maximum length of a prefix of a LD-derivation for Q that does not instantiate any variable of Q. The proof proceeds by induction on d_Q.

($d_Q = 1$). Let A be the leftmost atom in Q. We claim that $A \notin \mathcal{S}(P)$. Otherwise $A \in \mathcal{C}(P)$. Then by strong completeness of SLD-resolution, there exists a LD-refutation for A that does not instantiate any variable of A. As a consequence, $d_Q > 1$.

Therefore, A is a missing answer. By applying clause *(i)* the query `miss(`A`, Goal)` is resolved. We now distinguish two cases: either A is or not a variant of $A1\mu$ where $A1 \leftarrow \mathbf{B}$ is a clause of P and $\mu = mgu(\mathbf{B}, \mathbf{C})$ for some $\mathbf{C}$ in $\mathcal{I}$. In the latter case, we observe that by resolving `miss(`A`, Goal)` with clause *(iv)* we get a refutation, since there are finitely many oracle answers. In the former case, A unifies with a clause head without instantiating its variables, and then $d_A > 1$ and $d_Q > 1$. In conclusion, the latter case is impossible.

($d_Q > 1$). Let A be the leftmost atom in $Q = \mathbf{D}, A, \mathbf{E}$ such that $A \notin \mathcal{S}(P)$. By repeatedly applying clauses *(i, ii)* the query `miss(`A`, Goal)` is eventually resolved, since for acceptable programs the calls to `in_s` terminate. Moreover, since $\mathcal{S}(P) \subseteq \mathcal{C}(P)$, then by strong completeness of SLD-resolution, there exists a LD-refutation for $\mathbf{D}$ that does not instantiate any variable of $\mathbf{D}$, hence $d_Q \geq d_A$.

Again, we distinguish two cases: either A is or not a variant of $A1\mu$ where $c : A1 \leftarrow \mathbf{B}$ is a clause of P and $\mu = mgu(\mathbf{B}, \mathbf{C})$ for some $\mathbf{C}$ in $\mathcal{I}$. In the latter case, we observe that by resolving `miss(`A`, Goal)` with clause *(iv)* we get a refutation, since there are finitely many oracle answers.

In the former case, clause *(iii)* is applicable and `miss(`$\mathbf{C}$`, Goal)` is eventually resolved. In fact, by definition of `valid_s`, the query `pure(A, A1), clause(A1, B), valid_s(B, C), variants(A, A1)` succeeds under the stated hypothesis. As shown in the proof of Theorem 8, $\mathbf{C}$ cannot be in $\mathcal{S}(P)$, otherwise A would be in $\mathcal{S}(P)$, and then $\mathbf{C}$ is a missing answer. We claim that $d_A > d_{\mathbf{C}}$.

Let ξ be the prefix of a LD-derivation for P and $\mathbf{C}$ that does not instantiate any variable of $\mathbf{C}$. Since A and $A1\mu$ are variants, then there exists a renaming substitution σ such that $A = A1\mu\sigma$. Moreover, $A \leftarrow \mathbf{C}\mu\sigma$ is an instance of c. Let θ be a substitution mapping all variables of $A \leftarrow \mathbf{C}\mu\sigma$ into distinct fresh constants. Since no variable of $\mathbf{C}$ is instantiated in ξ, then there exists a prefix ξ' of a LD-derivation for $\mathbf{C}\mu\sigma\theta$ of the same length of ξ.

We observe that the LD-resolvent of $A\theta$ and c is more general than $\mathbf{C}\mu\sigma\theta$, since $A\theta \leftarrow \mathbf{C}\mu\sigma\theta$ is an instance of c. Therefore, there exists a prefix ξ'' of a LD-derivation for $A\theta$ longer than ξ'. By substituting in ξ'', every fresh constant introduced by θ with the variable it replaced, we get a prefix of a LD-derivation for P and A that does not instantiate any variable of A and whose length is greater than that of ξ. Summarizing, $\mathbf{C}$ is a missing answer and $d_Q \geq d_A > d_{\mathbf{C}}$. Therefore we can apply the inductive hypothesis on $\mathbf{C}$ to obtain the conclusion of the Theorem. □

Finally, we have termination of the debugger.

Theorem 13 ($\mathcal{S}$-Termination). *Let P be an acceptable program, and Q a query such that there are finitely many oracle's answers for Q. Then the LD-tree of* `missing_answers_s([Q], Goal)` *is finite.* □

Observe that the hypothesis that there are finitely many oracle's answers is rather restrictive in the case of $\mathcal{S}$-semantics. Looking at *Program 8*, we quickly realize that clauses *(iii,iv)* call `valid_s(B, C)` with `B` instantiated by the body of some program clause. In the case of a simple program such as `Append`, the resulting query `valid_s([append(Xs, Ys, Zs)], C)` has infinitely many LD-derivations. In general, we have that *"finitely many oracle's answers"* actually requires that $\mathcal{I}$ is a finite set. However, by noting that `variants(A, A1)` must succeed, a weaker assumption can be made by defining an oracle `valid_s(B, C, A, A1)` that stops a LD-derivation if $A1$ becomes more instantiated than A (or some sufficient condition that implies that, e.g. by checking that the size of $A1$ remains lower or equal than the size of A). By such an enhanced oracle, we have that the assumption of *"finitely many oracle's answers"* is less restrictive, and allows for reasoning on intended interpretations that are infinite sets. As an example, starting from the missing answer `append([X], Ys, [X|Ys])` for *Program 3*, the debugger finds out that it is an uncovered atom, when `valid_s` is as described above.

6 Discussion

A Completeness Result The following result is an immediate consequence of Theorems 9 and 12.

Theorem 14. *Let P be an acceptable program. If there is a missing answer Q w.r.t. $\mathcal{C}$-semantics (resp., $\mathcal{S}$-semantics) and there are finitely many oracle's answers for Q then there exists an uncovered atom.* □

A non-constructive proof has been established by Comini et al. [4] also in the case that there are not finitely many oracle's answers. Indeed, the proofs of Theorems 9 and 12 (non-constructively) show that result if we remove the hypothesis that there are finitely many oracle's answers.

Bounded Programs A declarative characterization of a class larger than acceptable programs is considered in [9], namely the class of *bounded logic programs*, and a decision procedure is provided with respect to $\mathcal{C}$- and $\mathcal{S}$-semantics. Unfortunately, the overall approach presented in this paper cannot be extended to bounded programs. The main problem is that Theorem 14 does not hold for bounded program, in general. As an example, `p` is a missing answer for the following (bounded) program, but there is no uncovered atom.

```
p ←p
%p   % missing
```

Future work is aimed at extending the ideas and the results presented here to larger classes of logic programs, by investigating subclasses of bounded programs.

Conclusions We presented two declarative debuggers of missing answers with respect to $\mathcal{C}$- and $\mathcal{S}$-semantics. They are correct for any program, and complete and terminating for a large class of logic programs. The implementations of the debuggers rely on decidability procedures for $\mathcal{C}$- and $\mathcal{S}$-semantics which are adapted from [9].

The results presented in this paper improve on Shapiro's and Ferrand's proposals for completeness and termination. Moreover, efficiency is improved in the following sense. As shown in Theorem 8, only calls `miss([Q], Goal)` where Q is a missing answer are made. On the contrary, from the example *Program 3*, we realize that Shapiro's and Ferrand's debuggers search space include queries that are not missing answers, and then cannot lead to uncovered atoms.

Also, we introduced a debugger for $\mathcal{S}$-semantics. The only approach to debugging of missing answers w.r.t. $\mathcal{S}$-semantics is due to Comini et al. [4]. They present a method for finding all uncovered atoms starting from the intended interpretation $\mathcal{I}$ of an acceptable program. However, the approach of Comini et al. is effective iff $\mathcal{I}$ is a finite set, whilst we require a weaker condition, namely that there are finitely many oracle's answers.

References

1. K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 493–574. Elsevier, 1990.
2. K.R. Apt and D. Pedreschi. Reasoning about termination of pure prolog programs. *Information and computation*, 106(1):109–157, 1993.
3. K.L. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Imperial College, Dept. of Computing, 1979.
4. M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium*, pages 275–287. The MIT Press, 1995.
5. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 103(1):86–113, 1993.
6. M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. Declarative Modeling of the Operational Behaviour of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, December 1989.
7. G. Ferrand. Error Diagnosis in Logic Programming, an Adaption of E. Y. Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198, 1987.
8. L. Naish. Declarative Diagnosis of Missing Answers. *New Generation Computing*, 10(3):255–285, 1991.
9. S. Ruggieri. Decidability of Logic Program Semantics and Applications to Testing. In *Proc. of PLILP'96*, volume 1140 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, Berlin, 1996.
10. E. Shapiro. *Algorithmic program debugging.* The MIT Press, 1983.
11. J. Shoenfield. *Mathematical logic.* Addison Wesdley, Reading, 1967.
12. L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, second edition, 1994.