The Prolog Debugger and Declarative Programming

Włodzimierz Drabent[1,2]([✉]) [ID]

[1] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
drabent@ipipan.waw.pl
[2] IDA, Linköping University, Linköping, Sweden

Abstract. Logic programming is a declarative programming paradigm. Programming language Prolog makes logic programming possible, at least to a substantial extent. However the Prolog debugger works solely in terms of the operational semantics. So it is incompatible with declarative programming. This report discusses this issue and tries to find how the debugger may be used from the declarative point of view. The results are rather not encouraging. Also, the box model of Byrd, used by the debugger, is explained in terms of SLD-resolution.

Keywords: Declarative diagnosis/Algorithmic debugging · Prolog · Declarative programming · Program correctness · Program completeness

1 Introduction

The idea of logic programming is that a program is a set of logic formulae, and a computation means producing logical consequences of the program. So it is a *declarative* programming paradigm. The program is not a description of any computation, it may be rather seen as a description of a problem to solve. Answers of a given program (the *logic*) may be computed under various strategies (the *control*), the results depend solely on the former. This semantics of programs, based on logic, is called *declarative semantics*.

Programming language Prolog is a main implementation of logic programming. Its core, which may be called "pure Prolog", is an implementation of SLD-resolution under a fixed control. (SLD-resolution with Prolog selection rule is called LD-resolution.) For a given program P and query Q, Prolog computes logical consequences of P which are instances of Q. If the computation is finite then, roughly speaking, all such consequences are computed[1].

On the other hand, Prolog may be viewed without any reference to logic, as a programming language with a specific control flow, the terms as the data, and

---

[1] See e.g. [1] for details. We omit the issue of unification without occur-check; it may lead to incorrect answers (i.e. not being logical consequences of the program). Technically, by an answer of a program we mean the result of applying a (correct or computed) answer substitution to a query.

a certain kind of term matching as the main primitive operation. Such a view is even necessary when we deal with non logical features of full Prolog, like the built-ins dealing with input/output. Of course such *operational view* loses all the advantages of declarative programming.

In the author's opinion, Prolog makes declarative programming possible in practice. A Prolog program treated as a set of logical clauses is a logic program. The logic determines the answers of the program. At a lower level, the programmer can influence the control. This can be done by setting the order of program clauses and the order of premises within a clause (and by some additional Prolog constructs). Changing the control keeps the logic intact, and thus the program's answers are unchanged; the logic is separated from the control [9]. What is changed is the way they are computed, for instance the computation may be made more efficient. In particular, an infinite computation may be changed into a finite one.

In some cases, programs need to contain some non-logical fragments, for instance for input-output. But the practice shows that Prolog makes possible building programs which are to a substantial extent declarative; in other words, a substantial part of such program is a logic program. Numerous examples are given in the textbooks, for instance [14]. For a more formal discussion of this issue see [6].

It should be noted that the operational, low-level approach to Prolog programming is often overused. In such programs it is not the declarative semantics that matters. A typical example is the red cut [14] – a programming technique which is based on pruning the search space; the program has undesired logical consequences, which are however not computed due to the pruning. Understanding such program substantially depends on its operational semantics. And understanding the operational semantics is usually more difficult than that of the declarative semantics. In particular, examples of programs with the red cut are known, for which certain choices of the initial query lead to unexpected results [14, p. 202–203], [3, Chapter 4. In seems that some Prolog textbooks over-use such style of programming (like [2,3], at least in their earlier editions).

Debugging Tools of Prolog. We begin with a terminological comment. Often the term "debugging" is related to locating errors in programs. However its meaning is wider; it also includes correcting errors. So a better term for locating errors is *diagnosis*. However this text still does not reject the first usage, as it is quite common.

Despite Prolog has been designed mainly as an implementation of logic programming, its debugging tools work solely in terms of the operational semantics. So all the advantages of declarative programming are lost when it comes to locating errors in a program. The Prolog debugger is basically a tracing tool. It communicates with the programmer only in terms of the operational semantics. She (the programmer) must abandon the convenient high abstraction level of the declarative semantics and think about her program in operational terms.

Declarative Diagnosis. In principle, it is well known how to locate errors in logic programs declaratively, i.e. abstracting from the operational semantics (see e.g. [4, Section 7] and the references therein). The approach is called *declarative diagnosis* (and was introduced under a name *algorithmic debugging* by Shapiro [13]). Two kinds of errors of the declarative semantics of a program are dealt with: *incorrectness* – producing results which are wrong according to the specification, and *incompleteness* – not producing results which are required by the specification. We learn about an error by encountering a *symptom* – a wrong or missing answer obtained at program testing. Given a symptom, an incorrectness (respectively incompleteness) diagnosis algorithm semi-automatically locates an error in the program, asking the user some queries about the specification.

Unfortunately, declarative diagnosis was not adapted in practice. No tools for it are included in current Prolog systems.

Intended Model Problem. A possibly main reason for lack of acceptance of declarative diagnosis was discussed in [4, Section 7]. Namely, declarative diagnosis requires that the programmer exactly knows the relations to be defined by the program. Formally this means that the programmer knows the least Herbrand model of the intended program. (In other words, the least Herbrand model is the specification.) This requirement turns out to be unrealistic. For instance, in an insertion sort program we do not know how inserting an element into an unsorted list should be performed. This can be done in any way, as the algorithm inserts elements only into sorted lists. Moreover, this can be done differently in various versions of the program. See [6] for a more realistic example[2]. Let us call this difficulty *intended model problem.*

Usually the programmer knows the intended least Herbrand model of her program only approximately. She has an *approximate specification*: she knows a certain superset S_{corr} and a certain subset S_{compl} of the intended model. The superset tells what may be computed, and the subset – what must be computed. Let us call the former, S_{corr}, the *specification for correctness* and the latter, S_{compl}, the *specification for completeness.* Thus the program should be correct with respect to the former specification and complete with respect to the latter: $S_{compl} \subseteq M_P \subseteq S_{corr}$ (where M_P is the least Herbrand model of the program). In our example, it is irrelevant how an element is inserted into an unsorted list; thus the specification for correctness would include all such possible insertions (and the specification for completeness would include none).

Now it is obvious that when diagnosing incorrectness the programmer should use the specification for correctness instead of the intended model, and the specification for completeness should be used when diagnosing incompleteness [4]. The author believes that this approach can make declarative diagnosis useful in practice.

---

[2] In the main example of [6], the semantics of a particular predicate differs at various steps of program development.

Intended model problem was possibly first noticed by Pereira [12]. He introduced the notion of *inadmissible* atomic queries. A formal definition is not given[3]. We may suppose that ground inadmissible atoms are those from $S_{corr} \setminus S_{compl}$. Generally, this notion is not declarative; an inadmissible atom seems to be one that should not appear as a selected atom in an LD-tree of the program.

Naish [11] proposed a 3-valued diagnosis scheme. The third value, *inadmissible*, is related to the search space of a diagnosis algorithm, and to its queries. The form of queries depends on the particular algorithm, e.g. it may be an atom together with its computed answers. So the third value is not (directly) related to the declarative semantics of programs. It turns out that applying the scheme to incorrectness diagnosis ([11, Section 5.1]) boils down to standard diagnosis w.r.t. S_{corr}, and applying it to incompleteness diagnosis ([11, Section 5.2]) – to the standard diagnosis w.r.t. S_{compl} (where S_{compl} is the set of correct atoms, and S_{corr} is the set of correct or inadmissible ones). So introducing the 3-valued scheme seems unnecessary (at least for incorrectness and incompleteness diagnosis).

This Paper. The role of this paper is to find if, how, and to which extent the Prolog debugger can be used as a tool for declarative logic programming. We focus on the debugger of SICStus Prolog. We omit its advanced debugging features, which are sophisticated, but seem not easy to learn and not known by most of programmers.

The paper is organized as follows. The next section deals with the Prolog debugger and the information it can provide. Section 3 discusses applying the debugger for diagnosing incorrectness and incompleteness. The last section contains conclusions.

2   Prolog Debugger

In this section we present the Prolog debugger and try to find out how to use it to obtain the information necessary from the point of view of declarative programming. First we relate the computation model used by the debugger to the standard operational semantics (LD-resolution). We also formalize the information needed for incorrectness and incompleteness diagnoses. For incorrectness diagnosis, given an atomic answer A we need to know which clause $H \leftarrow B_1, \ldots, B_n$ have been used to obtain the answer A (A is an instance of H), and which top-level atomic answers (instances of $B_1, \ldots, B_n$) have been involved. For incompleteness diagnosis, the related information is which answers have been computed for each selected instance of each body atom B_i of each clause $H \leftarrow B_1, \ldots, B_n$ resolved with a given atomic query A. In Sect. 2.2 we describe the messages of the debugger. Section 2.3 investigates how to extract from the debugger's output the information of interest.

---

[3] "a goal is admissible if it complies with the intended use of the procedure for it – i.e. it has the correct argument types – irrespective of whether the goal succeeds or not" (p. 6 of the extended version of [12]).

2.1   Byrd Box Model and LD-Resolution

The debugger refers to the operational semantics of Prolog in terms of a "Byrd box model". Roughly speaking, the model assigns four ports to each atom selected in LD-resolution. From a programmer's point of view such atom can be called a procedure call. The model is usually easily understood by programmers. However it will be useful to relate it here to LD-resolution, and to introduce some additional notions. In this paper, we often skip "LD-" and by "derivation" we mean "LD-derivation" (unless stated otherwise).

Structuring LD-derivations. Let us consider a (finite or infinite) LD-derivation D with queries $Q_0, Q_1, Q_2 \ldots$, the input clauses $C_1, C_2, \ldots$, and the mgu's $\theta_1, \theta_2, \ldots$. By a **procedure call** of D we mean the atom selected in a query of D. Following [5,7], we describe a fragment of D which may be viewed as the evaluation of a given procedure call A.

Definition 1. Consider a query $Q_{k-1} = A, B_1, \ldots, B_m$ $(m \geq 0)$ in a derivation D as above. If D contains a query $Q_l = (B_1, \ldots, B_m)\theta_k \cdots \theta_l$, $k \leq l$, then the call A (of Q_{k-1}) **succeeds** in D.

  In such case, by the **subderivation** for A (of Q_{k-1} in D) we mean the fragment of D consisting of the queries Q_i where $k-1 \leq i \leq l$, and for $k-1 \leq i < l$ each Q_i contains more than m atoms[4]. We call such subderivation **successful**. The (computed) **answer** for A (of Q_{k-1} in D) is $A\theta_k \cdots \theta_l$.

  If A (of Q_{k-1}) does not succeed in D then the **subderivation** for A (of Q_{k-1} in D) is the fragment of D consisting of the queries Q_i where $k-1 \leq i$.

  By a *subderivation* (respectively an *answer*) for A of Q in an LD-tree $\mathcal{T}$ we mean a subderivation (answer) for A of Q in a branch D of $\mathcal{T}$.

  Now we structure a subderivation D for an atom A by distinguishing in D top-level procedure calls. Assume A is resolved with a clause $H \leftarrow A_1, \ldots, A_n$ in the first step of D. If then an instance of A_i becomes a procedure call, we call it a top-level call. More precisely:

Definition 2. Consider a subderivation D for A, with first two queries $Q_{k-1} = A, Q'$ and $Q_k = (A_1, \ldots, A_n, Q')\theta_k$, where $n > 0$. So $A_1, \ldots, A_n$ is the body of the clause used in the first step of the subderivation. Let $|Q_k|$ be the length of Q_k (the number of atoms in Q_k).

  Consider an index j, $1 \leq j \leq n$. If there exists in D a query of the length $|Q_k| + 1 - j$ and $Q_{i_j} = (A_j, \ldots, A_n, Q')\theta_k \cdots \theta_{i_j}$ is the first such query then we say that $A_j\theta_k \cdots \theta_{i_j}$ (of Q_{i_j}) is a **top-level call** of D, and the subderivation D' for $A_j\theta_k \cdots \theta_{i_j}$ (of Q_{i_j}) in D is a **top-level subderivation** of D.

A top-level call of a subderivation D for A will be also called a top-level call for A.

---

[4] Thus each such Q_i is of the form $A_1, \ldots, A_{m_i}, (B_1, \ldots, B_m)\theta_k \cdots \theta_i$ where $m_i > 0$. This implies that the least $l > k$ is taken such that Q_l is of the form $(B_1, \ldots, B_m)\theta_k \cdots \theta_l$.

Notice that if A is resolved with a unary clause ($n = 0$, and D consists of two queries) then D has no top-level subderivations. Also, if a top-level subderivation D' of D is successful then the last query of D' is the first query of the next subderivation, or it is the last query of D.

We are ready to describe what information to obtain from the debugger in order to facilitate incorrectness and incompleteness diagnosis. First we describe which top-level answers correspond to an answer for A; we may say that they have been used to obtain the answer for A.

Definition 3. If subderivation D for A as in Definition 2 is successful then it has n top-level subderivations, for atoms $A_j\theta_k\cdots\theta_{i_j}$ ($j = 1, \ldots, n$). Their answers in D are, respectively, $A'_j = A_j\theta_k\cdots\theta_{i_{j+1}}$ (where i_{n+1} is the index of the last query $Q_{i_{n+1}} = Q'\theta_k\cdots\theta_{i_{n+1}}$ of D). In such case, by the **top-level success trace** for A (in D) we mean the sequence $A'_1, \ldots, A'_n$ of the answers.

Top-level success traces will be employed in incorrectness diagnosis. For diagnosing incompleteness, we need to collect all the answers for each top-level call.

Definition 4. Consider an LD-tree $\mathcal{T}$ with a node Q. Let A be the first atom of Q. By the *top-level search trace* (or simply **top-level trace**) for A (of Q in $\mathcal{T}$) we mean the set of pairs

$$\left\{ (B, \{B_1, \ldots, B_k\}) \,\middle|\, \begin{array}{l} B \text{ is the first atom of a node } Q' \text{ of } \mathcal{T}, \\ Q' \text{ occurs in a subderivation } D' \text{ for } A \text{ of } Q \text{ in } \mathcal{T}, \\ B \text{ is a top-level call of } D', \\ B_1, \ldots, B_k \text{ are the answers for } B \text{ of } Q' \text{ in } \mathcal{T} \end{array} \right\}.$$

2.2 Debugger Output

For the purposes of this paper, this section should provide a sufficient description of the debugger. We focus on the debugger of SICStus. For an introduction and further information about the Prolog debugger see e.g. the textbook [3] or the manual http://sicstus.sics.se/.

Prolog computation can be seen as traversal of an LD-tree. The Prolog debugger reports the current state of the traversal by displaying one-line items; such an item contains a single atom augmented by other information. A procedure call A is reported as an item

$$n \quad d \text{ Call: } A$$

and a corresponding answer $A' = A\theta_k\cdots\theta_i$ as

$$n \quad d \text{ Exit: } A'$$

Here n, d are, respectively, the unique invocation number and the current depth of the invocation; we skip the details. What is important is that, given an Exit item, the invocation number uniquely determines the corresponding Call item.

Note that a node in an LD-tree may be visited many times, and usually more than one item correspond to a single visit. For instance, to the last node Q_l of a

successful subderivation for A (say that from Definition 1) there correspond, at least, an Exit item with atom $A\theta_k \cdots \theta_l$ and a Call item with atom $B_1\theta_k \cdots \theta_l$ (provided $m > 0$). Note that such a node is often the last query of more than one successful subderivations (cf. Definition 2). In such case other Exit items correspond to Q_l. They are displayed in the order which may be described as leaving nested procedure calls. More formally, the order of displaying the Exit items is that of the increasing lengths of the corresponding successful subderivations. (The displayed invocation depths of these items are decreasing consecutive natural numbers.)

An Exit item is preceded by `?` when backtrack-points exist between the corresponding Call and the given Exit. Thus more answers are possible for (the atom of) this Call.

At backtracking the debugger displays Redo items of the form

$$n \quad d \ \texttt{Redo:} \ A'$$

Such item corresponds to an Exit item with the same numbers n, d and atom A'. Both items correspond to the same node of the LD-tree. The Redo item appears, speaking informally, when the answer A' is abandoned, and the computation of a new answer for the same query begins. SICStus usually does not display a Redo item when the corresponding Exit item was not preceded by `?`.

A Fail item

$$n \quad d \ \texttt{Fail:} \ A$$

is displayed when no (further) answer is obtained for A. This means that a node with A selected is being left (and will not be visited anymore). The numbers and the atom in a Fail item are the same as those in the corresponding Call item. Both the Call and Fail items correspond to the same node of the LD-tree.

We described the output of the debugger of SICStus. Commands of the debugger will be described when necessary. The debuggers of most Prolog systems are similar. However important differences happen. For instance the debugger of SWI-Prolog (http://swi-prolog.org/) does not display the invocation numbers. This may make difficult e.g. finding the Call item corresponding to a given Exit item. On the other hand, the debuggers or Ciao (http://ciao-lang.org/) and Yap (https://github.com/vscosta/yap-6.3) seem to display such numbers.

2.3   Obtaining Top-Level Traces

We are ready to describe how to obtain top-level traces using the Prolog debugger. We first deal with the search trace.

Algorithm 1 (All answers). Assume that we are at a Call port; the debugger displays

$$n \quad d \ \texttt{Call:} \ B$$

We show how to obtain all the answers for B. Do repetitively the following.

1. Type **s** to skip the details of processing the query B and to go to the corresponding `Exit` or `Fail` port.

2. If the obtained port is $n \ d$ `Exit:` B' then B' is a computed answer for B.
   Type **jr** (to jump to the `Redo` port; $n \ d$ `Redo:` B' is displayed). Repeat
   (step 1) to compute further answers
   If the obtained port is a `Fail` then all the answers have been obtained. To
   come back to the initial `Call` port, type **r**.

   An alternative to using this algorithm is to simply run Prolog on query B
(e.g. using the "break" option of the debugger).

Algorithm 2 (Top-level trace). Assume that we are at a Call port

$$n \quad d \ \texttt{Call:} \ A$$

We show a way of obtaining the top-level search trace for A. Repetitively do the
following.

1. If an item

   $$n \quad d \ \texttt{Call:} \ A \qquad \text{or} \qquad n \quad d \ \texttt{Redo:} \ A'$$

   is displayed then type enter to make one step of computation.[5]
2. If

   $$n \quad d \ \texttt{Fail:} \ A$$

   is displayed then the search is completed. The trace has been obtained.
3. If

   $$n \quad d \ \texttt{Exit:} \ A'$$

   is displayed then type **jr** (to jump to the `Redo` port of A, in order to continue
   the search).
4. If

   $$n_i \quad d{+}1 \ \texttt{Call:} \ B_i$$

   is displayed then employ Algorithm 1 to obtain the answers for B_i. Query B_i
   together with the answers is an element of the top-level trace for A.
   Now we are again at the same `Call:` B_i item. Type **s** to arrive at the first
   answer for B_i, (or to a `Fail` if there is none).
5. If

   $$n_i \quad d{+}1 \ \texttt{Exit:} \ B_i' \qquad \text{or} \qquad n_i \quad d{+}1 \ \texttt{Fail:} \ B_i$$

   is displayed then type enter, to make a single step.[6]

---

[5] In the case of `Call` there are three possibilities. If the result is an item
$n_1 \ d{+}1$ `Call:` B_1 then B_1 is an instance of the first atom of the body of the
clause used in the resolution step. Obtaining $n \ d$ `Exit:` A' means that a unary
clause was used and A succeeded immediately. Obtaining $n \ d$ `Fail:` A means that
A failed immediately, as it was not unifiable with any clause head.
In the case of `Redo:` A', we deal with backtracking after having obtained an answer
A' for A. Then there is a fourth possibility: obtaining a `Redo:` B_j' item, where B_j'
is an answer obtained for (an instance of) an atom B_j from the body of the clause
used to obtain the answer A'.

[6] After an `Exit`, this leads to a `Call:` B_{i+1} item, or to an `Exit:` A' item; the latter
when B_i is (an instance of) the last atom of the used clause. After a `Fail`, this leads
(in a simple case) to a `Redo:` B_{i-1}.
Here B_{i-1}, B_i, B_{i+1} are instances of three consecutive atoms of the used clause.

6. If

$$n_i \quad d+1 \text{ Redo: } B_i'$$

is displayed then type **s** to arrive at the next answer for B_i (or to a **Fail** if there is none).

The algorithm outputs the same answers (**Exit** items) twice (by Algorithm 1 and after an **s** at steps 4 and 6). So all the details of the trace are displayed even if we do not invoke Algorithm 1. But obtaining the top-level trace from such output seems too tedious; we need to group each query with its answers (e.g. by sorting by the invocation numbers), and remove unnecessary items. This can be done by a shell command `cut -b 2- | sort -nk 1 | egrep 'Call:|Exit:'`.

Algorithm 3 (Top-level success trace). Assume that we obtained an Exit item containing an answer A'. The item corresponds to the last query of a successful subderivation D for an atom A. In order to extract from the debugger output the top-level success trace for A in D, we need that the debugger has displayed the Call and Exit items containing the top-level calls of D and the corresponding answers. If this is not the case then, at the n d **Exit:** A' item, type **r** to arrive to the corresponding Call item, n d **Call:** A. Then perform Algorithm 2 until arriving again to the **Exit:** A' item (all the invocations of Algorithm 1 may be skipped).

To select a top-level success trace from the printed debugger items, do repetitively the following. The trace will be constructed backwards. Initially the current item is n d **Exit:** A'. Repetitively do the following:

The current item is

$$n \quad d \text{ Exit: } A' \qquad \text{or} \qquad n_j \quad d+1 \text{ Call: } B_j$$

Consider the preceding item. If the immediately preceding item is

$$n_{j'} \quad d+1 \text{ Exit: } B_{j'}'$$

then $B_{j'}'$ is obtained as an element of the success trace. Find the corresponding

$$n_{j'} \quad d+1 \text{ Call: } B_{j'}$$

item, and make it the current item.
Otherwise, the preceding item is

$$n \quad d \text{ Call: } A \qquad \text{or} \qquad n \quad d \text{ Redo: } A''$$

and all the elements of the top-level success trace for A have been found.

The construction of a top-level success trace can be made more efficient, by re-starting the computation with A' as the initial query. Then the search space to obtain a success of A' (and the corresponding top-level success trace) may be substantially smaller than that for original atomic query from the Call item.

3 Diagnosis

This section first discusses diagnosis of incorrectness, and then that of incompleteness. In each case we first present the diagnosis itself, and then discuss how it may be performed employing the Prolog debugger.

3.1 Diagnosing Incorrectness

A *symptom* of incorrectness is an incorrect answer of the program. More formally, consider a program P and an Herbrand interpretation S_{corr}, which is our specification for correctness. A symptom is an answer Q such that $S_{corr} \not\models Q$, where S_{corr} is the specification for correctness. (In other words, Q has a ground instance $Q\theta$ such that $Q \notin S_{corr}$.) When testing finds such a symptom, the role of diagnosis is to find the error, this means the reason of incorrectness. An error is a clause of the program which out of correct (w.r.t. S_{corr}) premises produces an incorrect conclusion. More precisely:

Definition 5. Given a definite program P and a specification S_{corr} (for correctness), an **incorrectness error** is an instance

$$H \leftarrow B_1, \ldots, B_n \qquad (n \geq 0)$$

of a clause of P such that $S_{corr} \models B_i$ for all $i = 1, \ldots, n$, but $S_{corr} \not\models H$.
    An *incorrect clause* is a clause C having an instance $C\theta$ which is an incorrectness error.

In other words, C is an incorrect clause iff $S_{corr} \not\models C$. In what follows, by a *correct atom* we consider an atom A such that $S_{corr} \models A$ (where S_{corr} is the considered specification for correctness).
    Note that we cannot formally establish which part of the clause is erroneous. Easy examples can be constructed showing that an incorrect clause C can be corrected in various ways; and each atom of C remains unchanged in some corrected version of C [4, Section 7.1].
    The incorrectness diagnosis algorithm is based on the notion of a proof tree, called also implication tree.

Definition 6. Let P be a definite program and Q an atomic query. A **proof tree** for P and Q is a finite tree in which the nodes are atoms, the root is Q and

if $B_1, \ldots, B_n$ are the children of a node B then $B \leftarrow B_1, \ldots, B_n$ is an instance of a clause of P $(n \geq 0)$.

Note that the leaves of a proof tree are instances of unary clauses of P.
    Now diagnosing incorrectness is rather obvious. If an atom Q is a symptom then there exists a proof tree for P, Q. The tree must contain an incorrectness error (otherwise the root of the tree is correct, i.e. $S_{corr} \models Q$). A natural way of searching for the error, in other words an incorrectness diagnosis algorithm, is as follows: Begin from the root and, recursively, check the children $B_1, \ldots, B_n$

of the current node whether they are correct (formally, whether $S_{corr} \models B_i$). If all of them are correct, the error is found; it is $B \leftarrow B_1, \ldots, B_n$ (where B the parent of $B_1, \ldots, B_n$). Otherwise take an incorrect child B_i, and continue the search taking B_i as the current node.

Obviously, such search locates a single error. So correcting the error does not guarantee correctness of the program.[7]

3.2 Prolog Debugger and Incorrectness

Now we try to find out to which extent the algorithm described above can be mimicked by the standard Prolog debugger. Unfortunately, the debugger does not provide a way to construct a proof tree for a given answer. We can however employ top-level success traces to perform a search similar to that done by the incorrectness diagnosing algorithm described in Sect. 3.1.

A Strategy for Incorrectness Errors. Here we describe how to locate incorrectness errors using the Prolog debugger.

Algorithm 4. Assume that while tracing the program we found out an incorrect answer A' (for a query A). So we are at an Exit item containing A'. Type **r** to arrive to the corresponding Call item n d `Call: A`. Do repetitively the following:

1. Construct the top-level success trace $B'_1, \ldots, B'_m$ for the subderivation D (for an atom A, where A' is the answer for A in D), as described in Algorithm 3.
2. Check whether the atoms of the trace are correct (formally, whether $S_{corr} \models B'_i$). If all of them are, then the search ends.
   Otherwise take an item n_i $d+1$ `Exit: B'_i`, in which B'_i is incorrect, and find the corresponding Call item n_i $d+1$ `Call: B`. Now repeat the search, with A, A' replaced by, respectively, B, B'_i, by typing a command **jc** n_i, or by starting new tracing from query B (in some cases **jc** n_i does not lead to the expected Call item).

The last obtained top-level success trace $B'_1, \ldots, B'_m$ points out the incorrect clause (Definition 5) of the program. The clause is $C = H \leftarrow B_1, \ldots, B_m$ such that the obtained answers are instances of the body atoms of C: each B'_j is an instance of B_j, for $j = 1, \ldots, m$. The head H of C is unifiable with the last call B for which the top-level success trace was built.

Obviously, the algorithm can be improved by checking the correctness of each element B'_i of the trace as soon as it is located. (So the success trace needs to be constructed only until an incorrect element is found.)

The approach of Algorithm 4 is rather tedious. A more natural way to locate incorrectness errors is as follows.

---

[7] This does not even guarantee that the symptom we began with would disappear – there may be some other errors involved.

Algorithm 5

1. Assume, as above, that an incorrect answer A' was found. Begin as in Algorithm 4: arrive to the `Call:` A that resulted in the incorrect answer, and start constructing a top-level search trace.
2. For each obtained item n_i $d+1$ `Exit:` B' check if B' is correct.
3. If B' is an incorrect answer, then restart the search from B'.
4. If no incorrect answer has appeared until arriving to the incorrect answer A' then the error is found. It is the last clause C whose head was unified with A in the computation. (Formally, an instance of C is an incorrectness error.) The clause may be identified, as previously, by extracting the top-level success trace (for the subderivation that produced A').

Comments. In Algorithms 4 and 5, it is often not necessary to know the (whole) top-level success trace to identify the incorrect clause in the program. In many cases, knowing the last one or two answers of the trace is sufficient. For instance, let n' d' `Call:` B be the last call for which top-level trace was inspected. The last item displayed by the debugger is n' d' `Exit:` B' (where B' is incorrect). Assume that the previous item is n_j $d'+1$ `Exit:` B'_j. Then the top-level trace of interest is not empty, B'_j is its last atom and is an instance of the last body atom of an erroneous clause. If the program has only one such clause, then finding the rest of the top-level success trace is unnecessary.

The error located by the second approach (Algorithm 5) may be not the one that caused the initial incorrect answer A'. This is because the search may go into a branch of the LD-tree distinct from the branch in which A' is produced. Anyway, an actual error has been discovered in the program. This outcome is useful, as each error in the program should be corrected.

Note that the approach is complete, in the sense that the error(s) responsible for A' can be found. This is due to the nondeterministic search performed by the algorithm. The error(s) will be located under some choice of incorrect answers in the top-level search traces.

The search may be made more efficient if, instead of tracing the original computation, we re-start it with an incorrect answer as a query. The corresponding modification (of both algorithms) is as follows. Whenever an incorrect answer B' is identified, instead of continuing the search for the corresponding call B, one interrupts the debugger session and begins a new one by starting Prolog with query B'. The query will succeed with B' (i.e. itself) as an answer, but the size of the trace may be substantially smaller (and is never greater). Moreover, any incorrect instance of B' may be used instead of B'.

The Prolog debugger does not facilitate searching for the reason of incorrectness. Finding a top-level success trace is tedious and not obvious. In particular, there seems to be no way of skipping the backtracking that precedes obtaining the wrong answer. The abilities of the debugger make Algorithm 5 preferable; this approach in a more straightforward way uses what is offered by the debugger.

Looking for the reason of an incorrect answer is a basic task. It is strange that such a task is not conveniently facilitated by the available debugging tools.

3.3   Diagnosing Incompleteness

A specification for completeness is, as already stated, an Herbrand interpretation which is the set of all required ground answers of the program. A symptom of incompleteness is lack of some answers of the program. More formally, given a program P and a specification S_{compl}, by an incompleteness symptom we may consider a ground atom A such that $S_{compl} \models A$ but $P \not\models A$. As a symptom is to be obtained out of an actual computation, we additionally require that the LD-tree for A is finite. We will consider a more general notion of a symptom:

Definition 7. Consider a definite program P and a specification S_{compl} (for completeness). Let A be an atomic query for which an LD-tree is finite and let $A\theta_1, \ldots, A\theta_n$ be the computed answers for A from the tree. If there exists an instance $A\sigma \in S_{compl}$ such that $A\sigma$ is not an instance of any $A\theta_i$ ($i = 1, \ldots, n$) then $A, A\theta_1, \ldots, A\theta_n$ is an **incompleteness symptom** (for P w.r.t. S_{compl}).

We will often skip the sequence of answers, and say that A alone is the symptom. The definition can be generalized to non-atomic queries in an obvious way.

Definition 8. Let P be a definite program, and S_{compl} a specification. A ground atom A is **covered** by a clause C w.r.t. S_{compl} if there exists a ground instance $A \leftarrow B_1, \ldots, B_n$ of C ($n \geq 0$) such that all the atoms $B_1, \ldots, B_n$ are in S_{compl}.
   A is *covered by the program P* (w.r.t. S_{compl}) if A is covered by some clause $C \in P$.

Informally, A is covered by P if it can be produced by a rule from P out of some atoms from the specification.
   If there exists an incompleteness symptom for P w.r.t. S_{compl} then there exists an atom $p(\boldsymbol{t}) \in S_{compl}$ uncovered by P w.r.t. S_{compl} [4,13]. Such an atom locates the error in P. This is because no rule of P can produce $p(\boldsymbol{t})$ out of atoms required to be produced. This shows that the procedure p (the set of clauses beginning with p) is the reason of the incompleteness and has to be modified, to make the program complete. Note that similarly to the incorrectness case, we cannot locate the error more precisely. Various clauses may be modified to make $p(\boldsymbol{t})$ covered, or a new clause may be added. An extreme case is adding to P a fact $p(\boldsymbol{t})$.
   Incompleteness diagnosis means looking for an uncovered atom, or – more generally – for an atom with an instance which is uncovered: Such atom localizes the procedure of the program which is responsible for incompleteness.

Definition 9. Let P be a definite program, and S_{compl} a specification. An **incompleteness error** (for P w.r.t. S_{compl}) is an atom that has an instance which is not covered (by P w.r.t. S_{compl}).

Name "incompleteness error" may seem unnatural, but we find it convenient.

A class of incompleteness diagnosis algorithms employs the following idea. Start with an atomic query A (which is a symptom) and construct a top level trace for it. Inspect the trace, whether it contains a symptom B. If so then invoke the search recursively with B. Otherwise A is an incompleteness error; we located in the program the procedure that is responsible for the incompleteness. Such approach (see e.g. [8,12]) is sometimes called *Pereira-style* incompleteness diagnosis [10].

3.4   Prolog Debugger and Incompleteness

We show how Pereira-style diagnosis may be done using the Prolog debugger.

Algorithm 6 (Incompleteness diagnosis). Begin with a symptom A. Obtain the top-level search trace for A. In the trace, check if the atom B from a Call item together with the answers $B_1, \ldots, B_n$ from the corresponding Exit items is an incompleteness symptom. If yes, invoke the same search starting from B. If the answer is no for all Call items of the trace, the search is ended as we located A as an incompleteness error.

Comments. Standard comments about incompleteness diagnosis apply here. To decrease the search space, it is useful to start the diagnosis from a ground instance $A\theta \notin S_{compl}$ of the symptom A (instead of A itself). The same for each symptom B found during the search – re-start the computation and the diagnosis from an appropriate instance of B.

Often an incorrectness error coincides with an incompleteness error – a wrong answer is produced instead of a correct one. The programmer learns about this when facing an incorrect answer B_i (appearing in a top-level trace). A standard advice in such case [8,10] is to switch to incorrectness diagnosis. This is because incorrectness diagnosis is simpler, and it locates an error down to a program clause (not to a whole procedure, as incompleteness diagnosis does). The gain of such switch is less obvious in our case, since the effort needed for incorrectness diagnosis (Algorithm 5) may be not smaller than that for incompleteness (Algorithm 6).

4   Conclusions

Prolog makes declarative logic programming possible – programs may be written and reasoned about in terms of their declarative semantics, to a substantial extent abstracting from the operational semantics. This advantage is lost when it comes to locating errors in programs, as the Prolog debugger works solely in terms of the operational semantics. We may say that logic programming would not deserve to be called a declarative programming paradigm if debugging had to be based on the operational semantics.

This paper is an attempt to study if and how the Prolog debugger can be used for declarative programming. It presents how the debugger can be used to

perform incorrectness and incompleteness diagnosis[8]. Examples, missing here, are available at http://arxiv.org/abs/2003.01422/. The debugger used is that of SICStus; the presented approach may be difficult to apply with the debugger of SWI-Prolog, as the latter does not display unique invocation numbers (needed in incorrectness diagnosis, Algorithms 3 and 4).

The results are rather disappointing. Declarative diagnosis based on the Prolog debugger is tedious and unnatural. Rather obvious information (like the proof tree leading to a given answer, or a top-level success trace) is impossible or difficult to obtain. Possibly, this drawback is a substantial obstacle for employing declarative logic programming in practice.

This drawback particularly concerns incorrectness diagnosis. Additionally, debugging of incorrectness seems more important than that of incompleteness. This is because incompleteness is often caused by producing incorrect answers instead of correct ones. Also, incorrectness diagnosis is more precise, as it locates a smaller erroneous fragment of the program than incompleteness diagnosis does. Hence the first step towards making Prolog debugging declarative is to implement a tool supporting incorrectness diagnosis. Experiments show that it is sufficient to provide a tool for convenient browsing of a proof tree (which provides an abstraction of the part of computation responsible for the considered incorrect answer).

The Introduction contains a discussion about how to avoid the "intended model problem", which is possibly the main reason why declarative diagnosis of logic programs was abandoned. The author believes that the proposed solution [4] can make declarative diagnosis useful in practice. What is missing are tools.

References

1. Apt, K.R.: From Logic Programming to Prolog. International Series in Computer Science, Prentice-Hall (1997)
2. Bratko, I.: Prolog Programming for Artificial Intelligence, 4th edn. Addison-Wesley, New York (2012)
3. Clocksin, W., Mellish, C.: Programming in Prolog: Using the ISO Standard, 5th edn. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-55481-0
4. Drabent, W.: Correctness and completeness of logic programs. ACM Trans. Comput. Log. **17**(3), 18:1–18:32 (2016). https://doi.org/10.1145/2898434
5. Drabent, W.: Proving completeness of logic programs with the cut. Formal Aspects Comput. **29**(1), 155–172 (2017). https://doi.org/10.1007/s00165-016-0392-0
6. Drabent, W.: Logic + control: on program construction and verification. Theory Pract. Logic Program. **18**(1), 1–29 (2018). https://doi.org/10.1017/S1471068417000047

---

[8] We may informally present the underlying idea of this paper in a different way: To understand what the Prolog debugger can tell us about the declarative semantics of the program, we need to be able to obtain the following information. 1. For a given atomic answer A, what are the top-level answers that have lead to A? (This is formalized as top-level success trace.) 2. For a given atomic query Q, and for each top-level atomic query B in the computation for Q, what are all the answers for B?

7. Drabent, W., Małuszyński, J.: Inductive assertion method for logic pro-
   grams. Theor. Comput. Sci. **59**, 133–155 (1988). https://doi.org/10.1016/0304-
   3975(88)90099-0
8. Drabent, W., Nadjm-Tehrani, S., Małuszyński, J.: Algorithmic debugging with
   assertions. In: Abramson, H., Rogers, M.H. (eds.) Meta-Programming in Logic
   Programming, pp. 501–522. The MIT Press, Cambridge (1989)
9. Kowalski, R.A.: Algorithm = logic + control. Commun. ACM **22**(7), 424–436
   (1979). https://doi.org/10.1145/359131.359136
10. Naish, L.: Declarative diagnosis of missing answers. New Generation Comput.
    10(3), 255–286 (1992). https://doi.org/10.1007/BF03037939
11. Naish, L.: A three-valued declarative debugging scheme. In: 23rd Australasian
    Computer Science Conference (ACSC 2000), pp. 166–173. IEEE Computer Society
    (2000). https://doi.org/10.1109/ACSC.2000.824398
12. Pereira, L.M.: Rational debugging in logic programming. In: Shapiro, E. (ed.) ICLP
    1986. LNCS, vol. 225, pp. 203–210. Springer, Heidelberg (1986). https://doi.org/
    10.1007/3-540-16492-8_76. Extended version at https://userweb.fct.unl.pt/~lmp/
13. Shapiro, E.: Algorithmic Program Debugging. The MIT Press, Cambridge (1983)
14. Sterling, L., Shapiro, E.: The Art of Prolog, 2nd edn. The MIT Press, Cambridge
    (1994)