Chapter 25
Algorithmic Debugging with Assertions

Włodek Drabent ⋆ *Simin Nadjm-Tehrani*
Jan Małuszyński

⋆ *Institute of Computer Science, Polish Academy of Sciences*
Dept. of Computer and Information Science, Linköping University

Abstract

Algorithmic debugging, as presented by Shapiro, is an interactive process where the debugging system acquires knowledge about the expected meaning of a program being debugged and uses it to localize errors. This paper suggests a generalization of the language used to communicate with the debugger. In addition to the usual "yes" and "no" answers, formal specifications of some properties of the intended model are allowed. The specifications are logic programs. They employ library procedures and are developed interactively in the debugging process. An experimental debugging system incorporating this idea has been implemented. In contrast to some other systems, its diagnosis algorithms do not require instantiation of unsolved goals by the oracle. This is achieved by generalization of the oracle in the incorrectness algorithm, and by adopting a new approach in the insufficiency algorithm. A formal proof of correctness and completeness of the new insufficiency algorithm is presented. Extensions for some Prolog features are discussed.

25.1 Introduction

This paper deals with diagnosis of logic programs and extends the pioneering work of Shapiro (Shapiro 1983) by studying a way of partial automating of the oracle by means of assertions. The paper also includes extensions of the approach for some Prolog features. Implementation issues and preliminary experimental results are reported in Drabent et al. (1988b).

Logical foundations of algorithmic debugging can be found in Ferrand (1987) and Lloyd (1987). Our basic notions, though slightly different, have been strongly influenced by these papers. (For discussion of the differences see Section 6).

Every (pure) logic program P has a model (see e.g. (Lloyd 1987)). P is often considered to be the specification of the least Herbrand model M_P. On the other hand, the program should properly reflect the intentions of the user. These can be thought of as the "intended model" and can be viewed as a subset I_P of the Herbrand base. If I_P differs from M_P the program is erroneous.

The program P is said to be

- incorrect iff $M_P - I_P \neq \emptyset$, i.e. iff it specifies some element which is not in the intended model, and
- insufficient iff I_P - $M_P \neq \emptyset$, i.e. iff some elements of the intended model are not specified by the program.

In this paper we do not deal with the termination aspect; we concentrate on tracing incorrectness and insufficiency of a logic program. The objective of debugging is to find a cause of an error in the program.

In the case of incorrectness it is a clause which "produces" elements not in I_P. More precisely, it is a clause whose body is valid in I_P and whose head is not valid in I_P. Such a clause will be called *incorrect*. It has a ground instance, such that all atoms of its body are in I_P and its head is not in I_P.

In the case of insufficiency it is a predicate p for which some atom $p(t_1, \ldots, t_n)$ valid in I_P cannot be produced by the clauses defining p. More precisely, there is no ground instance $H \leftarrow B_1, \ldots, B_m$ $(m \geq 0)$ of a clause of P such that H is an instance of $p(t_1, \ldots, t_n)$ and $B_1, \ldots, B_m$ are in I_P. The atom $p(t_1, \ldots, t_n)$ is called *uncovered*.

The elements of M_P can be computed using SLD-resolution. To discover an error and to localize its cause in the program one has to compare the results of computations, including failures, with the intended model. However, the latter is generally not formalized. To solve this problem Shapiro introduces the concept of oracle (Shapiro 1983). The *ground oracle* decides whether an atom is in the intended model. The *existential oracle* decides whether there is a solution to a given goal and is capable of producing elements of the intended model which are instances of the given goal. In practice, it is the user who answers the questions concerning the intended model.

Shapiro's debugging system acquires knowledge about the intended model through necessary interactions with the oracle. This knowledge consists of:

1. a finite subset of the intended model — YES answers of the ground oracle and the solutions produced by the existential oracle;
2. a subset of the complement of the intended model — NO answers of the ground oracle;
3. a finite set of atoms satisfiable in the intended model — YES answers of the existential oracle;

4. a finite set of atoms unsatisfiable in the intended model — NO answers of the existential oracle.

The language of the oracles does not allow to specify infinite subsets of the intended model nor infinite sets of atoms satisfiable in the intended model. The negative answers of the existential oracle are not used for tracing incorrectness although they specify infinite subsets of the complement of the intended model. This language is rather low level — the knowledge about the intended model is communicated in form of examples. There may therefore exist many queries concerning similar atoms.

Shapiro pointed out that incorporation of "constraints and partial specifications" into the algorithmic debugging scheme may reduce the number of interactions with the user (Shapiro 1983, p.79). This paper develops and formalizes this idea. In the approach presented the user is allowed to provide the system with formal specifications of some properties of the intended model. These formal specifications may be developed interactively in the debugging process. The diagnosis system uses its actual knowledge about the intended model to localize errors. Whenever this is not sufficient for evaluation of results of the computation the system queries the user. The answer augments system knowledge about the intended model. This scheme includes as special cases the answers used in Shapiro's system. But generally the language of answers is more powerful. If the user is able to provide the system with some general properties of the intended model, the number of interactions decreases dramatically.

Another aspect of our debugging methodology is the relative ease with which the user can interact with the system. As it is unlikely that the complete specification of the model is conveniently provided by the user, there will be a number of interactions between the system and the user. Our insufficiency diagnoser, in contrast to Shapiro's, will not require the user to provide instances of unsolved goals. Instead, the user is expected to recognize the solutions to a goal and to identify a case where some answer is missing.

The rest of the paper is organized as follows. In Section 2 a language of assertions is introduced as a natural generalization of the language of the oracle and the use of assertions for algorithmic debugging is discussed. Section 3 describes the necessary oracle interactions. In section 4 debugging algorithms based on assertions are presented. The section also includes a formal proof of correctness and completeness of our insufficiency diagnosis algorithm. Extensions for some Prolog features are discussed in section 5 and comparisons with related work are presented in section 6. Sections 7 and 8 contain conclusions and topics for future research.

25.2 Assertions

We suggest to extend the communication language of the algorithmic debugger. In addition to the simple YES and NO answers we want to provide the user with a possibility to describe some properties of the intended model. For this we introduce assertions as a device to specify (not necessarily finite) sets of (not necessarily ground) atoms of the object language. An obvious choice is to use logic programs to provide executable specifications of such sets using, as much as possible, existing library procedures.

Let S be a set of (not necessarily ground) atoms in the language L of a given logic program P. The objective is thus to construct a logic program Q with a unary predicate s "specifying" the set S. The clauses of Q beginning with the symbol s will be called assertions (for S).

More precisely, there should be a one-to-one correspondence between set S and the set of all ground atoms of the form $s(\ldots)$ that are logical consequences of Q. In other words, each atom A of L should be coded as a ground term A' of the language M of the program Q. We adopt the following coding scheme:

> if A is an atom and $X_1, \ldots, X_n$ are variables occurring in A , the image of A is $A' = A\{X_1/var(1), \ldots, X_n/var(n)\}$ $(1 \leq i \leq n)$

where var is a functor not used in the object language. (To obtain uniqueness it may be assumed that $X_1, \ldots, X_n$ are ordered according to their first occurrences in A).

This is a ground representation in terms of (Hill and Lloyd 1988). Notice that all the predicate symbols and functors of L become functors of M. Atoms which are not the same up to variable renaming have different images.

One may argue that assertions can be used for full specification of the intended model. This would amount to giving an alternative correct version of the buggy program (Dershowitz and Lee 1987). Such a solution is completely unrealistic in most cases.

It is often suggested that while developing a new version of an existing program the existing version can be used as an oracle (e.g. (Sterling and Shapiro 1986)). Taken literally, this idea is also unrealistic because it requires that every procedure of the new program has its counterpart with the same intended meaning in the old program. Sterling and Shapiro (1986, p. 323) mention *permutation_sort* and *quicksort* (Sterling and Shapiro 1986 - p.55,56) as an example. However, *quicksort* contains procedures *partition* and *append* that are not specified by the *permutation_sort* program.

We suggest to employ four properties of the intended model I_P in our debugging framework. They generalize the four types of answers given by the oracles as discussed in section 1. The properties are specified in the above-mentioned sense by four fixed predicate symbols in a program $As(I_P)$. This gives rise to four types of assertions. The ways the debugging algorithms use $As(I_P)$ are described in section 3.

We now give definitions of the 4 types of assertions.

> *Positive assertions.* These are used to define sets of (not necessarily ground) atoms valid in the intended model. We specify positive assertions using the predicate symbol *true*. If $true(A')$ is a logical consequence of $As(I_P)$ and A' is the image of A then for all substitutions θ, $A\theta \in I_P$ provided $A\theta$ is ground.

Example 1.

Consider the intended relation *insert* as in Shapiro (1983). It includes (as a proper subset) all triples (X, Y, Z) such that X is an integer, Y is a sorted list of integers and Z is a sorted list whose elements are X and all elements of Y. This property can be formalized as the following assertion:

$true(insert(X, Y, Z)) \leftarrow$
$\quad integer(X),$
$\quad sorted\_integer\_list(Y),$
$\quad sorted\_integer\_list(Z),$
$\quad permutation([X \mid Y], Z).$

(It is assumed that $As(I_P)$ contains procedures with the obvious meaning for the predicate symbols of the body.)

> *Negative assertions.* These are used to specify sets of atoms not valid in the intended model. We specify negative assertions using the predicate symbol $false$. If $false(A')$ is a logical consequence of $As(I_P)$ and A' is the image of A then there exists a substitution θ , such that $A\theta$ is ground and $A\theta \notin I_P$.

Example 2.

Consider the intended relation *sort* where the arguments are integer lists and the second one is a sorted permutation of the first. The following assertion characterizes an infinite set of atoms not valid in the intended model:

$false(sort(X, Y)) \leftarrow$
$\quad member(var(N), Y),$
$\quad not\_member(var(N), X).$

(If there is a variable that is a member of the second list but not a member of the first then the atom has a ground instance outside the intended model.)

Note that YES and NO answers to questions asked by the ground oracle can be seen as singleton positive and negative assertions respectively.

> *Positive existential assertions.* These are used to specify sets of atoms satisfiable in the intended model. We define positive existential assertions using the predicate symbol *posex*. If $posex(A')$ is a logical consequence of $As(I_P)$ and A' is the image of A then there exists a substitution θ such that $A\theta \in I_P$.

Example 3.

The intended *isort* predicate of Shapiro (1983) has the property that whenever it is called with the first argument being a list of integers and the second argument being an uninstantiated variable then there exists an instance of this call which is in the intended model. This can be formalized as the following assertion:

$posex(isort(X, var(Y))) \leftarrow integer\_list(X).$

The positive existential assertions generalize YES answers of the existential oracle.

> *Negative existential assertions.* These are used to specify sets of atoms unsatisfiable in the intended model. We define negative existential assertions using the predicate symbol *negex*. If $negex(A')$ is a logical consequence of $As(I_P)$ and A' is the image of A then for all substitutions $\theta, A\theta \notin I_P$.

Example 4.

The intended *isort* predicate of Shapiro (1983) has the property that none of its success instances have an unsorted list as the second argument. In the language of assertions this can be formalized as follows:

$negex(isort(X, Y)) \leftarrow$
$\quad integer\_list(Y),$
$\quad not\_sorted(Y).$

The negative existential assertions generalize NO answers to existential queries. It is worth noticing that various notions of types for logic programs discussed in the literature e.g. (Zobel 1987), (Mycroft and O'Keefe 1984), (Nilsson 1983), can be related to negative existential assertions (if an argument in an atom is of a wrong type then the atom should be unsatisfiable).

At every stage of development the program $As(I_P)$ should describe the intended model. A necessary condition for that is that it describes some model. This is not the case if, for example, both $true(A')$ and $false(A')$ are logical consequences of $As(I_P)$. That corresponds to A being both valid and not valid in I_P. The responsibility for providing consistent assertions is on the user. In the next section it is described how (partial) consistency checking of $As(I_P)$ is performed. Notice, that even the basic Shapiro algorithms are not free of the danger of inconsistent answers. Let A be an atom with variables and B its ground instance. When tracing incorrectness the answer concerning B may be YES, i.e. B is in I_P. Independently, when tracing insufficiency the answer concerning A may be NO, i.e. there is no instance of A in I_P.

25.3 Oracle interactions

In order to show how the assertions are used, we set out the questions that are posed by the diagnosis algorithms and the way they are answered. The basic idea is that a question is first attempted to be answered with the help of $As(I_P)$. Only if this attempt fails, the user is queried. She may either answer with YES/NO or specify a relevant property of I_P by adding new clauses to $As(I_P)$.

The ground image of atom A in the coding scheme used by assertions is denoted by A'.

(1) Universal questions:

This type of question is asked by the incorrectness diagnoser:

> "Is the atomic formula A valid in the intended model?" (i.e. are all its ground instances members of I_P?)

The insufficiency diagnoser requires answers to two additional types of questions:

(2) Existential questions:

> "Is A satisfiable in the intended model?" (i.e. is there a ground instance of A which is a member of I_P?)

(3) Incompleteness questions:

The algorithm needs the information whether certain solved goals have produced all the expected answers in the intended model. This is obtained by asking:

> "For the atom A, is there an instance $A\theta \in I_P$ such that $A\theta$ is not an instance of some member of the set $\{A\theta_1, \ldots, A\theta_n\}$?" (Substitutions $\theta_1, \ldots, \theta_n$ are (all the) computed answer substitutions for $\leftarrow A$ and P).

The system uses the knowledge explicitly represented in $As(I_P)$ for answering the above questions before querying the user. Moreover, some queries to the user may be avoided by exploiting the information that is implicit in the assertions. For instance, it may happen that $true(A')$ is a logical consequence of $As(I_P)$ but $posex(A')$ is not. However, in this case the answer to the existential question for A is YES and querying the user is unnecessary.

Let A be an atom and B its instance. The following properties hold:
(1) If A is valid in I_P then it is also satisfiable in I_P.
(2) If A is unsatisfiable in I_P then it is not valid in I_P.
(3,4) If atom A is valid (unsatisfiable) in I_P then B is also valid (unsatisfiable) in I_P.

(5,6) If B is not valid (satisfiable) in I_P then A is not valid (satisfiable) in I_P.
(7) A ground atom is satisfiable iff it is valid.

The procedure for answering *universal* questions employs properties (2), (6) and (7). For atom A:

- If $true(A')$ is a logical consequence of $As(I_P)$ then the answer to this question is YES.
- If A is ground and $posex(A')$ is a logical consequence of $As(I_P)$ then the answer is YES.
- If $false(B')$ or $negex(B')$ is a logical consequence of $As(I_P)$ for some B being an instance of A then the answer to this question is NO.
- Otherwise the user is queried.

The procedure for answering *existential* questions employs properties (1),(5) and (7). For atom A:

- If $posex(B')$ or $true(B')$ is a logical consequence of $As(I_P)$ for some B being an instance of A then the answer to this question is YES.
- If $negex(A')$ is a logical consequence of $As(I_P)$ then the answer to this question is NO.
- If A is ground and $false(A')$ is a logical consequence of $As(I_P)$ then the answer is NO.
- Otherwise the user is queried.

The properties (3) and (4) are not used by the answering procedures due to implementation difficulties. However, if the decoded set of atoms defined by $true$ (resp. $negex$) is closed under substitution then employing properties (3) and (4) does not change anything. In practice, these two sets are closed under substitution in every reasonable $As(I_P)$.

The YES/NO answers to the incompleteness questions are to be provided by the user.

Let $Assertion$ be any of the predicate letters $true, false, posex, negex$. The answering procedures require checking whether there exists an instance B of a given atom A such that $Assertion(B')$ is a logical consequence of $As(I_P)$. To do this, program $As(I_P)$ is queried with the goal $\leftarrow Assertion(A)$ (note: not coded A). This is because the coded image of any instance of A is also an instance of A and if the coded image of a term is an instance of A then the term is an instance of A.

Consider a universal (or an existential) question to be answered by the system. The answering procedures may sometimes be able to give both YES and NO answers to this question. In this case $As(I_P)$ is inconsistent and debugging is aborted.

To accumulate the knowledge implied by user YES/NO answers to universal and existential queries, new assertions can be added to $As(I_P)$ by the system. If the user answer to the *universal* query for A is NO then assertion $false(A') \leftarrow$ is added. If the user answer to the *universal* query for A is YES then any instance of A is valid in I_P. So assertion $true(A) \leftarrow$ is added. Now for any instance B of A, $true(B')$ is logical consequence of $As(I_P)$ (since coded image of any instance of A is also an instance of A and vice versa).

If the user answer to the *existential* query for A is YES then assertion $posex(A') \leftarrow$ is added. If the user answer to the *existential* query for A is NO then the assertion $negex(A) \leftarrow$ is added (since any instance of A is unsatisfiable in the intended model).

User answers to incompleteness questions can also be recorded. If the answer is NO then unary clause $complete\_solutionsA) \leftarrow$ is recorded. Then a success of $\leftarrow complete\_solutionsB')$ implies a NO answer to the incompleteness question concerning B since the answer is NO for any instance of A. (It turns out that recording of YES answers is unnecessary.)

25.4 Diagnosis algorithms

(1) Incorrectness diagnosis

If the SLD-refutation procedure of a goal $\leftarrow A$ with program P produces a substitution θ such that $A\theta$ is not valid in I_P, then an incorrect clause instance has to be found.

The original algorithm (Shapiro 1983) finds an incorrect clause by systematic traversal of a ground proof tree whose root is not in I_P. In actual computations of logic programs the proof trees constructed need not be ground. The idea can be extended for non-ground trees in two different ways. The suggestion of (Shapiro 1986, p.325) is that the oracle should instantiate the visited node, if possible to an instance not included in the intended model, otherwise to any instance. The solution used in this paper is to require that the oracle decides whether the visited node is *valid* in the intended model or not. (This is a generalization of the original *ground oracle* since validity of a ground atom in I_P means that it is an element of I_P.)

We use the top down version of Shapiro's basic algorithm as presented in (Sterling and Shapiro 1986) with this generalization. The queries posed by the algorithm are dealt with in the manner described above. The input to the algorithm is an atom A for which the program gives a wrong answer (this means a success instance of A which is not valid in I_P). The algorithm returns a (not necessarily ground) instance of a clause in P such that the atoms in the body of the clause are valid in I_P and the head is not.

The algorithm is sound because it always returns an incorrect clause instance. It is also complete in the sense that it terminates and returns such an answer for any input that satisfies the input condition stated above.

(2) Insufficiency diagnosis

We will say that program P is $insufficient$ for $\leftarrow A$ if there exists θ such that $A\theta \in I_P$ and no answer more general than θ is a computed answer substitution for $\leftarrow A$ and P. An atom A is *completely covered* by program P if for every θ such that $A\theta \in I_P$, P contains a clause which has an instance where $A\theta$ is the head and all the atoms in its body are in I_P. (Hence A is not completely covered if there exists θ such that $A\theta \in I_P$ and there is no clause instance of P with the head $A\theta$ and the body atoms in I_P.)

If a program P is insufficient for a goal $\leftarrow A$ then the insufficiency diagnoser is called to identify an atomic formula C not completely covered by the program P. The algorithm is based on the assumption that Prolog computation rule is used in the resolution.

Before describing the algorithm we introduce the following definition:

The **search forest** for A consists of a tree for each non-unary clause of P whose head is unifiable with A. Let $H \leftarrow B_1, \ldots, B_n \quad (n > 0)$ be a variant of such a clause.
Then

> (B_1, γ), where γ is an mgu of H and A, is the root of the corresponding tree

and
if (B_i, θ) is a node of the tree, and program P gives $\{\sigma_1, \ldots, \sigma_m\} \quad (m \geq 0)$ as computed answer substitutions to goal $B_i\theta$

> then $(B_{i+1}, \theta\sigma_j)$ for $j = 1, \ldots, m$ is a child of this node if $i < n$
> and $(\Box, \theta\sigma_j)$ for $j = 1, \ldots, m$ is a child of this node if $i = n$.

Note that (B_i, θ) is a node in the forest iff B_i instantiated to $B_i\theta$ is a selected goal (on the top level) in the computation for A. Note also that $(\Box, \ldots)$ leaves correspond to successes of $\leftarrow A$. If $(\Box, \theta)$ is a leaf in the forest then goal $\leftarrow A$ succeeds with computed answer substitution $\theta \mid variables(A)$ (where $variables(A)$ stands for the set of variables occurring in A and $\theta \mid S$ stands for the restriction of θ to the elements of S). For a given A, the search forest is unique up to variable renaming.

The Algorithm

The input to the algorithm is an atomic formula A for which the program is insufficient and the computation for $\leftarrow A$ is finite (under Prolog computation rule); the output is a *not completely covered* atom.

The insufficiency diagnoser asks questions about the nodes of the search forest for A. The types of questions asked have been discussed in section 3. (The order of visiting the nodes is irrelevant to the correctness of the algorithm).

For (B, θ) being a leaf of the forest, $B \neq \Box$, the existential question is asked about $B\theta$. If the answer is YES then the algorithm is recursively called with $B\theta$. (No questions are asked about a success leaf.)

For (B, θ) being an internal node with children $(C, \theta\sigma_1), \ldots, (C, \theta\sigma_m)$ (where C is an atomic formula or $\Box$), the incompleteness question is asked about the set $\{B\theta\sigma_1, \ldots, B\theta\sigma_m\}$ and the goal $B\theta$. If the answer is YES then the insufficiency diagnoser is called recursively on $B\theta$.

If for all nodes of the forest the answers for all the questions are NO, then A is returned as a not completely covered atom and the algorithm terminates. Otherwise, a not completely covered atom is found by the recursive call(s) of the algorithm.

Correctness and completeness of the algorithm

Lemma

Consider program P and a search forest for atom C.

If

> for every node in the forest the answer to the question asked by the algorithm is NO and
>
> C is completely covered by P

then

> P is sufficient for $\leftarrow C$.

Proof

Assume that the premises of the lemma hold. Let $C\gamma \in I_P$ (without loss of generality it may be assumed that the domain of γ is $variables(C)$). We show that there exists a computed answer substitution for $\leftarrow C$ that is more general than γ. As C is completely covered by P, there exists

$$A \leftarrow B_1, \ldots, B_n \qquad (*)$$

which is a variant of a clause of P and there exists a substitution δ such that

$$C\gamma = C\delta = A\delta,$$

$$B_1\delta, \ldots, B_n\delta \in I_P.$$

If $n = 0$ then an mgu of C and A (restricted to $variables(C)$) is the required computed answer substitution. Assume $n > 0$. We show that in the search forest for C, in the tree corresponding to $(*)$ there exists a leaf $(\Box, \theta)$ such that θ is more general than δ. Assume that this does not hold. Note that for the root (B_1, θ_1) substitution θ_1 is more general that δ. Let i be the greatest number for which there exists a node (B_i, θ) in the tree where θ is more general than δ. Then $B_i\delta$ is an instance of $B_i\theta$. Two cases are possible.

1. (B_i, θ) is a leaf of the tree. The answer to the existential question about $B_i\theta$ is YES (as $B_i\delta \in I_P$). Hence contradiction.

2. (B_i, θ) has sons $(C, \theta\sigma_1), \ldots, (C, \theta\sigma_m)$ where $C = B_{i+1}$ or $C = \Box$. As the answer to the incompleteness question about $B_i\theta$ is NO, for some substitution j, $\theta\sigma_j$ is more general than δ. Contradiction.

Now, the computed answer substitution corresponding to $(\Box, \theta)$ is $\theta \mid variables(C)$. This solution is more general than $\delta \mid variables(C) = \gamma$. This concludes the proof.

$\Box$

As the algorithm is (recursively) called for atom C only if P is insufficient for C, it follows by the Lemma that the atom returned by the algorithm is not completely covered by P. Hence the algorithm is correct.

Note that each search forest is finite. So is the recursion depth of the algorithm (otherwise the computation for $\leftarrow A$ would be infinite). Thus the algorithm always returns an answer. So the algorithm is complete, in the sense that it returns a correct answer for any atom satisfying the input conditions of the algorithm. This notion of completeness is weaker than the usual one (where a diagnoser answer is required even for goals that loop). However, the difference is insignificant if the program in question is run under the Prolog computation rule. This is because the insufficiency diagnoser is not used for a goal for which the program loops (under this computation rule). It should be mentioned that all practical insufficiency diagnosing algorithms are incomplete with respect to this stronger completeness (Naish 1988).

It remains to discuss a situation where a program is both incorrect and insufficient. An example is a program giving a wrong answer and missing a correct one. In such cases it is more convenient to perform incorrectness diagnosis first. The incorrectness diagnoser usually searches a smaller search space, does not ask incompleteness questions and produces more informative answers: an incorrect clause instance refers to a wrong clause while a not completely covered atom refers to a whole procedure.

A particular case is when one of the atoms displayed by an incompleteness question is not valid in I_P. Then it is convenient to interrupt the insufficiency diagnosis and start diagnosing incorrectness with such an atom. This usually leads to a faster and more informative result.

25.5 Extensions

In this section extensions of the method for some Prolog features are discussed. The approach presented above is declarative: the intended meaning of a logic program is its intended model (and the actual meaning is its least Herbrand model). Extensions of the method will be discussed within the same framework. This excludes programs with side effects: those using input-output or *assert-retract*.

25.5.1 Built-in predicates

Many Prolog built-in predicates can be treated declaratively since they can be specified by the relations they define (over the Herbrand universe). Programs employing such built-ins can be treated as logic programs by including unary clause $p(t)$ for any built-in p and any term tuple t in the relation corresponding to p. This includes predicates such as *functor, arg* , *integer* etc. , $= ..$ $(univ)$, $=$, $\backslash=$ and Prolog arithmetic (since uninstantiated arguments to arithmetic procedures are detected as run-time errors). No questions are posed by the debugging algorithms about such built-ins. They are assumed to be implemented correctly.

The built-in predicates $var, nonvar, ==$ (exact equality) and $\backslash==$ (exact inequality) cannot be described by the declarative semantics. They will be referred to here as *extralogical predicates*. Operationally, their role is to succeed without binding their arguments or to fail. This may be seen as (conditional) pruning of a part of a search tree.

Programs using such predicates can be dealt with by our method if the programmer knows the intended model of the program with extralogical predicates removed. Such a model provides an approximation of the expected behaviour which is inexpressible in a declarative way. Obviously, only those bugs that lead to insufficiency or incorrectness with respect to this model can be found by the diagnoser. In such a setting, an extralogical predicate call can be a reason for insufficiency but not for incorrectness. The insufficiency diagnoser returns an atom and the error is in the procedure corresponding to this atom. Either the reason for insufficiency is an extralogical call in this procedure or the atom is not completely covered with respect to the procedure with extralogical calls removed. In the first case the programmer has to decide whether the behaviour of the procedure is actually erroneous.

25.5.2 Cut

Here we discuss introducing *cut* into our debugging framework. As the framework is declarative, cut should be treated declaratively. This means that its role is understood as cutting away part of the search space. If inserting a cut removes some program's answers then this cut is called red. Otherwise it is called green (van Emden 1982). Obviously, the declarative semantics is no longer valid for programs with red cuts. However, the declarative debugging approach is still able to provide meaningful information about bugs in such programs, as shown below.

For incorrectness diagnosis the same algorithm is used. It analyses a proof tree that lead to a wrong answer. As a result an incorrect instance of a program clause, say $p(T) \leftarrow B$, is obtained. Now, wrong clauses are allowed in correct programs provided the clauses are protected by (red) cuts. The user has to decide whether the error has to be treated as a wrong clause or as a wrong usage of the cut. In the second case the reason is a not activated cut. It can be either a cut missing (or misplaced) in the program text or a cut not executed due to a failure of preceding calls. The inactivated cut should occur in the clauses of procedure p since p succeeds with a goal instance $p(T)$ not in I_P (and all the goals in B gave correct answers).

A possible treatment at this stage is to search for insufficiency that could lead to a cut not being executed. A search forest is built for the preceding clauses of procedure p that contain cuts. Incompleteness and existential questions are then asked about its nodes as in the insufficiency diagnoser. If there is a YES answer then some answers to the corresponding subgoal are missing. The insufficiency diagnoser has to be called for this subgoal. If all the answers are NO then procedure p has to be corrected.

This is similar to a suggestion of (Huntbach 1987) (with a modification that non-failing subgoals also have to be examined) and to the approach of (Pereira and Calejo 1988). A similar procedure is performed by a programmer in the examples of (Takahashi and Shibayama 1985). The difference is that in our approach the user is guided towards a declaratively correct program. A construction "red cut + incorrect clause" is accepted by a debugger only after an explicit decision of the user.

For insufficiency diagnosis, a variant of the algorithm of section 4 is used. Only the goals that actually occurred during the computation are represented in the search forest. (This means a node (B_i, θ) has sons $(B_{i+1}, \theta\sigma_1), \ldots, (B_{i+1}, \theta\sigma_j)$ only if during the actual computation $B_i\theta$ succeeded j times (with answer substitutions $\sigma_1, \ldots, \sigma_j$); j may be less than the number of answers of the program to goal $B_i\theta$ due to interruption of backtracking by a cut.) For leaves of the form $(B, \theta), B \neq \square$, the existential questions are asked as in the basic algorithm. Incompleteness questions are asked only about those internal nodes (B, θ) for which it is known that the corresponding goal $\leftarrow B\theta$ produced all the answers (in other words it eventually failed). A recursive call of the algorithm is made for a node for which the answer is YES. The algorithm returns an atom A for which all questions in the related search forest were answered NO. Either A is not completely covered (in the sense of section 4) or the reason for insufficiency is a cut in procedure p , where $A = p(\ldots)$. More precisely, the cut is either misplaced or is unnecessarily executed due to an incorrect success of the preceding procedure calls of the same clause. In the last case, the incorrectness diagnoser can be used to find an actual bug.

To assist the user, she is informed

1. which clauses with a head matching A were not used in the computation and
2. for which nodes (B, θ) in the search forest there exists a missing answer to $B\theta$ (together with a list of answers obtained for each such node).

Note that giving lists of answers missing due to cut may be informative but it may also lead to infinite computations for some programs.

If a cut is the reason for insufficiency then the information given to the user provides a compacted version of a trace of the procedure execution. It allows her to localize clauses and cut(s) executed and shows a history of backtracking in the clause containing this cut (these cuts). This makes it possible to decide whether a cut is misplaced and to localize a possible incorrect success.

25.5.3 Negation

For programs with negation (Lloyd 1987), the intended model I_P is a model of the completion of a correct program. For a sound implementation of negation it is necessary that computation of the program is safe or weakly safe (Lloyd 1987). This means that whenever $\neg A$ is selected and fails, A succeeds with an empty answer substitution. This requirement should be checked by the debugging system.

To incorporate negation, the algorithms of section 4 are extended in an obvious way suggested by McCabe (Shapiro 1983) (Lloyd 1987). If the incorrectness diagnoser, instead of an incorrect clause instance, finds a literal $\neg A$ that incorrectly succeeds then the insufficiency diagnoser is called with A. Whenever the answer to a question posed by the insufficiency diagnoser about a literal $\neg B$ is YES, the incorrectness diagnoser is called with input B (YES means missing solutions). The insufficiency diagnoser does not ask incompleteness questions about successful negative literals because there cannot be any missing solutions (negative literals succeed with empty substitutions).

Answering questions about negative literals refers to the property that $\neg A$ is valid (satisfiable) in I iff A is unsatisfiable (not valid) in I. Hence to answer the universal (existential) question about $\neg A$ the existential (universal) question about A is answered as described in section 3 and the answer is negated.

The diagnoser obtained by composing the extended incorrectness and insufficiency diagnosers in this way, is *sound* in the sense that for any atom satisfying the respective input conditions, the result (if any) is either an incorrect clause instance or a not completely covered atom. The definitions of these notions are obvious extensions of those for definite programs. The input condition is either that for the incorrectness diagnoser or that for the insufficiency diagnoser of section 4. The proof of soundness is a straightforward modification of the proofs for the definite program diagnosers.

The diagnoser is also *complete* in the sense that it returns an answer for any atom satisfying the relevant input condition. An outline of a proof is given below.

The Prolog computation rule is assumed. Let the diagnoser be called with an atom A as an input. From the input conditions it follows that there exists a finite computation D of the given object program P with goal $\leftarrow A$ such that D results in an incorrect answer or searches the whole search space.

The computation of the composed diagnoser can be seen as a sequence of mutual calls of the incorrectness and insufficiency diagnosers. The argument passed at such a call is an instance of an atom that actually occurred as a selected subgoal in D. The sequence of these atoms is a subsequence of the sequence of selected goals of D. Hence the number of mutual calls is finite.

The computation corresponding to a single call to the incorrectness diagnoser is finite since it is a search of a finite tree. By an argument analogous to that of section 4, the computation corresponding to a single call to the insufficiency diagnoser is finite. Hence the whole computation is finite.

25.6 Comparisons with related work

The types of assertions introduced originate from the analysis of the logical nature of answers given by the oracles of Shapiro. They also have their counterparts in the algorithms of Ferrand (1987) and Lloyd (1987) where the oracles are represented by the predicates *valid* and *unsatisfiable* (and to a certain extent *impossible* (Ferrand 1987)). But the oracles have complete knowledge of the intended model while the assertions only approximate it. For example the assertions *true* and *false* provide incomplete information about validity of a given atom in the intended model. The first of them specifies a set of atoms valid in the intended model, the other a set of atoms non-valid in the intended model. A given atom may belong to none of the sets while the validity oracles of Ferrand and Lloyd can always decide its validity. However, the oracles are outside the system, while the assertions constitute a part of the system (which is incrementally developed during the external interactions). External interactions are necessary in our system only if the actual assertions cannot produce the required answer. In this case the external interaction provides an increment for the existing assertions so that the question can be answered.

The only work known to us that uses a concept similar to our approximate specification is a recent paper by Lichtenstein and Shapiro (1988). It deals with debugging of concurrent programs and introduces an additional *abstract oracle*. The abstract oracle specifies a superset of the intended behaviour of a program while the concrete oracle specifies the intended behaviour exactly. The intention of introducing abstract oracle is to ask questions that are simpler to answer by a programmer, whereas the role of assertions in our approach is to automatically answer some of the questions. Assertions can specify not only supersets but also subsets of the set of interest. (The set of interest is either I_P or the set of atoms satisfiable in I_P.) For a given debugger, it is fixed which (super-) sets can be specified by the abstract oracle, while assertions can specify any set.

The algorithms of (Shapiro 1983) (Sterling and Shapiro 1986), Ferrand (1987) and Lloyd (1987) require that the oracle is able to deliver elements of the intended model. If the oracle is the user, this type of interaction may create difficulties or even lead to wrong answers. One of our objectives has been to free the user from this burden.

A new algorithm for insufficiency diagnosis presented in this paper automatically generates answers for atomic subgoals. Instead of generating bindings the user is (sometimes) asked whether the set of generated answers is complete. A similar approach is presented by Pereira (1986). However that work seems to rely on the procedural semantics of Prolog, while ours has a clean logical foundation and our algorithm is proved correct and complete.

Clearly, the bindings provided by the user can speed-up the diagnosis process. However, the decision whether a binding is to be given or not should be left to the user. Our algorithms can be easily extended with that option.

There are some differences in basic definitions used in this paper and the papers by Ferrand and Lloyd which give logical foundations for declarative debugging. We follow Lloyd in that our intended model is a ground Herbrand model in contrast to the nonground term model of Ferrand.

Another difference concerns the results produced by the debugger. Since we do not force the user to produce bindings during the debugging process, the final result may come out less instantiated than in the other systems. To be more precise, consider separately the form of our results in diagnosing incorrectness and insufficiency.

For incorrectness, the result returned is an incorrect instance of a program clause, that is $H \leftarrow B$ such that B is valid in I and H is non-valid in I. This is similar to Ferrand's definition of incorrectness ((Ferrand 1987) Definition 4). However, his debugger returns such $H \leftarrow B$ that B is valid and H is unsatisfiable. The results produced by the debugger of Lloyd also have this property. In the case of Ferrand (1987) this is due to representing variables of the program by variables of the debugger. (The diagnoser is a logic program, if it returns $H \leftarrow B$ then it is also able to return any instance of $H \leftarrow B$; hence H has to be unsatisfiable for the diagnoser to be sound.) The approaches are equivalent, since any incorrect clause (in our terminology) has an instance where the head is unsatisfiable and the body valid.

For insufficiency diagnosis the situation is similar. The results produced by our debugger are atoms which are not completely covered while Lloyd's debugger produces uncovered atoms (An atom A is called uncovered if A is valid in I_P and none of its instances is in $T_P(I_P)$; A is completely covered iff $A\theta \in I_P$ implies $A\theta \in T_P(I_P)$). Comparing the definitions one can see that every not completely covered atom has an instance which is uncovered. This instance is not produced by our system. This is because we do not force the user to produce bindings for subgoals during the debugging process. Ferrand's notion of insufficiency is a counterpart of Lloyd's uncovered atom but it is weaker than the latter (A is an insufficiency if A is valid in I_P and not all its ground instances are in $T_P(I_P)$). However, the answers really produced by Ferrand's algorithm are similar to those of Lloyd. More precisely, in both cases the result of insufficiency diagnosis is an uncovered atom.

Another difference to be mentioned concerns inputs for insufficiency diagnosis. Usually it is supposed to be a finitely failed goal which is satisfiable in the intended model. However, the finite failure of this goal may be caused by the fact that some subgoal of the computation does not fail but produces an insufficient number of answers. In most systems this situation is handled by asking the oracle to provide all intended answers for the subgoal. The answer not produced by the insufficient program will cause its failure and eventual localization of insufficiency. Our system does not require the user to provide correct subgoal instances. This also results in extending the allowed inputs for the diagnoser: the input is a goal whose computation terminates and delivers an incomplete set of computed answers.

A different debugging approach is presented in Pereira (1986), and Pereira and Calejo (1988). That approach is not declarative but operational. It does not refer to an intended model but to the intended *behaviour* of the program. A program is understood through its operational properties and not through the logical ones.

In addition to incorrectness and insufficiency, a third kind of program error is introduced in Pereira (1986), and Pereira and Calejo (1988), namely inadmissible call pattern. An example of such an error is violation of a mode declaration. The following is not made explicit there but is important from our point of view. Inadmissibility is related to an additional specification saying which call patterns are allowed during program computation. Inadmissibility is not related to the declarative semantics: a program P may be correct (that means $M_P = I_P$) but manifest inadmissible call pattern(s).

An earlier work using assertions within logic programming is (Drabent and Małuszynsński 1987). Here assertions are used to prescribe predicate call and success patterns. Preassertions in this sense describe all the predicate calls that are possible: those which succeed and those which fail. The described form of procedure calls is not expressible in terms of declarative semantics and is therefore, in general, not related to the assertions introduced in this paper. Nevertheless, it is possible to make use of such assertions in the debugging process by detecting inadmissible call patterns. We believe that this can be a generalization of Pereira's queries relating to admissibility of a goal (Pereira 1986), (Pereira and Calejo 1988).

In this paper we are interested in logic programming as a declarative programming paradigm. Thus we do not include inadmissibility into our debugging framework.

25.7 Conclusions

The main contribution of this paper is the formalization of the concept of assertion for algorithmic debugging. Assertions provide a formal description of some properties of the intended model, thus "approximating" it. They give a flexible framework for its formal description. On one end of the spectrum the yes/no oracle answers provide rudimentary but easy to produce information about the intended model. On the other end the full formal specification of the intended model can be used, if so desired. Assertions can be seen as generalizations of the simple oracle answers and include them as special cases.

It is worth noticing that the concept of assertion is orthogonal to the concept of debugging algorithm: any debugging algorithm based on oracle interactions can also use appropriate assertions.

A prototype debugger using assertions has been implemented. Algorithms which do not require correct instantiations of atoms by the user are incorporated in the implementation. Our experiments show a reduction in the debugging effort through the use of rather simple assertions and the improved algorithms. A more detailed account of experiments performed can be found in Drabent et al. (1988a) (1988b).

Modifications of the initial assertions may be preserved from session to session. In this way the debugging process gives as a side effect an interactively developed formal description of some properties of the intended model.

25.8 Future work

In practice it often happens that the intended model is not known to the programmer. Instead, she knows a set J_P of atoms that should be in M_P and a set K_P of atoms that should be in $B_P - M_P$ (where B_P is the Herbrand base). The rest, $B_P - J_P - K_P$, is irrelevant to program's specification. An interesting task is a declarative debugging methodology based on such an approximation of the intended model.

Further experiments with debugging of Prolog programs are needed to understand the debugging process better, to evaluate the presented approach and to develop pragmatics of declarative debugging with assertions.

Another subject of future work is to discuss testing of logic programs and correcting of errors. The objective would be a testing-diagnosing-correcting methodology. It should be based on declarative features of existing logic programming languages and may be a complement to methods of systematic construction and verification of programs. Although proving programs correct seems to be a more important target, programs still need debugging and providing sound methods and tools for this is a significant research task.

Acknowledgements

This work has been partially supported by the National Swedish Board for Technical Development, project number: 87-02926P, and a grant by The Royal Swedish Academy of Engineering Sciences (IVA). The first author was also supported by Polish Academy of Sciences. The editors of FGCS'88 proceedings (©ICOT, Tokyo 1988) kindly permitted us to use fragments of Drabent et al. (1988b) in this paper.

References

Dershowitz, N., and Lee, Y., Deductive Debugging, *Proceedings of the IEEE Symposium on Logic Programming* - San Francisco 1987 : 298-306.

Drabent, W., and Małszyński, J., Inductive Assertion Method for Logic Programs, *Proceedings of the International Conference on Theory and Practice of Software Development* (TAPSOFT) 1987, LNCS 250, Springer Verlag : 167-181.

Drabent, W., Nadjm-Tehrani, S., and Małuszyński, J., (1988a) Algorithmic Debugging with Assertions, Research Report LiTH-IDA-R-88-04, Linköping University, March 88.

Drabent, W., Nadjm-Tehrani, S., and Małuszyński, J., (1988b) The Use of Assertions in Algorithmic Debugging, *Proceedings of the FGCS conference* - Tokyo, November 88 : 573-581.

van Emden, M., Warren's Doctrine on the slash, Logic Programming Newsletter, December 1982.

Ferrand, G., Error Diagnosis in Logic Programming, an Adaptation E.Y. Shapiro's Method, *Journal of Logic Programming* 1987(4): 177-198.

Hill, P.M., Lloyd, J.W., Analysis of Meta-Programs, *Proceedings of the workshop on Meta-Programming in Logic Programming* , Bristol, 1988: 27-42.

Huntbach, M., Algorithmic PARLOG debugging, *Proceedings of the IEEE Symposium on Logic Programming* - San Francisco, 1987 : 288-297.

Lloyd, J.W., *Foundations of Logic Programming*, Springer Verlag, Second edition, 1987.

Lichtenstein Y., Shapiro E., Abstract Algorithmic Debugging, *Proceedings of the fifth International Conference and Symposium on Logic Programming* - Seattle, 1988: 512:531.

Mycroft, A., O'Keefe, R.A., A Polymorphic Type System for Prolog, *Artificial Intelligence* 23 , 1984 : 295-307.

Naish, L., Declarative Diagnosis of Missing Answers, Department of Computer Science, University of Melbourne, Technical report 88/9.

Nilsson, J.F., On the Compilation of a Domain-based Prolog, in: Mason, R.E.A.(ed), *Information Processing 83*, North Holland 1983 : 293-298.

Pereira, L. M. and Calejo, M., A Framework for Prolog Debugging, *Proceedings of the fifth International Conference and Symposium on Logic Programming* - Seattle, 1988: 481:495.

Pereira, L. M., Rational Debugging in Logic Programming, *Proceedings of the 3rd International Conference on Logic Programming* , LNCS 225, Springer Verlag , 1986 : 203-210.

Takahashi, H. and Shibayama, E., PRESET- A Debugging Environment for Prolog, *Proceedings of the fourth Logic Programming Conference* - Tokyo, 1985: LNCS 221, Springer Verlag: 90-99.

Shapiro, E.Y., *Algorithmic Program Debugging*, MIT Press, 1983.

Sterling, L. and Shapiro, E.Y., *The Art of Prolog*, MIT Press 1986.

Zobel, J., Derivation of Polymorphic Types for Prolog Programs, in: Lassez, J.L.,(ed), *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne 1987 : 817-838.