New Generation Computing, 10 (1991) 255-285
OHMSHA, LTD. and Springer-Verlag

Declarative Diagnosis of Missing Answers

Lee NAISH
Department of Computer Science,
University of Melbourne,
Parkville, Victoria 3052, Australia.

Received 16 May 1988
Final manuscript received 6 June 1991

Abstract This paper investigates algorithms for declarative diagnosis of missing answers in Prolog programs, especially programs which use coroutines. The logic of the problem is first presented, in the form of the simplest possible debugger. Next, we compare several previously published declarative debuggers based on Shapiro's work. Examples showing incompleteness, incorrectness and equivalence of debuggers are given. Several enhancements to these debuggers are presented which can reduce the number and complexity of questions asked of the oracle, while still supporting coroutines. Although no debugger considered is best in all cases, the new algorithms are a practical contribution. Finally, we discuss diagnosis algorithms based more on Pereira's work. These algorithms ask easier questions than Shapiro's algorithms but rely on the standard left to right computation rule. We discuss possible ways to adapt these algorithms to handle coroutining. Completeness of debuggers is also discussed.

Keywords: Debugging, Coroutines, Failure, Programming Environments.

§1 Introduction

One of the potential advantages of logic programming is that it enables one to easily separate the concerns of correctness (the declarative semantics, or what instances of a goal are true) and efficiency (the procedural semantics, or how the instances of a goal are computed).[9] To realise this potential, we must be able to initially code, test (partial) correctness and debug programs using just the declarative semantics. Declarative debugging (also called algorithmic, rational and deductive debugging) enables precisely this, at the debugging phase. Unfortunately, the standard left to right computation rule used in Prolog tends to result in infinite loops, making testing and (purely declarative) debug-

ging almost impossible. The primitives used to implement negation in Prolog (including cut) also cause problems for declarative debugging, due to the lack of clear declarative semantics.[16] Programmers must consider the procedural semantics in the initial coding, so at least some of the advantages of declarative debugging are lost.

A partial solution to these problems has been provided by Prolog dialects which have more flexible control strategies (most commonly coroutining, rather than strict left to right evaluation) which can help avoid infinite loops.[15] Most of these languages also have constructs for negation which have well defined declarative semantics and are soundly implemented. Without declarative debugging, some of the advantages of these languages are also lost. Although initial coding can be easy, if any bugs are introduced they can be very difficult to locate using conventional means because the procedural semantics is so complex. The difficulties become even more acute when parallel execution is used. Thus declarative debugging and Prolog dialects with flexible control (including parallel execution) have a natural affinity. Despite this, there has been little work on building practical declarative debuggers for these dialects, though some work has been done on declarative debugging for other (parallel) logic programming languages.

The first work on declarative debugging of Prolog was done by Shapiro.[23] It introduced algorithms for diagnosing wrong and missing answers which relied only on knowledge of the intended declarative semantics. Given a goal which succeeds or fails incorrectly, the algorithms trace through the execution, querying an oracle (most commonly the programmer) about the truth of various subgoals, and eventually isolate the error to a single clause or procedure. These algorithms were originally applied to debugging Prolog programs which used the standard computation rule but can also be applied to programs which use coroutining. Any wrong answer diagnosis algorithm can be easily adapted to support coroutining. Given a successful (wrong) derivation found using one computation rule, the sequence of calls can be reordered to simulate any other computation rule, including the standard one. However, a goal which fails finitely with one computation rule may loop with another, so missing answer diagnosis algorithms which rely on the standard computation rule cannot (necessarily) be adapted to handle coroutining.

There have been a great number of papers analyzing and extending Shapiro's work. A significant improvement in missing answer diagnosis was described in Ref. 20). The most important aspect of this algorithm was the introduction of a new kind of oracle query which resulted in easier questions to the user. The algorithm also uses dependency information to improve the search strategy and allows users to state that certain calls are "inadmissible". Despite the advantages of this algorithm, much of the subsequent work on declarative diagnosis was still based on Shapiro's algorithms, for several reasons. Shapiro's algorithms are simple and were expressed by short Prolog programs whereas

Pereira's algorithms are more complex, were not presented as clearly and use information such as admissibility which seemed procedural rather than declarative. Furthermore, Pereira's algorithm relies on the standard computation rule so it cannot be applied to coroutining (or stream and-parallel) systems.

Other areas which extend the original work include the following: theoretical issues such as soundness and completeness;[4,6,12,13,27] automating or extending the oracle interaction;[2,3,4,5,8,13,27] integration of testing and error diagnosis;[2,8,17] complexity of diagnosis algorithms;[22] integrating wrong and missing answer diagnosis with intelligent search strategies;[21] diagnosis of logic programs with arbitrary formulas in the clause bodies;[12] and declarative diagnosis of committed choice parallel logic programs.[7,10,11,19,25]

In this paper, we are primarily concerned with declarative diagnosis of missing answers, especially for (Prolog) programs which can use coroutines. Most of the ideas presented here have been implemented for the NU-Prolog system.[26] NU-Prolog has a flexible computation rule and allows arbitrary formulas in the bodies of clauses. However, to assist in presentation, the code we discuss will only deal with Horn clauses. It can be extended to deal with the more general case simply by adding more clauses, as in Ref. 12). We start by defining the simplest possible declarative debugger, then progressively make enhancements following the style of Shapiro's debuggers. Later we discuss possible extensions to debuggers more in the style of Pereira's.

§2 The Simplest Possible Declarative Debugger

There are two basic kinds of errors detected by most declarative debuggers. The simplest is an *incorrect clause instance*, which can be defined by the following clause:

```
% debugger N.0, part I
bug((A:- B)):- is_clause(A, B), unsatisfiable(A), valid(B).
```

This states that, in the intended interpretation of the program, the head of the clause instance is unsatisfiable but the body is valid. That is, the clause is incorrect in the intended interpretation. is_clause returns instances of object program clauses. valid and unsatisfiable are calls to an oracle, which may be implemented either by querying the user (as we assume in this paper) or by using some specification of the program.[2,5,8] Incorrect clause instances generally lead to incorrect answers returned by the program although if negation is present, they can also result in incompleteness of the program (missing answers). The clause above is not only a definition of an incorrect clause instance; with appropriate definitions of the three primitives it can be used as a logic program to find incorrect clause instances. It is sound and complete, since it is its own specificaiton. To be complete in practice, the order in which clause instances are considered must be fair, so that every clause instance is considered eventually. Generally, a large number of oracle queries will be needed to find a bug (or all

bugs). Previously published algorithms reduce the number of oracle queries to a practical level.

The second type of bug is an *uncovered atom*, which can be defined by the following NU-Prolog clause (all denotes universal quantification in NU-Prolog):

```
        % debugger N.0, part 2
bug(atom(A)):-
        valid(A),
        all B not (is_clause(A, B), valid(B)).
```

This states that, in the intended interpretation, the atom A is valid but there is no matching clause instance with a valid body (later we will examine weaker versions of this definition). This definition implies that the completion of the procedure[1)] is incorrect in the intended interpretation. It can also be run as a logic program, though it has the problem of requiring many oracle queries. In conjunction with the clause for incorrect clause instances, this is the simplest program for declarative error diagnosis.

A third class of bugs, *inadmissible calls*, are detected by Pereira's debuggers.[20,21)] Calls are said to be inadmissible if they have incorrect argument types. It has generally been considered that this is not related to the declarative semantics of the program, and hence has not been adopted by other declarative debugging systems. In Ref. 18) it is argued that types play an important role in the intended semantics of logic programs. It seems that this work can be used as a theoretical basis for using admissibility in declarative debugging. We do not deal with types or admissibility here however.

To find incorrect clause instances, it is sufficient to have an oracle which can determine if a ground atom is valid. The definition of uncovered atoms is more complex, and debugging missing answers generally requires more complex oracle questions. Three types of questions have been used by debuggers in the literature. The first asks whether a (generally non-ground) atom is satisfiable. The second asks what valid instances a non-ground atom has. This requires the oracle to provide bindings for variables in the atom, rather than just a yes or no answer. If the atom has several valid instances, all of these may have to be given eventually. The third asks whether a set of instances of a non-ground atom (generally the set of answers returned by the program) contains all valid instances of the atom. Shapiro-style debuggers use the first and second types of questions. Pereira-style debuggers use the first and third types of questions. Although questions of the third type are more complex in some sense, they only require a yes or no answer and they are relatively easy for the user to answer.

§3 Using a Goal to Guide the Search

The simple declarative debugger(s) presented in the previous section essentially do a blind search for all possible bugs. They do not use a bug

symptom (a wrong or missing answer) to guide the search. All previous work on declarative debugging has used a goal which exhibits incorrect behaviour to guide the search for bugs. Most debuggers are divided into two procedures: wrong, for diagnosing goals with wrong answers, and miss, for diagnosing goals with missing answers (sometimes different names are used). We discuss these approaches in this section.

Lloyd's approach[12] was to first try to isolate the logic of the problem and later add control information to make the program search for bugs more efficiently. The core of his version without "control information" is as follows:

```
        % debugger L.1
miss((A, B), R):-
        miss(A, R).
miss((A, B), R):-
        miss(B, R).
miss(A, R):-
        user_pred(A),
        clause(A, B),
        miss(B, R).
miss(A, atom (A)):- % equivalent to N.0 part 2
        user_pred(A),
        valid (A),
        \+ (clause(A, B), valid(B)).
```

The full version of the program deals with negation, quantifiers etc., and includes a definition of wrong also. We will simply be concentrating on the portion used for diagnosing missing answers for Horn clause programs. The programs we introduce can all be extended in the way Lloyd has shown. Also, we will be using standard Prolog to express the programs. To make the theory clear, Lloyd distinguished between object and meta levels and used explicit quantifiers in the logic. In this paper we try to uncover some of the practical difficulties which can be obscured in more theoretical approaches.

It is interesting to compare L.1 with N.0 (part 2). The last clause of L.1 is equivalent to N.0. The other clauses use the goal and restrict the search space, so L.1 is not completely free of control information. Rather than returning all possible bugs, it will only return bugs which are connected to the top level goal by a series of calls. This is reflected in the soundness and completeness theorems Lloyd proves. The soundness criteria for miss and wrong do not mention the goal and are equivalent to N.0: if miss (wrong) succeeds then it returns a bug (as defined by N.0). However, the completeness criterion for miss is more restricted: if the goal is satisfiable but has a finitely failed SLD tree then miss will return a bug (not necessarily all bugs). Thus, if miss is called with a goal which succeeds, no bug needs to be returned, and if the program contains several bugs, miss need only return one of them, whatever the goal is. The following example

shows how L.1 can fail, even though the program contains a bug:

```
p(a).
% q(a).        % missing (uncovered atom)

?- miss(p(a), R).          % fails (quite reasonably)
```

This seems a reasonable (partial) specification for miss. In practice, we call miss when we have found a goal which incorrectly fails, and just find and correct one bug at a time. The advantage of having a stricter completeness difinition is that the debugger can have a smaller search space. Instead of a blind search for bugs, L.1 only examines atoms which, for some computation rule, can be called (directly or indirectly) from the top level goal. For each of these atoms, it finds valid instances (by calling valid) and checks that there is no matching clause with a valid body. The search space of L.1 is still much larger than is needed to achieve completeness (it can be infinite, even for a finitely failing goal). In fact, L.1 was never intended as a practical debugger. Instead, Lloyd proposed the following program, which has additional control information, as a more practical debugger which supports coroutines.

```
        % debugger L.2
miss((A, B), R):-
        \+ call(A),              % extra control
        miss(A, R).
miss((A, B), R):-
        \+ call(B),              % extra control
        miss(B, R).
miss(A, R):-
        user_pred(A),
        clause(A, B),
        valid(B),                % extra control
        \+ call(B),              % extra control
        miss(B, R).
miss(A, atom(A)):-
        user_pred(A),
        valid(A),
        \+ (clause(A, B), valid(B)).
```

L.2 is identical to L.1 except that before each call to miss there is a check to make sure the goal fails and there is a check in the third clause to make sure the goal is valid (in practice this finds instances of the goal which are valid—see below). If the initial goal is valid but fails then this invariant will be maintained in all subsequent calls to miss. The search for bugs is restricted to clause instances which may cause the initial goal to fail, making it a practical debugger.

Procedurally, L.2 works as follows. Suppose miss is called with a valid failing atomic goal. The debugger searches for a matching clause, then searches

for a valid instance of the clause body (which must fail) then finds a failing atom in this goal and recurses. If there is no valid instance of a matching clause then the atomic goal is an uncovered atom.

The search for matching clauses and valid instances can be nondeterministic. To find valid instances of a complex goal, the goal can be broken down into atomic subgoals and the user (or another oracle) is asked about these. The user is asked whether the subgoal is satisfiable (has any valid instances), and if the reply is yes, the user must type an instance (unless the subgoal is already ground). On backtracking, the user is asked if there are more valid instances and the process is repeated. Thus, if there are N valid instances of a subgoal, the user may have to answer up to N + 1 yes/no questions and N questions where instances are given (*instance* questions). Answers typed by the user can be saved, so even if calls to valid are repeated the questions to the user are not. One of the key performance criteria of declarative debuggers is the number of questions asked, especially instance questions, as these require more effort to answer.

Ferrand[6] also proves a form of completeness for his debugger. Though written rather differently, it is essentially very similar to Lloyd's first version, but with some extra control added to the third clause (a subset of that in L.2):

```
        % debugger F.I
miss((A, B), R):-
        miss(A, R).
miss((A, B), R):-
        miss(B, R).
miss(A, R):-
        user_pred(A),
        clause(A, B),
        valid(B),              % extra control
        miss(B, R).
miss(A, atom(A)):-
        user_pred(A),
        valid(A),
        \+ (clause(A, B), valid(B)).
```

Another minor syntactic difference is the use of a predicate called satisfiable, which actually returns all valid instances of a goal. We have renamed it valid to minimise confusion (we do the same for Shapiro's debuggers given later).

Lloyd leaves the completeness of L.2 as an open question. As the program stands, it is not complete: the call to the debugger fails in following example.

```
p(a).
% p(b).              % missing
q(b).

?- miss((p(X), q(X)), R).          % fails in L.2
```

The reason for failure is that the top level call to miss contains a goal which fails and is satisfiable but not valid. In practice, this apparent bug in L. 2 can be avoided simply by calling miss via an interface procedure which finds a valid instance of the goal, rather than calling miss directly. Another source of incompleteness is the use of \+ and call to check for failure. Theoretically we should use a fair computation rule though in practice it does not seem worthwhile. The use of an unfair computation rule can result in an infinite loop instead of a call to \+ succeeding and the possible subsequent detection of a bug. However, if this was the case, the top level goal would have looped rather than finitely failed (assuming the same computation rule is used). We shall discuss various kind of infinite loops these debuggers can get into later.

Dershowitz and Lee's debugger[2] also fails to find the bug in this example. It assumes that incorrect failure is caused by the incorrect finite failure of a single atom in the computation. In this example, p(X) is missing a solution but it does not finitely fail. If we extend the example slightly, this debugger will return an incorrect answer. Instead of returning an atom of the procedure which has an incorrect completion, it returns an atom from which the procedure was called (this procedure is correct).

```
p(a).
% p(b).              % missing
q(b).
r:- p(X), q(X).

?- miss(r, R). % returns atom(r), not atom(p(b))
```

Huntbach's debugger for Parlog[7] makes a similar assumption, though this may be reasonable in Parlog, since calls cannot return more than one answer. The error may be due to a misunderstanding of the terminology used in Ref. 20), which uses the term *finitely failed* to describe goals which miss some answers, even if they succeed with some other answers. Shapiro[23] also uses the term finite failure, but in a way which is consistent with (but not identical to) the normal use.

The code of Shapiro's original debugger[23] is as follows:

```
        % debugger S.I
miss((A, B), R):-
        (call(A)->
            miss(B, R)
        ;
            miss(A, R)).
miss(A, R):-
        user_pred(A),
        (clause(A, B), valid(B) ->
            miss(B, R)
```

```
    ;
        R = atom(A)).
```

Like L.2, an interface procedure shoud be used, which makes sure the goal is valid but fails. Without this, S.1 can succeed when there are no errors:

```
p(a).

?- miss(p(b), R).          % returns atom(p(b)) but should fail
```

If the top level goal is valid and fails then this invariant will be preserved, as with L.2. In fact, the control is equivalent in the two debuggers. S.1 can be rewritten as follows:

```
miss((A, B), R):-
    \+ call(A),
    miss(A, R).
miss((A, B), R):-
    call(A),              % equivalent to \+ call(B)
    miss(B, R).
miss(A, R):-
    user_pred(A),
    clause(A, B),
    valid(B),
                          % \+ call(B) must succeed
    miss(B, R).
miss(A, atom(A)):-
    user_pred(A),
                          % valid(A) must succeed
    \+ (clause(A, B), valid(B)).
```

The first clause is identical to that of L.2. The second call has call(A) instead of \+ call(B), but the two are equivalent in the case where the first clause fails, since A succeeds and (A, B) must fail. The third clause is missing the test \+ call(B), but this must succeed anyway, since A fails. The last clause is missing the test valid(A) but this must also succeed. Thus, L.2 and S.1 have essentially the same logic and ask exactly the same questions if the normal cluase selection rule is used. The extra call to valid in L.2 does not result in more questions, since it is the same as a previous call which saves the information typed by the user. L. 2 is more similar to the specification but S.1 is more efficient.

§4 Rearranging and Eliminating Some Control

The S.1 and L.2 debuggers can be rewritten in a more logical and straightforward way. Instead of using an interface procedure then maintaining the invariant that each goal is valid and fails, this condition can simply be put into the specification (or definition) of miss. That is, miss should succeed only when (an instance of) the goal is valid and fails, and an uncovered atom should

be returned. The following debugger is an implementation of this idea. This debugger does not require an interface procedure: it will find a valid failing instance of the goal (if one exists) then proceed to find the bug.

```
        % debugger N.1
miss((A, B), R):-
        % valid(B),                    % like S.1/L.2 (see below)
        miss(A, R).
miss((A, B), R):-
        valid(A),
        miss(B, R).
miss(A, R):-
        user_pred(A),
        valid(A),
        \+ call(A),
        clause(A, B),
        miss(B, R).
miss(A, atom(A)):-
        user_pred(A),
        valid(A),                      % the goal (instance) must be valid
        \+ call(A),                    % and must fail
        \+ (clause(A, B), valid(B)).
```

It is simple to show that this program, including the commented out call to valid, is correct with respect to the specification above. Comparing it with L. 1, we see that the calls to valid in the first three clauses and the calls to \+ call(A) in the last two clauses are control information used to restrict the search. There are some repeated calls to valid (the call to valid in the first clause overlaps with the other calls to valid), but these will not result in extra questions for the user. In fact, if the first two clauses are reversed, N.1 will ask exactly the same questions (in the same order) as S.1 and L.2. Although this debugger is slightly more verbose than the others, it gives a very clear statement of the logic behind them. Another consequence of rearranging the control is that the calls to valid contain simple atoms rather than complete clause bodies. This makes it easier to modify the debugger so as to reduce the number of questions the user is asked.

By eliminating the call to valid in the first clause, repetition of calls to valid can be reduced and the number of questions can also often be reduced. The behaviour of the program is then as follows. Suppose miss is called with an atomic goal. The debugger searches for a valid instance of the atom which fails. If there are none, it fails. Otherwise, it searches for a matching clause, then searches for an instance of the clause body such that all subgoals to the left of a given atom are valid and that atom has a valid failing instance (it then recurses on that atom). If there are no such matching clause instances then the valid failing instance of the atomic goal is an uncovered atom.

This behaviour is essentially the same as Shapiro's improved debugger for

missing answers[23] reproduced below (with slight modifications so the last clause fails rather than give an error when the goal succeeds):

```
        % debugger S.2
miss(A, R):-
        clause(A, B),
        (miss1(B, R)->
            true
        ;
            R = atom(A)).

miss1((A, B), R):-
        valid(A),
        (   call(A),
            miss1(B, R)
        ;
            \+ call(A),
            miss(A, R)).
miss1(A, R):-
        user_pred(A),
        valid(A),
        \+ call(A),
        miss(A, R).
```

S.1 and L.2 find a valid instance of the whole clause body (by getting valid instances of individual subgoals) before searching for the failing subgoal. S.2 and N.1 check whether a valid instance of a subgoals fails as soon as the instance is found. Consider the following example:

```
p(X):- q(x), r(Y), s(Y).

% q(a).   missing
r(a).
s(a).

?- miss(p(a), R).
```

S.1 calls valid((q(a), r(Y), s(Y))). The user is first asked whether q(a) is valid then is asked whether the other subgoals have valid instances and what those instances are. N.1 and S.2 ask the first question then immediately check whether q(a) fails. It does, so the debuggers look for matching clauses and, finding none, return atom(q(a)). In this example, two yes/no questions and one instance question are avoided. If there were more subgoals to the right of q(X) this saving would be increased. If backtracking was needed to find a valid instance of the subgoals, the difference in the query complexity of S.1/L.2 and S.2/N.1 could be exponential.

In practice, S.2 asks fewer questions than S.1. However, the difference in

behaviour is essentially due to the elimination of some control information (the call to valid in the first clause of N.1), so it is worth considering if the search space of S.2 (and the number of questions asked) can ever be larger than that of S.1. Shapiro claims that S.2 never has worse query complexity than S.1, but the following example disproves this.

```
m:- m4, fail.
m:- m1.

m4:- m3.
m3:- m2.
m2:- m1.

% m1.     % missing

?- miss(m, R).
```

Since the body of the first clause of m has no valid instances, miss(m4, R) will never be called in S.1. In S.2 it will be called, which leads to the discovery of a (the) bug, but not as directly. If we consider the same program with the second clause of m is removed (so m should fail, and hence no answer is missing) S.2 still succeeds. Thus S.2 (and N.1) can return bugs even when given a goal which behaves correctly. It seems reasonable to consider this a feature.

Even when the search space of S.1 and S.2 are identical, S.2 may ask more questions because the order of searching is subtly different. To debug nondeterministic programs, it is important that valid returns all valid instances of the goal. S.1 gets a valid instance of the whole clause body then checks that all subgoals succeed. On backtracking another instance is generated. S.2 gets all valid instances of the first subgoal in a clause body and checks that each one succeeds before proceeding to the second subgoal. Consider the following example:

```
p(a).
p(b).
p(c).
% q(a).     % missing

m:- p(X), q(X).

?- miss(m, R).
```

If we assume p(a) is the first valid instance of p(X) returned in S.1 then S.1 will require fewer questions than S.2. Even if p(a) was the only valid instance, more questions would be asked in S.2, since a question must be asked to verify that there are no more valid instances. Questions of this kind are avoided in some of Shapiro's examples, because some predicates are assumed to be deterministic. This requires additional information from the programmer, which we do not consider in this paper.

There is a final subtle difference between S.1/L.2 and S.2/N.1 which concerns the behaviour of valid with atoms that have a non-ground valid instance. Althogh from a theoretical point of view we might say that valid should return all valid instances of an atom (or at least all ground valid instances), in practice it is important to represent an infinite set of ground instances by a single non-ground instance. Shapiro actually assumes there are always a finite number of (ground) valid instances, which makes a significant restriction on the class of programs which can be debugged. For example, if the debugger calls valid(length(Xs, 1)), the user shoud be able to give the single substitution Xs = [_] rather than the infinite number of ground lists of length one. Unfortunately, if this is done in S.2/N.1, the debuggers can fail to find a bug (and even return spurious bugs in the case of S.2). Consider the following example, where q(X) is true for all X, but the fact has been misstyped:

```
p:- q(X), r(X).

q(x).      % 'x' should be 'X'
r(a).

?- miss(p, R).
```

The debuggers call valid(q(X)), followed by \+ call(q(X)). If the first call succeeds without instantiating X, the second call fails, preventing the bug in q being found. In N.1 the top level call to miss fails and in S.2 it returns atom(p), which is incorrect. The basic problem is that we want to check if q(X) succeeds for all X, but \+ checks if it succeeds for some X. This is the basic problem with negation in Prolog. There have been many partial solution proposed, but none are completely satisfactory. Rather than fixing the code in these and subsequent debuggers in this paper, we will simple point out the problem and leave it to other implementors to add some hacks or perhaps even find an elegant solution. The problem is actually less severe in some of the later debuggers we discuss, because valid is not used as much.

§5 Using Call Instead of Valid

The call to valid in the second clause of N.1 is useful for instantiating variables in the goal so that calls to valid for subsequent subgoals have fewer solutions (that is, the search space is restricted). However, it does result in questions being asked. To avoid this, it is possible to use call to generate instances. If there is no bug in the first subgoal, then call will return the same answers as valid. If the call does not return all valid instances, some bugs in the rest of the goal may not be discovered. However, the first clause of the debugger will find a bug in the first subgoal. Although the search space is reduced and a call to miss may return fewer bugs than before, we can still guarantee that *a* bug will be returned if the goal has a valid failing instance. This satisfies the completeness definition of Lloyd and causes no problems if bugs are found and

removed one at a time.

Although the use of call reduces the number of questions to the user, it can markedly increase the cpu time used by the debugger. If the computation of the top level goal takes N resolution steps then the debugger can take $O(N^2)$ steps. Many debuggers avoid this problem by saving a representation of the original computation so that recomputation is avoided. The tradeoff between time and and space is analysed in Ref. 22). When a left to right computation rule is used, the search space of N.2 is a subset of the SLD tree of the top level goal (unlike the debuggers presented so far). Recomputation could therefore be avoided using this technique. However, coroutining can reduce the size of the search space without reducing the search space of the debugger, so some extra computation may have to be done in the debugger. We discuss the problems associated with coroutining later in this section. Restricting the search space to that of the original SLD tree is discussed in Section 8.

There are two other useful modification to N.1 which we now make. The first two clauses are swapped to improve the search order (we explain this in a moment) and the last two clauses are combined using an if-then-else construct (this makes the debugger more efficient).

```
    % debugger N.2
miss((A, B), R):-
    call(A),              % was valid(A) in N.1
    miss(B, R).
miss((A, B), R):-         % was first clause in N.1
    miss(A, R).
miss(A, R):-              % last two clauses in N.1 combined
    user_pred(A),
    valid(A),
    \+ call(A),
    (clause(A, B),
     miss(B, R1)
    ->
        R = R1
    ;
        R = atom(A)
    ).
```

In the following example, N.1 would ask the user for valid instances of q(Y), then ask if r(a) is valid. Using call instead of valid avoids questions concerning q (one yes/no and one instance question) if the debugger clauses are ordered as shown above.

```
p(X):- q(Y), r(X), s(Y).

q(a).
% r(a). missing
```

```
s(a).

?- miss(p(a), R).
```

N.2 calls subgoals in a clause from left to right until one fails. It then asks if that subgoal has any valid instances. If not, it backtracks to find different solutions to the calls of subgoals to the left or other valid instances of subgoals to the left which fail. If the first two caluses are reversed in N.2, the search order is reversed. The first failing subgoal is only examined after all previous subgoals are examined. This usually results in more questions being asked, particularly with deterministic computations (only one clause matches at each stage).

If the computation is deterministic and no procedures return incorrect answers, the first failing subgoal is always the subgoal with the bug. Thus N.2 can locate a bug in at most D yes/no questions and D instance questions, where D is the depth of the failed proof tree. S.1 requires C.D yes/no questions, where C is the number of subgoals per clause used in the incorrect failing branch of the proof tree, and up to C.D instance questions (depending on how many calls to valid are ground). The best case for N.1 is the same as N.2, and occurs when the first subgoal contains the bug at each step. The worst case for N.1 occurs when the last subgoal contains the bug at each step. C.D instance questions and (2C−1).D yes/no questions are asked (if the computation is known to be deterministic in advance, this can be reduced to C.D, by avoiding "any more instances" questions).

In Prolog systems which allow coroutines, the use of call can cause complications. Generally, such systems have a mechanism for delaying the evaluation of a call until certain variables in the call are instantiated. Thus call(A), in the first clause of N.2 can terminate with some subgoals still delayed, rather than A being executed completely (call(A) is said to *flounder*). Any variables bindings that would normally have been computed by the delayed calls are not passed on to B. In the worst case, calling A binds no variables in B, effectively eliminating the control component of the first clause and hence significantly increasing the search space of the debugger. Another practical problem is that the delayed subgoals of A can wake up later and interfere with the execution of the debugger.

The most practical solution to these problems is to be more flexible about ordering conjuncts. Rather than always considering the first conjunct first, it is better to try to find a conjunct which executes completely and, effectively, reorder the conjunction so this call appears first. In practice, it is rare for all the conjuncts to flounder and in the uncommon case when it occurs, we can accept a larger search space and either not call any subgoals or use hacks such as copying terms to avoid delayed calls causing problems. The first two clauses of N.2 can be combined in a way similar to what follows (again, we refrain from complicating this and subsequent debuggers by including the details of desirable hacks):

```
miss((A0, B0), R):-
    pick_nondelay_subgoal((A0, B0), A, B),
    (   call(A),
        miss(B, R)
    ;
        miss(A, R)
    ).
```

By combining the last two clauses of N.1 into a single if-then-else construct in N.2, the (repeated) work of the last clause is avoided. Unfortunately, from a theoretical viewpoint, this makes the debugger incomplete:

```
m:- m1, fail.
% m.    % fact missing

m1:- m.

?- miss(m, R).
```

The computation rule could be such that m and m1 both finitely fail. The last clause for N.1 could then succeed, returning the uncovered atom m. The complete SLD tree for the miss goal in N.1 has an infinite branch because of the third clause, but in theory, this will not prevent the answer from being returned (we could use a fair search for example). Because the last two clauses are combined in N.2, the recursive call to miss (in the condition of the if-then-else) neither succeeds or finitely fails, so there is no success branch on the SLDNF tree. Both N.2 and S.2 must loop rather than returning the bug. If we use standard Prolog to run the debuggers, even N.1 loops, because of the unfair depth first search which Prolog uses. Hence, in practice, introducing the if-then-else does not cause and more infinite loops.

Avoiding this kind of infinite loop seems quite difficult without calling valid with a whole clause body (and thus asking more questions). In a standard Prolog system, with a left to right computation rule, the problem is disguised because the debugger does not loop unless the top level goal also loops. With a different computation rule however, the debuggers can loop even when the goal finitely fails. The difficulty lies in the fact that the debugger splits a conjunction into parts and searches for bugs in each part independently (perhaps passing some variable bindings from left to right). When the conjunction is called however, coroutining make the interaction of the conjuncts much more complex. The operation of the debugger is unable to mirror the execution of the goal as can be done with a left to right computation rule. Examining the conjuncts in a more flexible order, using pick_nondelay_subgoal as suggested above, is an effective way of avoiding (most) infinite loops of this kind in practice.

§6 Testing Validity of Answers from Call

It is quite common for failures to be caused by subcomputations returning incorrect answers. Checking that the answers returned from call are correct has several advantages. First, if we know that a call has returned an incorrect answer, we can use wrong to find a bug. This is generally better than using miss, since it requires only simple yes/no questions, and is more likely to find an error in a single clause, rather than a whole procedure. Second, if we know that the answer obtained by calling a subgoal is correct, then a later call to valid does not need to ask the user so many questions (especially instance questions). Third, asking the validity of answers from call immediately can prevent incorrect bindings to variables showing up in other questions. These questions can be confusing, since they can ask the validity of subgoals which should never occur. In Pereira's debuggers, if such calls are declared inadimissible then the wrong answer diagnosis algorithm is invoked. However, wrong answers do not necessarily lead to inadmissible calls, so they are not always detected.

Checking that answers are valid generally involves asking questions. In programs which do not return incorrect answers, more yes/no questions will be asked. A reasonable compromise exists which delays (and often avoids) asking the extra questions. It has the first two advantages mentioned but not the third. The compromise is to ask if a result of a call is valid if and when the subsequent call to miss fails. This is implemented in the program below (the previous comments concerning call(A) floundering apply here also). If the order of disjuncts in the first clause is changed then the questions are asked as soon as call returns, gaining the third advantage but, we believe, asking more questions on average. Immediate detection of wrong answers and efficient diagnosis of failing deterministic code (as discussed in the previous section) seem to be conflicting requirements.

The algorithm of Refs. 8) and 13) checks the validity of all answers to a call before checking if it misses answers. The order of examining the different atom instances in a failed conjunction is not specified, but if our ordering is used the extra questions are delayed even longer than in our algorithm. The validity of answers is checked only after all answers have been considered. This strategy can be implemented in the code below by putting call(A) in both disjuncts of the first clause instead of before the whole disjunction.

```
        % debugger N.3
miss((A, B), R):-
        call(A),
        (
            miss(B, R)
        ;               % could reverse order of;
            unsatisfiable(A),
```

```
            wrong(A, R)
        ).
miss((A, B), R):-
        miss(A, R).
miss(A, R):-
        user_pred(A),
        valid(A),
        \+ call(A),
        ( clause(A, B),
          miss(B, R1)
        ->
            R = R1
        ;
            R = atom(A)
        ).
```

The order of the first two clauses is important for two reasons. First, the order of the search is superior (as with N.2). Second, it reduces the number of instance questions (the second advantage mentioned above). Suppose miss is called with a conjunction and that the first subgoal is missing a solution but the rest of the conjunction is free of bugs. The first subgoal will be called and for each solution unsatisfiable will (eventually) be called. This will result in one yes/no question about the validity of each solution. When all solutions to the subgoal are exhausted, the debugger backtracks to the second clause of miss and valid is called with the buggy subgoal. Since all instances of the subgoal which succeed are known to the system, it can just ask if there are any more valid instances. The first instance typed by the user will fail so miss can proceed recursively. If the clause order were reversed, or one of the previous debuggers were used, the user may have to type many valid instances of the subgoal, all of which succeed. Thus the number of instance questions asked by this debugger is one per atom checked, compared with an unbounded number for the other debuggers presented so far.

The following example (from Lloyd) is useful for comparing S.1/L.2, S.2/N.1, N.2 and N.3.

```
        % "quick" sort
qsort([], []).
qsort(A.B, C.D):-
        part(A, B, L1, L2),
        qsort(L2, S2),
        qsort(L1, S1),
        app(S1, A.S2, C.D).

        % partition
part(A, [], [], []).
```

```
part(A, B.C, B.D, E):-
                                    A>= B,
                                    part(A, C, D, E).
part(A, B.C, D, E):-      % should be part(A, B.C, D, B.E)
    A < B,
    part(A, C, D, E).

    % append
app([], A, A).
app(A.B, C, A.D):-
    app(B, C, D).

?- miss(qsort([3, 1, 2], [1, 2, 3]), R).
```

The following shows the questions asked by S.1, and user responses. The error returned is atom(part(1, [2], [], [2])). There are nine yes/no questions and five instance questions (note that each line showing an instance question also has an associated yes/no question). valid(qsort([], A)) is called twice but questioning the user is avoided on the second occasion, since the information is stored.

```
qsort([3, 1, 2], [1, 2, 3])          y
part(3, [1, 2], A, B)                y;    A = [1, 2], B = [].
qsort([], A)                         y;    A = [].
qsort([1, 2], A)                     y;    A = [1, 2].
app([1, 2], [3], [1, 2, 3])          y
part(1, [2], A, B)                   y;    A = [], B = [2].
qsort([2], A)                        y;    A = [2].
app([], [1, 2], [1, 2])              y
part(1, [], [], [2])                 n
```

The following shows the questions asked by N.1, and user responses. The error returned is the same. There are nine yes/no questions and four instance questions. All questions about app and some questions about qsort are avoided, but there are three extra questions verifying there are no more valid instances of a goal (these are put on separate lines).

```
qsort([3, 1, 2], [1, 2, 3])          y
part(3, [1, 2], A, B)                y;    A = [1, 2], B = [].
part(3, [1, 2], A, B)                no more
qsort([], A)                         y;    A = [].
qsort([], A)                         no more
qsort([1, 2], A)                     y;    A = [1, 2].
part(1, [2], A, B)                   y;    A = [], B = [2].
part(1, [], [], [2])                 n
part(1, [2], A, B)                   no more
```

The following shows the questions asked by N.2, and user responses. The error returned is the same. There are eight yes/no questions and three instance

questions. Even though the example is not well suited to the search strategy of N.2, checking the first failing subgoal first still reduces the number and complexity of questions asked.

```
qsort([3, 1, 2], [1, 2, 3])          y
app([1], [3], [1, 2, 3])             n
qsort([1, 2], A)                     y;    A = [1, 2].
app([], [1], [1, 2])                 n
qsort([], A)                         y;    A = [].
qsort([], A)                         no more
part(1, [2], A, B)                   y;    A = [], B = [2].
part(1, [], [], [2])                 n
```

The following shows the questions asked by N.3, and user responses. The error returned is an incorrect clause instance:

```
part(1, [2], [], []):-
        1 < 2,
        part(1, [], [], []).
```

There are five yes/no questions and no instance questions. When it is discovered that qsort([1, 2], [1]) is not valid, wrong is called (which locates the bug with only two further yes/no questions).

```
qsort([3, 1, 2], [1, 2, 3])          y
app([1], [3], [1, 2, 3])             n
qsort([1, 2], [1])                   n
part(1, [2], [], [])                 n
part(1, [], [], [])                  y
```

§7 Using Satisfiable Instead of Valid

If we know that a subgoal is satisfiable but fails, this is sufficient information to tell us it contains a bug. We do not need to know a particular valid instance of the subgoal, and asking the user for such an instance should clearly be avoided if possible. Using this idea, we can generalise the notion of an uncovered atom:

```
        % debugger N.0.1 (part 2; part 1 is the same as in N.0)
bug(atom(A)):-
        satisfiable(A),
        all [H, B] not (subsumes(A, H), is_clause(H, B), valid(B)).
```

An atom is uncovered if it is satisfiable and no clause instance whose head is an instance of the atom has a valid body. Note that this definition is equivalent to N.0 when A is ground.

Unfortunately, we cannot simply take any of the previous debuggers and replace valid by satisfiable everywhere and expect them to work. We need some

way of getting from an incorrectly failing goal to an incorrectly failing subgoal in the body of a matching clause. The previous debuggers use valid to get instances of subgoals in the bodies of matching clauses. It is pointless asking an oracle for satisfiable instances of goals: the most general answer is the goal itself. Some other method(s) must be used to find instances of subgoals which incorrectly fail.

The first method, used implicitly in all the debuggers, is the unification of the goal with the head of the clause. If the first subgoal contains the bug, this often instantiates it sufficiently. Another method is to use call, as in N.2 and N. 3. This is sufficient if the failure at the top level is caused by the failure of a single subgoal, rather than one of several answers being missed. This is an assumption of Refs. 2) and 7) which, as we mentioned previously, can cause incompleteness and even incorrectness.

The algorithm we suggest is to first use call to generate instances, but if this fails to find an incorrectly failing subgoal, to resort to using valid as before. This algorithm will find a bug whenever N.3 finds a bug, but will only ask the user instance questions when yes/no questions have failed to find the bug. In particular, if the computation is deterministic and no subgoal returns an incorrect answer, no instance questions will be asked. However, if yes/no questions are not sufficient to find the bug, N.4 may ask more yes/no questions than N.3. A proof of completeness of a variant of N.4 is sketched in Ref. 27).

```
    % debugger N.4
miss(A, R):-
    miss_c(A, R).
miss((A, B), R):-
    miss_v((A, B), R).

    % miss for atoms
miss_atom(A, R):-
    user_pred(A),
    \+ call(A),              % valid(A),
    satisfiable(A),          % \+ call(A) in N.3
    ( clause(A, B),
      miss(B, RI)
    ->
        R = RI
    ;
        R = atom(A)
    ).

    % uses call to get instances of subgoals which incorrectly
    % fail (incomplete for non-atomic goals)
miss_c((A, B), R):-          % same as N.3
    call(A),
```

```
        (
            miss_c(B, R)
        ;
            unsatisfiable(A),
            wrong(A, R)
        ).
miss_c((A, B), R):-
        miss_atom(A, R).
miss_c(A, R):-
        miss_atom(A, R).

        % uses valid to get instances of subgoals which incorrectly
        % fail (complete)
miss_v((A, B), R):-
        valid(A),
        (   miss_atom(A, R)
        ;
            miss_v(B, R)
        ).
miss_v(A, R):-              % valid is not needed for atoms.
        miss_atom(A, R).
```

The key part of the program, miss_atom, is the same as before except that is uses satisfiable instead of valid. Therefore it never asks instance questions. miss calls miss_c, which uses call to generate instances of subgoals and, if that fails, it calls miss_v, which uses valid. miss_v is not used for atomic goals because miss_c will not fail in this case, assuming the goal at the top level is satisfiable but fails.

As with N.2 and N.3, miss_c should be modified to handle the case when call(A) flounders. The best solution is to find a non-floundering subgoal and call that instead of A. Even if a more restrictive solution is used, such as failing if call(A) flounders, bugs can still be found by miss_v (though this asks more questions). Since miss_v is only tried when miss_c has failed to find a bug, it may be worthwhile making miss_v more robust, at the expense of asking more questions. For example, it could be made more like S.1/L.2, thus avoiding some problems with nonground valid atoms etc. Hopefully, the extra questions would rarely be asked, because most bugs would be found by miss_c.

We will use the following program (from Ref. 23)) as our next example:

```
isort([X|Xs], Ys):- isort(Xs, Zs), insert(X, Zs, Ys).
isort([], []).

insert(X, [Y|Ys], [Y|Zs]):- X < Y, insert(X, Ys, Zs).
insert(X, [Y|Ys], [X, Y|Ys]):- X =< Y.
% insert(X, [], [X]).          % missing

?- miss(isort([3, 2, 1], X), R).
```

S.1 (L.2) requires seven yes/no questions and four instance questions to find the uncovered atom insert(1, [], [1]). S.2 (N.1) requires the following six yes/no questions (including one "any more valid instances" question) and four instance questions to find the same uncovered atom:

```
isort([3, 2, 1], A)          y;    A = [1, 2, 3].
isort([2, 1], A)             y;    A = [1, 2].
isort([1], A)                y;    A = [1].
isort([], A)                 y;    A = [].
isort([], A)                 no more
insert(1, [], [1])           y
```

N.2 and N.3 avoid the three questions associated with isort([], A), since that call succeeds with the (only) correct answer. This results in four yes/no questions and three instance questions. N.4 avoids all instance questions and needs only four yes/no questions also. The uncovered atom returned, insert(1, [], A), contains a variable, meaning that the atom is satisfiable but the bodies of all matching clauses are unsatisfiable.

```
isort([3, 2, 1], A)          y
isort([2, 1], A)             y
isort([1], A)                y
insert(1, [], A)             y
```

§8 Avoiding Instance Questions Entirely

All the debuggers we have examined so far rely on the user providing valid instances of goals in some circumstances. We now examine two ways we may be able to eliminate instance questions entirely by making the computation of the debugger follow the search of the original computation more closely. Failed derivations of the original computation are used to limit the search space of the debugger. Both the ideas need considerably more research before they can be used as the basis of practical debuggers. The first technique has been used previously in some debuggers for conventional Prolog, with a left to right computation rule. We discuss the possibility of adapting it to Prolog systems which have coroutines. The second technique is new, and asks even simpler questions than the first technique. It also overcomes some of the reasons for incompleteness of the debuggers we have discussed, though it is also incomplete.

8.1 Using Incompleteness Questions

As mentioned previously, the Pereira-stye debuggers[3,4,8,13,14,20,21] do not ask instance questions. Instead, they ask if calls are *incomplete*. That is, not all valid instances are returned by the program. This implies that some valid instance of the call fails. The debuggers find all solutions to the call then print the call and the solutions, and ask the user if any valid solutions are missing.

The idea of incomplete calls, or *not completely covered atoms*,[3] can be used to further generalise the definition of bugs:

```
        % debugger N.0.2 (part 2; part I is the same as in N.0)
bug(atom(A)):-
        incomplete(A),
        all [H, B] not (subsumes(A, H), is_clause(H, B), valid(B)).
```

Note that this is equivalent to N.0.1 in the case where A has at most one valid instance. The following debugger, written in the style of N.2, uses this more general definition. It implements a similar algorithm to those described in the references above, without enhancements associated with avoiding recomputation, inadmissible calls, integration with wrong answer diagnosis and intelligent search strategies. Intelligent search strategies could be used with the logic of this program if alternative evaluation strategies are used, such as a mixture of top down and bottom up evaluation and an intelligent clause election rule.

```
        % debugger N.5
miss((A, B), R):-
        call(A),
        miss(B, R).
miss((A, B), R):-
        miss(A, R).
miss(A, R):-
        user_pred(A),
        incomplete(A), % valid(A), \+ call(A), in N.2
        ( clause(A, Y),
          miss(Y, RI)
        ->
            R = RI
        ;
            R = atom(A)
        ).
```

In N.2, the last clause uses valid and call to find a valid failing instance of the goal (if one exists). This requires instance questions. In contrast, N.5 uses incomplete to check if a valid failing instance of the goal exists. However, the actual instance is not required, so no instance questions are asked. For Prolog systems with left to right computation rules, this debugger is very effective. The algorithms presented in Refs. 13) and 4) are proved sound and complete. The if-then-else in the last clause does not cause incompleteness because the top level goal is assumed to have a finite SLD tree using a left to right computation rule.

Unfortunately, the basic idea of incompleteness questions runs into difficulties in systems which allow coroutines. The problem is that incomplete can be called with an atom whose execution flounders. It would be possible for

incomplete to use a different computation rule to find the set of successful instances of the goal without floundering. However, this is likely to cause infinite loops, since one of the main reasons for delaying calls is that they have an infinite number of solutions.[15] The alternative is to only partially execute the goal, but this can cause incompleteness of the debugger since the fact that some answers are missed may not be detected by incomplete.

We propose two ways to deal with this incompleteness. The first is to use a debugger in the style of N.4, which tries incomplete methods of bug location which ask simple questions, then, if these fail, tries another method which is (more) complete. Ideally, we should try to locate the bug using satisfiable first, then incomplete, then valid. Reordering the examination of conjuncts, using pick_nondelay_subgoal, should make this method quite effective, though occasionally instance questions would still be asked. The second method we propose is more complex, but avoids all instance questions. First, we note that the substitution returned by a floundered goal can be considered a *partial* answer substitution (as opposed to the normal *total* answer substitutions returned by non-floundering goals). Similarly, partial answer substitutions can be produced by a goal whose execution is interrupted by previously delayed calls being woken. The set of partial answers to a goal includes or subsumes all the total answers found by a computation rule which executes the goal completely without interruption. Therefore, if the set of partial answers to a goal is incomplete then the goal is incomplete. The converse is not true however, and this is the source of incompleteness of the debugger.

The new algorithm which avoids this source of incompleteness is as follows. We assume the goal in the call to miss is satisfiable but the set of partial answers is incomplete. Atomic clause bodies can be treated as before (since they must also be incomplete). For simplicity, we just describe the conjunctive case when there are two subgoals, p(X) and q(X). First, we find the set of partial answers to p(X) and check if it is incomplete (if so, we recursively search for the bug in p(X)). Next, for each partial answer substitution θ_i^1, for p(X), we find the set of partial answers for $q(X)\theta_i^1$ (stopping forward execution whenever a delayed call from p would be woken) and check if it is incomplete. So far, this is very similar to N.5. However, if no incompleteness is found, we continue by considering each partial answer substituition θ_j^2, for $q(X)\theta_i^1$, and finding the set of partial answers for $p(X)\theta_i^1\theta_j^2$ (and checking if it is incomplete). This conrresponds to the coroutining execution where part of p is executed, followed by part of q, followed by part of p. We continue to compose partial answer substitutions from p(X) and q(X) until one of the sets is incomplete or total answer substitutions for both calls are found (this corresponds to a whole branch of the SLD tree).

Each derivation of p(X), q(X) can be written as a sequence $A_1B_1A_2B_2 \cdots$, where A_i (B_i) is a sequence of calls derived from p(X) (q(X)). The algorithm searches for a prefix for which the last element exhibits incompleteness. Since each derivation is finite and there are a finite number of derivations, the

algorithm must terminate. For simplicity, we have described an algorithm which considers shortest prefixes first. It is better to reverse the search order, as was done in N.2. With left to right execution, p and q are considered at most once (the behaviour is identical to N.5 if the better the search order is used). As an example, consider the following program. It contains two NU-Prolog *when declarations*, which make calls to p and q delay until their first argument is a non-variable.

```
?- p(I, J) when I.          % only call when I instantiated
% p(0, 0).                  % fact missing
p(s(I), s(J)):- p(J, I).       % note args swapped

?- q(I, J) when I.          % only call when I instantiated
q(0, 0).
q(s(I), s(J)):- q(J, I).       % note args swapped

?- miss((p(s(s(0)), X), q(X, s(s(0)))), R).
```

The computation of the goal to be debugged is as follows. Initially, p is called, binding X to s(X1). The recursive call to p then delays. Next, q binds X1 to s(X2) and delays in the same way. Finally, p is called and fails because the base case is missing. The following table explains how the debugging algorithm works in this example. It gives the calls examined and the corresponding set of partial answers.

```
p(s(s(0)), X)          {p(s(s(0)), s(X1))}
q(s(X1), s(s(0)))      {q(s(s(X2)), s(s(0)))}
p(s(s(0)), s(s(X2)))   {}
```

The first two partial answer sets are complete, but the last one is incomplete (the debugger considers this first if the better search order is used). We can therefore recursively look for a bug with the atom p(s(s(0)), s(s(X2))). Since the body of the clause for p has only one atom, the debugger can then simply identify the uncovered atom p(0, X2). This example is simplified because no set of partial answers has more than one atom. This is because the finitely failed SLD tree has only a single branch. In general, several combinations of alternative substitutions must be examined, corresponding to different failed SLD derivations.

8.2 Using Satisfiability Questions Only

The debuggers which use call presented so far are asymmetric and have some problems when coroutining is used heavily (they have to resort to using valid more). A purely declarative debugger should be symmetric with respect to the order of subgoals in the goal being debugged. For example, the first two clauses of N.2 should be:

```
miss((A, B), R):-
    call(A),
    miss(B, R).
miss((A, B), R):-
    call(B),      % added for symmetry
    miss(A, R).
```

Although B is sometimes able to generate useful instances of A, improving the performance of the debugger, calling B before A often causes infinite loops. In practice, logic programs are not purely declarative. They have a declarative component but are written so as to be executed using a computation rule which (by default) selects subgoal from left to right. Thus, N.2 is a practical debugger but adding the extra call makes it impractical.

However, it is possible to generate instances of A by partially executing B (and obtaining partial answer substitutions) and vice versa. The algorithm proposed in the previous section used prefixes of the computation trace to obtain partial answer substitutions. Only the longest prefix can lead to finite failure, so incompleteness questions are generally asked more than satisfiability questions. Also, if the interleaving of the execution is very fine, which is common in many coroutining systems, many questions may be asked. An alternative method of obtaining partial answer substitutions is to use the sequences $A_1A_2A_3 \cdots$ and $B_1B_2B_3 \cdots$ to find instances of B and A, respectively. The number of questions is then not affected by the degree of interleaving. Furthermore, if the whole conjunction finitely fails then each instance of A and B generated in this way must also finitely fail, so satisfiability questions are sufficient. A disadvantage is that calls in these sequences are generally less instantiated than in the original execution trace, and hence may unify with more clause heads. Even a deterministic execution of A, B may result in many partial answer substitutions when considering A and B separately. Consider the following example:

```
?- p(I) when I.              % only call when I instantiated
p(0).
p(I):- p(J), I = s(J).

q(a).
q(f(A)):- q(A).
% q(0).      % missing

m:- p(X), q(X).

?- miss(m, R).
```

Using the NU-Prolog the computation rule, the computation proceeds as follows. First m is called, then q(X), then p(I), then a = s(J), which fails. On backtracking, q(X) is retried and matches with the second clause, then p(f(A)) is called, then f(A) = s(J) fails. On backtracking, there are no further matching

clauses to any call so the goal finitely fails.

The debugger can first verify that the goal is satisfiable but fails, so a bug must be manifest in the body of the clause for m. q(X) can then be used to generate instances of the rest of the body. Rather than execute q(X) completely, it is only executed as much as it was executed in the failing computation outlined above. That is, the initial call is done (which matches with two clauses), but the recursive call is ignored. This leads to the two instances: p(1) and p(f(A)). Note that both these instances finitely fail. Since both p(1) and p(f(A)) are unsatisfiable (that is, they fail correctly), the bug is not in p.

Next, p(X) can be used to generate instances of q(X). In the finitely failed computation, one call to p was done and, when the second clause of p was used, = was called. Thus, the (finitely failing) instances q(0) and q(s(J)) are generated. Note that in the original computation p was deterministic and X was never bound to 0 or s(J), due to the bindings made by q. q(s(J)) is unsatisfiable but q(0) is satisfiable. A satisfiable failing atom has been discovered so search for the bug progresses recursively with q(0).

This algorithm can be applied to arbitrary coroutining in pure Prolog, and no instance (or even incompleteness) questions are required. This example also illustrates a problem with the debuggers which use valid to generate instances of atoms, despite proofs of completeness in theory. Although the body of the clause for m has only one valid instance, each subgoal has an infinite number of valid instances (and an infinite nunber which succeed) and this set cannot be represented by a finite set of atoms even if we allow variables. Even with a left to right computation rule, subgoals can have an infinite number of valid instances (though only a finite number can succeed, otherwise the top level goal would loop rather than fail).

Thus, the user can be asked to give a valid instance of an atom which has an infinite number of valid instances. Relying on the user to eventually type the lucky instance which shows up the bug is obviously problematic. Generating all possible instances in some fair manner is also impractical. One way to avoid the problem is to use a debugger like S.1 or L.2 and ask the user give a valid instance of the whole clause body. This is obviously difficult for the user, especially when there are long clauses. The debugger given in Ref. 24) actually requires the user to provide instances of whole clauses (it is not intended to be a practical debugger, however). If valid can return an unbounded number of solutions then the execution of the debugger must use a fair search strategy if it is to be complete.

Although our algorithm which only asks satisfiability questions overcomes some of the completeness problems introduced by call looping, incompleteness of valid, coroutines and the unfair search strategy of Prolog, it is not complete itself. The reason is that there may be more than one bug manifest in a single clause. In the following example, p cannot generate the buggy instance of q because p has a bug, and vice versa.

```
% p(a).          % missing
p(b).

% q(a).          % missing
q(c).

m:- p(X), q(X).

?- miss(m, R).
```

It seems that extending this technique for debugging programs with negation and other system predicates would introduce further incompleteness. It would be possible to use this method to generate instances instead of call, but still default to using valid if no bugs are found. Alternatively, incompleteness questions could be asked regarding the partial executions $A_1A_2A_3 \cdots$ and $B_1B_2B_3 \cdots$.

§9 Conclusions

Declarative debugging, especially for systems which support flexible control, is important for fully exploiting the benefits of logic programming. Theoretical work has studied the soundness and completeness of declarative debuggers which allow coroutines, but the practicality of these debuggers has received little attention. We have presented successive modifications to previously published debuggers which, in general, significantly reduce the number and complexity of oracle queries. We have also dicussed examples where various debuggers have an infinite search space, due to infinite recursion or a call to valid having an infinite number of solutions. This causes incompleteness in practice.

The reason why the search space can be infinite is that the operation of the debugger does not mirror the (finite) computation of the goal which exhibits the error symptom. If a left to right computation rule is used, these problems are more easily avoided. We have outlined two algorithms which have a search space limited by the size of the SLD tree of the goal, and which use simpler queries. One of these is an extension of an algorithm which has been used successfully for left to right computation rules. However, much more work is needed to enhance these algorithms, so that fewer questions are asked, establish their practicality and prove their correctness.

Acknowledgements

I would like to thank Ann Nicholson for numerous comments on a draft of this paper. John Lloyd originally interested me in declarative debugging and in the course of this research I have had useful discussions with S. Y. Yan. This research was supported by the Australian Commonwealth Department of Science and ARGS (now ARC).

References

1) Clark, K. L., "Negation as Failure," in *Logic and Data Bases* (H. Gallaire and J. Minker, eds.), Plenum Press, pp. 293-322, 1978.
2) Dershowitz, N. and Lee, Y., "Deductive Debugging," *Proceedings of the 4th IEEE Symposium on Logic Programming*, San Francisco, California, pp. 298-306, August 1987.
3) Drabent, W., Nadjm-Tehrani, S. and Maluszynski, J., "The Use of Assertions in Algorithmic Debugging," *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, pp. 573-581, December 1988.
4) Drabent, W., Nadjm-Tehrani, S. and Maluszynski, J., "Algorithmic Debugging with Assertions," in *Meta-Programming in Logic Programming* (J. W. Lloyd, eds.), MIT Press, 1989.
5) Edman, A. and Tärnlund, S. -Å., "Mechanization of an Oracle in a Debugging System," *Proceedings of 8th IJCAI*, Karlsruhe, Germany, pp. 553-555, August 1983.
6) Ferrand, G., "Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method," *Journal of Logic Programming 4, 3*, pp. 177-198, September 1987.
7) Huntbach, M. M., "Algorithmic Parlog Debugging," *Proceedings of the 4th IEEE Symposium on Logic Programming*, San Francisco, California, pp. 288-297, August 1987.
8) Kanamori, T., Kawamura, T., Maeji, M. and Horiuchi, K., "Logic Program Diagnosis from Specification," *ICOT Technical Report, TR-447*, Institute for New Generation Computer Technology, Tokyo, Japan, March 1989.
9) Kowalski, R. A., *Logic for Problem Solving*, North Holland, New York, 1980.
10) Lichtrenstein, Y., "Algorithmic Debugging of Flat Concurrent Prolog," *M. Sc. thesis*, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel, August 1987.
11) Lloyd, J. and Takeuchi, A., "A Framework for Debugging GHC," *ICOT Technical Report, TR-186*, Institute for New Generation Computer Technology, Tokyo, Japan, 1986.
12) Lloyd, J. W., "Declarative Error Diagnosis," *New Generation Computing, 5, 2*, pp. 133-154, 1987.
13) Maeji, M. and Kanamori, T., "Top-Down Zooming Diagnosis of Logic Programs," *ICOT Technical Report, TR-290*, Institute for New Generation Computer Technology, Tokyo, Japan, August 1987.
14) Nadjm-Tehrani, S., "Contributions to the Declarative Approach to Debugging Prolog Programs," *thesis No. 187*, Department of Computer and Information Sciences, University of Linköping, Linköping, Sweden, 1989.
15) Naish, L., "Automating Control of Logic Programs," *Journal of Logic Programming, 2, 3*, pp. 167-183, October 1985.
16) Naish, L., "Negation and Quantifiers in NU-Prolog," *Proceedings of the 3rd International Conference on Logic Programming*, Imperial College of Science and Technology, London, England, pp. 624-634, July 1986. published as *Lecture Notes in Computer Science 225* by Springer-Verlag.
17) Naish, L., Dart, P. W. and Zobel, J., "The NU-Prolog Debugging Environment," *Proceedings of the 6th International Conference on Logic Programming*, Lisboa, Portugal, June 1989.
18) Naish, L., "Types and the Intended Meaning of Logic Programs," *Technical Report, 90/4*, Department of Computer Science, University of Melbourne, Melbourne,

Australia, February 1990. to appear in *Types in Logic Programming*, MIT press.

19) Nicholson, A. E., "Declarative Debugging of the Parallel Logic Programming Language GHC," *Proceedings of the 11th Australian Computer Science Conference*, Brisbane, Australia, pp. 225-236, February 1988.

20) Pereira, L. M., "Rational Debugging in Logic Programming," *Proceedings of the 3rd International Conference in Logic Programming*, London, England, pp. 203-210, July 1986. published as *Lecture Notes in Computer Science 225* by Springer-Verlag.

21) Pereira, L. M. and Calejo, M., "A Framework for Prolog Debugging," *Proceedings of the 5th International Conference/Symposium on Logic Programming*, Seattle, Washington, pp. 481-495, August 1988.

22) Plaisted, D. A., "An Efficient Bug Location Algorithm," *Proceedings of the 2nd International Logic Programming Conference*, Uppsala, Sweden, pp. 151-157, July 1984.

23) Shapiro, E. Y., *Algorithmic Program Debugging*, MIT Press, Cambridge, Massachusetts, 1983.

24) Sterling, L. and Shapiro, E., *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, Massachusetts, 1986.

25) Takeuchi, A., "Algorithmic Debugging of GHC Programs and Its Implementation in GHC," *ICOT Technical Report, TR-185*, Institute for New Generation Computer Technology, Tokyo, Japan, 1986.

26) Thom, J. and Zobel, J. eds., "NU-Prolog Reference Manual, Version 1.0," *Technical Report, 86/10*, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.

27) Yan S. Y. and Naish, L., "Completeness of an Improved Declarative Debugger," in *Advances in Computing and Information*, Canadian Scholar's Press, pp. 132-135, May 1990. to appear in *Applied Mathematical Letters*.