Debugging Logic Programs Using Specifications

Yuh-jeng Lee[1]* and Nachum Dershowitz[2]**

[1] Computer Science Department, Naval Postgraduate School
Monterey, CA 93943, U.S.A.
[2] Department of Computer Science, University of Illinois
1304 W. Springfield Ave., Urbana, IL 61801, U.S.A.

Abstract. We show how executable specifications may be used to generate test cases for bug discovery, locate bugs when test data cause a program to fail, and guide deductive and inductive bug correction.

1 Introduction

Logic programming has gained in popularity in recent years. This style of programming, using Horn clauses to express procedural information, allows one to reason easily about the effects of executing program statements.

We present a methodology for reasoning about the relationship between logic programs and their specifications, to help debug erroneous programs. To allow for debugging, the specifications must describe the relationships between the values of input and output variables. Since Horn clauses are a powerful subset of first-order logic, a program's specifications can oftentimes be written in Prolog itself and executed by the Prolog interpreter or compiler directly. (Whereas specifications are intended to emphasize clarity and simplicity, in implementing programs, efficiency is a major consideration.)

The debugger follows the pattern of Shapiro's ([12]). We focus on the use of executable specifications to generate test cases for bug discovery, locate bugs when test data cause a program to fail, and guide deductive and inductive bug correction ([3]). We also ask that specifications provide information on the well-founded ordering of input arguments for recursive procedures. (A well-founded ordering $\succ$ is a transitive and irreflexive binary relation on elements of a set S such that the S has no infinite descending sequences.) The ordering specifies, for each recursive call, which arguments should be decreasing. This is used for detecting looping.

Programs, for our purposes, are presumed to obey their Horn clause declarative semantics, i.e., extra-logical features, such as cuts, clause order, and subgoal order, may affect efficiency and termination, but not correctness. We also presume that specifications faithfully reflect the intended requirements of a program. To obtain the desired effect, it is sometimes necessary to use impure features, i.e., non-logical control structures, of Prolog.

* Research supported in part by the U. S. Naval Postgraduate School and the U. S. Army Artificial Intelligence Center.

** Research supported in part by a Meyerhoff Visiting Professorship at the Weizmann Institute of Science and by the U. S. National Science Foundation under Grants CCR-90-07195 and CCR-90-24271.

More expressive languages, like EQLOG ([5]) or RITE ([6]), which use equational Horn clauses, may be even more suited to the kind of program manipulations advocated here.

Other work on declarative debugging includes [4, 10, 7, 2, 11, 1, 8, 9].

2 Using Executable Specifications

Executable specifications of a program not only serve to check the output, but also generate useful test cases for that program, provided that axioms for primitive predicates are supplied. The information contained in specifications regarding the expected output behavior is indispensable for checking the correctness of the results of program execution, while test cases help reveal instances of incorrect output.

We assume that the properties of each procedure in the program have been described in the program's specifications, which detail the relationships between program variables. In other words, they define all legal input/output pairs for each procedure. To check for termination, we also need a well-founded ordering under which successive input values to recursive procedures are intended to form a descending sequence. Any unspecified procedures are presumed correct and terminating.

A program is *(partially) correct* with respect to its specification if each clause of the program can be proved from the specifications and given domain facts by first-order reasoning. If we can find an atomic formula that follows logically from the program (and domain facts), but which cannot be proved from the specifications (and domain facts), then we have shown that the program is *incorrect* with respect to the given specifications.

A program is *complete* with respect to its specifications if each clause of the specification follows from the program and domain facts. If we can find an atomic formula that follows from the specifications (and domain facts) which can not result from executing the program, then that formula is "uncovered" and the program is *incomplete.*

If during a computation, the program generates an infinite sequence of procedure calls, it is *nonterminating.* Otherwise, it is said to *terminate.*

We test for correctness and completeness by checking a program's computation results against its specifications, for a sequence of test inputs. Of course, finding no instances of incorrectness or incompleteness does not prove correctness and completeness. *Termination* is tested for by routines that compare the inputs with respect to the specified well-founded ordering whenever a procedure is invoked.

To generate test cases for a given goal, we first run the specifications of that goal to obtain a pair consisting of an input along with its expected output. We then use only the input value to run the goal on the program to be debugged. If the execution fails, goes into a loop, or returns an incorrect output value, then this test case has shown us that there is at least one bug in the program. In other words, a test case consisting of a correct input/output pair can be used to discover bugs should they cause the program to fail to compute the correct answer. If one of the predicates in the specifications of a program is defined in the form of a "generator", then we can generate alternate test cases by utilizing Prolog's built-in backtracking facility.

3 Bug Location

When a Prolog program does not compute correct results, it may be that the program contains incorrect clauses, is incomplete in defining certain relationships between program variables, or has an infinite procedure invocation sequence.

We constructed a meta-interpreter which executes programs, diagnoses errors according to the specifications of programs, and locates and reports bugs. Figure 1 summarizes the algorithm in pseudo-Prolog code.

```
execute( (Goal1, Goal2), Message ) :-
      execute( Goal1, Msg_Goal1 ),
      if Msg_Goal1 = ok( Goal1 )
           then execute( Goal2, Message )
           else Message = Msg_Goal1
execute( Goal, ok(Goal) ) :-
      system( Goal ), call( Goal )
execute( Goal, looping(Goal) ) :-
      not decreasing( Goal )
execute( Goal, Message ) :-
      not system( Goal ),
      clause( Goal, Subgoals ),
      execute( Subgoals, Msg_Subgoal ),
      if Msg_Subgoal = ok( Subgoals )
           then if spec( Goal )
                     then Message = ok( Goal )
                     else Message = incorrect( (Goal :- Subgoals) )
           else Message = Msg_Subgoal
execute( Goal, uncovered(Goal) ) :-
      spec( Goal )
```

Fig. 1. Algorithm for Automated Bug Location

The procedure *execute*(*Goal*, *Message*) serves two functions: goal reduction and bug location. The first clause deals with conjunctive goals. If the first conjunct executes correctly, the remaining conjuncts will be tried in order; otherwise, it just returns the error found to the top level. The second clause executes built-in primitives directly. The next three clauses detect bugs of nontermination, incorrect clauses, and uncovered goals, respectively.

The procedure first checks if the input variables violate the well-founded ordering defined in the specification of the procedure that covers the goal. To accomplish this, the parameters to each recursive call are recorded (asserted) during computation. If such is the case, we have an instance of a looping goal.

If the input cannot cause an infinite sequence of procedure calls, the interpreter will proceed to check if the program can actually complete the computation on the given input. It first finds a clause whose head can be unified with *Goal* and then recursively executes the subgoals in the body of that clause. If a bug is found in the body of a clause, it will be returned to the top level (for subsequent correction). If all the subgoals complete successfully, then all the output variables in *Goal* will be instantiated. The interpreter then checks if the output value is correct with respect to the specifications of *Goal*. If not, then we have found an incorrect clause.

If the goal fails because there is no clause in the program that covers it for the given input data (i.e., no unifying clause or a subgoal fails in every unifying clause), then, provided *Goal* is satisfiable according to the specifications, the program must be incomplete and we have an instance of an uncovered goal.

4 Bug Correction

Bug correction requires reasoning with knowledge of the domain and intended algorithm, the semantics of the programming language and the input/output specifications. Some automatic debugging systems use information stored in their system's knowledge base for bug correction by matching (maybe partially) and replacing the buggy program with the established code fragments. In our case, we have only the knowledge contained in the specifications of the individual procedures, plus some heuristics that suggest possible causes of errors. Deductive and inductive corrective measures are employed.

If a clause $p(x, y) :- p_1, ..., p_n$ returns an incorrect output y' on input x', we execute the specification of p with goal $p(x', Y)$, to get a correct output y''. If $p(x', y'')$ is covered by another clause in the program, then the incorrect clause should presumably have failed for this input. We, therefore, attempt to find extra conditions to prevent computation for input x', by trying to construct a proof that the right hand side of the clause implies the left hand side. If the proof fails because of some missing conditions, we add them as subgoals to the clause. (In the worst case, we can always add the subgoal **fail** to the clause. Although this might be too strong a fix and might result in some other goals becoming uncovered, we will see below how to deal with any uncovered goals.)

If the atom $p(x', y'')$ is only covered by the incorrect clause, then we proceed to add conditions that preclude computation of the wrong answer y', with input x', as above, or an inductive approach may be taken. If $p(x', y'')$ is not covered by any clause, then the fix proceeds in different directions, depending on whether $p(x', y'')$ can be unified with the head of the incorrect clause. If the head does unify, but some of the subgoals fail for y'', then we presume that the incorrect clause should cover the goal $p(x', y)$ and compute y'' instead of y'. In this case, we can combine fixes for the uncovered goal, $p(x', y'')$, and the incorrect clause that computes the erroneous solution $p(x', y')$. We check, for $p(x', y'')$ (i.e., under the current input and *correct* output), which of the subgoals in the clause fail with the output constrained to be y''. After identifying any such incorrect subgoals, we try to fix them by either applying a heuristic rule or an inductive method. We rearrange, replace, delete, or add new variables within subgoals until the original incorrect clause computes $p(x', y'')$ correctly, as in the refinement method of [12].

The last possibility is that $p(x', y'')$ cannot be unified with the head of the incorrect clause, nor is it covered by other clauses in the program. In this case, we assume that the incorrect clause we have identified should cover this goal. Accordingly, the only way to correct the bug is to first fix (i.e., weaken) the clause head so that it is unifiable with $p(x', y'')$. The methods described above can then be used to fix any incorrect subgoals.

To find subgoals that correct a faulty clause, we modify the approach of [13]. Our (incomplete) prover employs the following rules, in which G (possibly with a subscript) represents a goal, H (possibly with a subscript), an hypothesis, $\wedge$, $\vee$, and $\neg$ stand for logical "and", "or", and "not", respectively, $H \rightarrow G$" for "if H then G", and "$A \Leftarrow B$" for "to prove A, it is sufficient to prove B".

Rule 1. $H \rightarrow G_1 \wedge G_2 \Leftarrow (H \rightarrow G_1) \wedge (H \rightarrow G_2)$
Rule 2. $H \rightarrow G_1 \vee G_2 \Leftarrow (H \rightarrow G_1) \vee (H \rightarrow G_2)$
Rule 3. $(H_1 \vee H_2) \rightarrow G \Leftarrow (H_1 \rightarrow G) \wedge (H_2 \rightarrow G)$
Rule 4. $H \rightarrow (G_1 \rightarrow G_2) \Leftarrow (H \wedge G_1) \rightarrow G_2$
Rule 5. $(H_1 \rightarrow H_2) \rightarrow G \Leftarrow (\neg H_1 \rightarrow G) \wedge (H_2 \rightarrow G)$
Rule 6. $\neg H \rightarrow \neg G \Leftarrow G \rightarrow H$
Rule 7. $\neg H_1 \wedge H_2 \rightarrow \neg G \Leftarrow G \wedge H_2 \rightarrow H_1$

We replace a goal with its definition, as given in the goal's specification:

Rule 8. $H \rightarrow G \Leftarrow H \rightarrow G'$, *if* $G = G'$.

A logical simplifier is invoked after each reduction step and performs tasks such as removing nested conjunctions, duplicate goals, and tautologies. Also, domain facts can be used to replace a goal with something equivalent.

We use transitivity of implication:

Rule 9. $H \rightarrow G \Leftarrow H \rightarrow G'$, *if* $G' \rightarrow G$.

In particular, if the head of a correct program clause matches the goal, we can replace the goal with the subgoals obtained from that clause. Also, when a specific domain fact is known, it can be used to strengthen a goal.

Similarly, domain facts may be used to weaken hypotheses:

Rule 10. $H \rightarrow G \Leftarrow H' \rightarrow G$, *if* $H \rightarrow H'$.

An effort was also made to build in some domain knowledge about lists and inequalities so that it can employ a bit of common sense when reasoning about programs.

The proof process terminates when the original goal is reduced to **true**, in which case the clause is proved correct; when the original set of hypotheses (that is, the subgoals in the body of the clause) is reduced to **false**, meaning that there are conflicting subgoals in the clause, and that the clause is vacuously correct; when the goal is reduced to a subset of the hypotheses, in which case the implication is also established; or when the original goal is reduced to primitives and hypotheses, in which case those goals not appearing as hypotheses are added as subgoals to the original clause. In the latter case, we have identified those missing subgoals which will make the clause correct.

Once we identify an incorrect subgoal, we can correct it using either a heuristic rule or an inductive method, besides using the deductive methods outlined above.

We employ heuristics meant to correct certain commonly made, easily corrected, errors. For example, one of the rules is to swap the variables if there are only two variables in the subgoal. Other rules include moving a simple variable to a different position, replacing simple variables with more complicated terms, deleting seemingly redundant variables, and adding free variables that have appeared elsewhere in the same clause.

When the heuristic rules cannot correct the errors in a subgoal, a general inductive strategy is employed with the hope of fixing the bugs. This is done by applying some refinement operations on terms within the subgoals. For example, we can try to unify two free variables, or unify a compound term with variables appearing elsewhere in the same clause.

It should be noted that all heuristic fixes will be tested immediately after the changes are made; and if the fixes cannot correct the errors, all the changes will be undone.

To remedy the problem of an uncovered goal, we first check if the goal can be unified with the head of a clause. If indeed such a clause exists, then we presume that it should cover this goal. Since the original clause might be useful for other goals, instead of modifying the clause directly, we make local changes on a copy. We locate the subgoal that causes this clause to fail and either try to fix it inductively (by rearranging, replacing, deleting, or adding variable within the subgoal) or eliminate the offending subgoal entirely and use deductive means to correct it, if necessary.

When there is no clause whose head unifies with the uncovered goal, we use the specifications to synthesize a new clause. This can be done by using the uninstantiated goal as the clause head and the specifications as the clause body, simplifying the resulting clause as much as possible, or by an inductive method such as that in [12], using the specifications to guide the search. We can also fix a clause head so that it can be unified with the uncovered goal, and then debug the subgoals in the clause.

When the input to a procedure call violates the well-founded ordering defined for that procedure, a likely cause is that the input argument of the call is too general. For example, it may contain an irrelevant variable that does not appear in either the clause head or other subgoals of the same clause. Other possibilities are that some variables are missing or that the order of arguments is wrong. In any of these cases, what we have is a clause that contains a looping call caused by incorrect arguments. We try to fix the offending subgoal, using the same inductive method as for fixing incorrect subgoals. Alternatively, we can weaken it and employ deductive techniques to ensure that the well-founded condition is met.

It is also possible that a subgoal that would preclude the looping case is missing (and that the goal is covered by another clause). This can be treated in the same way as an incorrect clause.

5 The Constructive Interpreter

We integrated the functions of test case generation, bug discovery, bug location, and bug correction into an automated debugging environment, called the *Constructive Interpreter*. Its structure is described in Fig. 2 in pseudo-Prolog code. Upon receiving a goal, the interpreter first examines the input variables. If the input is symbolic (partially uninstantiated), then by executing the specifications of the procedure,

```
interpret( Goal ) :-
      spec( Goal ),
      freeze_input_variables( Goal, Goal' ),
      execute( Goal', Message ),
      if Message ≠ ok(Goal')
           then fix_bug( Message )
```

Fig. 2. The Constructive Interpreter

the interpreter will generate test cases. If the input variables are instantiated, then running the specifications on the given input checks if the input values are satisfiable. Once the legality of the input is established or a legal test input generated, the interpreter proceeds to execute the program on the input. Note that the interpreter will "freeze" the variables at this point, treating them as constants so that they will not be changed by the Prolog system. If execution completes successfully, the interpreter returns correct output values. In the case of symbolic input, the user can continue to generate alternate test cases and execute the program on different inputs. If the execution ever fails, that is, if the program contains an incorrect, incomplete, or nonterminating procedure, then the interpreter returns a diagnostic message with the location. Bug-fixing routines will then be invoked to correct the bug that has been identified and located.

Procedure $fix\_bug(Message)$ implements the bug correction heuristics discussed in Section 4.

This interpreter is constructive in the sense that it assumes an active role during the debugging process and actually tries to complete the construction of the program being debugged, all with very little user involvement. It consists of the three major components: test case generator, bug locator, and bug corrector. The test case generator executes specifications to either generate test input or verify the satisfiability of user-supplied input. The bug locator also carries out the computation. It has a run-time stack that records all the procedure invocations. This information and the specified well-founded ordering are used to check against looping. The execution is simulated to perform depth-first search and backtracking upon failure. A message stack is maintained during execution, and an error message is recorded whenever an error occurs. The bug corrector contains three main procedures, dealing with three different kind of errors respectively. In addition to performing error analysis and suggesting fixes, they all have access to the deductive theorem prover and inductive subgoal refiner.

In the remainder of this section, we illustrate the integrated functions, including test case generation, bug location, and correction, of the *Constructive Interpreter*. Our experimental implementation is able to generate test cases that reveal errors and locate bugs for all the sorting examples in [12].

Consider the quicksort program in Fig. 3, with the specifications in Fig. 4. The specifications say that $qsort(X, Y)$ holds if Y is sorted and Y is a permutation of X, that $part(L, E, X, Y)$ holds if Y is the list obtained by removing elements of X from L and E is greater than all the elements in X and smaller then all the elements in Y, and that $append(X, Y, Z)$ is true if Z is the combination of lists X and Y, in their

```
        qsort([X|L], L0) :- part(L, X, L1, L2), qsort(L1, L3), qsort(L2, L4),
                            append([X|L3], L4, L0)
 part([X|L], Y, L1, [X|L2]) :- part(L, Y, L1, L2)
 part([X|L], Y, [X|L1], L2) :- X ≤ Y, part(L, Y, L1, L2)
         part([ ], X, [X], [ ])
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3)
         append([ ], L, L)
```

Fig. 3. A Buggy Quicksort Program

```
                 spec(qsort(X, Y)) :- ordered(Y), perm(X, Y)
             spec(part(L, E, X, Y)) :- rm_list(X, L, Y), gt_all(E, X),
                                       lt_all(E, Y)
            spec(append(X, Y, Z)) :- length(X, N), front(N, Z, X),
                                     rm_list(X, Z, Y)
        wfo(qsort(X, Y), qsort(U, V)) :- shorter(X, U)
  wfo(part(X, A, B, C), part(Y, D, E, F)) :- shorter(X, Y)
  wfo(append(X, A, B), append(Y, C, D)) :- shorter(X, Y)
```

Fig. 4. Specifications for the Quicksort Program

original order. The predicate *wfo* specifies the well-founded ordering for sequences of input values. For procedures *qsort*, *part*, and *append*, the number of elements in the input list should decrease with each recursive call. The predicates *perm*, *ordered*, *rm_list*, *gt_all*, *lt_all*, and *shorter* can be defined as standard Prolog procedures, as in Fig. 5.

Invoking the debugger on the symbolic goal $qsort(U, V)$ generates a test case $qsort([\,], X)$ which it tries to satisfy. It discovered that this goal should have a solution $qsort([\,], [\,])$ according to the specification of *qsort*, but cannot get it from the program supplied. It then The debugger uses the specification for *qsort* to synthesize the clause $qsort([\,], [\,])$:-$ordered([\,]), perm([\,], [\,])$ to cover that goal. Since the body of this clause can be reduced to *true*, the debugger added a unit clause to the program (by asserting it to the database).

The debugger generates a one element list $qsort([x], X)$ as its next test input. (Note that the generated input, $[x]$, contains a Skolem constant x.) This time, it finds an incorrect clause in the procedure *part*, because partitioning an empty list should result in two empty sublist, so the result of $parti([\,], x, X, Y)$ should be $part([\,], x, [\,], [\,])$ instead of $part([\,], x, [x], [\,])$. After further analysis, the debugger concludes that $part([], X, [X], [])$:-*true* is incorrect. Since it can not fix the head, it retracts the clause. After synthesizing a clause that covers $part([\,], x, [\,], [\,])$, the debugger reexecutes all the test goals generated so far to make sure the changes do not destroy anything. (Note that there is no way a correctly synthesized clause can cause a problem; retracting an incorrect clause, however, can cause some goals to become uncovered.)

$$\begin{aligned}
\mathit{ordered}([\,]) & \\
\mathit{ordered}([X]) & \\
\mathit{ordered}([X1, X2|Xs]) & \text{ :- } \mathit{lt}(X1, X2),\ \mathit{ordered}([X2|Xs]) \\
\\
\mathit{perm}([\,], [\,]) & \\
\mathit{perm}([X|Xs], Ys) & \text{ :- } \mathit{del}(X, Ys, Zs),\ \mathit{perm}(Xs, Zs) \\
\mathit{del}(X, [X|Xs], Xs) & \\
\mathit{del}(X, [Y|Xs], [Y|Ys]) & \text{ :- } \mathit{del}(X, Xs, Ys) \\
\\
\mathit{rm\_list}([], Y, Y) & \\
\mathit{rm\_list}([H|T], Y, Z) & \text{ :- } \mathit{remove}(H, Y, YY),\ \mathit{rm\_list}(T, YY, Z) \\
\mathit{remove}(A, [A|T], T) & \\
\mathit{remove}(A, [H|T], [H|U]) & \text{ :- } \mathit{remove}(A, T, U) \\
\\
\mathit{gt\_all}(E, [\,]) & \\
\mathit{gt\_all}(E, [H|T]) & \text{ :- } E > T,\ \mathit{gt\_all}(E, T) \\
\\
\mathit{lt\_all}(E, [\,]) & \\
\mathit{lt\_all}(E, [H|T]) & \text{ :- } E < T,\ \mathit{lt\_all}(E, T) \\
\\
\mathit{front}(N, Z, X) & \text{ :- } \mathit{append}(X, Y, Z),\ \mathit{length}(X, N) \\
\mathit{shorter}(X, Y) & \text{ :- } \mathit{length}(X, Lx),\ \mathit{length}(Y, Ly),\ Lx < Ly
\end{aligned}$$

Fig. 5. Some Utility Procedures

The next test case generated is $qsort([0,1], X)$. Unlike the previous two test cases, the goal $qsort([0,1], X)$ is solved directly by the clauses currently in the program. Continuing with $qsort([1,0], X)$ results in the location of an incorrect clause in the procedure *part*. A trace of the procedures shows that the correct solution to $part([0], 1, X, Y)$ can be obtained from the other clause of *part*. Thus, this incorrect clause should have failed, but did not because of a missing subgoal. The corrected clause is $part([X|Y], Z, U, [X|W])\text{:-}Z <= X, part(Y, Z, U, W)$.

Rechecking the previous goal $qsort([1,0], X)$, the instance $qsort([1,0],[1,0])$:- $part([0], 1, [0], [])$, $qsort([0], [0])$, $qsort([], [])$, $append([1,0], [], [1,0])$ is found false. That is, $qsort([X|W], U)$:- $part(W, X, U1, V1)$, $qsort(U1, Y)$, $qsort(V1, Z)$, $append([X|Y], Z, U)$ contains an incorrect subgoal $append([X|Y], Z, U)$. The local fix is $qsort([X|Y], Z)$:- $part(Y, X, W, X1)$, $qsort(W, Z1)$, $qsort(X1, V1)$, $append(Z1, [X|V1], Z)$.

Up to this point, all the bugs in the original program have been detected and corrected. If we now continue to debug the program, the debugger will keep on generating arbitrarily long lists as test input without reporting an error. We would be led to believe, in this case, that the program is correct with respect to its specifications.

6 Conclusion

In this work, we have explored a distinctive feature of logic programming, namely the ability to use logic for both specification and computation. We have shown how user-supplied executable program specifications are used to define the intended behavior of a program and to generate test cases for bug discovery. We have employed the execution mechanism of a Prolog machine to locate bugs, using specifications to validate computation results. We have also devised heuristics to analyze bugs and suggest fixes, and used deductive theorem proving and inductive synthesis to mechanize the bug correction process, again with the help of specifications.

References

1. Paul Brna, Alan Bundy, and Helen Pain. A framework for the principled debugging of Prolog programs: How to debug non-terminating programs. In D. R. Brough, editor, *Logic Programming: New Frontiers*, chapter 2, pages 22–55. Intellect, Oxford, 1992.
2. W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In *Proceedings of a Workshop on Meta-Programming in Logic Programming*, Bristol, June 1988.
3. Nachum Dershowitz and Yuh-jeng Lee. Deductive debugging. In *Proceedings of the Fourth IEEE Symposium on Logic Programming*, pages 298–306, San Francisco, CA.
4. Gérard Ferrand. Error diagnosis in logic programming, an adaptation of E. Y. Shapiro's method. Technical Report 375, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France, March 1985.
5. Joseph A. Goguen and José Meseguer. EQLOG: Equality, types, and generic modules for logic programming. In D. DeGroot; G. Lindstrom, editor, *Logic Programming: Relations, Functions, and Equations*. Prentice Hall, Englewood Cliffs, NJ, 1986.
6. N. Alan Josephson and Nachum Dershowitz. An implementation of narrowing: The RITE way. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 187–197, Salt Lake City, UT, September 1986.
7. J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5:133–154, 1987.
8. L. Naish. Declarative debugging of lazy functional programs. Report 92/6, Department of Computer Science, University of Melbourne, Australia, 1992.
9. H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, pages 385–399, Leuven, Belgium, August 1992. Available as Vol. 631 of Lecture Notes in Computer Science, Springer-Verlag.
10. L. M. Pereira. Rational debugging in logic programming. In *Proceedings of the Third International Conference on Logic Programming*, pages 203–210, London, United Kingdom, July 1986. Available as Vol. 225, Lecture Notes in Computer Science, Springer-Verlag.
11. L. M. Pereira and M .C. Calejo. A framework for Prolog debugging. In *Proceedings of Fifth International Conference and Symposium on Logic Programming*, Seattle, August 1988.
12. Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.
13. Douglas R. Smith. Derived preconditions and their use in program synthesis. In *Proceedings of the Sixth Conference on Automated Deduction*, pages 172–193, New York, NY, June 1982.