Declarative debugging of lazy functional programs

Lee Naish

(lee@cs.mu.OZ.AU)

Technical Report 92/6

Department of Computer Science
University of Melbourne
Parkville, Victoria 3052
Australia

We show how declarative (or algorithmic) debugging can be applied to lazy functional programming and describe a prototype implementation. We first present a declarative debugger for logic programs which relies on three primitives that determine if an atom is valid in the intended interpretation, return the successful clause instance used by a call and return single atoms from a conjunction of atoms. By simply using different interpretations and versions of these three primitives the debugger can be used to locate errors in strict and non-strict functional programs. Debugging strict code is essentially the same as debugging Horn clause programs. Debugging non-strict code can result in questions containing unevaluated expressions. These questions can be simplified by using quantified variables.

The prototype system is based on NUE-Prolog, which supports a functional language on top of NU-Prolog and is implemented using the flattening transformation. The implementation has been modified so functional code returns a representation of the computation which is then passed to the debugger. The basic algorithm is applicable to other functional programming languages also.

Keywords: functional programming, lazy evaluation, declarative debugging, logic programming.

---

1 Introduction

Declarative debugging (also called algorithmic, rational and deductive debugging) [Sha83, Per86, SS86, DL87, Llo87] is a debugging methodology which was originally developed for logic programming. Bugs can be located using only declarative knowledge of the program (what results returned by procedures are correct) rather than procedural knowledge (the sequence of operations performed during execution). This approach is particularly advantageous when the evaluation mechanism is complicated. Prolog, for example, uses unification, backtracking and (in some systems) coroutining, so finding errors from a trace of the execution can be very difficult. In contrast, the declarative semantics of pure Prolog programs is straightforward. A program can be seen as a set of formulas in first order logic. A declarative debugger for Prolog asks the user (or some other oracle) if results returned by procedures are true in the intended interpretation of the program. In this way the bug can isolated to a particular procedure or clause.

Many functional programming languages have well defined declarative semantics or, like Prolog, have "pure" subsets with this property. Referential transparency allows us to determine the correctness of the result of a function call without knowing the actual steps in the execution. This is exactly what is required for declarative debugging. It is rather surprising how little work there has been on adapting declarative debugging techniques to these languages. The debugging support for many functional languages is poor, and many languages use lazy evaluation. Lazy evaluation, like the complex evaluation mechanism of Prolog, makes debugging based on procedural semantics very difficult. Declarative debugging algorithms for functional programming languages should therefore be very useful.

This paper is organized as follows. First a simple declarative debugger for diagnosing incorrect answers in pure Prolog programs is presented. The debugger is somewhat more abstract than other published debuggers and is written in a subset of NU-Prolog which has well defined declarative semantics. Next a simple functional programming language with (initially) non-lazy evaluation is described. We show how such a language can be debugged in an equivalent way to Prolog. Lazy evaluation is then introduced and we show how the debugger can be generalised to handle this. A refinement which reduces the complexity of questions to the user is also presented. The functional programming language we use is implemented by transformation into Prolog. Though our discussion is based on this language and implementation, we hope it will be obvious how the ideas can be adapted. A summary of the debugging algorithm in abstract terms is given after the description of our implementation. Finally, we consider related work and possible extensions then conclude.

2 Declarative debugging of logic programs

Wrong answers in pure Prolog programs without negation result from the use of an instance of an incorrect clause in the program. That is, the body of the clause instance is valid in the intended interpretation but the head is not. This can be formalised by the logic below (written in Prolog syntax; a generalisation of the logic presented in [Nai92]). The variables range over representations of atoms and conjunctions in the program being debugged. `valid(A)` is true if `A` is the representation of a formula which is valid in the intended interpretation. This can be implemented by querying the user or using a runnable specification. Theoretically, this logic can be used to find bugs in a program by generating clause instances and checking if they are (in)correct. In practice the search space is infinite for non-trivial programs.

```
bug((H :- B)) :-
        clause_instance(H, B),
        not valid(H),
        valid(B).
```

Practical debuggers for Prolog use a goal which behaves incorrectly to guide the search for an incorrect clause instance. The debugger below returns an incorrect clause instance as before but uses an additional input: the representation of a goal which has succeeded incorrectly (the incorrect answer substitution has been applied). It is essentially the same as the top down debugger of [SS86] but is refined so as to avoid using concrete data structures and non-logical primitives such as cut. The if-then-else construct of NU-Prolog [Nai86] (which has well defined declarative semantics) is used. The code relies on three additional procedures: `valid/1`, `successful_clause/2` and `c_member/2`. In this debugger, `valid` is only called with atoms. `successful_clause(H,B)` is true if `H:-B` is the representation of a clause instance whose body has succeeded. `c_member(A,B)` is true if `B` is the representation of the body of a successful clause instance (or goal) and `A` is the representation of an atom in it.

```
        % from representation of incorrectly successful atom,
        % return incorrect clause instance
wrong_atom(A, B) :-
        not valid(A),
        successful_clause(A, G),
        (if some [B1] wrong_rhs(G, B1) then
                B = B1
        else
                B = (A :- G)
        ).

        % as above, but for an arbitrary goal (eg, RHS of a clause)
wrong_rhs(G, B) :-
        c_member(A, G),
        wrong_atom(A, B).
```

The program can be read declaratively as follows. Given that a successful atom `A` is not valid and `A:-G` is a successful clause, if `G` is wrong due to some bug `B1` then `A` is wrong due to `B1`, otherwise `A` is wrong due to the incorrect clause instance `A:-G`. A successful clause body instance is wrong due to some bug `B` if some atom in it is wrong due to `B`.

It is helpful to consider the proof tree of a goal, defined as follows. Each node in the proof tree consists of a Prolog goal (either the top level goal or the body of a clause instance). Associated with each atom in the goal is a child node, consisting of the body of the clause instance which matched the atom. The debugger searches the proof tree for an atom which is not valid but whose child is valid. This corresponds to a clause instance whose body is valid but whose head is not. If the logic of the debugger is interpreted as a Prolog program the search is conducted top down, with the ordering defined by the order in which solutions to `c_member` are returned. The same logic can also be used with more intelligent search strategies.

It is also helpful to use proof trees, or parts of them, to represent goals, atoms and clause instances in the debugger. `successful_clause(H,B)` then simply matches `B` with the child of `H`, rather than recomputing any of the tree. The top level of a debugger can then be written as follows:

```
        % given a goal, find a buggy successful instance
        % (if one exists) and display the bug
wrong(Goal) :-
        compute_proof(Goal, Proof),
        wrong_rhs(Proof, Bug),
        display_bug(Bug).
```

As an example of the operation of the debugger, consider the following buggy program and call to `wrong`. We assume that `valid` is implemented by querying the user.

```
plus(0, I, I).
plus(s(I), J, K) :- % K should be s(K)
        plus(I, J, K).

?- wrong(plus(s(s(0)), s(0), X)).
% X = s(0) is the computed answer to the goal
Is plus(s(s(0)), s(0), s(0)) valid?  n
Is plus(s(0), s(0), s(0)) valid?  n
Is plus(0, s(0), s(0)) valid?  y

Incorrect clause instance (body is valid but head is not):
plus(s(0), s(0), s(0)) :-
        plus(0, s(0), s(0)).
```

Note that there may be calls to `valid` in which the argument is (the representation of) a non-ground atom such as `append([A],[B],[A,B])`. As in [DNTM88], the user is asked if the universal closure of the atom is valid (the user can simply be told that all variables in questions are implicitly universally quantified). This contrasts with [Llo87] in which `valid` returns all valid instances of a satisfiable atom.

3 A simple strict functional programming language

NUE-Prolog [Nai91] allows evaluable functions to be defined by sets of mutually exclusive equations. The definitions are converted to NU-Prolog by a preprocessor which implements the "flattening" transformation. This takes an evaluable function with N arguments and converts it into a predicate with N+1 arguments. Terms containing several occurrences of evaluable functions are converted into conjunctions of atoms. Left to right evaluation of the Prolog code is equivalent to leftmost innermost evaluation of the term. For example, the definitions below are converted into the following Prolog code.

```
plus(0, I) = I.
plus(s(I), J) = s(plus(I, J)). % correct version
% plus(s(I), J) = plus(I, J). % buggy version

times(0, I) = 0.
times(s(I), J) = plus(J, times(I, J)).

% Transformed code

plus(0, I, I).
plus(s(I), J, s(K)) :- plus(I, J, K). % correct version
% plus(s(I), J, K) :- plus(I, J, K). % buggy version

times(0, I, 0).
times(s(I), J, K) :- times(I, J, L), plus(J, L, K).
```

NUE-Prolog also supports an if-then-else construct (implemented using NU-Prolog if-then-else) and *apply* for defining higher order functions (implemented using *call*) and equations can have additional constraints in the form of Prolog code. For simplicity, these features will be ignored. Lazy evaluation is also supported and will be discussed later.

4 Declarative debugging of strict functional programs

The debugger for logic programs we presented can be used for debugging functional programs with *no* changes to the code. All that is necessary is to change the interpretation. Instead of clauses, think of equations. Therefore when a bug `H:-B` is returned, it represents an incorrect equation `H=B`. Instead of atoms, think of terms in which the top level function symbol is an evaluable function. Instead of general goals, think of general terms. `c_member(A,C)` is therefore true if `A` is (the representation of) a sub-term of `C` with an evaluable function symbol at the top level. Instead of a proof tree we have a computation tree in which each node is a term and each sub-term with an evaluable function at the top level has a child associated with it (corresponding to the use of an equation instance). Implicitly associated with each evaluable sub-term is the expression it is finally evaluated to. This can be found by traversing the tree. Instead of success, think of evaluation of a functional expression. Instead of truth, think of correctness of the evaluation. `valid(A)` is therefore true if the term `A` evaluates correctly. Assuming the top level term to be evaluated contains no variables, `valid` will only be called with ground terms so quantifiers are not needed.

Note that the relationship between the two interpretations is exactly the flattening transformation. The functional interpretation of a call to the debugger with a functional expression is equivalent to the Prolog interpretation of a call to the debugger with the flattened version of the expression. Thus running the functional version of the debugger is equivalent to running the Prolog version on the flattened version of the functional program and expression.

As before, it is helpful for the representation of the objects to encode the successful computation. Rather than having a separate meta interpreter which returns a proof tree, we have augmented the flattening process so that each predicate derived from a function returns two values. One is the result of evaluating the function, as before, and the other represents the computation in a similar way to a proof tree (the result of evaluating the function can be derived from this also). If there is a term containing an evaluable function `f(...)` which is rewritten to an expression `rhs(...)` then the representation of the computation contains `$rewrite(f(...),RHS,PG)`, where `RHS` represents the computation of `rhs(...)` and `PG` represents an additional Prolog goal which was solved at this stage (typically it is simply `true`). The result of transforming the definitions above is as follows.

```
plus(0, A9, A9, A9).
plus(s(A9), B9, s(A), s($rewrite(plus(A9, B9), B, true))) :-
        plus(A9, B9, A, B).

times(0, A9, 0, 0).
times(s(A9), B9, A,
        $rewrite(plus(B9, $rewrite(times(A9, B9), B, true)), C, true)) :-
        times(A9, B9, D, B),
        plus(B9, D, A, C).
```

Special data structures are returned for if-then-else and apply and there is extra code in the debugger to handle them. To evaluate the expression `plus(s(s(0)),s(0))` it is first flattened to the Prolog goal `plus(s(s(0)),s(0),Res,Comp)` which is then called, resulting in the answer:

```
Res =           s(s(s(0)))
Comp =          s($rewrite(plus(s(0), s(0)),
                        s($rewrite(plus(0, s(0)),
                                s(0), true)), true))
```

Calls to `valid/1` and `successful_clause/2` have a term with `$rewrite` as the top level function symbol. From this we can extract the original term (corresponding to the left side of an equation instance), the right side of the equation instance and what the term is eventually evaluated to. `c_member(A,C)` binds `A` to sub-terms of `C` which have `$rewrite` as their top level functor and are not sub-terms of the second or third argument of a `$rewrite` term. Given the incorrect functional version of plus, our functional debugger behaves as follows:

```
?- wrong(plus(s(s(0)),s(0))).
answer = s(0) valid?  n
plus(s(s(0)), s(0)) = s(0) valid?  n
plus(s(0), s(0)) = s(0) valid?  n
```

```
plus(0, s(0)) = s(0) valid?  y

Incorrect equation instance:
plus(s(0), s(0)) = plus(0, s(0)).
```

5 Implementing lazy evaluation

NUE-Prolog allows functions to be declared *lazy*. Such functions simply return a *closure* when called. If the value is needed at some point, an auxiliary function is called to force one step of the evaluation. A closure needs to contain representations of the arguments of the call and what function to call. If the predicate implementing the auxiliary function is `auxpred`, the representation of a closure we use is `$lazy(auxpred(`Arg_1`,...`Arg_N`,Res),Res)`. To force evaluation we use additional clauses whenever the left side of an equation contains (non-evaluable) function symbols. The following code illustrates the transformation:

```
        % returns infinite list of successive integers
?- lazy ints_from/1.
ints_from(N) = N.ints_from(s(N)). % correct version
% ints_from(N) = s(N).ints_from(N). % buggy version

        % returns first N elements of a list
front(0, L) = [].
front(s(N), H.L) = H.front(N, L).

% Transformed code

        % just returns closure
ints_from(A, $lazy($lazy$ints_from(A, B), B)).

        % auxiliary predicate
$lazy$ints_from(A9, [A9|A]) :-
        ints_from(s(A9), A).

front(0, A9, []).
front(s(A9), [B9|C9], [B9|A]) :-
        front(A9, C9, A).
        % additional clauses to force evaluation
front($lazy(B, A), C, D):- call(B), front(A, C, D).
front(s(C), $lazy(B, A), D):- call(B), front(s(C), A, D).
```

To evaluate the expression `front(s(s(0)),ints_from(0))` it is first flattened to the Prolog goal `ints_from(0,Is), front(s(s(0)),Is,Res)`. `Is` is initially bound to the closure `$lazy($lazy$ints_from(0,V1),V1)`. The call to `front` matches with the last clause which forces the closure to be evaluated one step then recursively calls `front` with `V1` which has been bound to `[0|$lazy($lazy$ints_from(s(0),V2),V2)]`. This call matches with the second clause, constructing the first element of the output list `Res` to `0`. The recursive call forces one more step in the evaluation of `ints_from` and binds the second element of `Res` to `s(0)`. `front` is finally called with the first argument `0`, which binds the tail of `Res` to `[]` without evaluating `Is` any further. The final binding for `Is` is a nested structure containing cons cells and closures. The innermost closure contains an uninstantiated variable as the last argument, indicating it was never evaluated.

6 Declarative debugging with lazy evaluation

To support declarative debugging of lazy code we use the same technique as for strict code: an extra argument is returned which returns a representation of the computation. A key point is that this representation tells us what expressions were evaluated and what expressions simply resulted in closures which where not evaluated. The debugger can then avoid further evaluating these expressions, which could cause infinite loops. We use the same `$rewrite` structure as before. The only change we have to make is to add an extra argument to the closures, since two values rather than one are now returned from each function. The `ints_from` function results in the following code.

```
ints_from(A, $lazy($lazy$ints_from(A, B, C), B, C), C).

$lazy$ints_from(A9, [A9|A],
        [A9|$rewrite(ints_from(s(A9)), B, true)]) :-
        ints_from(s(A9), A, B).
```

Closures which have been evaluated can be treated transparently by the debugger code. A term of the form `$lazy(auxpred(...),Res,Comp)`, where `Comp` is a nonvariable, is treated in the same way as the term `Comp`. The difference between debugging lazy and strict code is the treatment of unevaluated closures (`Comp` will be a variable in our system) by `valid`. The most straightforward approach is to use the unevaluated expression in questions asked by valid. The following example, using the buggy version of `ints_from`, illustrates this.

```
?- wrong(front(s(s(0)), ints_from(0))).
answer = [s(0), s(0)] valid?  n
front(s(s(0)), [s(0), s(0)|ints_from(0)]) = [s(0), s(0)] valid?  y
ints_from(0) = [s(0), s(0)|ints_from(0)] valid?  n
ints_from(0) = [s(0)|ints_from(0)] valid?  n

Incorrect equation instance:
ints_from(0) = [s(0)|ints_from(0)].
```

7 Simplifying questions

The questions asked in the approach above contain details of the closures which can be eliminated without compromising the precision of the debugging algorithm. Rather than using unevaluated expressions, which may be quite complex, variables can be used. However, it is necessary to carefully consider quantification of the variables. `valid` is always called with a term of the form `$rewrite(f(Args),RHS,Goal)`. There are two cases to consider. The first is when an unevaluated closure appears in `Args`. This corresponds to an expression which was passed into the function but never evaluated.

For example, the third and subsequent members of the list of integers in the `front` expression above are never evaluated. The representation of the computation is a term of the form `$rewrite(front(s(s(0)),...$lazy(...,Rn,Cn)...), ..., true)`, where `Cn` is a variable. Such cases are equivalent to a Prolog goal which succeeds without completely instantiating its arguments. The unevaluated closure can be represented by a universally quantified variable. Note that the same closure can appear in both the first and second arguments of a `$rewrite` term. For example, the expression `front(s(0),[ints_from(0)])` will result in a closure which appears in both the second argument of `front` and the result which `front` is eventually evaluated to. Multiple occurrences of a closure should be treated as multiple occurrences of a universally quantified variable. In our implementation, two closure occurrences can be compared by checking if the variables in them are identical.

The second case is when there is an unevaluated variable which occurs in the result of a function evaluation but not in the arguments of the function. This corresponds to a function which is not completely evaluated. For example, in the `front` computation again there is a term of the form `$rewrite(ints_from(0), ...$lazy(...,Rn,Cn))..., true)`. Such cases are similar to a Prolog computation in which some satisfiable subgoal is not executed. `valid` can treat these unevaluated closures as existentially quantified variables. The example illustrates the use of both universal and existential quantification of variables.

```
?- wrong(front(s(s(0)), ints_from(0))).
answer = [s(0), s(0)] valid?  n
front(s(s(0)), [s(0), s(0)|A]) = [s(0), s(0)] valid?  y
some [A] ints_from(0) = [s(0), s(0)|A] valid?  n
some [A] ints_from(0) = [s(0)|A] valid?  n

Incorrect equation instance:
ints_from(0) = [s(0)|ints_from(0)].
```

8 Summary of algorithm

We now summarize the algorithm for declarative debugging of lazy functional programs. We start with a representation of the functional computation. This is a tree where each node contains a term. Each evaluable sub-term is referred to as a *call* term. Each call term has a *result* term and a child associated with it. The child is the right hand side of the equation instance that the term was matched with, if it was evaluated at all, and a unique closure identifier otherwise. The result term is the final result of evaluation of the term. If the final result contains evaluable functions which have not been evaluated, these are tagged with the identifiers associated with the call terms in which they were introduced. The root node is a special node just containing *answer* as the call term and the result of the whole computation as the result term. It has one child, which has the top level expression as the call term.

The debugger searches the computation tree for a call term C which is incorrect, but all evaluable terms which are in the associated child are correct. An instance of the equation used to match with C is returned. To determine the correctness of call terms we assume the existence of an oracle which can determine truth in the intended model of the equational theory on which the program is based. Given a call term C with result term R, let L be C with all proper sub-terms replaced by their associated result terms. C is correct iff $L{=}R$ is true in the intended model. The instance of the equation to be returned is formed by matching L with the head of the equation.

For the refinement which results in simplified questions, each sub-term of L and R tagged with closure identifier i is replaced by variable V_i. C is correct iff the universal closure of $\exists V_{t1} ... \exists V_{tn} L = R$ is true in the intended model, where $V_{t1} ... V_{tn}$ are the variables appearing in R but not L. The instance of the equation to be returned is formed as before (but now may contain variables).

9 Related work

The only other attempt at applying declarative debugging to functional programming that we know of is [NF92]. An earlier version of this paper actually inspired our work. It gave an algorithm which results in unacceptably complex questions to the user. The questions are of the form: should term *T1* reduce to term *T2*, where *T1* and *T2* are both arbitrary terms, typically containing several evaluable function symbols. The questions asked depend on the evaluation order. The Prolog analogue of this algorithm would be to ask questions of the form: is formula *F1* implied by formula *F2*, where *F1* and *F2* are resolvents used in the computation. In contrast, the debugging method we use asks questions about single atoms in the Prolog case and equations with one evaluable function symbol in the functional case.

The algorithm presented in [NF92] is very similar to our algorithm. The evaluated versions of subterms are used wherever possible in questions. Underscores and elipsis (...) are used, though quantification is not mentioned explicitly. The starting point for their work was a debugger with a specified and somewhat “smart” search strategy. The details of the search strategy detracted from a clear statement of what the debugger was actually searching for and how it related to similar debuggers for logic programs. We had the advantage of starting from a very clear specification of the logic behind the debugger. Also, working in Prolog rather than C meant that our implementation required approximately one tenth the amount of code and it was possible to investigate algorithms and implement the prototype system in just a few days.

10 Further work

There are a great number of worthwhile extensions to the work we have done. The most important from a practical point of view is to implement similar systems for more widely used functional programming languages. For portability and flexibility, the ideal implementation language would be these functional programming languages. For languages which do not use lazy evaluation it should be quite easy to use a transformational approach. The source code could be transformed so functions return a representation of the computation, as is done in our system. A debugger written in the functional language could then use this data structure in a similar way to our debugger written in Prolog.

For systems using lazy evaluation, implementation of our debugging algorithm in the functional language seems more difficult. The key to debugging lazy code is the handling of closures. We make use of Prolog's meta level predicate `var/1` (which has dubious semantics at best). At the very least, it seems that an implementation in a pure functional language would have to encode the (lazy) evaluation mechanism in the debugger, either as a meta interpreter or implicitly in some form of program transformation.

A second implementation oriented extension of our work would be a declarative debugger for full NUE-Prolog, which allows mixing of Prolog with evaluable functions. A combined logic and functional computation can be represented as a tree which can be passed to a combined debugger. Such a system should be able to debug programs which miss answers as well as returning wrong answers. This is standard practice in declarative debuggers for Prolog. However, there are additional issues when functional code is involved. Consider errors such as attempting to extract the head of an empty list. In our implementation this simply causes failure at the Prolog level, so a natural way to debug such code is to treat it as a missing answer. An alternative would be to return a special value, *bottom*, from failed functional computations and use wrong answer diagnosis.

A final obvious area for research is the search strategy used by the debugger. Some work has already been done for debugging Prolog [Sha83, Per86, DNTM88, PC88] and this should be easy to apply to debuggers for functional languages. It may be that there are additional benefits in the functional case. For example, the debugger described in [Per86] allows users to specify what subterms of an answer are wrong. Information concerning what procedures produced what variable bindings is then used to locate the bug more directly. Since data flow information is simpler and seems more readily available in functional programming languages, such ideas may be easier to apply.

11 Conclusion

Declarative debugging is a very useful debugging paradigm, especially for programs which have simple declarative semantics but complex procedural semantics. Many functional programs fall into this category. We have shown that a suitably abstract formulation of a debugger for Horn clause logic can be reinterpreted as a debugger for functional programs. Without lazy evaluation the correspondence between the functional interpretation and the predicate logic interpretation is simply the flattening transformation which has been used to transform functional programs into logic programs. When lazy evaluation is used the correctness of computations which only partially evaluate terms must be ascertained. This can be done by asking the oracle questions containing partially evaluated expressions. Alternatively, the questions can be simplified by using both universal and existential quantifiers (the debugger for logic programs only uses universal quantifiers and the debugger for strict functional programs uses no quantifiers).

We have implemented a prototype declarative debugger for a simple functional language which is implemented by transformation into Prolog. The language supports basic pattern matching, if-then-else, higher order functions and lazy evaluation. The basic principles we have presented are applicable to other functional languages also. The important definitions are: the representation of a computation (a tree of terms), the "validity" of an evaluable term (roughly, it evaluates correctly), a bug instance in the computation (a term is not valid but its child is) and the algorithm used to search for a bug instance (we use top down search for simplicity, but any algorithm could be used). We hope to see the development of declarative debuggers for more popular functional programming languages in the near future.

Acknowledgments

This work was supported by a grant from the Australian Research Council.

References

[DL87] Nachum Dershowitz and Yuh-Jeng Lee. Deductive debugging. In *Proceedings of the Fourth IEEE Symposium on Logic Programming*, pages 298–306, San Francisco, California, August 1987.

[DNTM88] Wlodek Drabent, Simin Nadjm-Tehrani, and Jan Maluszynski. The use of assertions in algorithmic debugging. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 573–581, Tokyo, Japan, December 1988.

[Llo87] J.W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[Nai86] Lee Naish. Negation and quantifiers in NU-prolog. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 624–634, Imperial College of Science and Technology, London, England, July 1986. published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Nai91] Lee Naish. Adding equations to NU-Prolog. *Proceedings of The Third International Symposium on Programming Language Implementation and Logic Programming*, pages 15–26, August, 1991. Technical Report 91/2, Department of Computer Science, University of Melbourne.

[Nai92] Lee Naish. Declarative diagnosis of missing answers. *New Generation Computing*, to appear, 1992. Technical Report 88/9 (revised), Department of Computer Science, University of Melbourne.

[NF92] Henrik Nilsson and Peter Fritzson. Algorithmic debugging of lazy functional languages. *Proceedings of The Fourth International Symposium on Programming Language Implementation and Logic Programming*, to appear, August, 1992.

[PC88] Luis Moniz Pereira and Miguel Calejo. A framework for prolog debugging. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, pages 481–495, Seattle, Washington, August 1988.

[Per86] Luis Moniz Pereira. Rational debugging in logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 203–210, London, England, July 1986. published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Sha83] Ehud Y. Shapiro. *Algorithmic program debugging.* MIT Press, Cambridge, Massachusetts, 1983.

[SS86] Leon Sterling and Ehud Shapiro. *The art of Prolog: advanced programming techniques.* Logic Programming series. MIT Press, Cambridge, Massachusetts, 1986.