A three-valued declarative debugging scheme

Lee Naish
Dept. of Computer Science and Software Engineering,
University of Melbourne,
Parkville, 3052, Australia
lee@cs.mu.oz.au, http://www.cs.mu.oz.au/~lee

Abstract

Declarative debugging has many advantages over conventional approaches to debugging for logic and functional programs. This paper extends a previously defined scheme for declarative debugging in which computations were considered either correct or erroneous. We argue that a third value, "inadmissible", should be supported and show how this can be done. Two classes of bugs are defined: one equivalent to the bugs defined by the two valued scheme, the other associated with inadmissibility. It is shown how different instances of the scheme can be used to diagnose type errors, mode errors, violated assertions and abnormal termination as well as the more familiar classes of bugs detected by declarative debuggers.

1 Introduction

Declarative debugging was first introduced in [13] for diagnosing wrong and missing answers in Prolog. The fundamental idea of declarative debugging is that the programmer (or some other *oracle*) has an *intended interpretation* of the program (how the program should behave) and debuggers can query the programmer to obtain this information. The debugger compares the intended interpretation of a (buggy) program with its (incorrect) actual behaviour on some computation. The cause of the difference is isolated to a small section of code which must contain a bug.

Shapiro's work has been extended and refined in many ways. The two developments which are the key to this paper are the early work of Pereira [11] and our relatively recent work [7]. Our work, summarised in the next section, presented a scheme for declarative debugging which generalised many previous approaches. By viewing the problem at a more abstract level, diagnosing a variety of bugs (for example, wrong answers and missing answers) in a variety of languages (for example, logic and functional languages) can be put in the same framework.

One of the contributions of Pereira was to introduce the idea of *inadmissible* calls. In the classical view of logic programming, intended interpretations assign the values true or false to every ground atom. That is, the behaviour of the program (in terms of what things should succeed) is completely specified. The idea of inadmissibility is that the behaviour is not completely specified in the case where procedures are called in unexpected ways (for example, ill-typed calls). The original work of Pereira did not discuss how the concept of inadmissibility fitted with the declarative semantics of logic programs (and he used the term rational debugging rather than declarative debugging). This has now been addressed in a companion paper which gives a detailed account of a three-valued semantics for logic programs [9] inspired by the debugging scheme presented here.

The main contribution of this paper is to combine Pereira's idea of inadmissibility with our high level debugging scheme approach. After describing our previous "two-valued" scheme we point out some deficiencies and show how it can be adapted to support inadmissibility. We then describe several instances of the scheme and show how previously defined type and mode errors can be diagnosed using the scheme. We also show how abnormal termination can be dealt with, an area which is not normally addressed by declarative debugging. Some sections have been condensed due to space limitations; more details can be found in [8]. We assume knowledge of Prolog.

2 The two-valued scheme

The first and simplest use of declarative debugging was for diagnosing wrong answers in Prolog programs. For every successful Prolog computation there is a corresponding *proof tree*. Each node contains an atom which succeeded; the children are the atoms in the body of the matching clause which succeeded. Some atoms may be wrong (not valid in the intended interpretation). If a node is wrong but all its children are correct then the matching clause must contain a bug. Declarative debuggers search for such nodes in the proof tree using information about the intended behaviour of the program (typically asking the user about the correctness of atoms in the tree).

The previous declarative debugging scheme [7] proposed generalises wrong answer diagnosis for Prolog and is based on two assumptions. The first is that a computation can be described using a tree. The root of the tree describes the overall results of the computation and its children describe the result of sub-computations. The second assumption is that the correctness of a computation can be classified as either correct or erroneous. A node in the tree is defined as buggy if it is erroneous but has no erroneous children. Declarative debugging is simply the search for a buggy node in the tree. A simple top-down debugger can be coded in NU-Prolog as follows. It will return all "topmost" buggy nodes (buggy nodes which are not descendants of correct nodes).

```
debug(Root, Bug) :-
    erroneous(Root),
    (if some [Child, Bug1] (
        child(Root, Child),
        debug(Child, Bug1))
    then
        Bug = Bug1
    else
        Bug = Root
    ).
```

The scheme has been applied to a variety of bug classes in several different languages. For wrong answer diagnosis of Prolog the tree is a proof tree and erroneous nodes are those which are not valid in the intended interpretation. For diagnosing wrong answers in functional languages a similar tree is used with nodes containing a function call and a result. Nodes are erroneous if the result is incorrect. For diagnosing missing answers in Prolog a different kind of tree is used. A node contains a call and a set of answers and is erroneous if some correct answers are missing. Clearly separating the tree definition from the way in which the tree is searched allows declarative debugging to be more easily applied and results in simpler, well structured code for debugger.

3 Deficiencies of two values

Consider the following coding of merge sort which (for efficiency) uses insertion sort if the list is short. If the insertion sort code has a bug which results in unsorted lists being returned then `merge` may be called with unsorted lists. If `merge` then succeeds (perhaps with an unsorted list as the result), conventional declarative debugging could conclude that `merge` is buggy.

```
merge_sort(As, SAs) :-
    short_list(As),
    insertion_sort(As, SAs).
merge_sort(As, SAs) :-
    long_list(As), % \+ short_list(As),
    split(As, As1, As2),
    merge_sort(As1, SAs1),
    merge_sort(As2, SAs2),
    merge(SAs1, SAs2, SAs).
```

What is the relationship between the specification, intended interpretation and behaviour of `merge`?[1] We note that discussing the semantics of a procedure which has not actually been defined is unorthodox. However, software engineering tells us that coding first and then thinking about what the code computes is not a good idea. Ideally we should first decide on the intended interpretation, then write the code. The specification certainly should allow an implementation to assume the input lists are sorted (testing them for sortedness would double the number of comparison needed for merge sort). Nearly all implementations of `merge` do not check for sortedness and may succeed with unsorted lists or even non-lists. However, an implementation which did check would also be considered correct. We argue that the intended interpretation of most programmers does not and should not specify exactly which calls to merge should succeed if the inputs are not sorted lists. Such behaviour depends only on coding details.

However, programmers are confronted with such questions when using (two-valued) declarative debuggers. We illustrate some of the issues in more detail with another example where sortedness is important. In the following buggy program there is a confusion between representing sets as unsorted lists and sorted lists (`oset` stands for ordered set). The procedures `gtmax` and `ltmax` both use the variable `SA` in an inconsistent way.

```
% A > max element in nonempty list As
gtmax(A, As) :-           % should use
    list_to_set(As, SA),  % oset here
    oset_max(SA, Max),    % or set here
    A > Max.

% A < max element in nonempty list As
ltmax(A, As) :-           % should use
    list_to_set(As, SA),  % oset here
    oset_max(SA, Max),    % or set here
    A < Max.
```

[1] This question lead to our previous work on types and ultimately to this paper

```
% As is a list; S is a list
% representation of the set of
% elements in As (ie, just As)
list_to_set(As, S) :- S = As.

% As is a list; S is the sorted list
% representation of the set of
% elements in As
list_to_oset(As, S) :- sort(As, S)

% S is the sorted list representation
% of a set;  M is the maximum element
oset_max(S, M) :- append(_, [M], S).
```

The goal `gtmax(2,[2,3,1])` succeeds incorrectly: `list_to_set` binds `SA` to `[2,3,1]` then the call `oset_max([2,3,1],Max)` binds `Max` to 1. The proof tree is shown in Figure 1, with erroneous nodes in bold type. According to the classical view of logic programming and declarative debugging `oset_max([2,3,1],1)` is not valid in the intended interpretation and so the definition must be buggy. The body of the clause succeeds with the instance `append([2,3],[1],[1,2,3])`, which is valid, so we have found an incorrect clause instance. This is a very unsatisfactory explanation of the bug. The definition of `oset_max` assumes it is called with a sorted list and an explicit check for this condition would significantly reduce efficiency.

If we use a somewhat less natural intended interpretation and say that all `oset_max` atoms with unsorted lists are true rather than false, the error is correctly diagnosed in this case. However, missing answers are then diagnosed incorrectly (see Section 5.2). To avoid incorrect diagnosis in general, calls which violate assumptions must be considered true if they succeed and false if they fail [4]. Thus the intended interpretation must depend on the implementation rather than the other way around. It is not possible to first decide on a simple intuitive intended behaviour for the program then derive an efficient implementation.

Some early work on declarative debugging introduced the idea of *inadmissible calls* [11], though no declarative meaning was given. Subsequent work built on these ideas [10] and work on types [4] gave a declarative reconstruction of this idea, identifying inadmissible calls with ill-typed atoms (where types could be defined using arbitrary predicates). This was the inspiration for the work presented here. Subsequently we devised a more theoretical basis for this work, exploring model-theoretic, procedural and fixed-point semantics in a three-value setting [9].

4 The three-valued scheme

We now describe our three-valued declarative debugging scheme. As in the two-valued scheme, the computation must be represented by a tree. Associated with each node is one of three possible truth values: *correct*, *erroneous* or *inadmissible*. Roughly speaking, inadmissible nodes correspond to computations for which some assumption has been violated (for example, the "input arguments" are not "well typed"). Erroneous nodes correspond to computations for which no assumption has been violated but the result is incorrect. Correct nodes correspond to correct computations.

We define two kinds of buggy nodes. The first is equivalent to the definition for the two-valued scheme. It corresponds to a bug which causes the results of correct subcomputations to be combined and returned in the wrong way.

Definition A node in a tree is *e-buggy* if it is erroneous but all children are correct.

The second kind of buggy node definition involves inadmissibility. It corresponds to the combination of subcomputations violating some assumption. It is consistent with the "inadmissible call bug" definition of [10], if we assume inadmissible calls are not considered "bug manifestations".

Definition A node in a tree is *i-buggy* if it is erroneous and has an inadmissible child and no erroneous children.

Consider the `gtmax` example. If we define `oset_max(S,M)` to be inadmissible iff `S` is not a sorted list, the `oset_max` node in the tree is inadmissible. The parent node, `gtmax`, is erroneous but has no erroneous children and hence is i-buggy. Thus a debugger based on this scheme would correctly locate the buggy clause and could provide information about the offending instance and what atoms were inadmissible.

The following top-down debugger is a simple adaptation of the debugger presented for the two-valued scheme. Given a tree it will return all topmost buggy nodes in the form `e_bug(Node)` (`Node` is e-buggy) or `i_bug(Node,Child)` (`Node` is i-buggy and `Child` is an inadmissible child).

```
debug(Root, Bug) :-
    erroneous(Root),
    (if some [Child, Bug1] (
        child(Root, Child),
        debug(Child, Bug1))
    then
        Bug = Bug1
    else if some [Child] (
        child(Root, Child),
        inadmissible(Child))
    then
        Bug = i_bug(Root, Child)
    else Bug = e_bug(Root)).
```

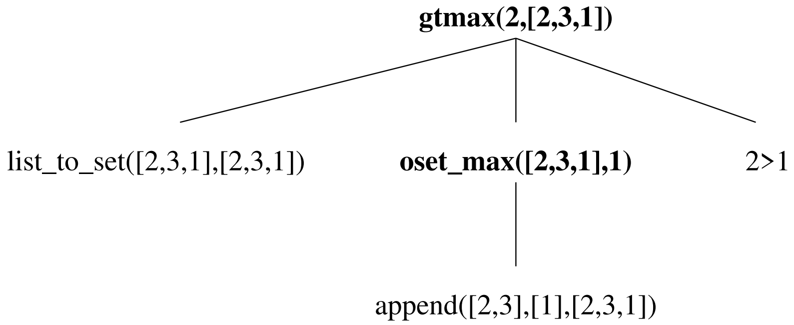

Figure 1. Proof tree for `gtmax`

Note that buggy node definitions and this debugger in some sense pay more attention to erroneous nodes than inadmissible nodes: an inadmissible node with erroneous siblings is ignored. The reason for this is that there are generally dependencies between siblings. An erroneous result from one node can be the cause of a sibling node being inadmissible (an example is discussed in Section 5.3). An inadmissible node should not cause a sibling to be erroneous, though it may cause a sibling node to be inadmissible.

If an erroneous node has no erroneous children but several inadmissible children then each such child shows the existence of a bug. Ideally, information on dependencies between these nodes can be used to find which node is the source of the error. The debugger above simply returns all the nodes using backtracking.

5 Instances of the scheme

We now present several instances of the three-valued debugging scheme to illustrate its usefulness and flexibility. Like [7], it can also be adapted to diagnose similar categories of bugs in other declarative languages.

5.1 Types — wrong answers

A very flexible approach to types in logic programming is presented in [4]. The main motivation was to clarify the intended declarative meaning of programs and reconcile the fact that typical logic programs have atoms which succeed even though they are not true in the intended interpretation (for example, calls to `merge` where arguments are not lists). The programmer supplies type declarations which define the set of *well typed* ground atoms (the *type set*). The declarative meaning of a program is given by a modified version of the program which has type checks added to every clause. Well typed answers computed by *type correct* programs are guaranteed to be correct according to this declarative semantics. That is, the type checks need not be executed at runtime.

A definite program is type correct if for every ground clause instance with a well typed head and a body which succeeds, the body is well typed [4]. The suggested type definition language was Horn clauses. This allows definition of the usual classes of types such as lists and trees and also allows more complex "types" to be defined, such as sorted lists. Type correctness is an undecidable property (even for very simple classes of types) so some runtime tools are desirable to detect and locate type-related bugs. One possibility is to perform the type checks at runtime and generate an error if they fail. There are some practical difficulties associated with this, related to execution order and implementation of negation. Also, the location of the failed type check does not precisely indicate the bug location — the type problem may be caused by a previous call returning the wrong answer.

The three-valued debugging scheme can be applied to debugging ground wrong answers with this model of types as follows. The tree is a Prolog proof tree. Nodes containing ill-typed atoms are inadmissible; other nodes are correct or erroneous depending on their validity in the intended interpretation. E-buggy nodes correspond to clause instances which are not true in the intended interpretation: the head is not valid but the body is valid. I-buggy nodes correspond to clause instances which are not type correct: the head is well typed and the body succeeds but is not well typed. For example, the type system can express the inadmissibility in the `gtmax` program. The buggy `gtmax` node corresponds to the following clause instance, which is not type correct:

```
gtmax(2, [2,3,1]) :-
    list_to_set([2,3,1], [2,3,1]),
    oset_max([2,3,1], 1),
    2 > 1.
```

For non-ground answers a node can be considered inadmissible if *some*[2] ground instance is ill-typed. I-buggy

[2] A universal quantifier here results in a less strict definition of admissibility but this can cause problems if programs contain negation and quantifiers.

nodes correspond to (possibly non-ground) clause instances which have a further (ground) instance which is not type correct.

5.2 Types — missing answers

Type errors resulting in wrong answers are discussed in [4] but missing answers are ignored. However, the two-valued debugging scheme has been applied to diagnosing missing answers and can be adapted. A (possibly failed) computation is represented using a tree as follows. Each node in the tree contains an atom (a call) and a set of instances of the atom (the answers returned). The children are all the calls made to atoms in the bodies of matching clauses. An sample tree is given in Figure 2. In the two valued scheme a node is erroneous if the set of answer atoms is incomplete. The third value can be introduced by saying that a node is inadmissible if *all* ground instances of the call atom are ill-typed.

The tree in Figure 2 represents the failed execution of `ltmax(2,[2,3,1])`. With the three-valued scheme the `oset_max` node is inadmissible and the bug is correctly diagnosed in `ltmax`. Using the two-valued scheme, if ill-typed atoms are considered true then the diagnosis would be that `oset_max([2,3,1],Y)` misses the answer `Y=3` (the `oset_max` node is buggy). See [7] or [3] for a more technical description of this class of bugs. If ill-typed atoms were considered false then the better declarative debuggers (which combine wrong and missing answer diagnosis) would still conclude `oset_max` is buggy due to the wrong answer `oset_max([2,3,1],1)`.

Note that despite the ill-typed call, the clause is type correct. The bug can only lead to missing answers, not wrong answers, and type correctness is related to wrong answers (when negation is present there is a stricter definition [4]). Nevertheless, types allow us to diagnose the error better.

5.3 Directed types and modes

In the directed types framework [1][12] each argument of each predicate is declared to be a particular type and mode (either input or output). Various sufficient conditions have been defined to ensure well-behaved programs. The basic intuition is that calls with well typed inputs should

1. produce only well typed outputs, and
2. result in only well typed calls in the body of any matching clause (which succeeds[3]).

The declarative view of modes presented in [6] can be seen as a higher level version of directed types. Mode declarations define a set of atoms (the *mode set*) which should be a superset of the success set. If it is not a superset of the success set then mode errors may result (for example, a well typed call may produce ill-typed answers).

Consider the following variation on the `gtmax` program. Declarations have been added to define the type (integers, lists of integers and sorted lists of integers) and modes. The code behaves in the same way as previously but due to a bug in `sort` rather than a confusion over set representation.

```
% A > max element in nonempty list As
:- pred gtmax(int:in, int_list:in).
gtmax(A, As) :-
    list_to_oset(As, SA), % bug fixed!
    oset_max(SA, Max),
    A > Max.

% as before
:- pred list_to_oset(int_list:in,
                     sorted_list:out).
list_to_oset(As, S) :- sort(As, S)

% As is a list; S is sorted version
:- pred sort(int_list:in,
             sorted_list:out).
sort(As, S) :- S = As. % obvious bug
```

The proof tree for `gtmax(2,[2,3,1])` is shown in Figure 3. Considering types only, both the `list_to_oset` and `oset_max` are inadmissible and hence the bug would be diagnosed in `gtmax`. This is not a good diagnosis but it seems difficult to do better without additional information. The type scheme makes no assumptions about the execution strategy — it could be top down, bottom up, left to right, right to left or any combination. Similarly, the proof tree does not reflect the execution strategy. There is no way a debugger could determine how the unsorted list was introduced to an argument position where a sorted list was expected. Without viewing the program source it may have come directly from the call to `gtmax` and, in fact, the clause instance for `gtmax` is not type correct.

Not surprisingly, the diagnosis can be improved if more (declarative) information is provided. Using modes we can refine the definition of admissibility. A ground ill-typed atom should be considered inadmissible if the *input* arguments are ill-typed (technically, if it is in the mode set) otherwise it should be considered erroneous. Atoms with inadmissible instances are inadmissible and other atoms with erroneous instances are erroneous. Using this definition the `list_to_oset` node is erroneous rather than inadmissible. The inadmissibility of the `oset_max` node is therefore ignored and the bug is correctly diagnosed in the definition of `sort`.

[3] The condition that the clause succeeds is often ignored due to the focus on decidable sufficient conditions for correctness.

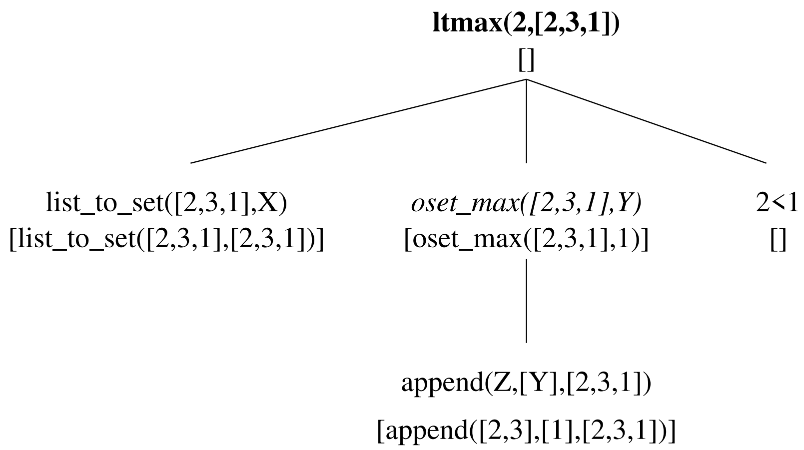

Figure 2. Missing answer tree for `ltmax`

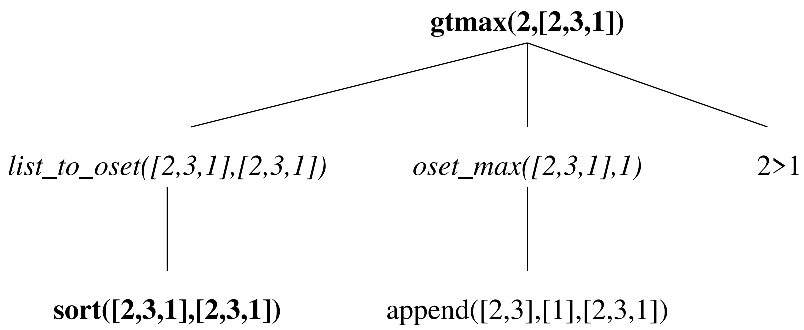

Figure 3. Proof tree for (revised version of) `gtmax`

This is an instance of the first condition of directed types being violated: the debugger finds a successful atom (`sort([2,3,1], [2,3,1])`) with well typed inputs but ill-typed outputs. From the perspective of declarative modes, this atom is not in the mode set (so the mode set is not a superset of the success set). Such bugs correspond to e-bugs in (this instance of) our debugging scheme.

Violation of the second condition of directed types corresponds to an i-bug. Our first `gtmax` program is an example of this. The declarative mode scheme is a refinement of our type scheme and from that perspective, i-bugs correspond to clause instances which are not type correct, as before.

5.4 Assertions

As discussed in [5], flexible types can be used in a similar way to assertions in imperative languages. Other forms of assertions have also been proposed for logic programs [2]. These assertions can express things such as a certain argument is a variable at the time of a call, which cannot be expressed by the type scheme. The standard top-down left to right execution of Prolog is assumed since this meta-level information depends on the execution order. Although it is against the spirit of purely declarative debugging, these meta-level assertions can be modelled by our three-valued debugging scheme.

Instead of a standard proof tree, each node contains two atoms. One is the instantiation of the atom at the time it was called, the other is the instantiation of the atom at the time it succeeded (in proof trees the instantiation at the end of the whole proof is used). The children of a node are the atoms in the body of the clause which was used, as in proof trees. Different execution orders result in trees which have the same shape but have different instantiations for the atoms other than those in the root.

Inadmissibility generally depends on the call atom in a node — does it satisfy some assertion or *precondition* of the predicate. Correctness of admissible nodes depends on the successful instance of the atom — is the instance valid or does it satisfy some *postcondition*. The two classes of bugs in (this instance of) our scheme correspond to failure of these two kinds of conditions. An e-bug corresponds to an unsatisfied postcondition, where all subcomputations are correct. An i-bug corresponds to an unsatisfied precondition in a subcomputation, where the enclosing computation has a satisfied precondition.

5.5 Abnormal termination

Computations which abort or generate exceptions due to division by zero, ill-typed or insufficiently instantiated calls et cetera, can be diagnosed using a slight extension of the method given in the previous section. Such errors can be seen as instances of assertions (preconditions) being violated and the calls can be treated as inadmissible. The tree which represents the computation can be generalised to allow for aborted computations in the following way. A node corresponding to a call which immediately aborts has no children, all nodes on the path back to the root contain the atom `abort` instead of a successful instance of the call, and there are no further branches to the right of this path. If we assume calls which immediately abort are inadmissible and any admissible calls which terminate abnormally are erroneous then the following search strategy is likely to be superior to the simple top-down search. Search up the rightmost branch of the tree (the leaf is the inadmissible aborting call) until an admissible node is found (it will be erroneous since the computation aborted). If any child is erroneous, search this subtree in the normal way (the subcomputation terminated normally with a wrong answer), otherwise we have found an i-bug (an erroneous node with no erroneous children but an inadmissible child).

5.6 Suspected infinite loops

If a programmer suspects a Prolog program is looping "infinitely" then the program is normally interrupted. The computation which was performed, and which terminated abnormally, can be represented as a tree as described in the previous section. However, admissible and erroneous nodes must be defined in a more subtle way We will discuss loops with reference to the following code.

```
p(...) :- q(...).
q(...) :- r(...), q(...). % loop here
...
```

The computation may be interrupted within the execution of `r`, and this call may have terminated normally if it had been allowed to proceed. Thus a call terminating abnormally does not necessarily indicate an error at that point. Assuming the whole computation was "non-terminating", some ancestor node (`q`) must be a call which would not have terminated, possibly because the call had unexpected arguments (for example, the arguments may not have been insufficiently instantiated). Such a call should be considered inadmissible. Higher ancestors, such as `p`, may be admissible (but not terminate normally). These nodes should be considered erroneous.

Thus all three truth values can apply to nodes which did not terminate normally. The relationship between intended behaviour, actual behaviour and the truth value can be defined as follows:

- A node is admissible if the call should terminate normally (for all solutions) according to the *intended* behaviour of the program. For example, we could have a definition of `reverse/2` with the intention that it works with either argument a ground list. Calls such as `reverse(X,[1,2])` would be admissible but `reverse(X,Y)` would be inadmissible.

- An admissible node is erroneous if the *actual* behaviour of the call is to loop "infinitely". Nodes may also be erroneous due to wrong answers. For example, if `reverse(X,[1,2])` actually loops or returns the answer `X=[]` it would be erroneous. Note that determining if a node is erroneous is not decidable (something must be undecidable when diagnosing infinite loops). Tools should be available to help the programmer determine the behaviour.

The search strategy presented in the previous section can be applied with the following addition: initially, we must search up the rightmost branch for an inadmissible node (since the leaf may not be inadmissible). We note that representing a user-aborted execution tree presents a challenge. In an emulated system an instruction count may provide a useful mechanism for implicit representation, allowing re-execution to the point of the abort.

6 Conclusion

Programmers generally write code with some assumptions about how it is to be used. If the code is used in an unexpected way, even the programmer is unlikely to know in advance how it will behave. Prolog code is no exception. This is not acknowledged in the conventional approach to the semantics of logic programming: intended interpretations are assumed to be two-valued. Strong typing goes some way towards eliminating "unexpected" calls but many assumptions cannot be expressed in decidable type systems. It is impossible to completely avoid unexpected calls so it is desirable for debuggers to be designed with this in mind.

We have devised a declarative debugging scheme which supports the idea of inadmissible subcomputations. This leads to two classes of bugs. From an algorithmic perspective they are best thought of as bugs concerning preconditions and postconditions, respectively. Using information which depends on Prolog execution strategy, the scheme can be applied to programs which terminate abnormally, either due to predicates aborting or (apparent) non-termination. At a purely declarative level (independent of the execution strategy) definitions of admissibility can be found which support debugging using existing type and mode systems. The two classes of bugs then correspond to previously defined classes of errors, including type and mode errors. Type information enables the debugger to produce better bug diagnoses compared with the two-valued scheme. Mode information can further improve diagnoses.

Fundamentally, we believe that intended interpretations are naturally three-valued and this should be reflected directly in debugging environments. This has lead to new theoretical work on semantics of logic programs which gives a solid theoretical basis for the debugging scheme.

Acknowledgements

This work was supported by a grant from the ARC

References

[1] J. Boye and J. Maluszynski. Two aspects of directional types. In L. Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming*, pages 747–761, Kanagawa, Japan, June 1995.

[2] W. Drabent and J. Maluszynski. Inductive assertions method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.

[3] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. The use of assertions in algorithmic debugging. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 573–581, Tokyo, Dec. 1988.

[4] L. Naish. Types and the intended meaning of logic programs. In F. Pfenning, editor, *Types in logic programming*, pages 189–216. MIT Press, Cambridge, Massachusetts, 1992.

[5] L. Naish. Verification of logic programs and imperative programs. In J.-M. Jacquet, editor, *Constructing logic programs*, pages 143–164. Wiley, Chichester, England, 1993.

[6] L. Naish. A declarative view of modes. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 185–199. MIT Press, September 1996.

[7] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.

[8] L. Naish. A three-valued declarative debugging scheme. Technical Report 97/5, Dept. of Computer Science, University of Melbourne, Australia, May 1997.

[9] L. Naish. A three-valued semantics for horn clause programs. In *this volume*, January 2000.

[10] L. Pereira and M. Calejo. A framework for Prolog debugging. In K. Bowen and R. Kowalski, editors, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, pages 481–495, Seattle, Washington, Aug. 1988.

[11] L. M. Pereira. Rational debugging in logic programming. In E. 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.

[12] Y. Rouzaud and L. Nguyen-Phuong. Integrating modes and subtypes into a Prolog type-checker. In *Proceedings of the Ninth Joint International Conference and Symposium on Logic Programming*, pages 85–97, 1992.

[13] E. Y. Shapiro. *Algorithmic program debugging*. MIT Press, Cambridge, Massachusetts, 1983.