Declarative Diagnosis of Floundering in Prolog

Lee Naish
Department of Computing and Information Systems
The University of Melbourne, Victoria 3010, Australia
lee@cs.mu.oz.au

Abstract

Many logic programming languages have delay primitives which allow coroutining. This introduces a class of bug symptoms — computations can *flounder* when they are intended to succeed or finitely fail. For concurrent logic programs this is normally called *deadlock*. Similarly, constraint logic programs can fail to invoke certain constraint solvers because variables are insufficiently instantiated or constrained. Diagnosing such faults has received relatively little attention to date. Since delay primitives affect the procedural but not the declarative view of programs, it may be expected that debugging would have to consider the often complex details of interleaved execution. However, recent work on semantics has suggested an alternative approach. In this paper we show how the declarative debugging paradigm can be used to diagnose unexpected floundering, insulating the user from the complexities of the execution.
Keywords: logic programming, coroutining, delay, debugging, floundering, deadlock, constraints

1 Introduction

The first Prolog systems used a strict left to right evaluation strategy, or computation rule. However, since the first few years of logic programming there have been systems which support coroutining between different sub-goals (Clark & McCabe 1979). Although the default order is normally left to right, individual calls can *delay* if certain arguments are insufficiently instantiated, and later *resume*, after other parts of the computation have further instantiated them. Such facilities are now widely supported in Prolog systems. They also gave rise to the class of *concurrent* logic programming languages, such as Parlog (Gregory 1987), where the default evaluation strategy is parallel execution and similar delay mechanisms are used for synchronisation and prevention of unwanted nondeterminism. Delay mechanisms have also been influential for the development of *constraint logic programming* (Jaffar & Lassez 1987). Delays are often used when constraints are "too hard" to be handled by efficient constraint solvers, for example, non-linear constraints over real numbers.

Of course, more features means more classes of bugs. In theory, delays don't affect soundness of Prolog[1] (see (Lloyd 1984)) — they can be seen as affecting the "control" of the program without affecting the logic (Kowalski 1979). However, they do introduce a new class of bug symptoms. A call can delay and never be resumed (because it is never sufficiently instantiated); the computation is said to *flounder*. Most Prolog systems with delays still print variable bindings for floundered derivations in the same way as successful derivations (in this paper we refer to these as "floundered answers"), and may also print some indication that the computation floundered. Floundered answers are not necessarily valid, or even satisfiable, according to the declarative reading of the program, and generally indicate the presence of a bug. In concurrent logic programs the equivalent of floundering is normally called *deadlock* — the computation terminates with no "process" (call) sufficiently instantiated to proceed. In constraint logic programming systems, the analogue is a computation which terminates with some insufficiently instantiated constraints not solved (or even checked for satisfiability). Alternatively, if some constraints are insufficiently instantiated they may end up being solved by less efficient means than expected, such as exhaustive search over all possible instances.

There is a clear need for tools and techniques to help diagnose floundering in Prolog (and analogous bug symptoms in other logic programming languages), yet there has been very little research in this area to date. There has been some work on showing floundering is impossible using syntactic restrictions on goals and programs (particularly logic databases), or static analysis methods (for example, (Marriott, Søndergaard & Dart 1990)(Marriott, García de la Banda & Hermenegildo 1994)). However, this is a far cry from general purpose methods for diagnosing floundering. In this paper we present such a method. Furthermore, it is a surprisingly attractive method, being based on the declarative debugging paradigm (Shapiro 1983) which is able to hide many of the procedural details of a computation. Declarative debugging has been widely used for diagnosing wrong answers in programming languages based on (some combination of) the logic, functional and constraint paradigms (Pope & Naish 2003, Caballero, Rodríguez-Artalejo & del Vado Vírseda 2006) and there has been some work on diagnosing missing answers (Naish 1992) (which mentions some problems caused by coroutining) (Caballero, Rodríguez-Artalejo & del Vado Vírseda 2007), pattern match failure (Naish & Barbour 1995) and some other bug symptoms (Naish 1997). However, floundering is not among the symptoms previously diagnosed using this approach.

The paper is structured as follows. We first give some examples of how various classes of bugs can lead to floundering. We then present our method of diagnosing floundering, give examples, and discuss how our simple prototype could be improved. Next we

[1]In practice, floundering within negation can cause unsoundness.

```
% perm(As0, As): As = permutation of
% list As0
% As0 or As should be input
perm([], []).
perm([A0|As0], [A|As]) :-
    when((nonvar(As1) ; nonvar(As)),
        inserted(A0, As1, [A|As])),
    when((nonvar(As0) ; nonvar(As1)),
        perm(As0, As1)).

% inserted(A, As0, As): As = list As0
% with element A inserted
% As0 or As should be input
inserted(A, As0, [A|As0]).
inserted(A, [A1|As0], [A1|As]) :-
    when((nonvar(As0) ; nonvar(As)),
        inserted(A, As0, As)).
```

Figure 1: A reversible permutation program

```
% Bug 1: wrong variable AS0 in recursive
% call
inserted(A, As0, [A|As0]).
inserted(A, [A1|As0], [A1|As]) :-
    when((nonvar(As0) ; nonvar(As)),
        inserted(A, AS0, As)).      % XXX

% Bug 2: wrong variable A in when/2
inserted(A, As0, [A|As0]).
inserted(A, [A1|As0], [A1|As]) :-
    when((nonvar(As0) ; nonvar(A)), % XXX
        inserted(A, As0, As)).

% "Bug" 3: assumes As0 is input       XXX
% (perm/2 intended modes incompatible)
inserted(A, As0, [A|As0]).
inserted(A, [A1|As0], [A1|As]) :-
    when(nonvar(As0),               % XXX
        inserted(A, As0, As)).
```

Figure 2: Buggy versions of inserted/3

briefly consider some more theoretical aspects, then conclude. Basic familiarity of Prolog with delays and declarative debugging is assumed.

2 Example

Figure 1 gives a permutation program which has simple logic but is made reversible by use of delaying primitives and careful ordering of sub-goals in `perm/2` (see (Naish 1986) for further discussion). The delay primitive used is the "when meta-call": a call `when(Cond,A)` delays until condition `Cond` is satisfied, then calls `A`. For example, the recursive call to `perm/2` will delay until at least one of its arguments are non-variables. Generally there are other features supported, such as delaying until a variable is ground; we don't discuss them here, though our method and prototype support them. A great number of delay primitives have been proposed (Clark & McCabe 1979, Naish 1986). Some, like the when meta-call, are based on calls. Others are based on procedures (affecting all calls to the procedure), which is often more convenient and tends to clutter the source code less. Our general approach to diagnosis is not affected by the style of delay primitive. The when meta-call is by far the most portable of the more flexible delay primitives, which is our main reason for choosing it. We have developed the code in this paper using SWI-Prolog.

We consider three separate possible bugs which could have been introduced, shown in Figure 2. They exemplify three classes of errors which can lead to floundering: logical errors, incorrect delay annotations and confusion over the modes of predicates. Bug 1 is a logical error in the recursive call to `inserted/3`. Such errors can cause wrong and missing answers as well as floundering. Due to an incorrect variable name, other variables remain uninstantiated and this can ultimately result in floundering. This bug can be discovered by checking for singelton variables, so in practice declarative debugging should not be required, but we use it as a simple illustration of several points. Despite the simplicity of the bug and the program, a complex array of bug symptoms results, which can be quite confusing to a programmer attempting to diagnose the problems.

```
?- perm([1,2,3],A).
Call: perm([1,2,3],_G0)
Call: when(...,inserted(1,_G1,[_G2|_G3]))
Exit: when(...,inserted(1,_G1,[_G2|_G3]))
Call: when(...,perm([2,3],_G1))
Call: perm([2,3],_G1)
Call: inserted(1,[_G4|_G5],[_G2|_G3])
Exit: inserted(1,[_G4|_G5],[1,_G4|_G5])
Call: when(...,inserted(2,_G6,[_G4|_G5]))
Exit: when(...,inserted(2,_G6,[_G4|_G5]))
Call: when(...,perm([3],_G6))
Call: perm([3],_G6)
Call: inserted(2,[_G7|_G8],[_G4|_G5])
Exit: inserted(2,[_G7|_G8],[2,_G7|_G8])
Call: when(...,inserted(3,_G9,[_G7|_G8]))
Exit: when(...,inserted(3,_G9,[_G7|_G8]))
Call: when(...,perm([],_G9))
Call: perm([],_G9)
Call: inserted(3,[],[_G7|_G8])
Exit: inserted(3,[],[3])
Exit: perm([],[])
Exit: when(...,perm([],[]))
Exit: perm([3],[3])
Exit: when(...,perm([3],[3]))
Exit: perm([2,3],[2,3])
Exit: when(...,perm([2,3],[2,3]))
Exit: perm([1,2,3],[1,2,3])
```

Figure 3: Trace with delayed and resumed calls

The call `perm([1,2,3],A)` first succeeds with answer `A=[1,2,3]`, which is correct. Figure 3 shows the execution trace generated using SWI-Prolog (some details on each line are removed to save space). The trace is the same for all versions of the program. The first call to `inserted/3` (wrapped in a when annotation) delays, shown in the first `Exit` line of the trace. It is resumed immediately after the recursive call to `perm([2,3],_G1)` because matching with the clause head for `perm/2` instantiates `_G1`. Subsequent calls to `inserted/3` also delay and are resumed after further recursive calls to `perm/2`. In this case, all resumed subcomputations immediately succeed (they match with the fact for `inserted/3`). To find other permutations the recursive clause for `inserted/3` must be selected, and the resumed subcomputations delay again when `inserted/3` is called recursively.

On backtracking, for Bug 1, there are four other successful answers found which are satisfiable but not valid, for example, `A=[1,2,3|_]` and `A=[3,1|_]`. An atomic formula, or atom, is *satisfiable* is some instance is true according to the programmer's intentions and *valid* if all instances are true. These answers could be diagnosed by existing wrong answer declarative debugging algorithms, though some early approaches assumed bug symptoms were unsatisfiable atoms (Naish 1997). An atom is *unsat-*

```
?- perm([A,1|B],[2,3]).
Call: perm([_G0,1|_G1],[2,3])
Call: when(...,inserted(_G0,_G2,[2,3]))
Call: inserted(_G0,_G2,[2,3])
Exit: inserted(2,[3],[2,3])
Exit: when(...,inserted(2,[3],[2,3]))
Call: when(...,perm([1|_G1],[3]))
Call: perm([1|_G1],[3])
Call: when(...,inserted(1,_G3,[3]))
Call: inserted(1,_G3,[3])
Call: when(...,inserted(1,_G7,[]))
Call: inserted(1,_G7,[])
Fail: inserted(1,_G7,[])
Fail: when(...,inserted(1,_G7,[]))
Fail: inserted(1,_G3,[3])
Fail: when(...,inserted(1,_G3,[3]))
Fail: perm([1|_G1],[3])
Fail: when(...,perm([1|_G1],[3]))
Redo: inserted(_G0,_G2,[2,3])
Call: when(...,inserted(_G0,_G4,[3]))
Call: inserted(_G0,_G4,[3])
Exit: inserted(3,[],[3])
Exit: when(...,inserted(3,[],[3]))
Exit: inserted(3,[2|_G5],[2,3])
Exit: when(...,inserted(3,[2|_G5],[2,3]))
Call: when(...,perm([1|_G1],[2|_G5]))
Call: perm([1|_G1],[2|_G5])
Call: when(...,inserted(1,_G6,[2|_G5]))
Exit: when(...,inserted(1,_G6,[2|_G5]))
Call: when(...,perm(_G1,_G6))
Exit: when(...,perm(_G1,_G6))
Exit: perm([1|_G1],[2|_G5])
Exit: when(...,perm([1|_G1],[2|_G5]))
Exit: perm([3,1|_G1],[2,3])
```

Figure 4: Trace for Bug 1

isfiable if no instance is true according to the programmer's intentions. These answers are interleaved with four floundered answers, such as A=[1,3,_|_], which are also satisfiable but not valid — when inserted/3 is called recursively, As0 remains uninstantiated because the incorrect variable is used, and after several more calls and some backtracking, floundering results. The call perm(A,[1,2,3]) succeeds with the answer A=[1,2,3] then has three floundered answers, also including A=[1,3,_|_]. The call perm([A,1|B],[2,3]) is unsatisfiable and should finitely fail but returns a single floundered answer with A=3. Figure 4 gives a trace of this computation.

Diagnosing floundering using execution traces is also made more challenging by backtracking. In Figure 4, lines 13–18 are failures, which cause backtracking over previous events. Another complicating factor is that when a predicate exits, it may not have completed execution. There may be subcomputations delayed which are subsequently resumed (see Figure 3), and these resumed subcomputations may also have delayed parts, etc — there can be coroutining between multiple parts of the computation. There are typically Exit lines of the trace which are not valid but are correct because the subcomputation floundered rather than succeeded (often this is not immediately obvious from the trace).

With Bug 2, an incorrect delay annotation on the recursive call to inserted/3, several bug symptoms are also exhibited. The call perm([X,Y,Z],A) behaves correctly but perm([1,2,3],A) succeeds with the answers A=[1,2,3] and A=[1,3,2], then loops indefinitely. We don't consider diagnosis of loops in this paper, though they are an important symptom of incorrect control. The call perm(A,[1,2,3]) succeeds with the answer A=[1,2,3] then has three further floundered answers, A=[1,2,_,_|_], A=[1,_,_|_] and A=[_,_|_], before terminating with failure.

Bug 3 is a more subtle control error. When inserted/3 was coded we assume the intention was the second argument should always be input, and the delay annotation is correct with respect to this intention. This is a reasonable definition of inserted/3 and we consider it is correct. However, some modes of perm/2 require inserted/3 to work with just the third argument input. When coding perm/2 the programmer was either unaware of this or was confused about what modes inserted/3 supported. Thus although we have modified the code for inserted/3, we consider the bug to be in perm/2. This version of the program behaves identically to Bug 2 for the goal perm(A,[1,2,3]), but the bug diagnosis will be different because the programmer intentions are different. The mistake was made in the coding of perm/2, and this is reflected in the diagnosis. The simplest way to *fix* the bug is change the intentions and code for inserted/3, but we only deal with diagnosis in this paper.

Because delays are the basic cause of floundering and they are inherently procedural, it is natural to assume that diagnosing unexpected floundering requires a procedural view of the execution. Even with such a simple program and goals, diagnosis using just traces of floundered executions can be extremely difficult. Subcomputations may delay and be resumed multiple times as variables incrementally become further instantiated, and this can be interleaved with backtracking. Reconstructing how a single subcomputation proceeds can be very difficult. Although some tools have been developed, such as printing the history of instantiation states for a variable, diagnosis of floundering has remained very challenging.

3 Declarative diagnosis of floundering

To diagnose unexpected floundering in pure Prolog programs with delays we use an instance of the three-valued declarative debugging scheme described in (Naish 2000). We describe the instance precisely in the following sections, but first introduce the general scheme and how it is applied the more familiar problem of diagnosing wrong answers. A computation is represented as a tree, with each node associated with a section of source code (a clause in this instance) and subtrees representing subcomputations. The choice of tree generally depends in the language and the bug symptom. For example, diagnosing wrong answers in Prolog generally uses proof trees (see Section 3.1) whereas diagnosing pattern match failure in a functional language requires a different kind of tree. The trees we use here are a generalisation of proof trees. The debugger searches the tree and eventually finds a node for which the associated code has a bug.

Each node has a truth value which expresses how the subcomputation compares with the intentions of the programmer. Normally the truth values of only some nodes are needed to find a bug and they are determined by asking the user questions. Three truth values are used for tree nodes: *correct*, *erroneous*, and *inadmissible*. To diagnose wrong answers, the user is asked about the atoms in proof tree nodes, which were proved in the computation. Correct nodes are those containing an atom which is valid in the intended interpretation, such as inserted(2,[1],[1,2]). The corresponding subcomputation returned the correct result and the subtree is eliminated from the search space, since it cannot be the cause of a wrong answer at the root of the tree. Erroneous nodes correspond to subcomputations which returned a wrong answer, such as inserted(2,[1],[1]). If such a

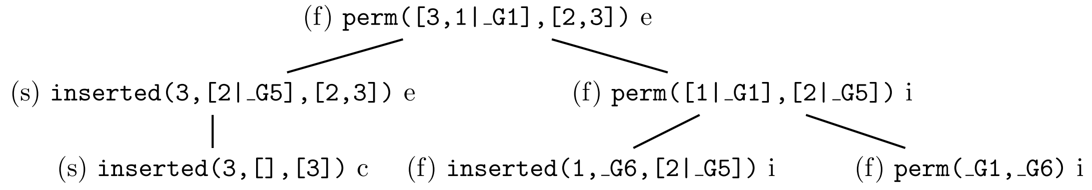
(f) perm([3,1|_G1],[2,3]) e
(s) inserted(3,[2|_G5],[2,3]) e
(f) perm([1|_G1],[2|_G5]) i
(s) inserted(3,[],[3]) c
(f) inserted(1,_G6,[2|_G5]) i
(f) perm(_G1,_G6) i

Figure 5: A Partial proof tree for Bug 1

node is found, the search space can be restricted to that subtree, since it must contain a bug. Inadmissible nodes correspond to subcomputations which should never have occurred. Inadmissibility was initially used to express the fact that a call was ill-typed (Pereira 1986). For example, `inserted` is expected to be called with a list in the second and/or third argument, so `inserted(2,a,[2|a])` would be considered inadmissible. Inadmissible means a *pre-condition* of the code has been violated, whereas erroneous means a *post-condition* has been violated. For inadmissible nodes, the subtree can be eliminated from the search space in the same way as a correct node. Here calls which flounder because they never become sufficiently instantiated are considered inadmissible.

Given a tree with truth values for each node, a node is *buggy* if it is erroneous but has no erroneous children. The computation associated with the node behaved incorrectly but none of its subcomputations behaved incorrectly, so there must be a bug in the code associated with that node. Buggy nodes can be classified into to kinds, depending on whether or not they have any inadmissible children. If there are no inadmissible children (all children are correct) the code has simply produced the wrong result, called an e-bug in (Naish 2000). If there are inadmissible children the code has resulted in other code being used in an unintended way, violating a pre-condition. This is called an i-bug. Diagnosis consists of searching the tree for a buggy node; such a node must exist if the root is erroneous and the tree is finite. Many search strategies are possible and (Naish 2000) provides very simple code for a top-down search. The code first checks that the root is erroneous. It then recursively searches for bugs in children and returns them if they exist. Otherwise the root is returned as a buggy node, along with an inadmissible child if any are found. In the next sections we first define the trees we use, then discuss how programmer intentions are formalised, give some simple diagnosis sessions and finally make some remarks about search strategy.

3.1 Partial proof trees

Standard wrong answer declarative diagnosis uses Prolog proof trees which correspond to successful derivations (see (Lloyd 1984)). Each node contains an atomic goal which was proved in the derivation, in its final state of instantiation, and the children of a node are the subgoals of the clause instance used to prove the goal. Leaves are atomic goals which were matched with unit clauses. We use *partial* proof trees which correspond to successful *or floundered* derivations. The only difference is they have an additional class of leaves: atomic goals which were never matched with any clause because they were delayed and never resumed.

Definition 1 ((Callable) annotated atom) *An* annotated atom *is an atomic formula or a term of the form when*(C, A)*, where A is an atomic formula and C is a condition of a when meta-call. It is* callable *if it is an atom or C is true according to the normal Prolog meaning (for ",", ";" and* `nonvar/1`*). atom*(X) *is the atom of annotated atom X.*

Definition 2 (Partial proof tree) *A* partial proof tree *for annotated atom A and program P is either*

1. *a node containing A, where atom(A) is an instance of a unit clause in P or A is not callable, or*

2. *a node containing callable atom A together with partial proof (sub)trees S_i for annotated atom B_i and P, $i = 1 \ldots n$, where atom(A) :- $B_1, \ldots B_n$ is an instance of a clause in P.*

A partial proof tree is floundered *if it contains any annotated atoms which are not callable, otherwise it is* successful.

Figure 5 gives the partial proof tree corresponding to the floundering of goal `perm([A,1|B],[2,3])` with Bug 1 (which corresponds to the trace of Figure 4). The "when" annotations are not shown. The single letter prefix for each node indicates whether the subtree is successful (s) or floundered (f). The single letter postfix relates to the truth values of the nodes: correct (c), erroneous (e), or inadmissible (i). Given these assignments, the only buggy node is that containing the atom `inserted(3,[2|C],[2,3])`. In Section 3.2 we explain how these assignments reflect the intentions of the programmer.

Representing a computation as a (partial) proof tree has several advantages over representing it as a linear trace if the goal is to diagnose incorrect successful or floundered derivations. First, backtracking is eliminated entirely, avoiding an important distraction. Second, the details of any coroutining are also eliminated. It has long been known this could be done for successful computations, but the realisation it can be done for floundered computations is relatively new and is the key to our approach. We retain information on what sub-goals were never called, but the order in which other subgoals were executed is not retained. The structure of the tree reflects the static dependencies in the code rather than the dynamic execution order. Because of this, each node gives us the final instantiation state of the atom, not just the instantiation state when it exited (at that time some subcomputations may have been delayed). Finally, the tree structure allows us search efficiently for buggy nodes by checking the truth value of nodes, which can be determine from the programmer's intentions.

Declarative debuggers use various methods for representing trees and building such representations. The declarative debugger for Mercury (Somogyi & Henderson 1999) is a relatively mature implementation. A much simpler method, which is suitable for prototypes, is a meta interpreter which constructs an explicit representation of the tree. Figure 6 is a very concise implementation which we include for completeness. Floundering is detected using the "short

```
% solve_atom(A, C0, C, AT): A is an
% atomic goal, possibly wrapped in
% when meta-call, which has succeeded
% or floundered; AT is the corresponding
% partial proof tree with floundered
% leaves having a variable as the list
% of children; C0==C iff A succeeded
solve_atom(when(Cond, A), C0, C, AT) :- !,
    AT = node(when(Cond, A), C0, C, Ts),
    when(Cond, solve_atom(A, C0, C,
                         node(_,_,_,Ts))).
solve_atom(A, C0, C, node(A,C0,C,AsTs)) :-
    clause(A, As),
    solve_conj(As, C0, C, AsTs).

% As above for conjunction;
% returns list of trees
solve_conj(true, C, C, []) :- !.
solve_conj((A, As), C0, C, [AT|AsTs]) :- !,
    solve_atom(A, C0, C1, AT),
    solve_conj(As, C1, C, AsTs).
solve_conj(A, C0, C, [AT]) :-
    solve_atom(A, C0, C, AT).
```

Figure 6: Code to build partial proof trees

circuit" technique — an accumulator pair is associated with each subgoal and the two arguments are unified if and when the subgoal succeeds. Tree nodes contain an annotated atom, this accumulator pair and a list of subtrees. A subcomputation is floundered if the accumulator arguments in the root of the subtree are not identical.

3.2 The programmer's intentions

The way truth values are assigned to nodes encodes the user's intended behaviour of the program. In the classical approach to declarative debugging of wrong answers the intended behaviour is specified by partitioning the set of ground atoms into true atoms and false atoms. There can still be non-ground atoms in proof tree nodes, which are considered true if the atom is valid. A difficulty with this two-valued scheme is that most programmers make implicit assumptions about the way their code will be called, such as the "type" of arguments. Although `inserted(2,a,[2|a])` can succeed, it is counter-intuitive to consider it to be true (since it is "ill-typed"), and if it is considered false then the definition of `inserted/3` must be regarded as having a logical error. The solution to this problem is to be more explicit about how predicates should be called, allowing pre-conditions (Drabent, Nadjm-Tehrani & Maluszynski 1988) or saying that certain things are inadmissible (Pereira 1986) or having a three-way partitioning of the set of ground atoms (Naish 2006).

In the case of floundering the intended behaviour of non-ground atoms must be considered explicitly. As well as assumptions about types of arguments, we inevitably make assumptions about how instantiated arguments are. For example, `perm/2` is not designed to generate all solutions to calls where neither argument is a (nil-terminated) list and even if it was, such usage would most likely cause an infinite loop if used as part of a larger computation. It is reasonable to say that calls to `perm/2` where neither argument is ever instantiated to a list should not occur, and hence should be considered inadmissible. An important heuristic for generating control information is that calls which have an infinite number of solutions should be avoided (Naish 1986). Instead, such a call is better delayed, in the hope that other parts of the computation will further instantiate it and make the number of solutions finite. If the number of solutions remains infinite the result is floundering, but this is still preferable to an infinite loop.

We specify the intended behaviour of a program as follows:

Definition 3 (Interpretation) *An* interpretation *is a three-way partitioning of the set of all atoms into those which are* inadmissible*,* valid *and* erroneous*. The set of admissible (valid or erroneous) atoms is closed under instantiation (if an atom is admissible then any instance of it is admissible), as is the set of valid atoms.*

The following interpretation precisely defines our intentions for `perm/2`: `perm(As0,As)` is admissible if and only if either `As0` or `As` are (nil-terminated) lists, and valid if and only if `As` is a permutation of `As0`. This expresses the fact that either of the arguments can be input, and only the list skeleton (not the elements) is required. For example, `perm([X],[X])` is valid (as are all its instances), `perm([X],[2|Y])` is admissible (as are all its instances) but erroneous (though an instance is valid) and `perm([2|X],[2|Y])` is inadmissible (as are all atoms with this as an instance). For diagnosing Bugs 1 and 2, we say `inserted(A,As0,As)` is admissible if and only if `As0` or `As` are lists. For diagnosing Bug 3, `As0` must be a list, expressing the different intended modes in this case.

Note we do not have different admissibility criteria for different sub-goals in the program — the intended semantics is predicate-based. Delay primitives based on predicates thus have an advantage of being natural from this perspective. Note also that atoms in partial proof tree nodes are in their final state of instantiation in the computation. It may be that in the first call to `inserted/3` from `perm/2`, no argument is instantiated to a list (it may delay initially), but as long as it is eventually sufficiently instantiated (due to the execution of the recursive `perm/2` call, for example) it is considered admissible. However, since admissibility is closed under instantiation, an atom which is inadmissible in a partial proof tree could not have been admissible at any stage of the computation. The debugger only deals with whether a call flounders — the lower level procedural details of when it is called, delayed, resumed *et cetera* are hidden.

Truth values of partial proof tree nodes are defined in terms of the user's intentions:

Definition 4 (Truth of nodes) *Given an interpretation I, a partial proof tree node is*

1. correct*, if the atom in the node is valid in I and the subtree is successful,*
2. inadmissible*, if the atom in the node is inadmissible in I, and*
3. erroneous*, otherwise.*

Note that floundered subcomputations are never correct. If the atom is insufficiently instantiated (or "ill-typed") it is inadmissible, otherwise it is erroneous.

3.3 Diagnosis examples

In our examples we use a top-down search for a buggy node, which gives a relatively clear picture of the partial proof tree. They are copied from actual runs of our prototype[2] except that repeated identical questions are removed and some white-space is changed.

[2]Available from `http://www.cs.mu.oz.au/~lee/papers/ddf/`

```
?- wrong(perm([A,1|B],[2,3])).
(floundered) perm([3,1|A],[2,3])...? e
(floundered) perm([1|A],[2|B])...? i
(succeeded)  inserted(3,[2|A],[2,3])...? e
(succeeded)  inserted(3,[],[3])...? v
BUG - incorrect clause instance:
inserted(3,[2|A],[2,3]) :-
    when((nonvar(A);nonvar([3])),
        inserted(3,[],[3])).
```

Figure 7: Bug 1 diagnosis for `perm([A,1|B],[2,3])`

```
...
(floundered) perm([1,2,3],[1,3,A|B])...? e
(floundered) perm([2,3],[3,A|B])...? e
(floundered) inserted(2,[3],[3,A|B])...? e
(floundered) inserted(2,[A|B],[A|C])...? i
BUG - incorrect modes in clause instance:
inserted(2,[3],[3,A|B]) :-
    when((nonvar([]);nonvar([A|B]))
        inserted(2,[A|_],[A|B])).
```

Figure 8: Bug 1 diagnosis for `perm([1,2,3],A)`

In section 3.4 we discuss strategies which can reduce the number of questions; the way diagnoses are printed could also be improved. The debugger defines a top-level predicate `wrong/1` which takes an atomic goal, builds a partial proof tree for an instance of the goal then searches the tree. The truth value of nodes is determined from the user. The debugger prints whether the node succeeded or floundered, then the atom in the node is printed and the user is expected to say if it is valid (`v`), inadmissible (`i`) or erroneous (`e`).

Figure 7 shows how Bug 1 is diagnosed for the goal `perm([A,1|B],[2,3])`. The root of the tree (shown in Figure 5) is erroneous, so the debugger proceeds to recursively search for bugs in the children. In this case it first considers the right child, which is inadmissible (so the recursive search fails), then the left child, which is erroneous (and the search continues in this subtree). Note that the call to `perm/2` in the root calls itself in an inadmissible way but this, in itself, does not indicate a bug. The cause of the inadmissible call is the other child, which is erroneous, and the root is not a buggy node. The recursive search in the left subtree determines the left-most leaf is correct and hence the buggy node is found. The diagnosis is a logical error in the `inserted/3` clause: the body of the clause is valid but the head is not.

Figure 8 shows how Bug 1 is diagnosed for the goal `perm([1,2,3],A)`. We assume the user decides to diagnose a floundered answer, backtracking over the previous answers. The diagnosis is ultimately a control error: the arguments of the clause head are sufficiently instantiated but the arguments of the recursive call are not. Both are diagnoses are legitimate. Even without delays, logical bugs can lead to both missing and wrong answers, which typically result in different diagnoses in declarative debuggers.

Figure 9 shows how Bug 2 is diagnosed. The first question relates to the first answer returned by the goal. It is valid, so the diagnosis code fails and the computation backtracks, building a new partial proof tree for the next answer, which is floundered. The root of this tree is determined to be erroneous and after a few more questions a buggy node is found. It is a floundered leaf node so the appropriate diagnosis is an incorrect delay annotation, which causes `inserted(A,B,[])` to delay indefinitely (rather than fail). Ideally we should also display the instance of the

```
?- wrong(perm(A,[1,2,3])).
(succeeded)  perm([1,2,3],[1,2,3])...? v
(floundered) perm([1,2,A,B|C],[1,2,3])...? e
(floundered) perm([2,A,B|C],[2,3])...? e
(floundered) perm([A,B|C],[3])...? e
(floundered) inserted(A,[3|B],[3]) ...? e
(floundered) inserted(A,B,[])...? e
BUG - incorrect delay annotation:
when((nonvar(A);nonvar(B)),inserted(B,A,[]))
```

Figure 9: Diagnosis of bug 2

```
?- wrong(perm(A,[1,2,3])).
(succeeded)  perm([1,2,3],[1,2,3])...? v
(floundered) perm([1,2,A,B|C],[1,2,3])...? e
(floundered) perm([2,A,B|C],[2,3])...? e
(floundered) perm([A,B|C],[3])...? e
(floundered) inserted(A,[3|B],[3])...? i
(floundered) perm([A|B],[3|C])...? i
BUG - incorrect modes in clause instance:
perm([A,C|D],[3]) :-
    when((nonvar([3|B]);nonvar([])),
        inserted(A,[3|B],[3])),
    when((nonvar([C|D]);nonvar([3|B])),
        perm([C|D],[3|B])).
```

Figure 10: Diagnosis of bug 3

clause which contained the call (the debugger code in (Naish 2000) could be modified to return the buggy node *and* its parent), and the source code location.

Figure 10 shows how Bug 3 is diagnosed, using the same goal. It proceeds in a similar way to the previous example (the partial proof trees are identical), but due to the different programmer intentions (the mode for `inserted/3`) the floundering call `inserted(A,[3|B],[3])` is considered inadmissible rather than erroneous, eventually leading to a different diagnosis. Both calls in the buggy clause instance are inadmissible. The debugger of (Naish 2000) returns both these inadmissible calls as separate diagnoses. For diagnosing floundering it is preferable to return a single diagnosis, since the floundering of one can result in the floundering of another and its not clear which are the actual culprit(s).

3.4 Search strategy

A top-down left to right search is the simplest search strategy to implement. In our prototype we have a slightly more complex strategy, searching floundered subtrees before successful ones (this is done by adjusting the order in which the childern of a node are returned — see Figure 11). More complex strategies for diagnosing some forms of abnormal termination

```
% returns children of a node,
% floundered ones first
child(node(_, _, _, Ts), T) :-
    nonvar(Ts), % not a floundered leaf
    (    member(T, Ts),
         T = node(_, C0, C, _),
         C0 \== C   % T is floundered
      ;
         member(T, Ts),
         T = node(_, C0, C, _),
         C0 == C    % T is not floundered
    ).
```

Figure 11: Finding children in partial proof trees

are given in (Naish 2000), and these can be adapted to floundering. From our definition of truth values for nodes, we know no floundered node is correct. We also know that floundering is caused by (at least one) floundered leaf node. Thus we have (at least one) path of nodes which are not correct between the root node and a leaf. It makes sense to initially restrict our search to such a path. Our prototype does a top-down search of such a path. There must be an erroneous node on the path with no erroneous children on the path. Both bottom-up and binary search strategies are likely to find this node significantly more quickly than a top-down search. Once this node is found, its other children must also be checked. If there are no erroneous children the node is buggy. Otherwise, an erroneous child can be diagnosed recursively, if it is floundered, or by established wrong answer diagnosis algorithms.

4 Theoretical considerations

We first make some remarks about the soundness and completeness of this method of diagnosis, then discuss related theoretical work. An admissible atomic formula which flounders has a finite partial proof tree with an erroneous root and clearly this must have a buggy node. Since the search space is finite, completeness is easily achieved. Soundness criteria come from the definition of buggy nodes (erroneous nodes with no erroneous children). The three classes of bugs mentioned in Section 2 give a complete categorisation of bugs which cause floundering. Incorrect delay annotations cause floundered buggy leaf nodes: they are admissible but delay. Confusions over modes cause floundered buggy internal nodes: they are admissible but have one or more floundered inadmissible children. Logical errors can also cause such nodes and can cause successful buggy nodes. Note that ancestors of a successful buggy node may also be floundered, as in Figure 5.

Our current work on diagnosis arose out of more theoretical work on floundering (Naish 2008). Nearly all delay primitives have the property that if a certain call can proceed (rather than delay), any more instantiated version of the call can also proceed (the set of callable annotated atoms is closed under instantiation). Our diagnosis method can be effectively applied to other delay primitives for which this property holds simply by changing the definition of callable annotated atoms. An important result which follows from this property is similar to the result concerning success: whether a computation flounders, and the final instantiation of variables, depends on the delay annotations but not on the order in which sufficiently instantiated call are selected. Non-floundering is also closed under instantiation, so it is natural for admissibility to inherit this restriction and partial proof trees provide a basis for intuitive diagnoses. Furthermore, there is a very close correspondence between floundered and successful derivations, and this is what enables our approach to diagnosis. A floundered derivation D for program P can be transformed into a successful derivation D' for a program P' which is identical to P except for the delay annotations. We briefly explain (a simplified version of) the mapping next.

The key idea is that a floundered derivation (or partial proof tree) using P will correspond to a successful derivation (or proof tree) in P' where the variables which caused floundering in P are instantiated to special terms which could not be constructed by the rest of the computation — the variables are *encoded* using special terms. This encoding has no effect on successful subcomputations; any subcomputation which succeeds with an answer containing variables will also succeed if any of those variables are further instantated. Because Prolog uses most general unifiers, the only terms constructed in a Prolog derivation contain function symbols which appear in the program. Thus "extraneous" function symbols, which do not occur in P, can be used to encode variables. For simplicty, we will just use `$`, and assume it does not appear in P (in (Naish 2008) multiple encodings are used, which makes the mapping between derivations more precise).

Each annotated atom $when(C, A)$ in P is transfomed so that the corresponding code in P' just calls A when C is satisfied but can also succeed when C is not satisfied, encoding the appropriate variables. Calling A is achieved by having A as a disjunct in the transformed code. The other disjunct implements the *negation* of C, using the encoding. The negation of `nonvar(V)` ensures `V` is an encoded variable, `$`, and De Morgan's laws handle conjunction and disjunction in delay conditions.

Definition 5 *The transformation T applied to when annotations is:*
$T(when(C, A)) = (T(C) ; A)$
$T((C_1, C_2)) = (T(C_1) ; T(C_2))$
$T((C_1 ; C_2)) = (T(C_1), T(C_2))$
$T(nonvar(V)) = (V = \$)$

For example, the recursive clause for the correct version of `inserted/3` is transformed to:

```
inserted(A, [A1|As0], [A1|As]) :-
   (As0 = $, As = $ ; inserted(A, As0, As)).
```

The goal `inserted(1,[2,3|A],[2,3|B])` has a derivation/partial proof tree in P which is floundered due to an annotated recursive call to `inserted/3` with variables `A` and `B` as the second and third arguments, respectively. There is a corresponding successful derivation/proof tree in P' where `A` and `B` are instantiated to `$`. This correspondence between floundered and successful computations means we can use the properties of successful derivations in the diagnosis of floundering — in particular, abstracting away backtracking and the order in which sub-goals are executed.

In section 3.2 we defined interpretations by partitioning the set of all atoms, rather than just the ground atoms. This is what allows us to say an insufficiently instantiated floundered atom is inadmissible. The ground encoded versions of such atoms would normally be considered ill-typed, hence it is reasonable to consider them inadmissible also. For example, `inserted(1,[2,3|A],[2,3|B])` is inadmissible and the encoded atom `inserted(1,[2,3|$],[2,3|$])` has non-lists in the last two arguments. Thus, by encoding atoms and using ill-typedness in place of under-instantiation, it is possible to define interpretations over just ground atoms. The way we described our intended interpretation in section 3.2 can remain unchanged. Encoding our example from that section, `perm([$],[$])` is valid, `perm([$],[2|$])` is erroneous and `perm([2|$],[2|$])` is inadmissible. By only using ground atoms, the three-valued semantics proposed in (Naish 2006) can be used unchanged (and our approach can indeed be considered "declarative"). Diagnosis of floundering becomes almost identical to diagnosis of wrong answers in the three-valued scheme. The only difference is the rare case of a valid ground atom which flounders rather than succeeds: when floundering is converted to success it appears there is no bug. For this case it is necessary to distinguish success from floundering, for example, with extra information in each node of the proof tree, as we have done in our implementation.

5 Conclusion

There has long been a need for tools and techniques to diagnose unexpected floundering in Prolog with delay primitives, and related classes of bug symptoms in other logic programming languages. The philosophy behind delay primitives in logic programming languages is largely based on Kowalski's equation: Algorithm = Logic + Control (Kowalski 1979). By using more complex control, the logic can be simpler. This allows simpler reasoning about correctness of answers from successful derivations — we can use a purely declarative view, ignoring the control because it only affects the procedural semantics. When there are bugs related to control it is not clear the trade-off is such a good one. The control and logic can no longer be separated. Since the normal declarative view cannot be used, the only obvious option is to use the procedural view. Unfortunately, even simple programs can exhibit very complex procedural behaviour, making it very difficult to diagnose and correct bugs using this view of the program.

In the case of floundering, a much simpler high level approach turns out to be possible. The combination of the logic and control can be viewed as just slightly different logic, allowing declarative diagnosis techniques to be used. The procedural details of backtracking, calls delaying and the interleaving of subcomputations can be ignored. The user can simply put each atomic formula into one of three categories. The first is inadmissible: atoms which should not be called because they are insufficiently instantiated and expected to flounder (or are "ill-typed" or violate some pre-condition of the procedure). The second is valid: atoms for which all instances are true and are expected to succeed. The third is erroneous: atoms which are legitimate to call but which should not succeed without being further instantiated (they are not valid, though an instance may be). A floundered derivation can be viewed as a tree and this three-valued intended semantics used to locate a bug in an instance of a single clause or a call with a delay annotation.

References

Caballero, R., Rodríguez-Artalejo, M. & del Vado Vírseda, R. (2006), Declarative diagnosis of wrong answers in constraint functional-logic programming, *in* S. Etalle & M. Truszczynski, eds, 'ICLP', Vol. 4079 of *Lecture Notes in Computer Science*, Springer, pp. 421–422.

Caballero, R., Rodríguez-Artalejo, M. & del Vado Vírseda, R. (2007), Declarative debugging of missing answers in constraint functional-logic programming, *in* V. Dahl & I. Niemelä, eds, 'ICLP', Vol. 4670 of *Lecture Notes in Computer Science*, Springer, pp. 425–427.

Clark, K. & McCabe, F. (1979), The control facilities of IC-Prolog, *in* D. Michie, ed., 'Expert systems in the microelectronic age', Edinburgh University Press, pp. 122–149.

Drabent, W., Nadjm-Tehrani, S. & Maluszynski, J. (1988), The use of assertions in algorithmic debugging, *in* 'Proceedings of the 1988 International Conference on Fifth Generation Computer Systems', Tokyo, Japan, pp. 573–581.

Gregory, S. (1987), *Design, application and implementation of a parallel logic programming language*, Addison-Weseley.

Jaffar, J. & Lassez, J.-L. (1987), From unification to constraints, *in* K. Furukawa, H. Tanaka & T. Fujisaki, eds, 'Proceedings of the Sixth Logic Programming Conference', Tokyo, Japan, pp. 1–18. published as Lecture Notes in Computer Science 315 by Springer-Verlag.

Kowalski, R. (1979), 'Algorithm = Logic + Control', *CACM* **22**(7), 424–435.

Lloyd, J. W. (1984), *Foundations of logic programming*, Springer series in symbolic computation, Springer-Verlag, New York.

Marriott, K., García de la Banda, M. & Hermenegildo, M. (1994), Analyzing Logic Programs with Dynamic Scheduling, *in* '20th. Annual ACM Conf. on Principles of Programming Languages', ACM, pp. 240–254.

Marriott, K., Søndergaard, H. & Dart, P. (1990), A characterization of non-floundering logic programs, *in* S. Debray & M. Hermenegildo, eds, 'Proceedings of the North American Conference on Logic Programming', The MIT Press, Austin, Texas, pp. 661–680.

Naish, L. (1986), *Negation and control in Prolog*, number 238 *in* 'Lecture Notes in Computer Science', Springer-Verlag, New York.

Naish, L. (1992), 'Declarative diagnosis of missing answers', *New Generation Computing* **10**(3), 255–285.

Naish, L. (1997), 'A declarative debugging scheme', *Journal of Functional and Logic Programming* **1997**(3).

Naish, L. (2000), 'A three-valued declarative debugging scheme', *Australian Computer Science Communications* **22**(1), 166–173.

Naish, L. (2006), 'A three-valued semantics for logic programmers', *Theory and Practice of Logic Programming* **6**(5), 509–538.

Naish, L. (2008), 'Transforming floundering into success'.
URL: *ww2.cs.mu.oz.au/~lee/papers/flounder*

Naish, L. & Barbour, T. (1995), A declarative debugger for a logical-functional language, *in* G. Forsyth & M. Ali, eds, 'Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems — Invited and Additional Papers', Vol. 2, DSTO General Document 51, Melbourne, pp. 91–99.

Pereira, L. M. (1986), Rational debugging in logic programming, *in* E. Shapiro, ed., 'Proceedings of the Third International Conference on Logic Programming', London, England, pp. 203–210. published as Lecture Notes in Computer Science 225 by Springer-Verlag.

Pope, B. & Naish, L. (2003), Practical aspects of declarative debugging in Haskell-98, *in* 'Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming', pp. 230–240. ISBN:1-58113-705-2.

Shapiro, E. Y. (1983), *Algorithmic program debugging*, MIT Press, Cambridge, Massachusetts.

Somogyi, Z. & Henderson, F. J. (1999), The implementation technology of the Mercury debugger, *in* 'Proceedings of the Tenth Workshop on Logic Programming Environments', Las Cruces, New Mexico, pp. 35–49.