Declarative Diagnosis of Floundering

Lee Naish
Computing and Information Systems
University of Melbourne

Slides, paper and code are on the web:

http://www.cs.mu.oz.au/~lee/papers/ddf/

Outline

Background

Motivation

Diagnosis method

Example diagnoses

Theory

Conclusion

Background

As well as the normal left to right execution, many Prolog systems support coroutining

A call can delay if it is insufficiently instantiated, and be resumed later, after some variables have been bound

But sometimes this never happens — the call is never resumed and the computation *flounders*

Similarly, concurrent logic programs can *deadlock* and constraint logic programs may never invoke (efficient) constraint solvers

Background (cont.)

Kowalski: Algorithm = Logic + Control

Instead of making the logic more complex, we can make the control more complex

Reasoning about correctness of successful derivations is made easier

Thats great if we never write buggy programs, or if bugs are never related to control

Floundering is a symptom of control-related bugs and the standard reasoning methods, based on declarative semantics, don't apply

The procedural semantics can be *very* complex

Maybe its not such a good trade-off after all?

Example SWI-Prolog Program

```
% 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 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)).
```

Example Program (cont.)

The program uses the "when meta-call" for delaying

Eg, the call `inserted(A0, As1, [A|As])` call delays until `As1` or `As` are instantiated

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

Three buggy versions

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

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

Bug Symptoms

Bug 1: `perm([1,2,3],A)` succeeds with `A=[1,2,3]`, then four satisfiable (but not valid) answers, eg `A=[1,2,3|_]`, and four floundered answers, eg `A=[1,3,_|_]`

Bug 1: `perm(A,[1,2,3])` succeeds with `A=[1,2,3]`, then three floundered answers, eg `A=[1,3,_|_]`

Bug 1: `perm([A,1|B],[2,3])` has floundered answer `A=3`

Bug 2: `perm([X,Y,Z],A)` behaves correctly but `perm([1,2,3],A)` succeeds with `A=[1,2,3]` and `A=[1,3,2]`, then loops

Bugs 2 and 3: `perm(A,[1,2,3])` succeeds with `A=[1,2,3]` then has three floundered answers, `A=[1,2,_|_]`, `A=[1,_|_]` and `A=[_|_]`, then fails

Declarative diagnosis of floundering

We can to avoid thinking about coroutining, backtracking and debugging strategy!

We use the three-valued declarative debugging scheme

The scheme represents a computation as a tree

Each node is *correct*, *erroneous* or *inadmissible* (as determined by the programmer)

A node is *buggy* if it is erroneous but has no erroneous children

The simplest search strategy is top-down

First, check the root is erroneous

Recursively search for buggy nodes in children; if found return them

Otherwise return the root as the bug (along with children, noting any inadmissible ones)

Partial Proof Trees

A proof tree corresponds to a successful derivation

Each node is at atom which was proved

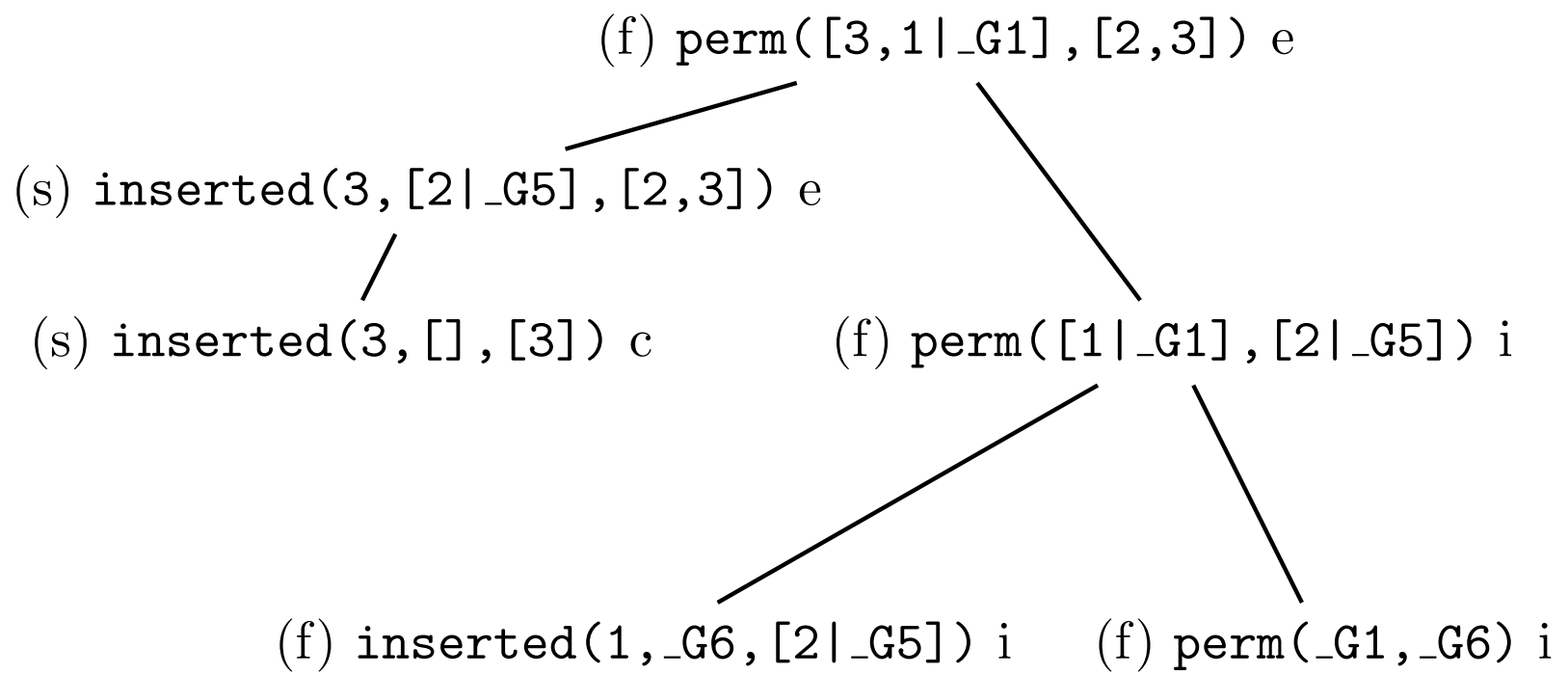
The children are the subgoals of the body of the clause instance used

Leaves are atoms matched with facts

A *partial proof* tree corresponds to a successful *or floundered* derivation

Leaves can also be calls which delayed but were never resumed

A Partial Proof Tree for Bug 1



(f) = floundered, (s) = succeeeded

e = erroneous, c = correct, i = inadmissible

The programmer's intentions

Diagnosis of wrong answers can be based in truth of ground atoms

Atoms in the proof tree are correct if they are *valid*

Inadmissibility can be used for ill-typed atoms, eg
`inserted(1,a,[1|a])`

For floundering we consider truth of non-ground atoms

The set of admissible (valid or erroneous) atoms is closed under instantiation (as is the set of valid atoms)

Successful partial proof tree nodes are correct (valid), erroneous or inadmissible, depending on the atom

Floundered partial proof tree nodes are erroneous if they are properly instantiated and inadmissible otherwise

Our intentions for `perm/2`

`perm(As0,As)` is admissible iff `As0` or `As` are (nil-terminated) lists and valid if `As` is a permutation of `As0`

eg: `perm([X],[X])`, `perm([X],[2|Y])`, `perm([1|X],[2|Y])`

For Bug 3 `inserted(A,As0,As)` is admissible iff `As0` is a list

For Bugs 1 and 2 its admissible iff `As0` or `As` are lists

Diagnosis example: Bug 1

```
?- 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])).
```

Diagnosis example: Bug 1

```
...
(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])).
```

Diagnosis example: 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]) ...? e
(floundered) inserted(A,B,[])...? e
BUG - incorrect delay annotation:
when((nonvar(A);nonvar(B)),inserted(B,A,[]))
```

Diagnosis example: Bug 3

```
?- 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])).
```

Theory

Soundness and completeness of diagnosis are straightforward

Floundering is only caused by incorrect delay annotations, confusion over intended modes, and logical errors

There is a floundered derivation iff there is a successful derivation using a transformed version of the program

Would-be floundered calls bind the insufficiently instantiated variables to special terms which don't occur elsewhere, eg `$`

`when((nonvar(As0) ; nonvar(As)), inserted(A, As0, As))` is transformed into `(As0 = $, As = $ ; inserted(A, As0, As))`

Our diagnosis algorithm is like three-valued wrong answer diagnosis using the transformed program

Conclusion

For floundering, Logic + Control = Logic$'$

Declarative diagnosis techniques can be applied quite easily

The complex details of calls delaying, interleaved execution, backtracking etc can be ignored