Declarative Diagnosis of Floundering

Lee Naish
Computer Science and
Software Engineering
University of Melbourne

Slides, paper and code are on the web:

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

Outline

Background

Example program

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.)

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

But for floundering we can't ignore control

The procedural semantics can be *very* complex

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

A Buggy 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)).
```

Uses the “when meta-call” for delaying

Eg, recursive `perm(As0,As1)` call delays until `As0` or `As1` are instantiated

A Buggy Program (cont.)

```
% inserted(A, As0, As): As = list As0 with A inserted
%......% As0 should be input          %... Bug 2
% As0 or As should be input
inserted(A, As0, [A|As0]).
inserted(A, [A1|As0], [A1|As]) :-
%...when(nonvar(As0),                 %... Bug 2
%...when((nonvar(As0) ; nonvar(A)),  %... Bug 1
    when((nonvar(As0) ; nonvar(As)),
%.......inserted(A, AS0, As)).        %... Bug 3
        inserted(A, As0, As)).
```

Bug Symptoms

Bug 1: `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 1 and 2: `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

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

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

Declarative diagnosis of floundering

An instance of the three-valued debugging scheme is used

The scheme represents a computation as a tree

Each node is *correct*, *erroneous* or *inadmissible*

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

WARNING

The following program contains material which may be offensive to some viewers

Building Partial Proof Trees

```
% solve_atom(A, C0, C, AT): A is an atomic goal
% (possibly wrapped in a when meta-call)
% which has succeeded or floundered;
% AT is the corresponding partial proof tree; floundered
% leaves have a variable as the list of children;
% C0==C if 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).
```

Building Partial Proof Trees (cont.)

```
% 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).
```

YUK!

The programmer's intentions

For diagnosing wrong answers the programmer can just consider 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 the we need to consider 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, erroneous or inadmissible, respectively

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([2|X],[2|Y])`

For Bug 2 `inserted(A,As0,As)` is admissible iff `As0` is a list

For Bugs 1,3 its admissible iff `As0` or `As` are lists

Diagnosis example: Bug 1

```
?- 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 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/types 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])).
```

Diagnosis example: Bug 3

```
?- wrong(perm(A,[1,2,3])).
(succeeded)  perm([1, 2, 3], [1, 2, 3]) ...? v
(floundered) perm([1, 3, A|B], [1, 2, 3]) ...? e
(floundered) perm([3, A|B], [2, 3]) ...? e
(floundered) perm([A|B], [2|C]) ...? 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 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/types in clause instance:
inserted(2, [3], [3, A|B]) :-
    when((nonvar([]);nonvar([A|B])),
        inserted(2, [A|_], [A|B])).
```

Search strategy

If the root is floundered there is a incorrect path down to a leaf

First find the bottom-most erroneous node on this path!

This path can be searched top-down by ordering children sensibly (as in the examples)

Or it can be searched bottom-up, or with binary search

Other children need to be checked (they can be diagnosed recursively)

Theory

Soundness and completeness of diagnosis are straightforward

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

A model-theoretic interpretation can be given by encoding variables with function symbols not appearing elsewhere, eg `perm([$],[$])`

When meta-calls can be interpreted as disjunctions

`when((nonvar(As0) ; nonvar(As1)), perm(As0, As1))` is interpreted as `(evar(As0), evar(As) ; perm(As0, As1))`

There is a floundered derivation iff there is a successful derivation using the encoding

Our diagnosis algorithm is like three-valued wrong answer diagnosis using the encoded derivation

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