RETROSPECTIVE ZOOMING:
A KNOWLEDGE BASED TRACING AND DEBUGGING METHODOLOGY FOR LOGIC PROGRAMMING

Marc Eisenstadt
Human Cognition Research Laboratory
The Open University
Milton Keynes, England

ABSTRACT

This paper describes new tracing and debugging facilities for logic prograramming (Prolog in particular), based on a selective retrospective analysis of an exhaustive run-time trace. The tracer uses an enriched repertoire of program success/failure 'symptoms' to improve the clarity of the trace, and identifies characteristic 'symptom clusters' in order to work out the true cause of a bug.

INTRODUCTION

In the course of debugging Prolog programs, the user can easily be overwhelmed by a plethora of tracing information. An overview of the behaviour of the user's program is sorely needed before engaging in any kind of single-stepping activity, even when a 'skip/retry' facility is provided. In addition, users can benefit from some intelligent tracing and debugging assistance, as amply demonstrated by the work of Shapiro (1982). The progression towards intelligent tracing facilities involves three main facets:

a) Symptomatic behaviour: A more detailed analysis of the behaviour of Prolog programs needs to be provided. This is because the four behaviours ('call', 'exit', 'fail', 'redo') provided by existing trace packages are Insufficient to provide clear signposts indicating the most likely cause of program failure.

b) Zooming: Once the behaviours are elaborated, the user needs to be protected from gory details on the one hand, yet allowed easy and rapid access to the relevant details as needed.

c) Suspicious symptom clusters: Characteristic 'symptom clusters' In the program trace need to be identified, so that particular kinds of behaviour can be singled out as being highly suspect.

SYMPTOMATIC BEHAVIOUR

The 'PTP' Prolog trace package (Eisenstadt, 1984) distinguishes among several different types of Prolog program failure (e.g. subgoals failed, no more backtrack solutions, backtrack encountered cut, no definition, wrong arity, variable unification failed, system primitive failed). In addition, PTP displays resolving clause numbers along with variables instantiated when the clause is attempted (rather than just when it exits).

In general, of course, the user may not want to observe program execution in such detail. The point of PTP's original 'symptomatic trace' facility was to develop a detailed analysis of Prolog program behaviour so that it could then be subjected to semi-automated inspection. The following sections describe how the latest implementation of PTP hides these details from the user while still capitalizing on the information contained therein.

ZOOMING

In the new PTP, the user sees the end results in advance, and can then 'zoom' in successively greater detail on subgoals of interest. Consider the following (buggy) quicksort program:

```
qsort([], []).
qsort([X|Xs], Result) :-
   split(Xs, X, Lo, Hi),
   qsort(Lo, Sorted_Lo),
   qsort(Hi, Sorted_Hi),
   append(Sorted_Lo, [X|Sorted_Hi], Result).

split([X|Xs], Crit, [X|Lo], Hi) :-
   X < Crit,
   split(Xs, Crit, Lo, Hi).
split([X|Xs], Crit, Lo, [X|Hi]) :-
   Crit > X,
   split(Xs, Crit, Lo, Hi).
split([], Crit, [], []).

append([X|Xs],Ys,[X|Zs]):-
   append(Xs,Ys,Zs).
append([],Ys,Ys).
```

Let's run the code through PTP. The symbol '?' below means 'attempting subgoal'. Failure due to inner subgoal failure is indicated by '-'• The symbol 'X' is a concatenation of the symbol '>' ('Entering the body of resolving clause whose head is shown') and the symbol '<' ('This clause didn't work, looking for next resolvent'). The notation 'S1/S2' refers to an invocation starting at trace step number S1 and finishing at trace step number S2. A number enclosed in braces, e.g. {2}, indicates which specific clause has been considered. User input to the 'PTP:' tracer prompt is shown underlined. Output from PTP is as shown, with '%' comments added retrospectively.

```
PTP:  qsort([2,1,3],R).
  1: ? qsort([2,1,3],_275)
 38: - qsort([2,1,3],_275) {2}
% Bombed out, 38 steps. Only clause 2 used.

SUBGOAL FAILURES:
  7/16: split([3],2,_184,_185)
  3/36: split([1,3],2,_184,_185)
  1/38: qsort([2,1,3],_275)
% Why should 'split', invoked at step 7,
% bomb out? Let's invoke the zoomer:

PTP:  zoom(7).  % home in on trace step 7
  7:   ? split([3],2,_127,_128)
 8/11: ><split([3],2,[3|_127],_128) {1}
12/15: ><split([3],2,_127,[3|_128]) {2}
 16:   - split([3],2,_127,_128)

% Clause 1 is tried at step 8, and loses (OK).
% Notice the partly-instantiated third arg shown
% at step 8. But why did clause 2 of 'split' fail?

PTP:  zoom(12).  % clause 2 of split is at step 12
 12: > split([3],2,_127,[3|_128]) {2}
 13:   @ 2>3          % '@' means system prim
 14:   --2>3          % '--' means prim loses
 15: < split([3],2,_127,[3|_128]) {2}

% why did I test 2 > 3 at step 13?  Aha...
% that was my mistake... so it loses
% and of course bails out of clause 2
```

Notice that the user has had to direct the process only twice to get to the culprit, by invoking 'zoom(7)' and 'zoom(12)'. This would have been true regardless of recursion depth, i.e. independent of the length of the list input to qsort at the top level. This 'constant zoom time' depends upon (a) the user being informed of 'suspect' subgoal failures from the innermost one outwards, and (b) the user deciding which of these failures is worthy of further perusal.

The retrospective zoom facility allows the user to catch a glimpse of overall program behaviour, and to make an informed decision on the basis of this selective view. The next section describes how PTP takes some of the debugging burden away from the programmer.

SUSPICIOUS SYMPTOMS

Two classes of suspicious code are pinpointed automatically by PTP for the user's benefit: (a) 'singleton suspects', which can be derived directly from symptoms exhibited in single lines of the trace, and (b) 'cluster suspects', which are suggested by the occurrence of characteristic clusters of symptoms distributed appropriately through the trace.

Singleton Suspects

The 'singleton suspect' analyser is automatically invoked when top-level goals are presented to the 'PTP:' prompt. The analyser walks over the full (internally stored) trace, and looks for missing definitions, wrong arity, unresolved goals, subgoal failures, and false (or wrong) successes. There is a one-to-one mapping from tracer symbols to singleton suspects, so these are easy to collect and point out to the user. False or wrong successes are pointed out if PTP is given prior assertions such as 'expect(qsort([2,1,3],[1,2,3]))'. The analysis becomes more interesting for 'cluster suspects', since the interaction of failure and success patterns needs to be considered.

Cluster Suspects

The complete trace sequence, with its symptom symbols and indenting pattern, forms a characteristic 'trace footprint', which can be analysed empirically to identify particular causes underlying observed program behaviour. The empirical analysis is done by using the PTP 'symptom' and 'zoom' facilities described above, and noting which patterns are indicative of deeper causes of failure. This analysis uses a repertoire of 'bug-cliches' representing PTP's best guess about particular program failures. These cliches are used to invoke a message frame instantiated with the specifics of the goal sequence to help the user identify the source of the problem.

To illustrate cluster symptoms, and PTP's explanation thereof, let's fix the offending line of code discovered earlier in clause 2 of 'split', so that it reads 'X > Crit' instead of 'Crit > X', and try another test run:

```
PTP:  qsort([2,1,3,2],R).
  1: ? qsort([2,1,3,2],_331)
 38: - qsort([2,1,3,2],_331) {2}

SUBGOAL FAILURES:
 15/24: split([2],2,_224,_225)
  7/28: split([3,2],2,_224,_225)
  3/36: split([1,3,2],2,_224,_225)
  1/38: qsort([2,1,3,2],_331)

PTP:  why(24).
I.E. WHY DID THE INVOCATION OF 'split'
AT STEP 15 FAIL AT STEP 24?

A SHARED FAILURE PATTERN HAS BEEN OBSERVED IN:
  (A) CLAUSE 1 OF split/4 (STEPS 16/19),
  (B) CLAUSE 2 OF split/4 (STEPS 20/23)

THE FAILURE PATTERN IS:  "FAILED PRIMITIVE".
IN (A) THE SUSPECTED TROUBLESOME INVOCATION WAS:
  17:  2<2
IN (B) THE SUSPECTED TROUBLESOME INVOCATION WAS:
  21:  2>2

POSSIBLE UNDERLYING CAUSES ARE:
 i) (SOME OF) THE TROUBLESOME INVOCATIONS ARE WRONG
ii) ADDITIONAL CASES HAVEN'T BEEN CATERED FOR
%End of running example
```

Cause (ii) above is the culprit: a '>=' test is needed to catch the missing case. Here is how the 'cluster suspect' is identified:

a) The user's request 'why(24)' is taken as expressing displeasure at the subgoal invocation/failure between steps 15 and 24. Therefore, it is assumed that the subgoal

'split([2],2,_139,_140)' should have succeeded.

b) The 'singleton suspect' subgoal failures (aside from the top level goal) differ only in terms of the list-lengths of their first arguments. Therefore, it is assumed that the root cause of all these failures is identical.

c) A 'zoom' of steps 15-24 is performed internally. The displayed version would have looked like this:

```
 15:   ? split([2],2,_139,_140)
16/19: ><split([2],2,[2|_139],_140) {1}
20/23: ><split([2],2,_139,[2|_140]) {2}
 24:   - split([2],2,_139,_140)
```

d) At this point, a characteristic symptom cluster is detected. The kernel of this cluster is the following four element collection:

```
['?', '>< {1}', '>< {2}', '-']
```

This kernel matches a known cluster pattern named 'subgoal fails after all resolving clauses tried and failed'.

e) The detection of the cluster invokes a set of rules which try to see whether there is a shared pattern underlying the failure of each clause. Intuitively, the analyser is looking for why a Prolog rule, viewed abstractly as a 'cases statement', has 'fallen off the end'. A further internally-performed zoom reveals the following kernel pattern:

```
[ '?',
  ['> {1}', ['@', '--'], '< {1}'],
  ['> {2}', ['@', '--'], '< {2}'],
  '-' ]
```

(Line 3 of the above pattern corresponds to trace steps 20-23, which are analogous to steps 12-15 of the trace presented earlier in the ZOOMING section.) This pattern provides sufficient grounds for the remainder of the messages displayed in the example. The declarative nature of this analysis enables it to work on more perverse definitions of 'split', such as ones where the greaterthan and lessthan tests come after the recursive invocation! The analysis can be performed even in the latter case because the internal zoomer inspects behaviour in terms of the program's declarative reading (which looks very similar in both the normal and perverse cases) before delving into sequence details.

Other cluster symptoms currently recognized are shown below:

* uncatered-for-case-with-bad-ordering (appropriate rule exists, but is not encountered due to miisordering)

* uncatered-for-case-with-rule-missing (like the 'uncovered goal' of Shapiro, 1982, but has specialists to identify missing tests for (a) null list, (b) atom, (c) last element)

* under-specified-unification (occurs for example when a variable accidentally can unify with either a list or an atom)

* infinite-loop-caused-by-loop-in-db (asserting 'tallerthan(joe,Joe)' will cause problems for naive transitivity code)

* infinite-loop-caused-by-left-recursive-rule, e.g. foo(X,Y):-foo(X,Z),foo(Z,Y).

COSCLUSIONS

'Retrospective zooming' enables a trace to remain faithful to the purely declarative reading of a logic program, yet allows appropriate probing of the procedural aspects as well. Suspect code can be identified by an empirical investigation of both single-line symptoms and, more importantly, clusters of co-occurring symptoms.

Our earlier work on automated program debugging (Laubsch & Eisenstadt, 1982) relied on the notion of a 'canonical effect description' which could be used to compare actual program behaviour with desired behaviour. In contrast to this, PTP, (like the system of Shapiro, 1982) leaves the notion of 'desirabilility' of program behaviour up to the programmer during debugging. PTP differs from Shapiro's work in maintaining an a priori repertoire of 'suspect' program behaviour, which itself is based upon a 'bug taxonomy' developed in the course of pilot studies of experienced Prolog programmers. The 'cluster suspects' detectable by PTP, while still in their earliest incarnation, have enabled the rapid development of a practical and empirically-motivated tracing and debugging facility for Prolog.

ACKNOWLEDGEMENTS

This work is supported by the UK Science and Engineering Research Council, Grant number GR/C/69344.

REFERENCES

[1] Eisenstadt, M. A powerful Prolog trace package. In T. O'Shea (Ed.), Advances in Artificial Intelligence (ECA1-84). Amsterdam: Elsevier/North-Holland, 1984.

[2] Laubsch, J., & Eisenstadt, M. Using temporal abstraction to understand recursive programs involving side effects. Proceedings of the National Conference on Artificial Intelligence (AAAI-82"), Pittsurgh, PA. 1982.

[3] Shapiro, E.Y. Algorithmic program debugging. Cambridge, MA: MIT Press, 1982.