RATIONAL DEBUGGING OF LOGIC PROGRAMS

Luis Moniz Pereira

Departamento de Informática

Universidade Nova de Lisboa

2825 Monte da Caparica

Portugal

July 1985

ABSTRACT

Information about Prolog data term's dependencies on derivation goals can be used for improved debugging, whether in the wrong solution or the missing solution type of faulty program behaviour.

A debugger for full Prolog (itself written in Prolog) has been developed that automates the reasoning ability required to pinpoint errors, resorting to the user only to ask about the intended program semantics. The debugger makes cooperative use of both the declarative and the operational semantics of Prolog programs.

In this writing the debugging algorithm is expressed in detail, session protocols are exhibited, comparison to other work is made, but the implementation is not examined.

INTRODUCTION

In logic programming many difficult problems remain whose statement is simple. One is: what's a rational way of debugging logic programs ?

The presuposition that there is some such a way is implicit in the question, but that should not be surprising. Logic deals with general rational forms of reasoning, irrespective of what the subject matter of its expressions may be.

Research in Logic Programming has shown that such independence regarding content manifests itself both in the forward direction of reasoning, as well as in the backwards one. Consequently, general interpreters for Prolog have been invented that perform more intelligently when backtracking, by avoiding derivation attempts that are bound to fail (on the basis of information about the dependency of bindings on previous derivation nodes) [Bruynooghe and Pereira 84].

In a similar vein, the debugging of logic programs is amenable to a general content independent approach. In the context of logic programming, debugging can be envisaged as a precise rational activity, whose purpose is to identify inconsistent or incomplete logic program statements, by systematically contrasting actual with intended program behaviour. A program can be thought of as a theory whose logical consequences engender its actual input/output behaviour. Whereas the program's intended input/output behaviour is postulated by the theory's purported models, i.e. the truths the theory supposedly accounts for.

The object of the debugging exercise is to pinpoint erroneous or missing axioms, from erroneous or missing derived truths, so as to account for each discrepancy between a theory and its models.

I provide here one answer to the above question, for full Prolog programs, which relies on information about each term's dependency on the set of previous derivation steps (such information is gleaned and retained by means of a suitably modified unification algorithm [Bruynooghe and Pereira 84]).

The debugger does so by contrasting theory and model, questioning the user about the intended program model. The types of questions are whether goals are admissible (i.e. have correct argument types), whether goals should be solvable or unsolvable (i.e. have some true instance or not), whether solved goals are true or false, and whether a set of solutions to a goal is complete. The user is not required to produce solutions to goals (as in [Shapiro 83]), nor to know about the operational semantics of Prolog programs (which renders the debugger helpful for knowledge acquisition validation by the non-programmer). All answers given by the user are stored, and repetition of the same or subsumed questions from the debugger is avoided.

The requirement of rationality, over and above the requirement of algorithmicity, imposes that no subderivations will be examined for errors which are irrelevant to the diagnosis of a bug, with respect to some given available information. Rationality requires that the debugging steps should be not only sufficient but necessary as well (relative to that information). Of course, a prescient being would be able to pinpoint erroneous clauses at once. More pragmatically, a

sensible trade-off must be found between the price of information and the economies it permits.

The present approach to rational debugging contends that the dynamic information regarding term dependency on derivation nodes, plus some of the operational semantics history, and a record of answers to system queries gleaned from the user, provide a basis for a practical method of rational debugging which can be automated.

Indeed, I have developed a Prolog debugger, itself written in Prolog, which incorporates the aforementioned rational debugging method. In particular, Shapiro's pioneering algorithmic debugging [Shapiro 83] is improved upon, though through an altogether different approach.

To some extent the rational debugger embodies people's debugging methodology. However, the availability of computing power makes practical more elaborate debugging procedures than most people care to use on their own, relying in fact on more information than the human debugger is expected to be aware of.

The debugger makes use of information about the dependency of terms on derivation steps to follow up on wrong solutions to goals, and to delimit the search to identify goals with missing solutions. The dependency information is generated by a modified unification algorithm [Bruynooghe and Pereira 84] that keeps track of where each binding (or match with an empty binding) occured. Such information is also used to provoke direct backtracking to a specific node in a derivation with the purpose of examining it for bugs, and for performing intelligent backtracking program execution (making the

debugger more efficient). In this respect, the ability to jump back over subderivations irrelevant to failures avoids unnecessary prompting of the user with questions. The debugger also knows about both the declarative and the operational semantics of Prolog and uses them cooperatively to achieve a faster detection of bugs, and further avoid unnecessary questioning of the user.

In the sequel, the debugging algorithm's rationale is presented in some detail, but for reasons of space, does not include cut treatment, and other Prolog extra-logical features such as its "ersatz" negation by non-provability, the "repeat...fail" construct, "assert" and "retract" usage, "var" and "nonvar", etc. These, as well as implementation details will be the subject of another paper. Comparison to other work is made, and session protocols are exhibited.

THE RATIONAL DEBUGGING ALGORITHM AND ITS RATIONALE

A) Wrong Solution Mode

A1. A goal known to have a wrong solution is presented to the debugger. The debugger solves it and asks you to state whether the solution found is wrong If not wrong, the debugger then produces the next solution and queries you again, until the wrong solution is obtained.

A2. A wrong solution has been identified and you point, with the cursor, at some wrong term within it.
(The debugger is easily modified to let you point at more than one wrong term ; I have prefered to track down one bug at the time so one can follow the debugger's behaviour more easily ; also, multiple wrong terms may have the same origin, as when they come from one same recursive procedure.)

A3. The debugger then uses the set of dependencies of the wrong term (i.e. derivation steps on which its binding depends [Bruynooghe and Pereira 84]) to thread back through the execution tree to the most recent dependency node for the term. In the wrong term debugging mode the derivation tree is "frozen", so that no alternatives are sought when going back. The dependency set includes both the productive dependencies (where bindings were produced) and the consumptive dependencies between functors (where bindings are consumed but not produced -- empty bindings).

The former are those used by selective backtracking Prolog [Pereira and Porto 82] in programs assumed correct, but not the latter. For debugging purposes, the empty bindings may be the source of bugs in those cases where a binding should have been rejected, thereby originating a failure, but wasn't and didn't.

A4. At the (next) most recent dependency node, the debugger queries you whether the goal at that node is "admissible" (a goal is admissible if it complies with the intended use of the procedure for it -- i.e. it has the correct argument types -- irrespective of whether the goal succeeds or not).

A5=A4.no. If the goal is not admissible, the debugger asks you to point at some wrong term within it and continues at A2 with the new term.

A6=A4.yes. If the goal is admissible, then the debugger matches the goal with the head of the clause used at that node in the frozen derivation, and queries you whether the resulting literal is "solvable" (i.e. if it should have a solution according to the intended program semantics). If the clause body is empty it asks instead whether the literal is "true", rather than "solvable", as there are no body goals to solve.

A7=A6.no. You answer "no" to the matched head literal being solvable or true

First note that the goal did solve. So at least one of the following holds:

- the clause head is wrong because it produced a wrong binding,
  or because it should not have matched the goal.
- some body goal should have failed but didn't or some goal that would fail
  is missing.

The debugger asks you to point at a wrong term in the literal, so it can detect whether it was passed by the goal on to and accepted by the clause, or is textually present in the clause head.

A8.1. If the body is empty, then a wrong unit clause has been found, and the debugger reports an error corresponding to your previous answer (i.e. whether you've pointed to a term textually present or not).

A8.2. If the body is not empty, next the debugger asks you whether each of the (non built-in) solved goal instances is i) inadmissible t) true or f) false. Note that the t) and f) questions assume universal quantification over any variables.

case i) It requests you to point at some wrong term within it, reverts to wrong solution debugging mode, and follows part A of the algorithm at step A3, with the new term.

case t) If all solved clause body goals are true, then the clause is wrong because it has produced a false conclusion from true premises. The debugger reports an error according to what term you pointed at in the matched clause head literal, at step A7.

case f) As the goal has solved, it asks if you want to debug the goal. If you do it starts the debugger on it at step A1; if not it aborts to Prolog.

A9=A6.yes. You answer "yes" to the matched head literal being solvable or true.

First, consider that the wrong term being tracked down is not a variable. In that case, the clause invocation and execution are not blameable for producing or not rejecting the wrong term being tracked down. Note that the wrong term cannot have been produced by the clause body. In that case, any node producing it would have already been examined and detected before, since the wrong term's dependencies are visited in reverse time order.
So continue at A4.

Alternatively, the wrong term being tracked down is variable. In that case a too general solution is being produced by some clause. If you were asked whether the matched clause head was "true" (that's when the clause body is empty) continue at A4 (the question implied universal quantification).

If the body is not empty, you were asked if the matched clause head was "solvable" (the question implied existencial quantification). To detect whether an overgeneral solution is being produced at the present clause, you are asked about each of the solved goals in the body, as in A8.2. Cases i) and f) are dealt with in the same way. Case t) is different: if all the solved body goals are true, you are asked if the solved clause head is true. If "yes" continue at A4, otherwise an erroneous clause (that produces an overgeneral solution) has been found.

B) Missing Solution Mode
==========

B1. Some solution is missing for a finitely failed goal which is presented to the debugger. It then attempts to find solutions to the goal until you state that the missing solution would be next. Infinite failure has no

special treatment yet. The depth-control method has not been improved upon.

B2. The debbuger initiates an attempt to find the next solution, aware that at least one goal which failed should not have done so. The top goal becomes the currently known lowest unsolving ancestor.

B3. When a failed goal is detected, the failure may be legitimate or unjustified.

B4=B3.legitimate. (B3.unjustified is dealt with in case s of B8.2)
If the failure is legitimate, then it cannot be the cause of a missing solution. Rather than asking you immediately which is the case, the debugger attempts to find some ancestor of the failed goal that solves (the list of ancestors is available at each step). Which one is appropriate ?

It picks the most recent ancestor, below the currently known lowest unsolving ancestor goal, that is above all the non-ancestor dependency nodes of the failed goal, so as to give opportunity for backtracking to take place to those nodes and eventually solve the failed goal. Another possibility is that, on backtracking, some alternative is found that makes the goal failure irrelevant (if the chosen ancestor does solve), by avoiding it altogether.

B5. The appropriate ancestor may be known to be unsolving though it should be solvable. If so, go to step B8. If not, it then considers whether it has backtracked over it to the failing goal or not.

B6=B5.no. If not, it attempts to solve the ancestor (by calling a Prolog Prolog interpreter) and continues at B7.

B6=B5.yes. If so, it finds out whether the ancestor has a next solution (i.e. solves again). It does this by calling on the Prolog Prolog

interpreter to find if there are at least N+1 solutions to the ancestor, where N is the number of times backtracking to it has taken place. Continues at B7.

B7.

B7.1. If the ancestor solves, then the debugger switches off debugging and backtracks from the failed goal continuing execution until that ancestor solves, whence it returns to the missing solution debugging mode at step B3.

B7.2.

If the ancestor does not solve, then it queries you whether this failure is legitimate or not. It asks if the ancestor goal is i) inadmissible u) unsolvable or s) solvable.

case i) It requests you to point at some wrong term within it, reverts to wrong solution debugging mode, and follows part A of the algorithm at step A3, with the new term.

case u) Failure of the ancestor was to be expected ; it then switches off debugging and backtracks from the failed goal until the ancestor is reached, and it returns to the missing solution debugging mode at step B4.

case s) The ancestor becomes the currently known lowest unsolving ancestor and debugging continues at step B8.

B8. The failed goal is reconsidered, and a failure analysis takes place :

B8.1. There are no clauses for the goal. You are told so and debugging terminates.

B8.2. The goal fails before any backtracking to it occurred (i.e. it did not produce a solution). The debugger inquires whether the goal is

i) inadmissible u) unsolvable or s) solvable.

case i) It prompts you to point at some wrong term within it and reverts to wrong solution debugging mode at step A3 with the new term.

case u) The failure is justified ; it then simply proceeds with the execution by backtracking from the goal (note that selective backtracking [Pereira and Porto 82] is used throughout by the debugger, so that subderivations irrelevant to failures are skipped and no unnecessary questions about them are posed).

case s) If there is no matching clause, this bug is reported and debugging terminates. Otherwise the bug is that no matching clause solves its body. All the failures in its execution are legitimate, for otherwise they would have been detected previously, since they must have occured and been considered before the present one. The goal is reported as having a missing solution and debugging terminates.

B8.3. The goal fails after backtracking to where it was spawned (i.e. it had produced a solution before). The debugger inquires whether the goal is i) inadmissible u) unsolvable or s) solvable.

case i) Same as in B8.2.
case u) Since it has solved before, that solution is wrong ; debugging reverts to wrong solution mode and tackles that goal starting at A1.
case s) Although it has solved before, not all intended solutions may have been produced ; the debugger calls on a standard interpreter to compute all produceable solutions and inquires if they are correct and a complete set, by asking you to indicate if some solution is m) missing w) wrong or n) neither missing nor wrong.

case m) If they are not all the solutions, a goal has been detected with

a solution missing whose descendents have no missing solutions, for otherwise they would have unjustifiedly failed before and been reported.

case w) If one of them is incorrect, wrong solution mode can be entered to debug the goal.

case n) If they are all the solutions, the failure is legitimate and backtracking ensues ; debugging proceeds at step B4.

COMPARISON TO OTHER WORK

The only other similar work is algorithmic debugging [Shapiro83]. The following are the main contrasts between rational (R) and algorithmic (A) debugging :

(A1) The level of discourse is that of literals.

(R1) The level of discourse is finer, being that of terms within literals.

(A2) Asks unnecessary questions because it cannot jump over subderivations irrelevant to a bug.

(R2) Follows dependencies of terms on derivation steps, enabling it to jump over irrelevant subderivations, so that its questions are to the point. Its questions are not only sufficient but necessary as well.

(A3) Cannot switch from missing solution mode to wrong solution mode.
(Incidently, that's why the user has to input all the correct ground instances in its missing solution algorithm.)

(R3) Does switch from one mode to another as shown in the rationale above.

(A4) Does not make use of Prolog's operational semantics.

(R4) Uses Prolog's declarative and operational semantics cooperatively.

In short, (R) converges faster than (A) on bugs, with less user effort, like

the protocols below exemplify. The difference becomes bigger with larger programs.

RATIONAL DEBUGGER PROTOCOLS (on the examples in [Shapiro 83])

The terminal cursor is used to point at some symbol by placing it over the symbol and pressing <RET>. Here the symbol "^" indicates the position immediately below the cursor. In the protocols that follow, the comments on the right refer to the the steps at issue of the debugging algorithm.

[Shapiro 83] page 41 example. The insert sort below is used for debugging :

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

isort([],[]).

insert(X,[Y|Ys],[Y|Zs]) :- Y>X, insert(X,Ys,Zs).    /* should be Y<X */

insert(X,[Y|Ys],[X,Y|Ys]) :- X=<Y.

insert(X,[],[X]).
```

Terminal session :

```
?- wrong isort([2,1,3],S).

isort([2,1,3],[2,3,1])
           ^
top goal solution ok ? n                          steps A1, A2
use arrow to point at wrong term.

insert(2,[3,1],X1)
         ^
admissible goal ? n                               steps A3, A4, A5, A2
use arrow to point at wrong term.                 insert expects a sorted
                                                  list as 2nd argument
insert(1,[3],X1)
admissible goal ? y                               steps A3, A4

insert(1,[3],[3|X1])
              ^
solvable goal ? n                                 steps A6, A7
use arrow to point at wrong term.

insert(1,[],[1])
                                                  step A8.2
```

```
state whether this goal is :
i) inadmissible       t) true   f) false
choose one : t

Error in clause

insert(X1,[X2|X3],[X2|X4]) :- X2>X1, insert(X1,X3,X4).

in misused variable having an occurrence in the head,
or goal missing in body.
```

[Shapiro 83] page 50 example. Another insert sort with an error :

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

isort([],[]).


insert(X,[Y|Ys],[X,Y|Ys]) :- X=<Y.

insert(X,[Y|Ys],[Y|Zs]) :- insert(X,Ys,Zs).          /* Y<X missing */

insert(X,[],[X]).
```

Terminal session :

```
?- wrong isort([4,1,2,3,5,6],S).

isort([4,1,2,3,5,6],[1,2,3,4,5,6])
top goal solution ok ? y                              step A1
S = [1,2,3,4,5,6] ;


isort([4,1,2,3,5,6],[1,2,3,5,4,6])
                      ^
top goal solution ok ? n                              steps A1, A2
use arrow to point at wrong term.

insert(4,[5,6],X1)
admissible goal ? y                                   steps A3, A4

insert(4,[5,6],[5|X1])
               ^
solvable goal ? n                                     steps A6, A7
use arrow to point at wrong term.

insert(4,[6],[4,6])
                                                      step A.8.2
state whether this goal is :
i) inadmissible       t) true   f) false
choose one : t
```

Error in clause

```
insert(X1,[X2|X3],[X2|X4]) :- insert(X1,X3,X4).
```

in misused variable having an occurrence in the head,
or goal missing in body.

[Shapiro 83] page 56 example. Yet another insert sort for debugging :

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

isort([],[]).


insert(X,[Y|Ys],[Y|Zs]) :- Y<X, insert(X,Ys,Zs).

insert(X,[Y|Ys],[X,Y|Ys]) :- X=<Y.

/* insert(X,[],[X]). is missing */
```

Terminal session :

```
?- fails isort([3,2,1],S).                          steps B1, B2, B3
missing solution next ? y                           insert(1,[],X1) failed

isort([1],X1)                                           ancestor found
                                                    steps B4, B5, B6, B7.2

state whether this goal is :
i) inadmissible       u) unsolvable   s) solvable
choose one : s

insert(1,[],X1)                                                 step B8.2

state whether this goal is :
i) inadmissible       u) unsolvable   s) solvable
choose one : s
```

That there is no matching clause is the reason for a missing solution to the goal

```
insert(1,[],X1)
```

activated in clause

```
isort([X1|X2],X3) :- isort(X2,X4), insert(X1,X4,X3).
```

ACKNOWLEDGEMENTS

This work was partly done during a sabbatical leave, at the Artificial Intelligence Technology Group, Digital Equipment Corporation, Hudson MA,USA.

It's also part of an DEC External Research Project on Expert Systems in Prolog, and of a contract with the Junta Nacional de Investigacao Cientifica (JNICT), Portugal. Thanks are due to Luis Monteiro, Roger Nasr, Michael Poe and Karl Puder for their comments.

REFERENCES

[Bruynooghe and Pereira 84] Bruynooghe,M. ; Pereira, L.M.

Deduction revision through intelligent backtracking
in "Issues in Prolog Implementation",
(J.Campbell ed.), Ellis Horwood Ltd. 1984

[Pereira and Porto 82] Pereira, L.M. ; Porto, A.

Selective backtracking
in "Logic Programming",
(K.Clark, S.Tarnlund eds.), Academic Press 1982

[Shapiro 83] Shapiro, E.

"Algorithmic Debugging"
M.I.T. Press 1983