RATIONAL LOGIC PROGRAMMING DEBUGGING

Luís Moniz Pereira

Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica
Portugal

(September 86)

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 which 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. The approach could be used for logic programming and theorem proving in general, and could be particularly important when adapted to distributed systems, where it is hard to use traditional sequential debugging tools.

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.

The relationship of debugging to fault finding and explanation giving is brought to light within this approach.

This work is augments on [Pereira 86].

Introduction

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

The presupposition 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 Pereira 84, Chen Lassez Port 86, Mannila Ukkonen 86].

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. Incorrections are diagnosed but not perfected, according to this definition.

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 Pereira 84]). The approach could conceiveably be adopted for logic programming and theorem proving in general, and distributed systems.

The debugger contrasts 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 82,83]), nor to know the operational semantics of Prolog programs (which renders the debugger helpful for knowledge acquisition validation by the non–programmer). All answers from 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 82,83] is improved upon, though through an altogether different approach. No improvement is made however in what regards detecting failure to terminate.

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 Pereira 84] that keeps track of where each binding (or match with an empty binding, or failure to match) 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 during debugging for efficiency. 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. Predicates may be stated correct, and thereby enhance debugging efficiency. Nevertheless, predicates incorrectly stated so can always be detected by the debugger.

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, all i/o, "assert" and "retract" usage, "var", "nonvar", "metavar", certain extra–logical built–in predicates, etc. (These, as well as implementation details must be the subject of another paper.) Suffice it to say that they are accounted for, indeed the whole of Prolog. In particular, all of a program's side-effects are undone or redone, including terminal i/o, when the debugger reinstates some derivation context. Of course, the debugger's own side–effects are untouched.

Before describing the algorithm though, the close relationships between debugging, fault finding and explanation giving are brought to light within this approach. Then the modified unification algorithm that keeps track of dependencies is recalled. Afterwards comparison to other work is made, and session protocols are exhibited, followed by a comment on future development.

Rational Fault Finding

Fault-finding is akeen to debugging. In the context of Logic, both arise as a confrontation between theory and model. Whereas in debugging one confronts an erroneous theory, in the form of a set of clauses, with models in the form of input/output pairs, in fault–finding one confronts a perfect theory (a set of clauses acting as a blueprint for some artifact) with the imperfect input/ouput behaviour of the artifact (which, if it were not faulty, would behave as a model of the theory).

What is common to both is the mismatch. The same techniques used in debugging to pinpoint faulty clauses can be used to find the clauses in a perfect blueprint which are at odds with artifact behaviour. Then, by means of the correspondance from the blueprint's clauses to the artifact's subcomponents whose i/o behaviour they emulate, the faulty ones can be identified.

Of course, the tracking down process may require several tests, and these are to be automatically generated so as to discriminate among competing faulty component hypotheses (complications will arise when several faults concur).

The modelling of artifacts is part of the problem. But the blueprints need only describe an artifact to the level of replaceable components desired, and the clauses need only describe a component's behaviour, not its internals. Qualitative modelling may be enough for the purpose at hand, dispensing with elaborate overdetailed mathematical physics.

These ideas on how debugging and fault-finding relate are new, the attractiveness of the approach being its basis on Logic. In the same vein that one can obtain a general rational debugger for Prolog programs, as previously described, irrespective of the program domain, one can aim now at constructing a general fault-finding procedure, whatever the faulty artifact may be, just as long as it can be modelled by clauses. This differs from the domain-specific approaches where the possible faults must be provided beforehand in each case.

An extension of the above stance concerns planning. To wit, an artifact may be faulty because it is wrongly put together. Removal of a fault requires then correction of the actions or action plan generator that constructed the artifact. The blueprint which prescribes and judges behaviour according to theory may be imaculate, but the action plan generator that constructs artifacts reproachable. When both blueprint and generator coexist in a system that detects errors in itself by finding fault in its deeds we come full circle to the rudiments of self-criticism.

Explanations

In what concerns explanations, one idea is again to use the basic technology behind the annotation of terms with their dependencies on derivation nodes, already in use in the rational debugger (and foreseen also for the fault-finder as explained above).

Another idea is to couple the previous one with the techniques we have developed for writing meta-interpreters in Prolog (in fact the rational debugger is written in Prolog using such techniques [Pereira 84, Porto 84]). With them, it is easy to provide at each derivation node information about the partial derivation so far and the outstanding goals, and hence HOW and WHAT-FOR or WHY explanations. WHY-NOT explanations require distinguishing relevant from irrelevant failures, which is what the debugger does when tracking down the reasons for a top goal failure.

In fact research can start by enhancing the debugger to explain its reasonings. When a program has no bugs, this will naturally lead to a more rational way of generating explanations.

Useful notions relevant for providing explanations in a logic programming setting are found in [Cox Pietrzykowski 86]

The Modified Unification Algorithm that Keeps Track of Dependencies

For a full treatment of this topic the reader is advised to consult [Bruynooghe Pereira 84, Pereira Porto 82]. In there, simplifications are described that render the algorithm computationally efficient.

A deduction tree is a skeletal proof tree, i.e. a proof tree with all the substitutions completely ignored. Each deduction tree is an extension of a previous one, provided by a modified unification algorithm. To obtain the wanted behaviour we associate a deduction tree with each term involved in unification. With a call, which is a term, we associate the nodes necessary for the existence of that call, i.e. those on the path from the root to the call (including the call). With the head of an applied clause, also a term, we associate the empty deduction tree (the clause is given). As a result of the unification algorithm below, each substitution has an associated deduction tree, which is the minimal one necessary for obtaining that substitution. When the unification algorithm consults a substitution component, it requires the existence of the associated deduction tree. The formal presentation of the algorithm follows, where t-T denotes a term t with associated deduction tree T.

1. matching t-T1 with t-T2 generate 'empty' (the empty substitution) with deduction tree $T1 \cup T2$(this information is important to the debugger; a possible bug could be that the t's should differ).
2. matching f(t1,...,tm)-T1 with g(t1,...,tn)-T2 generate 'failure' with $T1 \cup T2$ as an inconsistent deduction tree (this information is important to guide backtracking, but we'll not explain that here).
3. matching f(t1,...,tm)-T1 with f(r1,...,rn)-T2 (with `n>0` and some $ti \neq ri$) match, for each i, ti-T1 with ri-T2.
4. matching x1-T1 with t2-T2 where t2 is not a free variable, and a substitution component `x1<-t` exists with associated deduction tree T, match $t - T1 \cup T$ with t2-T2.
5. matching x1-T1 with t2-T2 with x1 a free variable, generate the substitution component `x1<-t2` with $T1 \cup T2$ as deduction tree.

The Rational Debugging Algorithm and Its Rationale

Two basic algorithm modes are used: one for tracking down wrong solutions, the other to account for missing solutions to derivations that terminate. For the failure to terminate case, we have no improvement over depth-bound checking.

A) Wrong Solution Mode

The basic idea in this mode is to thread back through a derivation tree, by successively following a wrong term's dependencies on derivation nodes, until at some node a clause instance is found whose premises are all true but whose conclusion is false and produces or passes back the wrong term.

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 found 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 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, but the bindings are undone all the same. The dependency set includes both the productive dependencies (where bindings were produced) and the consumptive dependencies between functors (where bindings are checked equal, i.e. consumed but not produced – empty bindings).

The former are those used by selective backtracking Prolog [Pereira 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 A3 with the new term, unless the term is textual in the goal, in which case a bug has been found.

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). No bindings beyond those of the match are shown, although the debugger still retains the clause instance used at this derivation node, including all the bindings produced by the solved body goals (these can be useful further on); all other bindings subsequent to the execution of the clause instance have been forgotten on backing to the present node.

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. (cf. A9 for the A6.yes case).

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 first inquires about the clause head by asking 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. Next, it inquires about the clause body.

A8.1. If the body is empty, then a wrong unit clause has been found, and the debugger, to be informative, reports an error according 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 solved subgoal instances (except for the built-in ones– these are assumed correctly solved) 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. If it's textual a bug is reported, otherwise it reverts to the wrong solution debugging mode 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 returns to Prolog.

Whatever the case, the algorithm does not continue below at A9, since it always jumps to some other step or returns to Prolog.

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

If you were asked whether the matched clause head was "true" (that's when the clause body is empty) continue at A4 (note the question implies universal quantification). Otherwise, if the body is not empty, you were asked if the matched clause head was "solvable" (note the question implies existencial quantification).

The wrong term being tracked down might not be a variable, or alternatively, it is a variable, and then an over general solution is being produced.

In either case, the clause invocation and execution are not blameable for producing the wrong term being tracked down. Note that the wrong term cannot have been produced by the clause body. If it were so, any node producing it would have already been examined and detected before, since the wrong term's dependencies are visited in reverse time order.

Although the matched head literal is solvable, a wrong solution may be being produced by the clause. So next you are asked if the solved clause head is true. If "yes" continue at A4. If "no", then to detect whether any of the body goals should have failed so that the wrong conclusion would not follow, you are queried about each of them in turn, as in A8.2.

B) Missing Solution Mode

The basic idea in this mode is that for a goal to have a missing solution, i.e. to fail to produce some solution, it is necessary that at least one goal that failed in the attempt should not have failed. The problem is then to find one such "uncovered" failing goal for which there is no "uncovered" subgoal beneath (which could be making it fail).

The user knows whether a goal should fail, but we don't want to prompt him about each goal that fails, i.e. we need to distinguish relevant from irrelevant failures. A failure is irrelevant if there is some ancestor of the goal that solves when enough backtracking is allowed. Too low an ancestor may not have enough backtracking chances. It is also irrelevant if some ancestor is unsolvable. Furthermore, if we find some solvable ancestor that doesn't solve, we can lower the ceiling on the bug, as it were, for we have found a goal with a missing solution lower on the tree than the original one. Thus, from that point of view, the lower the ancestor the better.

Given a failing goal, what the algorithm does, basically, is to identify an appropriate ancestor (using an heuristic spelled out in the sequel) about which it tries to obtain an answer to the above cases. Only if the failing goal is not found to be irrelevant is the user queried about it.

Whilst looking for a missing solution, a wrong solution to a goal may be found (that might in fact explain why the other is missing). The user can then revert to wrong solution mode on that goal immediately.

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. If there is no such ancestor continue at B8.

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. Continue at 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, where 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 or matching clauses for the goal. You are told which is the case and debugging terminates.

B8.2. There are matching clauses, but the goal fails without ever occurred producing a solution. (Cf. B8.3 otherwise.)

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 Porto 82] is used throughout by the debugger, so that subderivations irrelevant to failures are skipped and no unnecessary questions concerning them are posed).

case s) The bug is that no matching clause solves its body.

All the failures in the execution of any matching clause 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 had solved before, and fails after backtracking to where it was activated.

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 [Shapiro 82,83, Fernand 85, Av-Ron 84]. [Lloyd 86] adapts it to MU-Prolog, and [Lloyd Takeuchi 86, Takeuchi 86] to GHC [Ueda 85]. [Eisenstadt et al.84, Eisenstadt 85] are recent developments of traditional trace and spy methods.

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 does not know how to ignore other irrelevant to a bug.

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

(A3) Requires the user to supply correct and complete sets of argument instances in response to queries.

(R3) Requires the user only to recognise some wrong term in arguments, or incomplete sets of solutions, but not to produce them.

(A4) Intrinsically, it cannot dynamically switch from missing solution mode for a goal to wrong solution mode for another goal. (Incidently, that's why the user has to input all the correct ground instances in its missing solution algorithm, so as to guarantee that satisfiable goals are indeed correctly solved.)

(R4) Does switch from one mode to another as shown in the rationale above (Cf. B7.2, B8.2, B8.3 above).

(A5) Does not use knowledge about Prolog's operational semantics, except that it's top-down.

(R5) Extensively uses Prolog's declarative and operational semantics as the rationale above makes amply clear.

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

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. There are also comments regarding questions that are not asked because the answer to them is subsumed by some previous answer. **Example of append**

```
append([],L,L).
append([H|T1],T2,[H|T3]) :- append(T1,t2,T3).    % error: t2 should be T2

?- wrong append([1],[2],P).           % the debugger is invoked on a goal
                                      % with a wrong solution
append([1],[2],[1|t2])  top goal solution ok ? n           % steps A1, A2
                  ^     use arrow to point at a wrong term.

append([],t2,X)  admissible goal ? n                    % steps A3, A4, A5
          ^      use arrow to point at a wrong term.

Error in clause   append([H|T1],T2,[H|T3]) :- append(T1,t2,T3).
in textual functor in goal   append(T1,t2,T3)
```

[Shapiro 83] page 41 example

```
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]).

?- 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 a wrong term.

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

insert(1,[3],[3|X])  solvable goal ? n                  % steps A6, A7
              ^      use arrow to point at a wrong term.
insert(1,[],[1])                                        % step A8.2
```

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

Error in clause   insert(X,[Y|Ys],[Y|Zs]) :- Y>X, insert(X,Ys,Zs).
in misused variable having an occurrence in the head, or goal missing in
body.
```

[**Shapiro 83**] **page 50 example**

The same program with a different bug, where the isort clauses remain unchanged.

```
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]).

?- 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] ;                                  % Prolog top level answer

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],X)  admissible goal ? y                        % steps A3,A4

insert(4,[5,6],[5|X])  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(X,[Y|Ys],[Y|Zs]) :- insert(X,Ys,Zs).
in misused variable having an occurrence in the head, or goal missing in
body.
```

[**Shapiro 83**] **page 56 example**

The same program with a different bug, where the isort clauses remain unchanged.

```
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

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

```
isort([1],X)                      % ancestor found; steps B4, B5, B6, B7.2

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

insert(1,[],X)                                         % 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,[],Ys)   activated in clause
        isort([X|Xs],Ys) :- isort(Xs,Zs), insert(X,Zs,Ys).
```

Example sublist (shows you can have wrong overgeneral answers)

```
sublist([],X).                              % wrong definition for case []
sublist([X|Y],[X|Z]) :- sublist(Y,Error).   % 'Error' should be 'Z'
sublist(Y,[_|Z]) :- sublist(Y,Z).

?- wrong sublist(X,Y).

sublist([],X)  top goal solution ok ? n                % steps A1, A2
           ^   use arrow to point at a wrong term.

sublist(X1,X2)   admissible goal ? y                   % steps A3, A4, A6

        % sublist([],X) true? n  subsumed above        % step A7, A8.1

Error in clause   sublist([],X).
in misused variable having an occurrence in the head; or goal missing in
body.

% here the first clause is replaced by:  sublist([],[]).


?- wrong sublist(X,Y).                         % debugging continues


sublist([],[])  top goal solution ok ? y               % step A1
X=[]
Y=[] ;

sublist([X1],[X1|X2])  top goal solution ok ? y        % step A1
X=[X1]
```

```
Y=[X1|X2] ;


sublist([X1,X2],[X1|X3])  top goal solution ok ? n      % steps A1, A2
                    ^     use arrow to point at a wrong term.

        % sublist(X1,X2)  admissible goal? y  subsumed above; steps A3, A4

sublist([X1|X2],[X1|X3])  solvable goal ? y             % step A6

        % sublist([X1,X2],[X1|X3])  true? n  subsumed in top goal; step A9

        % sublist([X1],[X1|X2])  i) t) or f) ? t subsumed above; step A8.2

Error in clause   sublist([X|Y],[X|Z]) :- sublist(Y,Error).
in unsufficiently constrained variable present in the head.
```

[**Shapiro 82**] **quicksort example** (with 6 bugs)

```
qsort([X|L],L0) :- partition(L,X,L1,L2)   % qsort([],[]).   is missing
                   qsort(L1,L3), qsort(L2,L4),
                   append([X|L3],L4,L0).  % should be append(L3,[X|L4],L0)

partition([X|L],Y,L1,[X|L2]):- partition(L,Y,L1,L2)       % X>Y is missing
partition([X|L],Y,[X|L1],L2):- Y=<X, partition(L,Y,L1,L2) % should be X=<Y
partition([],X,[],[]).

append([X|L1],L2,[X|L3]) :- append(L1,L2,L2).  % should be append(L1,L2,L3)
append([],L,[]).                               % should be append([],L,L)

?- fails qsort([2,1],P).                       % steps B1, B2, B3
missing solution next ? y                      % qsort([],L3) failed
                                % steps B4, B8: no appropriate ancestor
qsort([],X)                                    % step B8.1
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  qsort([],L3)    activated in clause
      qsort([X|L],L0) :- partition(L,X,L1,L2),
         qsort(L1,L3), qsort(L2,L4), append([X|L3],L4,L0).

        % here   qsort([],[]).  is added to the program

?- fails qsort([2,1],P).           % debugging continues; steps B1,B2,B3
missing solution next ? y          % append([],[1|X],[1|X]) failed
```

```
                                % steps B4, B5, B6, B7, B7.2
append([2],[1|X1],X2)           % this is the ancestor
state whether this goal is:  i) inadmissible  u) unsolvable  s) solvable
choose one: s

append([],[1|X],[1|X])          % steps B8, B8.1
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   append([],[1|X1],[1|X1])   activated in clause
        append([X|L1],L2,[X|L3]) :- append(L1,L2,L2).

        % here the second clause of append is changed to  append([],L,L).

?- wrong qsort([2,1],P).                        % debugging continues

qsort([2,1],[2|X])  top goal solution ok ? n              % steps A1, A2
               ^    use arrow to point at a wrong term.

 % append([2],[1|X1],X2)  admissible goal? y  subsumed above; steps A3, A4

append([2],[1|X1],[2|X2])  solvable goal ? y              % step

append([2],[1|X1],[2|X2])  true ? n                       % step A9
                     ^     use arrow to point at a wrong term.

append([],[1|X],[1|X])  i) t) or f) ? t                   % step A8.2

Error in clause   append([X|L1],L2,[X|L3]) :- append(L1,L2,L2).
in unsufficiently constrained variable present in the head.

        % here the first clause for append is replaced by
        % append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

?- wrong qsort([2,1],P).                        % debugging continues

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

append([2],[1],X)  admissible goal ? y                    % steps A3, A4

append([2],[1],[2|X])  solvable goal ? y                  % step A6

append([2],[1],[2,1])  true ? y                           % step A9
```

```
partition([],2,X1,X2)  admissible goal ? y                   % step A4

partition([],2,[],[])  true ? y                              % steps A6, A9

partition([1],2,X1,X2)  admissible goal ? y                  % step A4

partition([1],2,X1,[1|X2])  solvable goal ? n                % steps A6, A7
                    ^       use arrow to point at a wrong term.

 % partition([],2,[],[])  i) t) or f) ? t  subsumed above; steps A8, A8.2

Error in clause   partition([X|L],Y,L1,[X|L2]) :- partition(L,Y,L1,L2).
in misused variable having an occurrence in the head; or goal missing in
body.

        % here the first clause for partition is replaced by
        % partition([X|L],Y,L1,[X|L2]) :- X>Y, partition(L,Y,L1,L2).

?- fails qsort([2,1],P).            % debugging continues; steps B1,B2,B3
missing solution next ? y           % partition([1],2,L1,L2) failed
                                    % steps B4,B8: no appropriate ancestor
partition([1],2,X1,X2)              % step B8.2
state whether this goal is  i) inadmissible  u) unsolvable  s) solvable
choose one: s

The clauses for it legitimately fail, so there is a missing solution to
the goal   partition([1],2,L1,L2)   activated in clause
    qsort([X|L],L0) :- partition(L,X,L1,L2), qsort(L1,L3), qsort(L2,L4),
       append([X|L3],L4,L0).

        % here the second clause for partition is replaced by
        % partition([X|L],Y,[X|L1],L2) :- X=<Y, partition(L,Y,L1,L2).

?- wrong qsort([2,1],P).                % debugging continues

 % qsort([2,1],[2,1])  top goal solution ok ? n       % steps A1, A2
 %              ^      use arrow to point at a wrong term; subsumed above

append([2,1],[],X)  admissible goal ? y                    % steps A3, A4

append([2,1],[],[2|X])  solvable goal ? y                  % step A6

append([2,1],[],[2,1])  true ? y                           % steps A9, A4

 % partition([],2,X1,X2)  admissible goal? y  subsumed above; step A6
```

```
 % partition([],2,[],[])  true? y            subsumed above; steps A9, A4

 % partition([1],2,X1,X2) admissible goal? y  subsumed above; step A6

partition([1],2,[1|X1],X2) solvable goal? y                 % steps A3, A4

partition([1],2,[1],[])  true ? y                           % steps A9, A4

qsort([2,1],X)  admissible goal ? y                         % step A6

qsort([2,1],X)  solvable goal? y                            % step A9

 % partition([1],2,[1],[])  true ? y  subsumed above; step A8.2

qsort([1],[1])  true ? y                                    % step A8.2

qsort([],[])  true ? y                                      % step A8.2

append([1],[],[1])  true ? y                                % step A8.2

Error in clause   qsort([X|L],L0) :- partition(L,X,L1,L2), qsort(L1,L3),
                                     qsort(L2,L4), append([X|L3],L4,L0).
in wrongly constrained variable present in the head.

   % here the first clause for qsort is changed to
   % qsort([X|L],L0) :- partition(L,X,L1,L2), qsort(L1,L3), qsort(L2,L4),
   %                    append(L3,[X|L4],L0).

?- qsort([2,1],P).
P = [1,2] ;
no              % debugging of this goal is complete; truth at last!
```

Future Developments

The approach could be used for logic programming and theorem proving in general. It can be particularly important to explore its use in distributed systems, where it is hard to employ traditional sequential debugging tools. We intend to do that for Delta Prolog [Pereira Nasr 84, Pereira et al. 86].

At present, it is being made more congenial and helpful through integration into a C-Prolog/Emacs multi-window environment [Abreu 85], including for example the highlighting of wrong terms or uncovered goals in detected bugs. It will unburden the user in his rôle as oracle to allow the giving of conditions on goals concerning admissibility, solvability and truth.

Fault finding and explanation giving, as detailed above, have strong ties with this approach to debugging. We intend to bring them to fruition shortly.

A formal account of rational debugging, and for that matter of intelligent backtracking, is needed. We believe it to be a worthwhile research topic.

Acknowledgements

Thanks are due to Ivan Bratko, Phil Cox, Luis Monteiro, Roger Nasr, Michael Poe and Karl Puder for their comments, and to Salvador Abreu for integrating the debugger into the C-Prolog/Emacs environment. This work is part of a contract with the Junta Nacional de Investigação Científica e Tecnológica(JNICT), Portugal, and other contracts.

References

[Abreu 85] Abreu, S.P.
The UNL Emacs/Prolog interface, Dept. of Computer Science, Universidade Nova de Lisboa

[Av-Ron 84] Av-Ron, E.
Top-down diagnosis of Prolog programs, MSc thesis, Weizmanm Institute

[Bruynooghe Pereira 84] Bruynooghe, M.; Pereira, L.M.
Deduction revision through intelligent backtracking, in "Issues in Prolog Implementation" (J.Campbell ed.), Ellis Horwood

[Chen Lassez Port] Chen, T.Y. ; Lassez J-L ; Port G.S.
Maximal unifiable subsets and minimal nonunifiable subsets, New Generation Computing vol.4, no.2

[Cox Pietrzykowski 86] Cox, P. ; Pietrzykowski, T.
Causes for events: their computation and application, Proc. 8th Conf. on Automated Deduction, Lecture Notes in Computer Science, Springer

[Eisenstadt et al.84] Eisenstadt, M.; Hasemer, T.; Kriwaczek, F.
An improved user interface for Prolog, INTERACT'84, 1st. IFIP Conf. on "Human-Computer Interaction"

[Eisenstadt 85] Eisenstadt, M.
Tracing and debugging Prolog programs by retrospective zooming, report 17, Human Cognition Lab, The Open University, UK.

[Ferrand 85] Ferrand, G.
Error diagnosis in logic programming, an adaptation of E.Y.Shapiro's method, Rapport de Recherche 375, INRIA, Rocquencourt, 78153 Le Chesnay, France

[Lloyd 86] Lloyd, J.
Declarative error diagnosis, Dept. of Computer Science report, Univ. of Melbourne

[Lloyd Takeuchi 86] Lloyd, J. ; Takeuchi, A.
A framework for debugging GHC, ICOT report TR-186

[Mannila Ukkonen 86] Mannila, H. ; Ukkonen,E.
Timestamped term representation for implementing Prolog, IEEE Symposium on Logic Programming, Salt Lake City

[Pereira 84] Pereira, L.M.
Logic control with logic, in “Issues in Prolog Implementation” (J.Campbell ed.), Ellis Horwood

[Pereira 86] Pereira, L.M.
Rational debugging of logic programs, in “3rd Int. Logic Programming Conf., Lecture Notes in Computer Science 225 (E.Shapiro ed.), Springer

[Pereira Nasr 84] Pereira, L.M.; Nasr, R.
Delta Prolog: a distributed logic programming language, in “Proc. of 1984 Int. Conf. on 5th Generation Computer Systems”, ICOT

[Pereira et al.86] Pereira, L.M.; Monteiro, L.; Cunha,J.C.; Aparício, J.N.
Delta Prolog: a distributed backtracking extension with events, in “3rd Int. Logic Programming Conf., Lecture Notes in Computer Science 225 (E.Shapiro ed.), Springer

[Pereira Porto 82] Pereira, L.M. ; Porto, A.
Selective backtracking, in “Logic Programming” (K.Clark, S.Tarnlund eds.), Academic Press

[Porto 84] Porto, A.
Two-level Prolog, in “Proc. of 1984 Int. Conf. on 5th Generation Computer Systems”, ICOT

[Shapiro 82] Shapiro, E.
Algorithmic program debugging, in “Proc. of 9th annual ACM Symp. on Principles of Programming Languages”

[Shapiro 83] Shapiro, E.
“Algorithmic Debugging”, M.I.T. Press

[Takeuchi 86] Takeuchi, A.
Algorithmic debugging og GHC programs and its implementation in GHC, ICOT report TR-185