The Use of Proof Planning Critics to Diagnose Errors in the Base Cases of Recursive Programs*

Louise A. Dennis
School of Computer Science and Information Technology, University of Nottingham,
lad@cs.nott.ac.uk

January 12, 2005

Abstract

This paper reports the use of proof planning to diagnose errors in program code. In particular it looks at the errors that arise in the base cases of recursive programs produced by undergraduates. It describes two classes of error that arise in this situation. The use of test cases would catch these errors but would fail to distinguish between them. The system adapts proof critics, commonly used to patch faulty proofs, to diagnose such errors and distinguish between the two classes. It has been implemented in $\lambda Clam$, a proof planning system, and applied successfully to a small set of examples.

The use of mathematical proof to show that a computer program meets its specification has a long history in Computer Science (e.g. [14, 13]). Considerable time and effort has been invested in creating computer-based tools to support the process of proving programs correct (e.g. [15, 8]). However the technique and tools are only used in very specialised situations in industry where programmers generally rely on testing and bug reports from users to assess the extent to which a program meets its specification.

There are several reasons why there is such poor uptake of the use of proof in industry. One is that the final proof will tell you if the program is correct, but failing to find a proof does not, on immediate inspection, help in locating errors. This problem can be particularly severe when using automated proof techniques which generally produce no proof trace in the case of failure. Many cases have been reported where the process of attempting a proof by hand has highlighted an error, for instance Paulson's discovery of new attacks against security protocols [16]. Anecdotal evidence suggests that errors are located by examining and reflecting on the process of the failed proof attempt.

It is worth noting the comparative success of model checking techniques (e.g. [11]). Model checkers are automated (though they require an expert user to convert the problem into an appropriate form) and return counterexamples when they fail. This confirms the analysis that automated support for error discovery is valuable and might aid a more widespread uptake of theorem proving technology.

This paper reports preliminary work using the proof planning paradigm (in particular the concept of proof critics) to diagnose the errors in program code. I focus on two classes

*This research was funded by EPSRC grant GR/S01771/01 and Nottingham NLF grant 3051

of error that can arise in the base case of a recursive program and show how a proof critic can be written to distinguish between these two situations.

1 Proof Planning

Proof planning [1] is an Artificial Intelligence based technique for the automation of proof. One aspect of proof planning is the inspection of failed proof attempts by means of *proof critics* [10] which attempt to patch the proof.

Proof planners use AI-style planning techniques to generate proof plans. A proof plan is a proof of a theorem at some level of abstraction. The main planning operators used by a proof planner are called *proof methods*.

The first proof planner, *Clam* [3], focused on proof by mathematical induction using the rippling heuristic (a form of rewriting constrained to be terminating by meta-logical annotations) [2]. λ*Clam* [19, 5], which I used for this work, is a higher-order descendant of *Clam*which incorporates both a hierarchical method structure and proof critics.

λ*Clam* works by using depth-first planning with proof methods. Each node in the search tree is a subgoal under consideration at that point. The planner checks the preconditions for the available proof methods at each node and applies those whose preconditions succeed to create the child nodes. The plan produced is then a record of the sequence of method applications that lead to a trivial subgoal. λ*Clam*'s proof methods are believed to be sound although they are not currently reducible to sequences of inference rule applications. This means that while λ*Clam* outputs something that can be considered a proof in a similar way to a pen-and-paper correctness proof it does not produce a fully-formal proof.

1.1 Proof Methods

Proof method application is governed by preconditions (which may be either legal or heuristic in nature) and by a *proof strategy* (or compound method) which restricts the methods available depending on the progress through the proof. For instance we may wish to simplify a goal as much as possible by applying a rewriting method exhaustively before considering other procedures such as checking for tautologies.

In λ*Clam* a proof method can be *atomic* or *compound*. If it is compound then it is a sub-strategy built up from other methods and *methodicals*[1] [18]. Methodicals exist for repeats, sequencing methods, creating OR choices etc. and so complex proof strategies for controlling the search for a proof can be created.

1.1.1 The Proof Strategy for Induction

The proof strategy for induction can be seen in figure 1[2]. The diagram shows a top level repeat which attempts a disjunction of methods (in λ*Clam* these are attempted

[1] Analogous to a tactical in an LCF style theorem prover.

[2] There is no clear semantics for the use of diagrams to represent proof strategies. In this case boxes are used to indicate methods (both atomic and compound) and arrows to indicate method sequencing. Methods within methods indicate the method hierarchy, arrows that branch show OR choices and methods with more than one exit arrow indicate that they produce several goals which are treated differently.

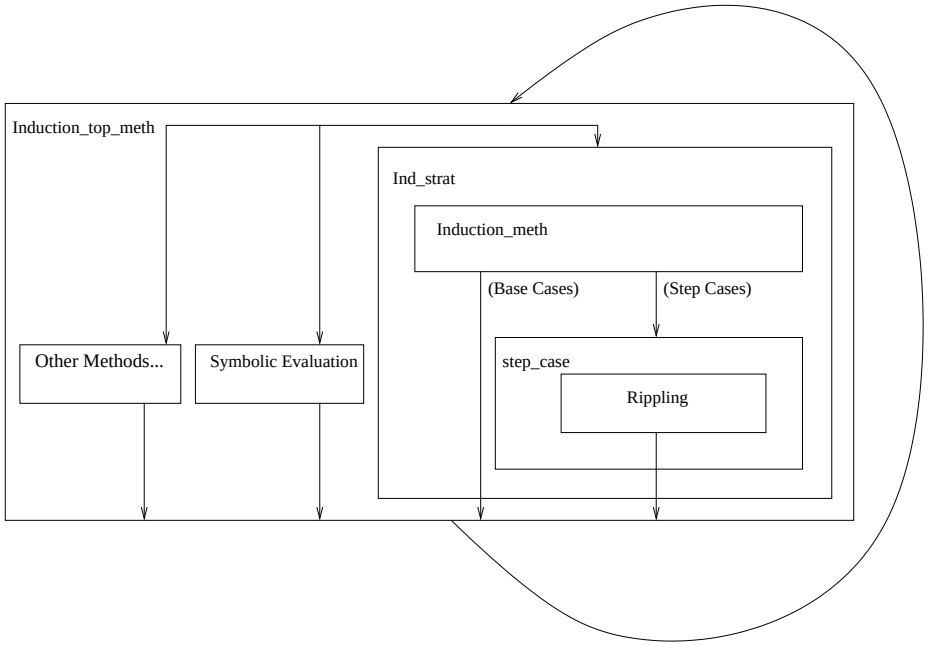

Figure 1: The Proof Strategy for Induction

from right to left, the planner backtracking out of failed choices). These include basic tautology checking, generalisation of common subterms and also symbolic evaluation and the induction strategy (`ind_strat`). Within the induction strategy, the induction method chooses an induction scheme and produces subgoals for base and step cases.

The top level strategy is reapplied to the base cases. The step cases are handled using rippling. The details of rippling are not important to the work described here and so are omitted from discussion. The results are then passed out to the top level strategy again. The process terminates when all subgoals have been reduced to *true*.

This proof strategy is used as the basis of the system for diagnosing errors in recursive programs discussed in this paper.

1.2 Proof Critics

A proof strategy provides a guide to which proof methods should be chosen at any given stage of the proof. Knowing which method is expected to apply gives additional information should the system generating the plan fail to apply it. Proof critics can be employed to analyse the reasons for failure and propose alternative choices. Critics are expressed in terms of preconditions and patches. The preconditions examine the reasons why the method has failed to apply. The proposed patch suggests some change to the proof plan or strategy. It may choose to propagate this change back through the plan and then continue from the current point, jump back to a previous point in the proof plan, or modify the current strategy being used by the planner, for instance by introducing new methods for consideration at that point.

In $\lambda Clam$v4, used for this work, critics can be built up into strategies using *criti-*

cals [9] in the same way that method strategies can be developed.

1.3 Related Work

Monroy [12] has already used proof planning to examine faulty conjectures. He follows work by Franova and Kodratoff [7] and Protzen [17] and attempts to synthesize a *corrective predicate* in the course of proof. The idea is that the corrective predicate will represent the theorem that the user intended to prove. This predicate is represented by a meta-variable, P, such that $P \rightarrow G$ where G is the original (non)theorem. P is instantiated during the course of a proof planning attempt. This approach assumes that, in some sense, the error arose because the original conjecture was too general.

The work reported here does not attempt to generate a corrective predicate. It seeks simply to diagnose the point in the code which is causing proof failure and allow a user to determine the appropriate modification. This allows for a more general class of errors to be identified beyond over-generalisations. Clearly there are advantages and disadvantages to both approaches and ideally they might at some point be combined into a system which both diagnosed the error and suggested a modification.

2 Novice Programs

In order to exploit the proof planning paradigm it is necessary to identify common patterns of proof in order to structure the proof strategy. In the case of faulty conjectures this would include identifying common patterns of proof failure which in turn requires the collection of a large body of data containing errors in order to observe the patterns of failure involved.

I have opted to study the programs produced by novice programmers, specifically undergraduates working on functional programming modules. It is relatively easy to acquire a large number of such programs and they are also likely to be well suited to proof by mathematical induction for which a mature proof strategy already exists (described above). On the downside such programs may not have a clear specification. Also, such programs may not contain the sort of bugs which we would expect to be generated by real programmers.

2.1 Errors in the Base Cases of Recursive Programs

I analysed a corpus of ML programs produced by students at the University of Edinburgh. This was a large set of approximately 150 scripts (attempting up to 4 problems) of which half were examined (the remainder being kept aside for later testing). These programs were all recursive in nature and a number of errors were identified and classified. This paper focuses specifically on errors occurring in the base cases of recursions.

An obvious approach to such problems is to filter the programs through test cases (in fact this is the approach adopted by many practitioners). However my analysis revealed two different ways in which errors may appear – these two different circumstances would not be distinguished by a counter-example alone.

The student programs contained some technical challenges for rippling which prevented the production of the proof plans showing that the step cases were correct. In

the rest of this paper I use a manufactured example in which the errors are duplicated but in which the basic problem is altered.

Consider the reverse function on lists[3] commonly defined as:

$$\begin{aligned} reverse(nil) &= nil, \\ reverse(X :: XS) &= reverse(XS) <> (X :: nil). \end{aligned}$$

The two errors that appeared to arise in student code were when either the base case of the recursion was incorrect in some way or it was omitted. A typical incorrect base case would be:

$$reverse(nil) = X :: XS.$$

The object of the research reported here was to distinguish between these two problems based on the failed proof attempt.

2.2 The Proof Strategy

The first challenge was to develop some sort of specification for the program. In this case I assume that the tutor has provided a "correct" version of the program and that the system is attempting to prove their equivalence[4]. In this case the initial proof goal is

$$\forall l.\, student\_reverse(l) = tutor\_reverse(l).$$

Hand proofs of these equivalences suggest that the proof stalls in the base case of the induction at either

$$X :: XS = nil$$

when the base case is incorrect or

$$student\_reverse(nil) = nil$$

when the base case is missing.

In the case of the incorrect base case both the goal has been reduced to a falsehood (assuming a free constructor specification).

In the second case (missing base case) we have two inequal terms, one of which has been reduced to variables and type constructors while the other still contains a defined term.

2.3 Critics for missing and incorrect base cases

This analysis suggested that there should be a critic on the symbolic evaluation method. Symbolic Evaluation is, in fact, a compound method consisting of repeated applications of a rewrite method and is shown in figure 2. I implemented a modification to this method so that a critic strategy is called if the `rewrite` method fails. This is shown in figure 3[5]. The critic strategy calls an atomic critic, `check_equalities`. This is

[3] In what follows we use *nil* to indicate the empty list, :: to indicate the cons function that joins an element to the front of a list and <> to indicate a built-in append function which joins two lists together.

[4] Obviously this scenario is unlikely in an industrial setting but was sufficient for the problem at hand.

[5] A dashed line is used here to indicate that the critic is invoked if the method fails.

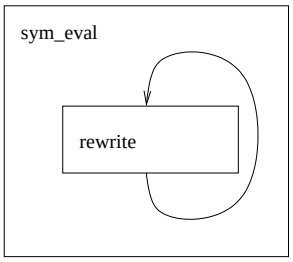

Figure 2: The Symbolic Evaluation Method

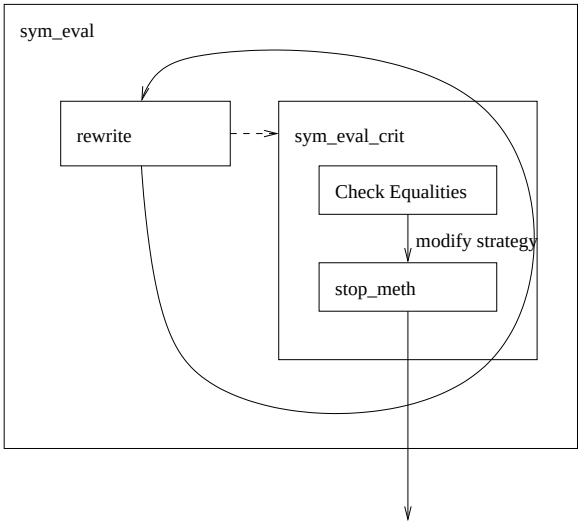

Figure 3: The Symbolic Evaluation Method and Base Case Diagnosis Critic

shown in figure 4 however I have chosen to present this in terms of preconditions and effects rather than patches since the critic does not patch the proof (or the theorem). The "known to be false" precondition checks a small internal list of non-theorems for a match – these assume a free constructor specification (ie. they contain $\neg(0 = s(N))$ where N is a variable) there are clearly some issues with this assumption and an obvious area for improving the critic is in making the implementation of this precondition more rigorous for instance by using I-Axiomatizations [4]. If its preconditions succeed then `check_equalities` processes its effects which in this case prints out an appropriate diagnosis message. If the critic succeeds the current strategy is changed so instead of proceeding as normal a method called `stop_meth` is invoked which closes the current proof branch immediately. Other proof branches are left open to be explored, potentially finding additional errors in the program (e.g. in the case where two base cases are incorrect or missing both are diagnosed – this occurs in some examples using the definition of *even*). If `check_equalities` fails then the system returns to the normal proof plan for induction.

Preconditions

Case 1 The current goal is known to be false.

OR

Case 2 The current goal is an equality and exactly one side of the equality is a simple term.

Effects

Case 1 Diagnose an incorrect base case.

Case 2 Diagnose a missing base case.

Figure 4: The Base Case Diagnosis Critic

3 Results

There are two sorts of results for the implementation of the base case diagnosis system. Firstly the system should correctly classify errors and secondly it should not diagnose actual theorems as faulty[6].

A handful of student programs were converted into λ*Clam*'s input format (modifying the programs in some cases because of the problems they posed to the step case proofs though preserving the errors) all of these were correctly diagnosed by the system. The student and tutor programs used are listed as an appendix to this paper. It should be noted that where a student has chosen and alternative base case, say 1 rather than 0, the program still diagnoses a missing base case since the two programs are not equivalent for 0 as input. It could be argued that this is instead an instance of an incorrect base case or even that it has arisen from an insufficiently well-defined specification on the part of the tutor. At the moment the system ignores these distinctions though it might be possible to extend it to identify rewrite rules defined from the student program that were not used in the proof.

The system was also run on λ*Clam*'s benchmark sets of theorems on lists and natural numbers – the new critic did not cause any of these to be incorrectly classified as faulty. The system can also prove the equivalence of the student program to the tutor one if if they have chosen a more complex form of recursion (eg. in the case of *reverse* having two base cases, one for the empty and one for the one element list and then a recursive case which removes two elements from the head of the list at a time).

[6]Within reason, I make no claims that this is a decision procedure which can never conclude that a true theorem has a flaw, however I want to be assured that the critic heuristic is *usually* correct.

4 Further Work

There is a risk that a proof has failed to go through because of some missing lemma, or inference rule. For instance suppose that constructors are not free but the system is unable to simplify $s(p(X))$ to X. In this case a falsehood could be detected where none exists. There are some obvious improvements which can be implemented to the existing falsehood detection system (already discussed above) but it would also be useful to link a counter-example generator into the system since the existence of an example where each program gave a different answer would guarantee that the student program was incorrect while the diagnosis system could provide additional information and guidance about the nature of the error.

Similarly in the detection of missing base cases it is possible that the student has supplied a base case which has been rewritten but to some term about which the system can not reason further (most obviously they may have used some built-in function which is not represented in the proof system). Tight integration between the student programming environment and the proof system would help overcome this but it would also be useful to detect whether the student side of the equation had been rewritten at all (in which case they have supplied some sort of base case in the program) or whether it was simply irreducible from the moment the goal was set up. It is also possible that they have implicitly made use of some equality between built-in functions which is not represented in the rewrite rules of the system. Once again the ability to generate a counter-example would provide a useful sanity check here.

My immediate intention is to port the existing proof plan and critic for base case diagnosis to IsaPlanner [6]. This is a newly developed proof planner containing much of the existing work on induction but also containing a wider knowledge base of rewrite rules, theorems and non-theorems and providing a more robust implementation base than $\lambda Clam$. Within this framework I hope to implement the more sophisticated ideas detecting falsehood and missing base cases discussed above. I also hope to investigate a wider set of examples and to use the actual programs produced by the students as the basis for theorems rather than porting their errors to functions more amenable to rippling.

4.1 Incorrect Step Cases

An obvious extension to this work is to look at errors occurring in the recursive case of functional programs. Several examples of these are also present in the data set. When the proof fails in these cases it takes place in the "tidying up" phase that follows the use of the `step case` method, once again failing during symbolic evaluation.

This will raise more serious issues about false positives since in a number of existing proof plans symbolic evaluation fails at this point, the method is backtracked out of and a subsidiary induction attempted[7].

[7] Lucas Dixon (personal communication) has suggested that a lemma speculation critic (already implemented in IsaPlanner) could solve this problem and would be called before the failure of symbolic evaluation.

5 Conclusion

This paper reported preliminary work to investigate the use of proof critics to diagnose program errors. It shows that, in principle at least, proof critics can be used to diagnose such errors and that they can be used to distinguish between different classes of error that would be picked up by the same counter-example. Potentially proof planning provides more information to a user about the nature of program error than a counter-example generator alone could.

References

[1] A. Bundy. A science of reasoning. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.

[2] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1. Elsevier, 2001.

[3] A. Bundy, F. van Harmelan, C. Horn, and A. Smaill. The oyster-clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 647–648. Springer, 1990.

[4] H. Comon and R. Nieuwenhuis. Induction = I-axiomatization + first-order consistency. *Information and Computation*, 159(1–2), 2000.

[5] L. A. Dennis and J. Brotherston. *λclam v4: User/Developer's Manual*. Mathematical Reasoning Group, Division of Informatics, University of Edinburgh, 2002.

[6] L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In F. Baader, editor, *19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Computer Science*, pages 279–283. Springer, 2003.

[7] M. Franova and Y. Kodratoff. Predicate synthesis from formal specification. In B. Neumann, editor, *10th European Conference on Artificial Intelligence*, pages 97–91. John Wiley and Sons, 1992.

[8] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[9] J. Gow. *The Dynamic Creation of Induction Rules Using Proof Planning*. PhD thesis, Centre for Intelligent Systems and their Applications, School of Informatics, University of Edinburgh, 2004.

[10] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

[11] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[12] R. Monroy. Predicate synthesis for correcting faulty conjectures: The proof planning paradigm. *Automated Software Engineering*, 10(3):247–269, 2003.

[13] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6:139–143, 1984.

[14] P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[15] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994.

[16] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[17] M. Protzen. Patching faulty conjectures. In M. A. McRobbie and J. K. Slaney, editors, *13th Conference on Automated Deduction*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 77–91. Springer, 1996.

[18] J. D. C. Richardson and A. Smaill. Continuations of proof strategies. In M. P. Bonancina and B. Gramlich, editors, *4th International Workshop on Strategies in Automated Deduction (STRATEGIES 2001)*, Sienna, Italy, June 2001. Available from `http://www.logic.at/strategies/strategies01/`.

[19] J. D. C. Richardson, A. Smaill, and I. Green. System description: Proof planning in higher-order logic with lambda-clam. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Computer Science*, pages 129–133. Springer, 1998.

6 Appendix

6.1 Tutor Programs

Insert Everywhere	$inserteverywhere(N, nil) = (N :: nil) :: nil$ $inserteverywhere(N, X :: XS) =$ $(N :: (X :: XS)) :: map(\lambda l.(X :: l), inserteverywhere(N, XS))$
Reverse	$reverse(nil) = nil$ $reverse(X :: XS) = reverse(XS) <> (X :: nil)$
Even	$even(0) = T$ $even(s(0)) = F$ $even(s(s(N)) = even(N)$

6.2 Student Programs

Insert Everywhere	
Version 1	$inserteverywhere(N, nil) = nil :: nil$ $inserteverywhere(N, X :: XS) =$ $(N :: (X :: XS)) :: map(\lambda l.(N :: l), inserteverywhere(N, XS))$
Version 2	$inserteverywhere(N, nil) = nil$ $inserteverywhere(N, X :: XS) =$ $(N :: (X :: XS)) :: map(\lambda l.(N :: l), inserteverywhere(N, XS))$
Version 3	$inserteverywhere(N, X :: XS) =$ $(N :: (X :: XS)) :: map(\lambda l.(N :: l), inserteverywhere(N, XS))$
Reverse	
Version 1	$reverse(nil) = X :: XS$ $reverse(X :: XS) = reverse(XS) <> (X :: nil)$
Version 2	$reverse(X :: XS) = reverse(XS) <> (X :: nil)$
Even	
Version 1	$even(0) = F$ $even(s(0)) = F$ $even(s(s(N)) = even(N)$
Version 2	$even(s(0)) = F$ $even(s(s(N)) = even(N)$
Version 3	$even(0) = T$ $even(s(s(N)) = even(N)$
Version 4	$even(s(s(N)) = even(N)$