What can be Learned from Failed Proofs of Non-Theorems?

Louise A. Dennis and Pablo Nogueira

School of Computer Science and Informaton Technology, University of Nottingham, Nottingham, NG8 1BB
lad@cs.nott.ac.uk
pni@cs.nott.ac.uk

Abstract. This paper reports an investigation into the link between failed proofs and non-theorems. It seeks to answer the question of whether anything more can be learned from a failed proof attempt than can be discovered from a counter-example. We suggest that the branch of the proof in which failure occurs can be mapped back to the segments of code that are the culprit, helping to locate the error. This process of tracing provides finer grained isolation of the offending code fragments than is possible from the inspection of counter-examples. We also discuss ideas for how such a process could be automated.

The use of mathematical proof to show that a computer program meets its specification has a long history in Computer Science (e.g. [13, 12]). However the techniques 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. One of the many reasons to which this poor uptake is attributed 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. However cases have been reported where the process of attempting a proof by hand has highlighted an error. 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. [9]), part of which has been attributed to the fact that model-checkers return counterexamples when they fail. As a result, there has been a great deal of interest recently in the detection of counter-examples for software programs and protocols (e.g. [17]). The purpose of the work reported here was to see if the process of attempting and then failing to find a proof could tell a user anything further about software errors than could be extracted from a counter-example.

We start by outlining our methodology (§1) and give a broad categorisation of the errors we encountered (§2), we then examine some specific examples of buggy programs, the failed proof attempts they generated, how these attempts can be used to assign "blame" to particular segments of the original source code

and in some cases to suggest patches(§3). Lastly we discuss ideas for automating this process and some challenges that such a system would face (§4).

1 Methodology

We have opted to study programs produced by novice programmers, specifically undergraduates working on functional programming modules. There were three main reasons why we selected this domain: it is relatively easy to acquire a large number of such programs; they are likely to be well suited to attempts at correctness proofs and easy to translate into the object language of theorem provers; and they tend to be relatively small so the associated correctness proofs can be generated comparatively rapidly allowing us to focus on surveying errors. There are clearly also some problems with selecting such a domain, in particular that the errors produced by novices may be dissimilar to those produced by professional programmers. It is also the case that conclusions drawn about the utility of failed proof attempts in functional programming may not tell us anything about their utility in other paradigms.

We gathered a test bed of such programs and used standard testing to identify those which were incorrect. This process also generated counter-examples for those programs. We then attempted hand proofs in the Isabelle/HOL theorem prover [15, 14] to see if anything further could be learned about the nature of the error above that already learned from the counter-examples.

Our choice of Isabelle was dictated by an interest in extending this work to an automated system to produce program error diagnoses based on failed proof attempts. We have already performed some initial investigations in this area using the proof planning paradigm [5]. IsaPlanner [6] is a proof planning system built on top of Isabelle so this made the combination of Isabelle/IsaPlanner an attractive option. Furthermore our test database contained, almost exclusively, recursive programs suitable for proof by mathematical induction, an area in which the IsaPlanner system specialises.

The proof planning paradigm [2] works by capturing patterns of proof across families of similar problems. The work reported here discusses hand proof attempts in Isabelle performed with the intention of defining patterns of failure which could then be expressed within a proof planning system.

Our initial investigations involved a set of ML and Haskell Programs generated by students at the Universities of Edinburgh[1] and Nottingham[2] respectively. We performed a naive shallow embedding by hand of these programs into Isabelle/HOL. We assumed that all datatypes mapped directly to the corresponding Isabelle datatype (i.e. Isabelle lists behave in the same way as ML lists etc.) and that built-in functions (such as list append, list membership etc.) could also be directly transferred. In some cases we had to edit definitions so they used

[1] The original author of these exercises is unknown. They were set as part of the Computer Science 2 module.

[2] Exercise authored by Dr. Graham Hutton for the Functional Programming module.

structural recursion – for instance when writing functions on the natural numbers students often used `n` and `n - 1` when defining the recursive case, rather than `n + 1` and `n`. However, given the similarity between the Isabelle/HOL theories and ML these assumptions are relatively safe and certainly adequate for an initial investigation. The Haskell programs provided greater challenges. We had issues with the representation of type classes and lazy evaluation and we also found that the match between the functions in the Haskell prelude and Isabelle's existing theories was often insufficient[3]. This reduced our confidence in the conclusions we could draw from the Haskell examples. In this paper we shall only examine ML examples however the Haskell examples do broadly support our conclusions.

In what follows we shall use the `typewriter font family` to represent actual fragments of program code from our corpus and *latex math environment* to represent Isabelle definitions and goals so it is clear when we are talking about actual ML and when we are discussing our translation into Isabelle/HOL.

2 Categorisation of Errors

2.1 Errors in the Basis Case of a Recursive Program

Our previous work investigated the errors that occur in the basis cases of recursive programs [5] and identified two common classes of error in novice programs: that one or more base cases are omitted or that a base case is incorrect. These are distinguished in proofs by reaching a proof goal in the base case of an inductive proof which contains a "user-defined" function which can not be simplified or alternatively by the derivation of $False$. These observations were largely confirmed by this study. In some cases we gained extra information from Isabelle's recursive definition mechanisms. For instance `primrec` (Isabelle's mechanism for defining primitive recursive functions) will issue a warning that there is no equation for a particular constructor if a case is omitted. However, often functions with missing cases required the use of the more general `recdef` definition mechanism[4] which did not issue such warnings. We also found that there were instances where a student would ensure that a sub-function was never called on particular cases and therefore the fact that it was only partially defined was not an issue for the overall correctness of the program.

2.2 Errors in the Recursive Case of a Recursive Program

Initially we observed three different kinds of error in recursive cases. Firstly, the recursive case contained insufficient information (it was embeddable within a

[3] We also had problems with an exercise specification which under-specified the type of bits to integers although it was clear the functions were only to be tested on 0 and 1.

[4] These errors tended to arise when the student was attempting a two step recursive scheme which `primrec` would not allow.

"correct" recursive case), secondly the recursive case contained too much information (a "correct" recursive case was embeddable within it), and thirdly there was no embeddability relation with the correct recursive case. We had hoped that these three errors would map to three different styles of failure observed in our proofs. However experimentation with a number of artificially created examples revealed that this was not the case. Furthermore experimentation with our actual corpus revealed that finer distinctions could be drawn (eg. indicating that a particular sub-expression in the recursive case was to blame). This made us realise that an approach based on a broad categorisation of types of error and then an attempt to sort programs into categories was misguided. Instead, where an error is attributed to the recursive case of a program, the trace of the proof attempt can be used to further localise the error to a sub-expression.

2.3 Specification Errors

Additionally some errors in the corpus could be broadly classified into programs which simply omitted to satisfy some part of the specification. In this case it is unclear that proof had much to offer over and above any sort of counter-example generation. A counter-example could quickly show that this part of the specification was not met and, usually, a major redesign of the program was required in order to accommodate the omitted feature so identifying a culpable program fragment was not really an issue.

3 Analysis of Some Specific Examples

In this paper we focus on three ML list processing exercises:

1. *Write a function* `removeAll x l` *which removes all occurrences of the item* `x` *in the list* `l`.
2. *Write a function* `onceOnly l` *which returns a list containing exactly one copy of every item that appears in* `l`.
3. *Write a function* `insertEverywhere x l` *which returns the list of all lists obtained from* `l` *by inserting* `x` *somewhere inside.*

We created several auxiliary functions in Isabelle for expressing the specifications of these exercises. In particular we used *count_list* and *sub_list* shown below. *count_list* counts the number of appearances of an element in a list and *sub_list* only evaluates to true if one list is contained in another and if the elements of that list appear in the same order as the super-list. Our Isabelle definitions of our specification functions are[5]:

[5] We have preserved much Isabelle syntax in this presentation. For instance the uncurried format; the use of # for list concatenation and the use of *Suc* to indicate the successor function on natural numbers. However in some cases with have used standard mathematical notation instead. Most notably $\in$ for list membership (the Isabelle function *mem*) and $\neq$ for inequality.

$$count\_list\ a\ [] = 0, \tag{1}$$

$$count\_list\ a\ (h\#t) = \textbf{if}\ \ a = h\ \textbf{then}\ \ Suc(count\_list\ a\ t) \quad \textbf{else}\ \ count\_list\ a\ t. \tag{2}$$

$$sub\_list\ l\ [] = \textbf{if}\ \ l = []\ \ \textbf{then}\ \ True\ \ \textbf{else}\ \ False, \tag{3}$$

$$sub\_list\ l\ h\#t = \textbf{if}\ \ l = []\ \ \textbf{then}\ \ True\ \ \textbf{else}\ \ \textbf{if}\ \ hd\ l = h\ \textbf{then}\ \ sub\_list\ (tl\ l)\ t \quad \textbf{else}\ \ sub\_list\ l\ t. \tag{4}$$

The full specification we used for each exercise can be found in Appendix A.

Following proof planning literature we will refer, in what follows, to the use of the induction hypothesis in an inductive proof as *fertilisation*.

3.1 Case 1: Fertilisation Fails

This example occurs in the `removeAll` exercise. It is important to note here that in a previous exercise the student had been asked to create a function `removeOne` which removed one occurrence of the item `x` from the list `l`. The student has defined `removeAll` as follows:

```
fun removeAll _ [] = []
 |  removeAll x (h::t) = if x = h then removeAll x t
                                  else h::removeOne x t;
```

Presumably the student is programming by analogy, starting with their `removeOne` function. They have forgotten to change the recursive step to `removeAll`.

This program generates rather obscure counter-examples. All the following calls succeed:

```
> removeAll 1 [];
val it = [] : int list
> removeAll 1 [1, 1, 1];
val it = [] : int list
> removeAll 1 [1, 2, 1];
val it = [2] : int list
```

Our first counter-example was:

```
> removeAll 1 [1, 1, 2, 3, 4, 1, 1];
val it = [2, 3, 4, 1] : int list
```

careful investigation with additional counter-examples reveals that the problem arises when the item to be removed occurs more than once in the list *after* an element that is not to be removed.

We translated the student code into Isabelle as:

$$removeAll\ x\ [] = [], \tag{5}$$

$$removeAll\ x\ (h\#t) = \textbf{if}\ \ x = h\ \textbf{then}\ \ removeAll\ x\ t \quad \textbf{else}\ \ h\#removeOne\ x\ t. \tag{6}$$

The attempted proof against the first part of the specification,

$$\neg(x \in removeAll(x, l)),$$

gets blocked at an unsuccessful fertilisation attempt,

$$\neg x \in removeAll\ x\ l\ \wedge x \neq a \Rightarrow \neg x \in removeOne\ x\ l.$$

It is easy to see from this goal that the "blame" lies with the use of the function `removeOne` and it is also possible to work out the correction that is needed to make the proof complete successfully. Indeed an automated technique such as difference unification [1] would probably be able to generate a patch.

In this case the proof trace much more directly localises the error in the program than the counter-example did.

3.2 Case 2: Fertilisation succeeds

This is a case where the student has been asked to write the `insertEverywhere` function. The attempt is

```
fun insertEverywhere x [] = [[x]]
  | insertEverywhere x (x1 :: xs) = (x :: x1 :: xs)
                                    :: insertEverywhere x (xs);
```

and they have, in fact, added the comment

> "this function only returns the list of lists given by inserting the value x before all the elements in the list. It does not take into account the values before which x has already been inserted. For example 1[2,3]; would return[[1,2,3],[1,3],[1]] not sure how to implement the function to include the previous values in the list"

indicating that they are well aware of the bug in their program. The problem can be solved by mapping `\l. x1::l` over all the lists produced by `insertEverywhere`[6].

As part of verifying this we attempted to prove

$$l \in insertEverywhere\ x\ l_1 \rightarrow count\_list\ x\ l = Suc(count\_list\ x\ l_1).$$

[6] Although the correct student programs generally used a sub-function with an accumulator argument to achieve the same effect.

The step case is

$$\forall x_a.x_a \in insertEverywhere\ x\ list \rightarrow count\_list\ x\ x_a = Suc(count\_list\ x\ list) \Rightarrow \forall x_a.x_a \in insertEverywhere\ x\ (a\#list) \rightarrow count\_list\ x\ x_a = Suc(count\_list\ x\ (a\#list)).$$

This simplifies to

$$\forall x_a.x_a \in insertEverywhere\ x\ list \rightarrow count\_list\ x\ x_a = Suc(count\_list\ x\ list) \Rightarrow x = a \rightarrow (\forall x.(a\#a\#list = x \rightarrow count\_list\ a\ x = Suc(Suc(count\_list\ a\ list))) \wedge (a\#a\#list \neq x \rightarrow x \in insertEverywhere\ a\ list \rightarrow count\_list\ a\ x = Suc(Suc(count\_list\ a\ list)))).$$

NB. This shows that Isabelle has automatically proved a branch where $x \neq a$. Repeated use of introduction rules then case splits this into two goals depending on whether $x_a = a :: a :: list$ or whether $x_a \in insertEverywhere\ a\ list$. These two branches can be mapped to sub-expressions of the original recursive case of the function definition: `(x :: x1 :: xs)` and `insertEverywhere x (xs)` respectively. The first of these goals is true and we can derive false from the second with the following sequence of steps:

$$\forall x_a.x_a \in insertEverywhere\ a\ list \rightarrow count\_list\ a\ x_a = Suc(count\_list\ a\ list) \wedge x = a \wedge a\#a\#list \neq x_a \wedge x_a \in insertEverywhere\ a\ list \Rightarrow count\_list\ a\ x_a = Suc(Suc(count\_list\ a\ list)).$$

Fertilisation occurs

$$count\_list\ a\ x_a = Suc(count\_list\ a\ list) \wedge x = a \wedge a\#a\#list \neq x_a \wedge x_a \in insertEverywhere\ a\ list \Rightarrow count\_list\ a\ x_a = Suc(Suc(count\_list\ a\ list)).$$

and we now have a contradiction between the first hypothesis and the conclusion.

In order to prevent this contradiction arising it is necessary to prevent the unification of $x_a \in insertEverywhere\ a\ list$ with the antecedent of $x_a \in insertEverywhere\ a\ list \rightarrow count\_list\ a\ x_a = Suc(count\_list\ a\ list)$. The goal, $x_a \in insertEverywhere\ a\ list$, was obtained by rewriting the formula $x_a \in insertEverywhere\ a\ (a\#list)$ with the definition of *insertEverywhere* and then taking the tail of the resulting term. Clearly this tail needed some extra structure somewhere to prevent immediate fertilisation. In this case the extra structure was needed around *insertEverywhere a list* but conceivably it could have required extra structure in one of the argument positions.

It is not clear that we know anything more at this point than we did from the counter-examples detailed by the student (and the ones we generated ourselves in testing) since these clearly indicate that successive calls to `insertEverywhere` are losing necessary information. It is, however, possible to see how an automated mechanism could isolate the responsible sub-expression using a proof trace more easily, perhaps, than it could from analysis of a counter-example alone.

It is also possible that some form of deductive synthesis [3] or corrective predicate construction [11] at this point might allow the correct sub-expression to be synthesized.

Case 3: Fertilisation not expected

In this case another student, also attempting to write `insertEverywhere`, has met problems. They have defined a subsidiary function, `ie`:

```
fun ie n R [] = [R@[n]]
 | ie n R (h::t) = (n::h::t)::[R]@(ie n (R@[h]) t);
```

Again they are aware that the function does not work correctly *"I have had massive problems trying to concatenate the list R and the tail"*. There are several problems here. `R` needs to appear within the head of the list of lists; and it should appear *before* and not after `(n::h::t)`.

This program produces an odd set of counter-examples:

```
> insertEverywhere 0 [1, 2];
val it = [[0, 1, 2], [], [0, 2], [1], [1, 2, 0]] : int list list
> insertEverywhere 0 [1, 2, 3];
val it = [[0, 1, 2, 3], [], [0, 2, 3], [1], [0, 3], [1, 2], [1, 2, 3, 0]]
: int list list
```

However `insertEverywhere []` produces the correct answer which together with the counter-examples strongly suggests that the problem lies with the recursive rather than the basis case of the functions. So the question is whether proof can isolate the problem further.

In our Isabelle development it was easy to establish that

$$insertEverywhere\ x\ l = ie\ n\ []\ l$$

We then attempted to establish a generalised version of our specification including

$$l \in (ie\ x\ l_1\ l_2) \Rightarrow count\_list\ x\ l = Suc(count\_list\ x\ l_2) + count\_list\ x\ l_1$$

The proof follows a similar pattern to that in the previous example. We perform induction on l_2 considering $a\#list$ in the step case and then case split on whether $l = x\#a\#list$. In this instance the first branch of the case split causes problems resulting in the goal

$$\begin{gathered}\forall x_a x_b.x_b \in ie\ x\ x_a\ list \rightarrow \\ count\_list\ x\ x_b = Suc(count\_list\ x\ list) + count\_list\ x\ x_a \\ \wedge\ l = x\#a\#list \Rightarrow \\ count\_list\ x\ (x\#a\#list) = Suc(count\_list\ x\ (a\#list)) + count\_list\ x\ l_1,\end{gathered}$$

which rewrites to

$$\begin{gathered}\forall x_a x_b.x_b \in ie\ x\ x_a\ list \rightarrow \\ count\_list\ x\ x_b = Suc(count\_list\ x\ list) + count\_list\ x\ x_a \\ \wedge\ l = x\#a\#list \Rightarrow \\ Suc(count\_list\ x\ (a\#list)) = Suc(count\_list\ x\ (a\#list)) + count\_list\ x\ l_1,\end{gathered}$$

which simplifies to

$$\forall x_a x_b . x_b \in ie\ x\ x_a\ list \rightarrow$$
$$count\_list\ x\ x_b = Suc(count\_list\ x\ list) + count\_list\ x\ x_a$$
$$\wedge\ l = x\#a\#list\ \Rightarrow\ count\_list\ x\ l_1 = 0$$

which is satisfiable but not always true.

In this case we wouldn't have expected fertilisation to be possible because of the structure of the case splits. However this has still shown there is a problem with `(n::h::t)`. In fact exploration of the remaining branches of the proof further suggests that `[R]` is also a problem while `(ie n (R@[h]) t)` is not. In this case the proof trace once again appears to have provided more information than the counter-example. For instance the fact that the actual recursive call itself *is* correctly formed is not at all obvious from the counter-examples.

3.3 Case 4: Combined Problems

We conclude by examining an example where there are a combination of errors in the program. In this case in the basis cases. This is an example where the student has been asked to write the `onceOnly` function. They have created a complex set of sub-functions, not required by the program specification, in order to achieve this:

```
fun insert x [] = []
  | insert x(h::t) =
          if x <= h then x :: h :: t
                    else h :: insert x t;

fun sort [] = []
  | sort (x :: xs) = insert x (sort xs);


fun Once [] = []
  | Once (x1 :: x2 :: xs) =
        if x1 = x2 then Once (x2 :: xs)
                   else x1 :: x2 :: Once xs;

fun onceOnly [] = []
  | onceOnly (x :: xs) = Once (sort (x :: xs));
```

There are two errors here. Firstly the basis case for `insert` should be `insert x [] = [x]` and secondly we are missing a basis case for `Once` where there is a singleton list. Our counter-examples indicate that all calls to `onceOnly` evaluate to `[]`.

It is fairly simple to prove that

$$onceOnly\ l = Once(sort\ l).$$

We then needed to establish a number of theorems about `sort` and `insert`. We introduced two new functions *min_list* and *less_min_list* (which evaluate tot he minimum element in a list, and a list less its minimum element respectively):

$$min\_list(a\#[]) = a, \tag{7}$$
$$min\_list(h\#t) = min\ h\ (min\_list\ t). \tag{8}$$

$$less\_min\_list\ [] = [], \tag{9}$$
$$less\_min\_list\ (h\#t) = \textbf{if}\ h = min\_list(h\#t)\ \textbf{then}\ t\ \textbf{else}\ h\#(less\_min\_list\ t). \tag{10}$$

and attempted to prove

$$l \neq [] \Rightarrow (sort\ l) = (min\_list\ l)\#sort(less\_min\_list\ l).$$

The step case of this proof introduces the goal:

$$list \neq [] \Rightarrow sort\ list = (min\_list\ list)\#(sort\ (less\_min\_list\ list))$$
$$a\#list \neq [] \Rightarrow sort\ a\#list = (min\_list\ a\#list)\#(sort\ (less\_min\_list\ a\#list))$$

which triggers a case split on whether $list = []$. In the case where this is true the induction hypothesis evaluates to true and rewriting the induction conclusion reaches the goal

$$insert\ a\ [] = [a]$$

which rewrites to

$$[] = [a]$$

and then we derive $False$. From this it is obvious that the appropriate fix is to edit the basis case of `insert` to `insert x [] = [x]`. It is important to note here that the lemma we have chosen is dependent our intuitions about the way `insert` and `sort` should behave. If the student had named these functions less informatively the process of locating the `insert` error would have been considerably complicated.

It isn't possible to detect the additional problem with `Once` even continuing from this point with enthusiastic use of Isabelle's `sorry` command in order to establish theorems that can't be proved. `Once` is only ever applied to expressions of the form `sort l` which, because of the bug in `insert`, all evaluate to `[]` so the missing case in `Once` is undetectable.

In other examples we were able to identify combinations of problems where they caused the proof to break down in different branches of the trace. However this example illustrates some limitations of the proof approach in terms of detecting all the errors within a program.

4 Discussion and Related Work

We have picked some representative examples of problems in our corpus of study. Unsurprisingly these illustrate that the structure of a program correctness proof

is related to the structure of the underlying program. It is important to realise that the structure of the proof is also affected by the structure of the specification functions and so a direct mapping between branches of a proof and cases within a program is not always possible. Also of note is the fact that there may not be an explicit `if ... then ... else` structure in a program and yet the structure of the code can still induce a case split on the proof allowing us to focus our attention on particular sub-expressions. In our examples by the interaction of Isabelle's list membership function and the head::tail structure of lists.

It is also possible to see that the proof traces provide hints about how a proof can be patched sometimes directly providing the correct evaluation of an expression (the `insert` example) and sometimes highlighting where information may be missing, etc.

Our examples raise a number of interesting questions:

1. How could the proof traces shown here be produced automatically. In particular how should such a system decide when it has reached an informative failure? and how can a programmer's intended behaviour for sub-functions be determined?
2. Given a trace how can useful information be extracted and presented to a user?
3. How might patches for problem areas of code be constructed?

Automating Proof Tracing Although in many cases the programs we are studying are only a few lines long the proofs we have produced raise a number of challenges for automation, even when the programs are correct. For instance we frequently needed to generalise our goals to accommodate the presence of accumulators in sub-functions and occasionally needed to speculate new functions and lemmas entirely (e.g. the need to provide a rule for expressing *sort l* in terms of the head and tail of a list). Proof planning already has an account of how new lemmas and generalisations can be found [8] but our examples present considerable challenges to the state-of-the-art in this area.

Leaving aside the issue of appropriate lemma speculation we also found that in order to extract the useful information from the failed proofs we had to bypass Isabelle's simplifier to step through a number of rewrite steps and other simplifications by hand before we reached a goal that was "informative". In general we used the simplifier to narrow the investigation to particular branch of the proof, but then found we needed to retract the simplification to gain finer grained information about exactly which case was causing problems and the steps that led to an unprovable goal. This process would need to be controlled carefully in any automated system. As a related issue an important part of this process was identifying goals that were satisfiable but not always true, existing counter-example discovery technology clearly has a role to play here. Possibly a call to `quickcheck` or similar should be employed each time a proof attempt branches in order to ascertain whether a system should attempt to prove that branch or gather error information.

Lastly there is the issue of programmer introduced sub-functions. In an ideal world a programmer would specify the behaviour of all sub-functions as well as the main program however there are many situations where this will not be the case, for instance if an error diagnosis system were used as a marking aid rather than as a program construction aid for students. In some cases it may be possible to use ontology matching and repair methods [10] to deduce the intended behaviour. In case 4 the fact that the function was named `insert` would allow us to compare its behaviour to a correct list insertion function[7]. It might also be possible, in some cases, to get a programmer to provide sample inputs and outputs and use conjecture forming technology [4] to deduce appropriate lemmas.

Extracting Information from Proof Traces Once a proof trace has been produced there is then the question of how useful information can be extracted from it. There would appear to be a number of ways in which this could work, some general and some related to specific forms of failure. For instance, when fertilisation has failed it seems plausible to attempt difference unification [1] of the induction conclusion and the induction hypothesis in order to highlight differences and suggest patches. Similarly where we are attempting to prove an equality, it may be possible to compare the LHS and RHS in order to adapt a function's output.

It is also often possible to extract generalised counter-examples or counter-example classes from a failed proof branch which may provide more focused information than individual counter-examples. For instance, in the `insert` example, it is possible to deduce there is a problem with all one element lists.

Patching Problem Code Monroy [11] has already used proof planning to examine faulty conjectures. He follows work by Franova and Kodratoff [7] and Protzen [16] 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. This process can correctly fill in missing base cases[8] but the approach would need modification if it were to remove a piece of faulty code and then replace it with a different correct fragment.

An alternative approach might be to look to deductive synthesis technology [3]. Deductive synthesis uses meta-variables in existence proofs to synthesise a program that meets its specification. Once an offending program fragment has been identified it should be possible to replace it with a meta-variable and attempt to use similar techniques to instantiate this variable. Similarly where premature fertilisation has occurred indicating missing structure, the possible lo-

[7] Our thanks to Fiona McNeill for this suggestion.

[8] Monroy and Dennis, *Fault Diagnosis*, Edinburgh Dream Group Blue Book Note 1485.

cations for this structure could be represented by meta-variables and deductive synthesis used to instantiate these.

5 Conclusion and Further Work

In this paper we have reported the results of of a case study in the use of proof to locate program errors. We have shown that the structure of the proof can be used to narrow the focus of attention to specific parts of program and made some suggestions about how such a trace could, in some situations, also be used to suggest appropriate patches. We have compared the information we could extract from our failed proof attempts with the information deducible from counter-examples and concluded that it is generally, although not always, possible to narrow the focus to the culpable piece of code better using a proof trace than it is using the counter-example.

We now intend to construct an automated system based on proof planning to produce these proof traces and then use this system to automatically generate diagnoses and patches.

Acknowledgements

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

References

1. D. A. Basin and T. Walsh. Difference unification. In R. Bajcsy, editor, *Proceedings of IJCAI-93*, pages 116–122. Morgan Kaufmann, 1993.
2. 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.
3. A. Bundy, L. Dixon, J. Gow, and J. D. Fleuriot. Constructing induction rules for deductive synthesis proofs. In *Constructive Logic for Automated Software Engineering*, ENTCS. Elsevier, 2005. To Appear.
4. S. Colton. The HR program for theorem generation. In A. Voronkov, editor, *18th International Conference on Automated Deduction*, volume 2392 of *LNCS*, pages 285–289. Springer, 2002.
5. L. A. Dennis. The use of proof planning critics to diagnose errors in the base cases of recursive programs. In W. Ahrendt, P. Baumgartner, and H. de Nivelle, editors, *IJCAR 2004 Workshop on Disproving: Non-Theorems, Non-Validity, Non-Provability*, pages 47–58, 2004.
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. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.
9. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.
10. F. McNeill, A. Bundy, and C. Walton. Diagnosing and repairing ontological mismatches. In *Starting AI Researchers' Symposium*, 2004. Also available as Edinburgh Informatics Report EDI-INF-RR-0251.
11. R. Monroy. Predicate synthesis for correcting faulty conjectures: The proof planning paradigm. *Automated Software Engineering*, 10(3):247–269, 2003.
12. F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6:139–143, 1984.
13. P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.
14. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
15. L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994.
16. 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.
17. G. Steel, A. Bundy, and E. Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. *AISB Journal*, 1(2):169–182, 2002.

Appendix A: Isabelle Specifications Used

1.1 `removeAll`

removeAll_spec1: $\neg(x \in removeAll(x, l))$
removeAll_spec2: $x \neq a \Rightarrow count\_list(x, removeAll(a, l)) = count\_list(x, l)$
removeAll_spec3: $\neg(x \in l) \Rightarrow removeAll(x, l) = l$

1.2 `onceOnly`

onceOnly_spec1: $\neg(x \in l) \Rightarrow count\_list\ x\ (onceOnly\ l) = 0$
onceOnly_spec2: $(x \in l) \Rightarrow count\_list\ x\ (onceOnly\ l) = 1$

1.3 `insertEverywhere`

insertEverywhere_spec1: $l_1 \in insertEverywhere\ x\ l \Rightarrow count\_list\ x\ l_1 = Suc(count\_list\ x\ l)$
insertEverywhere_spec2: $l_1 \in insertEverywhere\ x\ l \Rightarrow sub\_list\ l\ l_1$
insertEverywhere_spec3: $l_1 \in insertEverywhere\ x\ l \wedge x_1 \neq x \Rightarrow count\_list\ x_1\ l_1 = count\_list\ x_1\ l$
insertEverywhere_spec4: $(count\_list\ x\ l_1 = Suc(count\_list\ x\ l) \wedge sub\_list\ l\ l_1 \wedge \forall x_1.\ x_1 \neq x \rightarrow count\_list\ x_1\ l_1 = count\_list\ x_1\ l) \Rightarrow l_1 \in (insertEverywhere\ x\ l)$