Proof-Directed Debugging and Repair

Louise A. Dennis[1], Raul Monroy[2] and Pablo Nogueira[1]

[1] School of Computer Science and Information Technology, Univ. of Nottingham, Jubilee Campus, Wollaton Road, Nottingham, NG8 1BB, UK lad@cs.nott.ac.uk pni@cs.nott.ac.uk

[2] Computer Science Department, ITESM, Estado de México, Carr. Lago de Guadalupe Km 3.5 Atizapán, 52926, México raulm@itesm.mx

Abstract

We describe a project to refine the idea of proof-directed debugging. The intention is to clarify the mechanisms by which failed verification attempts can be used to isolate errors in code, in particular by exploiting the ways in which the branching structure of a proof can match the the structure of the functional program being verified. Our intention is to supply tools to support this process. We then further discuss how the proof planning paradigm might be used to supply additional automated support for this and, in particular ways in which the automation of proof-directed debugging with proof planning would allows code patches to by synthesised at the same time that a bug is located and diagnosed.

1 INTRODUCTION

Programmers are familiar with the concept of testing, running a program on a selection of inputs and comparing the actual with the expected output. The intention is that when all the tests are successfully passed by a program then there should be a reasonable expectation that the program is error-free. Intelligent use of testing can provide a programmer with a great deal of information about the location and nature of errors within programs. However testing only provides a *guarantee* of correctness for the specific test cases used, it does not provide a guarantee for other potential inputs. Programmers are less familiar with verification, which involves constructing a mathematical proof that a program is correct. The advantage of verification is that the end result is a much stronger guarantee of the correctness of the program than can be provided by testing. One short-coming of existing verification technology, however, is that it is unclear what the appropriate remedial action is when a verification attempt fails. Current verification techniques are unable to fulfill the error location role which is such a fundamental, though perhaps poorly recognised, function of standard testing.

We intend to develop a theory of *proof-directed debugging*, providing a principled framework by which verification (the process by which a program is shown to meet its specification using formal proof) can be conducted in a manner which will assist in the location and repair of programmer errors (as testing does). It will be possible to perform proof-directed debugging in a highly interactive fashion, where the user performs most of the proof guidance. However, to be of practical

use to programmers with limited experience of proof, automated versions of proof-directed debugging will be required. We therefore intend to integrate automated reasoning into the proof-directed debugging framework. We anticipate this will involve the development of new automated reasoning techniques. We also hope to develop and integrate methods for the automated patching of erroneous code.

The success of model checking techniques (e.g. [12]) has partly been attributed to the fact that model-checkers return counterexamples when they fail. As a result, there has been a great deal of interest in the automatic detection of counter-examples for software programs and protocols (e.g. [18, 4]). Our previous work [6] has shown that, in many cases, better debugging information can be obtained from failed proofs than can be obtained by testing and counter-example generation alone. It should be noted that we believe that testing and counter-example generation remain important techniques and propose the use of proof-directed debugging as an addition to rather than a replacement for these methodologies.

This project ties in with the aims of the Dependable Systems Evolution Grand Challenge for Computing Research[1]. To be truly useful, the *verifying compiler* [10] will need to incorporate this kind of technology in order to provide appropriate warnings and error messages to the user. Once again, to be useful in a verifying compiler, proof-directed debugging will need to be automated. This is an added incentive for the investigation of automated approaches.

Out intention is to develop a theory of proof-directed debugging by which failed proofs can be used to create rich debugging information. A particularly ambitious goal is the production of methods by which, in some cases, a proof-directed debugger will be able not only to *diagnose the problem* but also *automatically repair the program*. The ultimate intent is to define a methodology for the development of tools which provide rich debugging information to programmers; can suggest appropriate patches for erroneous code; and which, if run successfully to completion, also provide a guarantee of correctness. Our starting point in this project is the development of proof-directed debugging and repair for functional programs, particular for functional programs produced by novices.

2 EXAMPLE OF A FAILED VERIFICATION OF A FUNCTIONAL PROGRAM

Consider the following ML program written by a novice programmer. They are attempting to create a function, `removeAll`, which removes all occurrences of an element, `x`, from a list, `l`. 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 `x` from `l`

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

[1] http://www.fmnet.info/gc6/

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

We examined this program, among others, in [6]. Standard testing generated the following examples and counter-example:

Examples

`removeAll 1 [];`
output: `[]`
`removeAll 1 [1, 1, 1];`
output: `[]`
`removeAll 1 [1, 2, 1];`
output: `[2]`

Counter-Example
`removeAll 1 [1, 1, 2, 3, 4, 1, 1];` **output:** `[2, 3, 4, 1]`

These are not particularly informative in terms of diagnosing the error.

We translated the student code into Isabelle [15] – an interactive theorem prover – and attempted to prove (among other properties):

$$\neg(x \in \mathit{removeAll}\, x\, l),$$

The proof proceeds by induction immediately generating two branches (where l equals $[]$ and where l equals $h :: t$). The first branch is successfully discharged – establishing the correctness of `removeAll _ [] = []`. The second branch of the proof then splits again on a case analysis of whether $x = h$. The first branch is successfully discharged – establishing the correctness of **`if`** `x = h` **`then`** `removeAll x t`. The remaining branch becomes "blocked" at

$$\neg x \in \mathit{removeAll}\, x\, l \wedge x \neq h \quad \Rightarrow \quad \neg x \in \mathit{removeOne}\, x\, l.$$

We are "expecting" to be able to use the induction hypothesis $\neg x \in \mathit{removeAll}\, x\, l$ to discharge the goal at this point but we can't because the induction conclusion contains the function *removeOne* instead of *removeAll*. It is easy to see here that the "blame" lies with the use of the function `removeOne` and it is also possible to work out a correction. Indeed an automated technique such as difference unification [1] would be able to generate a patch.

Analysis of this, and similar examples, suggests that the branching of a program correctness proof serves a useful role in locating the section of code responsible for an error. However, in order to exploit this the user needs to be able to recognise when some goal is genuinely blocked rather than the result of a proof mis-step earlier on.

Another issue here, of course, is where the properties we seek to verify might come from. A novice, or even experienced, programmer unfamiliar with formal

specification and proof is likely to find this task as challenging as the proof-directed debugging process itself. For the present we are assuming that an appropriate (and correct) specification has been provided by an external source such as a tutor or specification engineer.

3 PROOF-DIRECTED DEBUGGING

Proof-directed debugging was first suggested by Harper [8]. Harper presents an example which demonstrates how the failure of the correctness proof leads to both the generation of a counter-example and a revision of the specification. This work does not appear to have been pursued and, as far as we are aware, no attempt has been made to refine Harper's ideas into a framework or process within which proof-directed debugging could take place.

The idea for developing such a framework, rather than relying on a user's skill at general proof, is based on the example of Algorithmic debugging [16]. Algorithmic debugging provides a tool-supported process for normal debugging. It has been primarily applied to logic programs but has also been evaluated in other paradigms, such as imperative and functional programming [7, 14]. The idea is to construct an execution tree of a run of the program on some input and query the user each time this tree branches. This identifies branches which are returning false results and so locates sections of code responsible for errors. There is a clear relationship here to the branch localisation effect we observe when attempting to verify buggy programs. There are also some key differences. We do not propose to construct a tree of the actual program execution trace on some input but rather to use the combined structure of program and proof to similar effect.

We have therefore some very specific challenges that we need to address. Proof-directed debuggers will need to guide a proof process rather than an execution process, a significantly more challenging task both for the user and the system. In general, in the examples reported in [6], we used the automated Isabelle simplifier to establish the truth of particular branches of a proof but, where the simplifier failed we needed to proceed in a more low-level step-by-step fashion until an "informative" dead end goal or another branching step was reached. This process would need to be controlled carefully, preferably with as much automation as possible, in any proof-directed debugger. A related and 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. A call to Isabelle's `quickcheck` [2] counter-example finder could assist with this. As mentioned earlier we believe that proof-directed debugging should be viewed as an addition to existing error location methodologies and, in fact, this analysis suggests that all these techniques should work *together* to assist in the diagnosis and location of an error.

[2] Isabelle's `quickcheck` should not be confused with Claessen and Hughes' QuickCheck tool. Both of these are clearly highly relevant to our research and this name overloading is therefore very unfortunate.

We anticipate that powerful existing automated reasoners such Isabelle's simplifier and `quickcheck` will be important components in proof-directed debugging but our examples also suggest that we will need to develop novel automated reasoning techniques specifically tailored towards the proof-directed debugging context in order that necessary user guidance of the proof process can not only be kept to a minimum, but also made reasonably comprehensible to a programmer. We believe such automation can be achieved using proof planning (see §4) but we think it important, at this point, to stress the need to separate the theory of proof-directed debugging from the use of proof planning to automate the process since other automation frameworks may be proposed and developed in future.

4 PROOF PLANNING

Proof planning [2] is an Artificial Intelligence based technique for the automation of proof. It aims to provide a generic framework for the automation of reasoning with a focus on explainable reasoning. Key features of proof planning [5] are the use of meta-logical information to make heuristic choices; hierarchical, customisable *proof strategies* which dictate the expected structure of the proof; and *proof critics* which modify the existing proof tree in response to a proof failure.

The ideas of expected structure, explainable reasoning and failure analysis make proof planning a good candidate underlying architecture for the automation of proof-directed debugging. As an additional advantage, proof planning has already made considerable progress in the automated discovery of loop invariants [17] and induction schemes [3] both of which will be necessary for proof-directed debuggers. A further incentive is the existing body of research on corrective predicate construction and middle-out reasoning (§4.1) which already exists in the proof planning field [13, 3, 9, 11]. A key part of our project, therefore, is the use of proof planning as the underlying automation technology for proof-directed debugging. Our plan is to start with a simple proof planning strategy in which a proof is guided in a step-by-step fashion using a limited set of simple techniques until a branch point occurs, individual branches are probed using brute-force automated reasoners and counter-example finders, and the user is queried when the system is unable to determine an appropriate way forward. We expect that evaluation of this on a corpus of functional programs produced by novices will reveal areas where additional automated support can be added.

4.1 Patching Erroneous Code

We also intend to develop a theory for how proof-directed debugging can produce candidate patches for erroneous code. There are a number of possibly technologies we could explore here (eg. [1]) however we intend to focus on two techniques.

4.1.1 Middle-out Reasoning

The deductive synthesis of a program from a constructive proof of its specification presents many challenges to automated reasoning methods. Work within the proof planning paradigm has focused on an approach termed *middle-out reasoning*. The key idea in middle-out reasoning is to postpone choices about key parts of the theorem or proof for as long as possible. This is typically performed by replacing some part of the theorem goal (generally created by the introduction of an existential witness) with a higher-order meta-variable. This meta-variable is then gradually instantiated during the course of the proof until the appropriate witness term is synthesised.

Consider our erroneous program for `removeAll`. We have identified a problem with the use of `removeOne` so we could replace this with a meta-variable, X:

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

The failing branch of the proof attempt discussed in §2 now simplifies to

$$\neg x \in \mathit{removeAll}\, x\, l \wedge x \neq h \Rightarrow \neg x \in (X\ h\ x\ t).$$

If we now attempt to exploit our induction hypothesis by unifying it with the conclusion of the goal then X is unified to $\lambda x, y, z.\ \mathit{removeAll}\, y\, z$. The process has correctly synthesised a patch for this particular program.

4.1.2 Corrective Predicates

Another technique for the correction of faulty theorems is *corrective predicate synthesis*. This method aims to build a definition for a *corrective predicate*, P, for any (potentially erroneous) theorem, G, such that $\forall x.\ P(x) \Rightarrow G(x)$. A prototype corrective predicate synthesis system already exists based on the *Clam* proof planner [13]. In this system any theorem proving attempt includes a corrective predicate, represented at the outset by a higher-order meta-variable. The definition of the corrective predicate is built using middle-out reasoning during the proof. The instantiation of the predicate divides into cases where the proof branches and when a case leads to a goal which is neither provable nor disprovable, the predicate is instantiated via *abduction*.

Consider the following program written by a novice ML programmer. They are attempting to write a program to produce all possible lists gained by inserting an element, `x`, at some point in a list, `l` (this is part of an exercise that builds up to a permutation function). The student has forgotten to provide a basis case for their recursive program.

```
fun inserteverywhere n (x::xs) =
          (n::(x::xs))::(map (fn l ⇒ (x::l))
                             (inserteverywhere n xs))
```

Using the *Clam*-based system we attempted to prove the equivalence of this program to one produced by a tutor

```
fun Tinserteverywhere n [] = (n::[])::[]
 | Tinserteverywhere n (x::xs) =
          (n::(x::xs))::(map (fn l ⇒ (x::l))
                             (Tinserteverywhere n xs))
```

The goal[3] we attempted was:

$$\forall n,l.P(n,l) \Rightarrow inserteverywhere(n,l) = Tinserteverywhere(n,l)$$

The system first attempted induction on l leaving two goals

$$P(n,[]) \Rightarrow inserteverywhere(n,[]) = Tinserteverywhere(n,[])$$

$$P(n,x :: xs) \Rightarrow inserteverywhere(n,x :: xs) = Tinserteverywhere(n,x :: xs)$$

The proof attempt of the second goal succeeds automatically instantiating one case of the predicate definition: $P(n,x :: xs) \Rightarrow P(n,xs)$.

The base case yields an "abducible" goal, namely:

$$inserteverywhere(N,[]) = (N :: []) :: []$$

which instantiates the other case of the corrective predicate (by abduction) to

$$P(n,[]) \Rightarrow inserteverywhere(N,[]) = (N :: []) :: []$$

So the final theorem synthesized is:

$$P(N,l) \Rightarrow inserteverywhere(N,l) = Tinserteverywhere(N,l)$$

where

$$\begin{aligned} P(N,[]) &= inserteverywhere(N,[]) = (N :: []) :: [] \\ P(N,H :: T) &= P(N,T) \end{aligned}$$

The recursive structure can be eliminated from this predicate so it becomes

$$P(N,L) = inserteverywhere(N,[]) = (N :: []) :: []$$

and so our theorem simplifies to:

$$\begin{aligned} inserteverywhere(N,[]) &= (N :: []) :: [] \Rightarrow \\ inserteverywhere(N,l) &= Tinserteverywhere(N,l) \end{aligned}$$

effectively synthesizing the correct answer.

[3]The *Clam* system uses curried syntax, hence the shift in style here.

We have applied this system to a number of our erroneous functional programs. The results are encouraging and, in particular, where parts of the code have been omitted (as above) the technique can successfully synthesise the missing code fragments. Several challenges remain: in the case of erroneous as opposed to missing information, such as appears in the `removeAll` example, the current system fails to synthesise an appropriate patch. Consider the following piece of code for `inserteverywhere`

```
fun inserteverywhere n [] = []::[]
 | inserteverywhere n (x::xs) =
          (n::(x::xs))::(map (fn l ⇒ (x::l))
                             (inserteverywhere n xs))
```

In this case the student has supplied an incorrect base case and the system synthesises the following predicate:

$$\begin{aligned} P([]) &= \text{false} \\ P(H :: T) &= P(T) \end{aligned}$$

which can be simplified to $P(L) = \text{false}$. The unsimplified structure of this predicate can be analysed to indicate which branch of the program is incorrect but no patch has been suggested. While this serves the error location function of a proof-directed debugger it is insufficient in terms of patch generation.

We are interested in extending the corrective predicate idea so that it can perform more consistently on our examples. One idea might be to interleave the construction of the corrective predicate with selective deletion of cases of the program allowing patches to be synthesized as if in the case of missing information. The corrective predicate methodology contrasts with the two-stage process outlined in §4.1.1 where we first attempted a proof to locate the error and then re-ran the proof with a meta-variable in place. We believe that both approaches deserve investigation.

5 CONCLUSION

We believe that the verification process can serve two important roles in program construction: that of guaranteeing the correctness of a program; and that of aiding in the identification, localisation and repair of errors. The second of these two roles is poorly supported by existing verification tools.

Functional programming appears to be an ideal paradigm within which to explore this second role of verification. Functional programs are comparatively easy to reason about, allowing us to focus on the processes necessary for diagnosing and repairing errors. Furthermore there are many aspects of functional programs that novices generally find challenging and it is therefore possible to acquire a large corpus of erroneous programs which nevertheless are extremely tractable from a reasoning point of view again allowing researchers to focus on the process of error location and repair.

Our intention therefore is use functional programming as a platform upon which we can develop a theory of proof-directed debugging and automated assistance for the task of using verification for error diagnosis and repair.

ACKNOWLEDGMENTS

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, *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 *CLASE'05*, ENTCS. Elsevier, 2005. To Appear.

[4] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00*, pages 268–279. ACM Press, 2000.

[5] L. A. Dennis, M. Jamnik, and M. Pollet. On the comparison of proof planning systems: Lambda-clam, omega and isaplanner. In *Calculemus 2005*, ENTCS, 2005. To Appear.

[6] L. A. Dennis and P. Nogueira. What can be learned from failed proofs of non-theorems? In J. Hurd, E. Smith, and A. Darbari, editors, *TPHOLs 2005: Emerging Trends Proceedings*, pages 45–58, 2005. Technical Report PRG-RP-05-2, Oxford University Computer Laboratory.

[7] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy. Generalized algorithmic debugging and testing. *ACM Lett. Program. Lang. Syst.*, 1(4):303–322, 1992.

[8] R. Harper. Proof-directed debugging. *Journal of Functional Programming*, 9(4):471–477, 1999.

[9] J. Hesketh, A. Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail-recursive programs. In D. Kapur, editor, *CADE 11*, volume 607 of *LNAI*, pages 310–324, 1992.

[10] C. A. R. Hoare. The verifying compiler, a grand challenge for computing research. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation 6th International Conference, VMCAI 2005*, volume 3385 of *LNCS*, pages 78–78. Springer, 2005.

[11] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *JAR*, 16(1–2):113–145, 1996. Also available from Edinburgh as DAI Research Paper 729.

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

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

[14] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, July 1994.

[15] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[16] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, 1983.

[17] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In P. Flener, editor, *Logic-Based Program Synthesis and Transformation*, volume 1559 of *LNCS*, pages 271–288. Springer, 1998.

[18] G. Steel, A. Bundy, and E. Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. *AISB Journal*, 1(2):169–182, 2002.