Program Slicing and Middle-Out Reasoning for Error Location and Repair

Louise A. Dennis [1,2]

School of Computer Science and Information Technology
University of Nottingham, Nottingham, UK

Abstract

This paper describes a proof-based approach to the location and repair of errors in functional programs. The approach is based on the use of program slicing to locate errors and middle-out reasoning to repair them.

An implementation in the $\lambda Clam$ proof planning system is described with some preliminary results.

Key words: Proof-Directed Debugging, Proof Planning, Middle-Out Reasoning, Program Slicing

1 Introduction

The integration of debugging and verification procedures in proof-directed debugging applications is a comparatively recent and little explored field. This paper outlines a methodology for proof-directed debugging and repair based on program slicing and middle-out reasoning.

The approach treats functional programs as sets of rewrite rules. We refer to subsets of these as *program slices*. We associate a program slice with each branch of a proof tree, representing the rewrite rules involved in that branch. It is then possible to compare the program slices from successful and unsuccessful proof branches in order to select a candidate for a repair attempt.

The repair is conducted by a technique known as *middle-out reasoning*. Middle-out reasoning is an approach to deductive synthesis problems which postpones choices about key parts of a theorem or proof for as long as possible. This is typically performed by replacing some part of the theorem goal with a

[1] Research funded by EPSRC grant GR/S01771/01 and Nottingham NLF grant 3051. My thanks to three anonymous referees whose comments greatly improved the paper.
[2] Email: `lad@cs.nott.ac.uk`

higher-order meta-variable. This meta-variable is then gradually instantiated during the course of the proof until the appropriate witness term is synthesised.

This paper shows how, once an incorrect rewrite rule has been detected by program slicing, its right hand side can be replaced by a meta-variable and the proof restarted. The new RHS is then instantiated by middle-out reasoning.

The paper is organised as follows: §2 gives background on program slicing; §3 discusses the the proof planning paradigm and middle-out reasoning; §4 discusses the proposed methodology for error location and repair in detail while §5 discusses an implementation; §6 presents some preliminary results; §7 discusses further work and §8 surveys related techniques.

2 Program Slicing

Program slicing is a technique from debugging which is used to locate errors within programs. In debugging applications program slicing works on an execution trace of the program. Traditional program slicing [19] identifies a variable of interest at some point in a program (the *slicing criterion*) and then extracts a fragment of the program (a *program slice*) containing, for instance, all those statements upon which the value of the variable at that point depended. Program slicing techniques for imperative languages use graph-based representations of programs with statements represented as nodes and a program slice as a set of nodes.

The HAT debugging tool [3] applies program slicing to Haskell programs in order to direct a user's attention to relevant parts of the program's source. HAT constructs an *evaluation dependency tree* showing the computation process of the program on some input. The nodes in this tree are equations indicating the reduction of a redex to a value. Implicitly the edges of this tree are the cases of the program's functional definition used to perform the reductions. HAT uses a polling system based on superimposing correct and incorrect branches of evaluation dependency trees to generate heuristic scores for the cases in order to identify errors [5]. This paper proposes to use a similar process except with branches in a proof tree used to generate appropriate slices rather than branches in a debugging trace.

3 Proof Planning

This paper uses proof planning as a framework within which the automatic detection of erroneous rewrite rules by program slicing and their repair using middle-out reasoning can be conducted.

Proof planning [1] is an Artificial Intelligence based technique for the automation of proof. *Proof methods* are the main planning operators used by proof planning systems. Proof method application is governed by preconditions and by a *proof strategy* which, among other things, restricts the methods available at any point in a proof.

A proof strategy provides a guide to which proof methods should be applicable. Knowing which method is expected to apply gives additional information should the system fail to apply it. *Proof critics* [10] can then 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. Traditionally the proposed patch suggests some change to the proof tree or strategy. In this paper we propose extending critics so they may also propose a change to the theorem.

Proof planning also employs the concept of a *proof context*. The proof context is a repository which stores additional heuristic information. The context may be global to an entire plan or local to goals.

3.1 Middle-Out Reasoning

The use of meta-variables to delay commitment in proof search is commonplace (eg. it is used in Lego and Isabelle [14,16]). This form of middle out reasoning has been widely investigated in proof planning. In particular it has been used in synthesis proofs of the form $\exists x.P(x)$ to generate a witnesses for the existentially quantified variable [9,12] and in work with critics to synthesise new lemmas and generalisations [10]. The instantiation and unification of such variables is naturally higher order. The central idea is that unknown parts of the proof goal (eg. the witness term in a synthesis proof) are replaced by a meta-variable which is instantiated as needed during the proof.

The introduction of meta-variables into a proof process increases the search space involved and it becomes necessary to control their instantiation carefully. The rippling heuristic (a form of rewriting constrained to be terminating by meta-logical annotations) [2,18] is used to control this instantiation in the step cases. After every rewriting step an attempt is made to instantiate all meta-variables appearing in the theorem goal. These meta-variables appear in our situation as functions applied to a list of candidate arguments. Instantiation attempts investigate their projection onto one of their arguments (eg. F appearing as $F(x, y, z)$ is instantiated in turn as $\lambda x \lambda y \lambda z.\, x$, $\lambda x \lambda y \lambda z.\, y$ and $\lambda x \lambda y \lambda z.\, z$). Candidate projections are filtered through a simple counter-example finder to exclude false instantiations.

4 A Methodology for Error Location and Repair

The central idea in this paper is that program slicing in the style of the HAT system can be used within a proof context to select a rewrite rule for modification. A proof critic can use a comparison of the program slices for each proof branch to nominate a rewrite rule for repair and to restart the proof with a new version of this rewrite rule which contains meta-variables. The new rewrite rule is then instantiated by middle-out reasoning in a re-run of the proof. The use of proof planning implicitly assumes that the user has

sufficient prior knowledge about the proof structure to provide an appropriate strategy. The proof strategy for induction used in this work has been widely studied and can be considered of general applicability in many situations.

4.1 Adapting Program Slicing to Verification

We treat the functional cases of a program definition as nodes in program slices. The slicing criterion is the nodes that have been used in a branch of a proof. If we think of the functional cases as rewrite rules then the slicing criterion creates a program slice for each branch of the proof tree consisting only of those rewrite rules used in the derivation of that branch.

In order for such a system to work reasonably within a verification context a user is expected to limit the set of rewrite rules which may appear in program slices. This is done by nominating some set of rules as "suspect". This prevents every definition and theorem in the system appearing in program slices, making the choice of rule for repair more constrained. It is assumed that the pre-existing definitions and theorems of the system within which a verification takes place are correct and it is only those elements introduced by the user [3] which are available for correction.

4.2 Adapting Critics and Middle-Out Reasoning for Program Repair

Our previous work [6,8] identified two typical places in which program errors become "obvious" to a human during a proof attempt. These are when a false goal is derived (typically during symbolic evaluation) or when the induction hypothesis can not be used (referred to in the proof planning literature as *fertilisation*). Therefore new critics can be attached to these two methods. The first only fires if a goal has been reached which contradicts an axiom while the second fires whenever fertilisation fails. These critics inspect the program slice information available for the whole proof tree to choose a particular rewrite rule for correction. We assume this rewrite rule has been derived from the case breakdown of a functional program. In general the pattern on the LHS of such rules are clear and errors are generally picked up by static analysis. Therefore we assume the LHS is correct and that it is the RHS that is incorrect. The RHS for a new version the rule is constructed as an application of a meta-variable to any variables appearing on the LHS and also to an additional recursive argument to represent the potential recursive structure of the proof.

The proof is then restarted with this new rule available instead of the presumed incorrect version.

[3] which could potentially include elements of the specification as well as the program.

5 An Implementation of Error Location and Repair

A prototype system which uses program slicing and middle-out reasoning to locate and repair erroneous rewrite rules has been implemented in $\lambda Clam$ [17] – a proof planning system written in λProlog. It uses a modification of $\lambda Clam$'s proof strategy for induction enhanced by two new critics.

$\lambda Clam$ works by using depth-first planning with proof methods and critics. Each node in the search tree is a subgoal [4]. The planner checks the preconditions for the available proof methods and critics at each node and applies those whose preconditions succeed to, in the case of methods, create the child nodes or, in the case of critics, modify the current proof (or, in this new extension, theorem) and proof strategy. In general proof strategies are constructed so that critics are attempted by the depth-first search only after all the available methods have failed. $\lambda Clam$ associates a heuristic proof context with individual goals.

$\lambda Clam$'s proof strategy for induction is shown in figure 1. The diagram

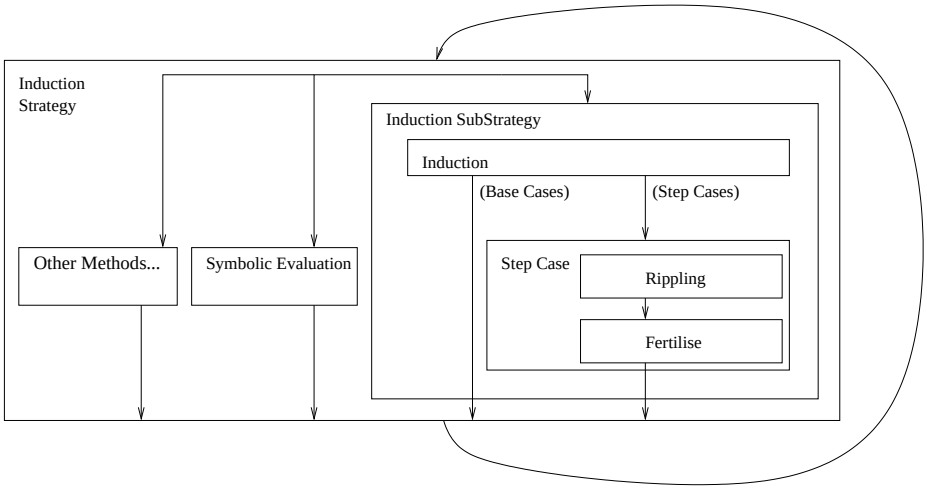

Fig. 1. The Proof Strategy for Induction

shows a top level repeat which attempts a disjunction of methods. These include tautology checking, generalisation of common subterms and also symbolic evaluation (rewriting) and the induction strategy. Within the induction strategy, the induction method chooses an induction scheme and produces subgoals for base and step cases. The top level strategy is re-applied to the base cases. The step cases are handled using rippling and then fertilisation which uses the induction hypothesis to simplify the conclusion further. The results are then passed out to the top level strategy again if necessary. The process terminates when all subgoals have been reduced to *true*.

[4] in general this means that the search tree can be considered to represent the proof tree although a separate global proof tree structure built up by the search process is also maintained which can be inspected and potentially modified by proof critics.

5.1 Examples

The discussion of this implementation is centered around two examples both derived from the list append function and a proof of its associativity. These are artificial examples based on theorems $\lambda Clam$ is known to be able to prove. The errors are based on errors found in novice ML programs [8] to which we ultimately intend to apply the system (see §7). We use here the λProlog convention where variables start with an upper case letter and constants with a lower case letter. The faulty definitions are:

Erroneous Basis Case (`app1`)

```
definition app11: app1([], Z) = [0].
definition app12: app1(H::T, Z) = H::(app1(T, Z)).
```

Erroneous Recursive Case (`app2`)

```
definition app21: app2([], Z) = Z.
definition app22: app2(H::T, Z) = H::(H::app2(T, Z)).
```

5.2 Extensions to the Proof Strategy for Induction

The preceding analysis requires that two extensions be made to the proof strategy for induction. Firstly a mechanism is required for constructing program slices and associating these with proof branches (§5.2.1). Then new critics need to be introduced (§5.2.2) to analyse these slices; choose a rewrite rule for correction; construct a new version of this rule containing meta-variables (§5.2.3) and then restart the proof. The subsequent process of middle-out reasoning was already implemented in $\lambda Clam$ and is based on the work of Ireland and Bundy [10] but was trivially extended to symbolic evaluation as well as rippling.

5.2.1 A Data Structure for Proof Branch Based Program Slices

The first extension required was a mechanism for constructing program slices associated with proof branches. This was achieved by introducing a new data structure into the local proof context associated with individual nodes in the proof tree. The basic data structure is a tuple relating a syntax constant (the suspect program) with a list of tracking information for the rewrite rules associated with that constant.

The tracking information is a further tuple of the name of the rewrite rule, a *good/bad tree*, a position in that tree, and a flag showing whether the rule has been used in this branch of the proof tree. The good/bad tree is a standard tree datatype representing the current proof branching structure, whose leaves are labelled either `good`, `bad` or `unused`.

Consider a proof by induction. Let us suppose that we are interested in constructing program slices for our `app2` program above. There are two

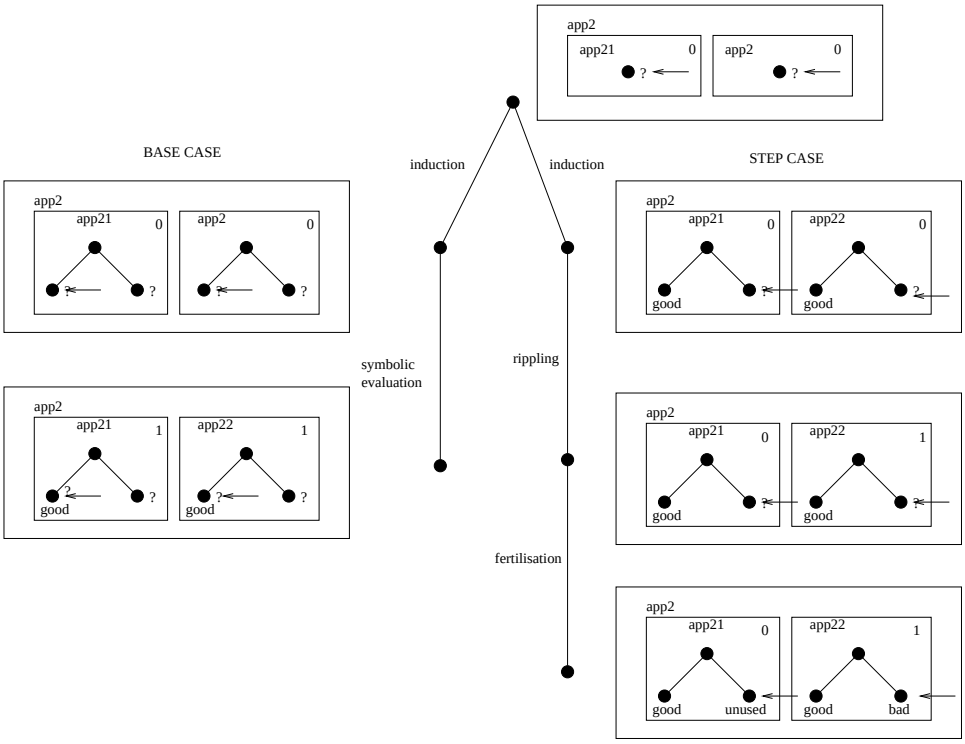

Fig. 2. Constructing Program Slices as a proof progresses

rewrite rules associated with this program, `app21` and `app22`. The good/bad trees for these rules are instantiated dynamically as a proof progresses. When a proof branch is successfully closed the relevant branch in the good/bad tree can be instantiated with a `good` or `unused` leaf node. Where a proof critic detects a problem with a proof branch then the good/bad tree can be instantiated with a `bad` or `unused` leaf node.

Figure 2 shows a schematic version of this new contextual information and how it is developed as a simple induction proof progresses. The figure shows the proof tree with nodes representing subgoals and edges labelled with the proof method that generates the child goals. We've omitted the goals from the node labels but have shown the proof context. The context consists of program slice information about `app2` consisting of individual information about `app21` and `app22`. At the start (the top node in the proof tree) of the proof nothing is known about the proof structure so the good/bad tree associated with both rewrite rules is a single node labelled with a question mark. There is a pointer to the top node of this tree, indicating where this proof node is in that structure. Neither rule has been used yet so the flags are set to 0. Induction is applied and the proof tree splits into a base case and a step case. The new context information for `app2` at these nodes has been extended. The good/bad tree now shows two branches. The pointers in the base case point to the left hand branch while those in the step case point to the right hand branch. Let us assume the base case is attempted first and concludes successfully having used both rewrite rules during symbolic

evaluation. The leaf nodes for the left hand branch are therefore instantiated to `good` for both rules. The step case is then attempted and uses only `app22` during rippling. The result is judged to be incorrect at the fertilisation stage and so the right hand branches are instantiated `bad` for `app22` and `unused` for `app21`.

In *λClam* the dynamic instantiation of the good/bad trees as the proof progresses is handled using prolog variables on the nodes in the tree structures. This should not be confused with the later use of middle-out reasoning where variables are used to represent unknowns in goals.

The program slices associated with each branch of the proof structure can thus be deduced from the final good/bad trees. So the program slice associated with the base case branch is `[app21, app22]` while that associated with the step case branch is `[app22]`.

5.2.2 Using Program Slices in Critics

As discussed in §4.2 two new critics were introduced. The first fires if symbolic evaluation produces a goal which contradicts an axiom while the second fires whenever fertilisation fails.

These critics compute a score for each rule from its good/bad tree. This score is a tuple of the number of bad nodes and the number of good nodes in the tree. These scores are then ordered by $\succ$ where

$$(b_a, g_a) \succ (b_b, g_b) \iff (b_a > b_b) \vee ((b_a = b_b) \wedge (g_a < g_b))$$

and $<$ and $>$ represent the standard order relation on natural numbers. The rule with the highest score is selected. So a rule which has 2 `bad` nodes in its good/bad tree will be chosen over one which has only 1 `bad` node. However when comparing two rules both with 2 `bad` nodes then one with only 1 `good` node will be chosen over one with 2 `good` nodes.

5.2.3 Middle-Out Reasoning for Rewrite Rule Repair

In order to use middle-out reasoning it was necessary to adapt the old, erroneous rules to contain meta-variables replacing the "bad bits". It is presumed that the case structure on the LHS of these rules is correct so these are left unchanged. The RHSs are then constructed as an application of a meta(prolog)-variable to the variables appearing on the LHS with an additional recursive argument including further meta-variables to represent a potential recursive structure for the new rule (i.e. $\forall x_1$`. app1([], `x_1`) = [0]` becomes $\forall x_1$`. app1([], `x_1`) = F(app1(G(`x_1`), H(`x_1`)), `x_1`)`).

The *λClam* proof context was also extended to maintain a list of "banned" rules. Rules nominated for revision by the critics can therefore be added to the banned list. Once this is done the proof is restarted.

We illustrate the middle-out reasoning process with two examples.

5.2.4 Incorrect Basis Case

Consider our example of an incorrect base case. λ*Clam* synthesises a new candidate rule, $\forall x_1$. `app1([], `x_1`) = F(app1(G(`x_1`), H(`x_1`)), `x_1`)`, bans the previous rule, `app11`, and restarts the proof. This proceeds by induction, the step case is discharged using the new rule version. The base case,

$\vdash$ $\forall x_1$. $\forall x_2$. `app1(app1([], `x_1`), `x_2`) = app1([], (app1(`x_1`, `x_2`)))`,

is rewritten using the new version to

$\vdash$ $\forall x_1$. $\forall x_2$. `app1(F(app1(G(`x_1`), H(`x_1`)), `x_1`), `x_2`) =`
`app1([], (app1(`x_1`, `x_2`))).`

λ*Clam* immediately tries to find a projection for `F`, `G` and `H` suggesting

$\forall x_1$. `app1([], `x_1`) =`
`(`λx_2. λx_3. x_3`) (app1(`λx_4`.`$x_4(x_1)$`, `λx_4`.`$x_4(x_1)$`), `x_1`)`

and the new rule beta-reduces to

$\forall x_1$. `app1([], `x_1`) =` x_1.

This makes the goal

$\vdash$ $\forall x_1$. $\forall x_2$. `app1(`x_1`, `x_2`) = app1([], (app1(`x_1`, `x_2`)).`

No counter-examples are found and so the instantiation is accepted. Further rewriting then successfully finishes the proof.

5.2.5 Incorrect Recursive Case

In the example of an incorrect recursive case λ*Clam* suggests the replacement rule

$\forall x_1$. $\forall x_2$. $\forall x_3$. `app2 (`x_3`::`x_2`, `x_1`) =`
`F(app2(G(`x_3`, `x_2`, `x_1`), H(`x_3`, `x_2`, `x_1`)), `x_3`, `x_2`, `x_1`).`

The proof is restarted and induction applied giving the step case

$\forall x_1$. $\forall x_2$. `app2(app2(t, `x_1`), `x_2`) = app2(t, app2(`x_1`, `x_2`))`
$\vdash$ $\forall x_1$. $\forall x_2$. `app2(app2(h::t, `x_1`), `x_2`) =`
`app2(h::t, app2(`x_1`, `x_2`)).`

Rippling rewrites this with the new rule version. `G` and `H` are immediately instantiated to λW_1. λW_2. λW_3. W_2 and λW_1. λW_2. λW_3. W_3 respectively to give the goal:

$\forall x_3$. $\forall x_4$. `app2(app2(t, `x_3`), `x_4`) = app2(t, app2(`x_3`, `x_4`))`
$\vdash$ $\forall x_1$. $\forall x_2$. `app2(F(app2(t, `x_1`), h, t, `x_1`), `x_2`) =`
`app2(h::t, app2(`x_1`, `x_2`)).`

There isn't room in this paper for a full discussion of the rippling heuristic. Briefly, it operates in two modes – outward and inward – and it attempts outward first (shown above). Outward rippling attempts to move differences between the induction hypothesis and conclusion towards the root of the term tree of the conclusion. In this case before the sub-expression `app2(h::t, `x_1`)` is rewritten the difference between it and the induction hypothesis is `h::` which

appears under app2. If G and H are not instantiated then not only is a new difference created "higher up" the tree (F) but there remains a difference on the first argument of app2 (G), and a new one has appeared on the second argument (H). In order to maintain differences in the "right places" for outward rippling, instantiation of G and H is forced. If rippling outward is unsuccessful then rippling inward is attempted in which case such differences would be allowed to remain under certain conditions.

An attempt is made to find a projection for F but in this case no suggestions get past the counter-example finder. The new rule is again used to rewrite the term. In this case it instantiates F to :: in order for the subexpression app2(F(app2(t, x_1), h, t, x_1), x_2) to match the LHS of the rule. However there remains a choice of arguments for :: from app2(t, x_1), h, t and x_1. Once again the rippling heuristic helps. Rippling insists that the induction hypothesis is embedded in the induction conclusion. This forces the selection of the recursive argument app2(t, x_1). Type checking forces the selection of the other argument instantiating the new rule to:

$\forall x_1$. $\forall x_2$. $\forall x_3$. app2(x_3::x_2, x_1) = x_3::app2(x_2, x_1)

thus the goal becomes:

$\forall x_3$. $\forall x_4$. app2(app2(t, x_3), x_4) = app2(t, app2(x_3, x_4))
$\vdash$ h::app2(app2(t, x_1), x_2) = app2(h::t, app2(x_1, x_2))

The rest of the proof proceeds in a straightforward fashion.

6 Results

The following tables show the performance of the current system in four scenarios. It is being tested on standard simple theorems from the $\lambda Clam$ benchmarks. Details of the theorems can be found in appendix A. In all cases the user has nominated the definitions of plus, times and exponentiation as "suspect". Theorems are only shown in a table where they were actually false given the error.

Scenario 1: $0 + X = 1$ **(rule plus1)**

Theorem	Rule Chosen	Instantiation	Notes
simple	plus1	$0 + X = X$	
assp	plus1	$0 + X = X$	
comp	plus1	$0 + X = X$	
plus2right	plus1	$0 + X = X$	
zeroplus	-	-	loops
comm	times1	$0 * X = X$	

Theorem	Rule Chosen	Instantiation	Notes
dist	times1	$0 * X = X$	
zerotimes	-	-	loops
times2right	plus1	$0 + X = X$	
expplus	exp1	-	heap overflow

Scenario 2: $s(Y) + X = s(s(Y))$ **(rule plus2)**

Theorem	Rule Chosen	Instantiation	Notes
assp	plus2	$s(Y) + X = X + Y$	heap overflow
comp	plus1	-	heap overflow
plus1lem	plus2	$s(Y) + X = s(X + Y)$	
plusxx	plus2	$s(Y) + X = s(X + Y)$	
comm	plus1	-	heap overflow
distr	plus2	$s(Y) + X = s(X + Y)$	
assm	plus2	$s(Y) + X = s(X + Y)$	heap overflow
expplus	exp2	$X^{s(Y)} = X^Y$	

Scenario 3: $0 * X = X + X$ **(rule plus3)**

Theorem	Rule Chosen	Instantiation	Notes
comm	times1	-	can't synthesise a constant
dist	times1	-	can't synthesise a constant
distr	times1	-	can't synthesise a constant
assm	times1	-	can't synthesise a constant
zerotimes	-	-	loops
times2right	times1	-	can't synthesise a constant
expplus	times1	-	can't synthesise a constant

Scenario 4: $s(Y) * X = (Y * X) + Y$ **(rule plus4)**

Theorem	Rule Chosen	Instantiation	Notes
comm	times1	-	heap overflow
dist	plus1	-	heap overflow

Theorem	Rule Chosen	Instantiation	Notes
distr	plus1	-	heap overflow
assm	plus1	-	heap overflow
zerotimes	-	-	loops
times2right	plus2	-	heap overflow

Out of 31 test runs the offending rewrite rule was correctly identified in 16 cases. In 3 further cases although a different rewrite rule was identified the system nevertheless managed to manufacture a patch that allowed the theorem to go through. Of the 16 correctly identified rules the system managed to synthesise the correct patch in 12 cases and complete the proof in 11 of these.

6.1 Discussion

The above results are, to an extent, discouraging. However analysis of the problems and some pen-and-paper working suggests several approaches.

Firstly, it is clear that the existing middle-out reasoning technology in λ*Clam* is incapable of synthesising a constant when a function has been implicitly suggested. This should not be too hard to correct. Secondly, choosing essentially random theorems in order to verify a program is, unsurprisingly, not terribly accurate. Furthermore, pen-and-paper working suggests the results are likely to improve dramatically if several proofs (or even all branches in one proof – since λ*Clam* works depth-first the critics are fired on the first "bad" proof branch encountered rather than inspecting the entire tree) were checked before a rule were selected for revision. This is technically difficult in λ*Clam* since Teyjus λProlog does not perform garbage collection and is inclined to run out of heap when attempting two proofs in a row.

The experiment also suggests that potential rule instantiations should be screened to disallow trivial suggestions (eg. the instantiation of plus2 to $s(Y)+X = X+Y$ in the attempt to prove assp in scenario 2).

Lastly a few bugs in λ*Clam* were revealed causing unnecessary looping and heap overflows.

7 Further Work

The experiment reported above reveals some improvements and bug-fixes required in the prototype system. This then needs to be evaluated on a wider range of examples. Our target data set is the examples reported in [8]. These contain several erroneous programs where either the error or the necessary repair does not strictly conform to the rippling heuristic. This would mean that the pre-existing λ*Clam* middle-out reasoning techniques would be unable to synthesise such rules. Investigation of the ways in which the heuristic can

be relaxed and/or the technique extended is therefore necessary. An obvious avenue to explore is the best-first rippling of Johansson [11] which allows the preconditions of the ripple method to be treated as soft rather than hard constraints on the process.

It is also desirable to work at a finer level than whole rewrite rules when generating program slices. Program slicing applied to functional programming typically uses function application/redexes as the nodes in program slices. Working at this level would have the advantage of allowing only subexpressions of the RHS of rules to be replaced which would again reduce the search space required for correct instantiations.

8 Related Work

Monroy [15] has previously used middle-out reasoning in the context of proof planning for repairing theorems. His work attempts to synthesize a *corrective predicate* in the course of proof. 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. In the presence of incorrect definitions this system may synthesise the predicate `False` since it has no mechanism for excluding rewrite rules, an example of this is shown in [7]. At best, when faced with an incorrect rule, Monroy's system synthesises a domain restriction limiting the theorem to those cases which are correct rather than altering the rule itself.

Colton and Pease [4] use techniques inspired by Imre Lakatos's 'Proofs and Refutations' [13] to modify theorems. As with Monroy's system the techniques focus on restricting the domain of the theorem by barring particular counter-examples (eg. "All primes *except 2* are odd"); identifying some property common to all counter-examples; or identifying some property common to all the examples. The revision process is driven by the discovery of counter-examples rather than an analysis of proof failure.

Both these approaches assume that it is the theorem statement that is in error not the definitions of the constants appearing in the statement. This latter scenario is more likely to be the case in proof-directed debugging.

9 Conclusion

This paper has put forward a proof-based methodology for locating and repairing errors in programs based upon program slicing (for locating errors) and middle-out reasoning (for repairing these errors). The approach is intended for use in proof-directed debugging applications.

A prototype implementation has been built and run on some simple examples. This demonstrates that the methodology can, in principle at least, correctly locate errors and that it is possible to achieve sufficient control over the middle-out reasoning process to synthesise appropriate patches.

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, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.

[3] O. Chitil. Source-based trace exploration. In C. Grelck, F. Huch, G. J. Michaelson, and P. Trinder, editors, *IFL 2004*, LNCS 3474, pages 126–141. Springer, March 2005.

[4] S. Colton and A. Pease. Lakatos-style methods in automated reasoning. In *Proceedings of the IJCAI'03 workshop on Agents and Reasoning, 2003.*, 2003.

[5] T. Davie and O. Chitil. One right does make a wrong. In H. Nilsson and M. van Eekelen, editors, *TFP 2006*, pages 27–40, 2006.

[6] 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.

[7] L. A. Dennis, R. Monroy, and P. Nogueira. Proof-directed debugging and repair. In H. Nilsson and M. van Eekelen, editors, *TFP 2006*, pages 131–140, 2006.

[8] 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.

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

[11] M. Johansson. A best-first planner for rippling. 4th Year Project Report, Division of Informatics, University of Edinburgh, 2005.

[12] D. Lacey, J. D. C. Richardson, and A. Smaill. Logic program synthesis in a higher order setting. In J. Lloyd, V. Dahl, U. Furbach, and P. J. Stuckey, editors, *CL 2000*, LNCS 1861. Springer, 2000.

[13] I. Lakatos. *Proofs and Refutations*. Cambridge University Press, 1976.

[14] Z. Luo and R. Pollack. Lego proof development system: User's manual. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, May 1992. See also `http://www.dcs.ed.ac.uk/home/lego`.

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

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

[17] 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, *CADE-15*, LNCS 1421, pages 129–133. Springer, 1998.

[18] A. Smaill and I. Green. Higher-order annotated terms for proof search. In J. von Wright, J. Grundy, and J. Harrison, editors, *TPHOLs'96*, LNCS 1275, pages 399–414. Springer-Verlag, 1996.

[19] M. D. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

A Test Theorems

Theorem Name	Theorem Statement
simple	$0 = 0 + (0 + 0)$
assp	$\forall x, y, z.\ (x + y) + z = x + (y + z)$
comp	$\forall x, y.\ x + y = y + x$
plus2right	$\forall x, y.\ x + s(y) = s(x + y)$
plus1lem	$\forall x.\ x + s(0) = s(x)$
plusxx	$\forall x.\ s(x) + x = s(x + x)$
zeroplus	$\forall x, y.\ x = 0 \wedge y = 0 \rightarrow x + y = 0$
comm	$\forall x, y.\ x * y = y * x$
dist	$\forall x, y, z.\ x * (y + z) = (x * y) + (x * z)$
distr	$\forall x, y, z.\ (x + y) * z = (x * z) + (y * z)$
assm	$\forall x, y, z.\ (x * y) * z = x * (y * z)$
zerotimes	$\forall x, y.\ x = 0 \vee y = 0 \rightarrow x + y = 0$
times2right	$\forall x, y.\ x * s(y) = x + (x * y)$
expplus	$\forall x, y, z.\ x^{y+z} = x^y * x^z$