Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 19–33
www.elsevier.com/locate/entcs

Enhancing Theorem Prover Interfaces with Program Slice Information

Louise A. Dennis[1,2]

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

Abstract

This paper proposes an extension to theorem proving interfaces for use with proof-directed debugging and other disproof-based applications. The extension is based around tracking a user-identified set of rules to create an informative program slice. Information is collected based on the involvement of these rules in both successful and unsuccessful proof branches. This provides a heuristic score for making judgements about the correctness of any rule.
A simple mechanism for syntax highlighting based on such information is proposed and a small case study presented illustrating its operation. No implementation of these ideas yet exists.

Keywords: Proof-Directed Debugging, Program Slicing, Verification

1 Introduction

The use of verification for locating errors in theorems, and more specifically programs, is a relatively neglected area as is the provision of interfaces to assist in this task. This paper considers the proof-directed debugging of functional programs and proposes an extension to current theorem proving interfaces to support this.

The extension is based on the assumption that the debugging process involves locating a program statement or, in the case of functional programs, function case which is incorrect. This incorrect statement will appear in a *program slice* which can be identified during verification. Other program slices leading to correct deductions may also be identified during proof. This information can then be used to create appropriate syntax highlighting of function cases in an interface. A potential highlighting scheme is put forward and a simple case study based around Isabelle/HOL [12] and ProofGeneral [1] is performed to show how this would work.

[1] This research was funded by EPSRC grant GR/S01771/01 and Nottingham NLF grant 3051.
[2] Email:lad@cs.nott.ac.uk

doi:10.1016/j.entcs.2006.09.019

No implementation has yet been performed however potential issues are discussed in the context of Isabelle Proof General.

Although the discussion in this paper is based around an application to proof-directed debugging it is likely that similar mechanisms may also be useful in other situations where the cause of a proof failure needs to be identified.

The paper is organised as follows: §2 discusses the concepts of proof-directed debugging and program slicing; §3 present a mechanism for tracking program slices through a proof and §4 presents examples of this mechanism at work via a simple case study; §5 discusses some results using a similar mechanism within an automated system; §6 looks at some related work and §7 discusses implementation issues and other further work.

2 Proof Directed Debugging and Program Slicing

Proof-directed debugging was first suggested by Harper [10] and work is underway to extend this into a framework for locating program errors through the proof process [6]. The idea of using a framework rather than relying on a user's skill at general proof, is based on the example of Algorithmic debugging [15,9,11]. Algorithmic debugging constructs an execution tree of a run of the program on some input and then queries the user each time this tree branches. This identifies branches which are returning false results and so locates sections of code responsible for errors.

Program Slicing was first suggested by Weiser [17]. The key idea was to identify a variable of interest at some point in a program (called the *slicing criterion*) and then extract a fragment of the program (a *program slice*) either containing all those statements upon which the value of the variable at that point depended or that fragment whose values were effected by the value of that variable at that point. Program Slicing techniques for imperative languages have generally followed this work [16] using control flow graphs, data flow graphs or other graph-based representations of programs with statements represented as nodes in the graph and a program slice as a set of nodes from the graph. In functional programs function application takes the place of program statements. The notion of a slicing criterion can also be generalised (e.g. to a projection as in [14]).

The intention behind proof-directed debugging is to use the branching structure of a proof to create program slices and use these to assist in the location of errors. There is clearly a need to provide appropriate tools (i.e. tactics/Isar methods) tailored to this task. This paper does not concentrate on this aspect but considers instead the way a theorem prover's interface could assist a user through the presentation of relevant program slices.

3 Proof Tree Branches as a Slicing Criterion

The verification of functional programs naturally involves splitting a program into a set of equational rules each corresponding to a case in its functional definition. The usage of these rules in the proof can thus be tracked, effectively creating a program

slice (ie. those parts of the program used in the proof of any program), and a "score" maintained indicating how many true and false branches of the proof have used that rule (typically as part of a simplification process). These scores can then be used by the interface to return additional information to the user. For simplicity, we shall continue to refer to these rule traces as program slices even though, in the context of theorem proving, there is no reason why they should not be a general collection of definitions, lemmas and theorems unrelated to a any program.

Let us consider a simple insertion sort program written in ML.

```
fun insert x [] = [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);
```

Each case of the definition becomes one of four equational rules:

(i) `insert x [] = [x]`

(ii) `insert x (h::t) = if x ≤h then x :: h :: t` else h :: insert x t

(iii) `sort [] = []`

(iv) `sort (x :: xs) = insert x (sort xs)`

We suggest that a proof-directed debugging interface should allow a user to nominate a selection of such definitions as "suspect" during a verification attempt. Obviously a user could choose to nominate all definitions involved in their development as suspect including ones related to the specification and even pre-existing definitions from the theorem prover's theory database however our suspicion is that this would lead to an overloading of information rendering program slicing of little use. This is an obvious subject for some experimental investigation once such an interface has been implemented.

We assume that a theorem proving system generates a sequence of *proof states* which, at the very least, contain lists of current open goals in the proof attempt. The central idea is to associate program slices with the goals in these proof states. Each goal, g, in a proof state is associated with a set of suspect rules (*a slice*), $\mathcal{S}(g)$, which have been used in the derivation of that goal. In addition to this the system also stores a set of triples in each proof state, s, associating each suspect rule, r, with two integers the first of which, the good integer, $good(r, s)$ is incremented whenever a proof branch is closed (because it has been successfully proved) and the second of which, the bad integer, $bad(r, s)$, is incremented whenever a goal is derived with a False conclusion (this can be revised if a contradiction is subsequently found in the hypotheses). Where it is obvious, the state argument will be dropped from these functions. These two scores can be used to form a probabilistic estimate of the chance that a rule is correct.

Initially $good(r)$ and $bad(r)$ are set to zero for all rules, r, and the initial goal, g_i, is associated with the empty program slice, $\mathcal{S}(g_i) = []$. As the proof progresses the system updates the information as follows:

Consider two proof states, s_n followed by s_{n+1}. s_{n+1} is derived from s_n by a

tactic t which replaces some parent goals with a set of child subgoals. For each new subgoal, g, in such a proof state with parent, g_p.

- Let R be the set of suspect rules used by t to derive g from g_p, $\mathcal{S}(g) = \mathcal{S}(g_p) \cup R$.
- If g has a False conclusion and g_p did not then for all rules r in $\mathcal{S}(g)$, $bad(r, s_{n+1}) = bad(r, s_n) + 1$.
- If g has a True conclusion then for all rules r in $\mathcal{S}(g)$, $good(r, s_{n+1}) = good(r, s_n) + 1$. Furthermore if the conclusion of g_p was False then $bad(r, s_{n+1}) = bad(r, s_n) - 1$.
  This last modification allows a False goal to become closed (by discovering a contradiction in the hypotheses) and then corrects the bad integer to cancel out the effect produced by the previous deduction of False.
- For all remaining rules, r, $good(r, s_{n+1}) = good(r, s_n)$ and $bad(r, s_{n+1}) = bad(r, s_n)$.

On the whole it would appear to be preferable if interfaces take on the task of tracking rule usage information rather than the underlying theorem prover since this information is extra-logical. However in automated, or semi-automated systems such as proof planners (e.g. IsaPlanner [8] and $\lambda Clam$ [13]) there would appear to be benefits in tracking such information in the proof system itself so that it can inform an automated debugging process [5].

The obvious mechanism for presenting this tracking information to a user is as a syntax highlighted list of rules associated with each goal. For instance this paper will use the monochrome conventions shown in Table 1. The categories have

	Highlighting convention
$r \in \mathcal{S}(g)$	**bold**
$bad(r) > good(r)$	underline
$bad(r) < good(r)$	*italics*

Table 1
Highlighting Conventions used in this Paper

been selected because they proved to be the most informative in the examples discussed below. The are interpreted as **used to derive this goal**, probably incorrect and *probably correct*. There is no reason, in principle, why such highlighting should be restricted to just three categories. Indeed, following results in an automated system, we argue in §5 for a further category of "worst" rule based on an ordering of tuples of bad and good integers.

4 Case Study

We now show some examples of proof attempts of incorrect theorems undertaken in the Isabelle/Isar system [12,18]. These examples are drawn from a corpus of buggy student ML programs [7].

We will consider the verification of the ML program shown in figure 1. 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));
```

Fig. 1. A Buggy ML Program

```
primrec
   insert_nil: "insert x  [] = []"
   insert_cons: "insert x (h#t) = (if x ≤ h then
                                   x#h#t else h#insert x t)"

primrec
   sort_nil: "sort [] = []"
   sort_cons: "sort (x#xs) = insert x (sort xs)"

recdef Once "measure length"
   once_nil: "Once [] = []"
   once_cons: "Once (x1#x2#xs) = (if x1=x2 then
                            Once (x2#xs) else x1#x2#Once xs)"

primrec
   onceOnly_nil: "onceOnly [] = []"
   onceOnly_cons: "onceOnly (x#xs) = Once (sort (x#xs))"
```

Fig. 2. Isabelle Formalisation of the Buggy program

is a real example submitted by a student as the solution to an exercise to provide a function, `onceOnly`, that when applied to a list, `l`, returned a new list containing only one copy of each element in `l`. There are three errors in this program. Firstly the basis case of the `insert` function is incorrect. Secondly a case is missing in the definition of the `Once` function (the case for lists of length one) and lastly in the `else` branch of the recursive case the expression should be `x1 :: Once (x2 :: xs)`.

An Isabelle formalisation of the student's program taken from [7] is shown in figure 2. It should be noted that this represents a naive shallow embedding of ML into Isabelle but one sufficient for proof-directed debugging at this scale. In order to verify this program a further function, `count_list` which counts the number of occurrences of its first argument in its second was used. The first theorem to be proved is:

$$\neg x \in l \Longrightarrow count\_list\ x\ (onceOnly\ l) = 0$$

For the purposes of this case study we assume that the definitions of `insert`, `sort`, `Once` and `onceOnly` are all considered suspect which gives us eight suspect rules: `insert_nil`, `insert_cons`, `sort_nil`, `sort_cons`, `once_nil`, `once_cons`, `onceOnly_nil` and `onceOnly_cons`. We also assume that the following theorem has been proved:

$$onceOnly\ l = Once(sort\ l) \tag{1}$$

The remainder of this section is organised as follows. §4.1 illustrates slice creation in the initial stages of the proof in order to give an idea of how the information updating works, §4.2 illustrates the effect of reaching a false goal, and §4.3 illustrates

what happens when cases are missing.

4.1 Basic Usage

The following table shows the information held in the initial proof state

Rule	good	bad	Rule	good	bad
`insert_nil`	0	0	`Once_nil`	0	0
`insert_cons`	0	0	`Once_cons`	0	0
`sort_nil`	0	0	`onceOnly_nil`	0	0
`sort_cons`	0	0	`onceOnly_cons`	0	0

From now on we will omit the full table but concentrate instead on the summary of the information that can be provided with syntax highlighting.

At the start of the proof there is one Isabelle goal

```
1. ¬ x ∈ l ⟹ count_list x (onceOnly l) = 0
```

to which is attached the empty slice. Presentationally it seems advisable to omit any rules defining constants not appearing in the current goal So the initial goal would display the additional information (NB. at present these rules do not fit into any of the categories described in Table 1 therefore neither is highlighted in any way):

- `"onceOnly [] = []"`
- `"onceOnly (x#xs) = Once (sort (x#xs))"`

The proof attempt proceeds by simplifying, replacing `onceOnly l` with `Once (sort l)`, according to (1), and then applying length induction on the list[3]. Since (1) isn't in our suspect list its use in simplification isn't recorded. We don't chain rule tracking back through additional lemmas so there is no record that, even implicitly, `onceOnly_nil` and `onceOnly_cons` were involved in the goal. Once again it will need experimentation with an implementation to determine whether this is a sensible choice. This gives us the following Isabelle goal:

```
1. !!xs. [| ∀ ys. length ys < length xs → ¬ x ∈ ys
                    → count_list x (Once (sort ys)) = 0;
            ¬ x ∈ xs |]
         ⟹ count_list x (Once (sort xs)) = 0
```

This introduces two new suspect constants but has so far used none of our rules. Furthermore the constant `onceOnly` is no longer mentioned and so its definitional rules are dropped from the display list. Hence the following suggested output.

- `"sort [] = []"`
- `"sort (x#xs) = insert x (sort xs)"`

[3] It takes some experience with these styles of proof to select length induction as the appropriate scheme. At present this work presumes a user with relatively sophisticated theorem proving ability yet paradoxically rather naive program debugging skills – providing further support in the choice of Isar methods is left to further work.

- `"Once [] = []"`
- `"Once (x1#x2#xs) = (if x1=x2 then Once (x2#xs)`
  `else x1#x2#Once xs)"`

The next step is a case split on `xs` using the Isar `cases` method followed immediately by simplification of all goals. This automatically discharges the first goal associated with the case split (for $xs = []$) leaving us with one goal:

```
1. !!a list. xs = a # list ⟹
             count_list x (Once (insert a (sort list))) = 0
```

Discharging the first goal creates a slice consisting of `sort_nil` and `once_nil` and updates the good integers so that $good(sort\_nil) = good(once\_nil) = 1$. The remaining goal was generated using the rule `sort_cons` and so its slice is `[sort_cons]`.

Following the syntax highlighting conventions, therefore, we get the following rule annotations:

- `"insert x [] = []"`
- `"insert x (h#t) = (if x < h then x#h#t else h#insert x t)"`
- *`"sort [] = []"`*
- **`"sort (x#xs) = insert x (sort xs)"`**
- *`"Once [] = []"`*
- `"Once (x1#x2#xs) = (if x1=x2 then Once (x2#xs)`
  `else x1#x2#Once xs)"`

already we are seeing information about program slices in which we can have some confidence and we get some information on the slice which is relevant to the current goal.

4.2 Inferring False

It becomes clear, while attempting the above proof, that some independent lemmas need to be established about the `sort` function. This provides a good example of how the system behaves when a goal evaluates to False. Let us consider a simple lemma to show that all members of a list, `l`, are also members of `sort l`.

We start with the goal:

```
theorem "x ∈ l ⟹ x ∈ (sort l)"
```

Following our previous rules and guidelines the displayed rules are:

- `"sort [] = []"`
- `"sort (x#xs) = insert x (sort xs)"`

The proof continues by length induction on `l` (which does not change the annotation) followed by a case split on `xs` and simplification of all goals. The first subgoal is discharged automatically, leaving:

```
1. !!a list.
      [| if a = x then True else x ∈ list; xs = a # list;
         ∀ ys. length ys < Suc (length list) →
                                  x ∈ ys → x ∈ sort ys;
```

```
        if a = x then True else x ∈ list |]
    ⟹ x ∈ insert a (sort list)
```

and the highlighted rules:

- `"insert x [] = []"`
- `"insert x (h#t) = (if x ≤h then x#h#t else h#insert x t)"`
- *`"sort [] = []"`*
- **`"sort (x#xs) = insert x (sort xs)"`**

It then proceeds by cases on `(sort list)` followed by simplification which gives two subgoals with their associated program slices:

```
1. [| ¬ x ∈ list; if a = x then True else x ∈ list;
      sort list = [];
      if a = x then True else x ∈ list |]
   ⟹ False
```

- **`"insert x [] = []"`**
- `"insert x (h#t) = (if x ≤h then x#h#t else h#insert x t)"`
- *`"sort [] = []"`*
- **`"sort (x#xs) = insert x (sort xs)"`**

```
2. [| if a = x then True else x ∈ list; sort list ≠ [];
      x ∈ list → x ∈ sort list;
      if a = x then True else x ∈ list |]
   ⟹ x ∈ insert a (sort list)
```

- `"insert x [] = []"`
- **`"insert x (h#t) = (if x ≤h then x#h#t else h#insert x t)"`**
- *`"sort [] = []"`*
- `"sort (x#xs) = insert x (sort xs)"`

This identifies a program slice that has been involved in producing the False goal ([`insert_nil`, `sort_cons`]) and therefore assists in the hunt for errors.

In some similar proofs the step case is automatically discharged in which case $good(sort\_cons) = bad(sort\_cons) = 1$ and the rule's annotation becomes **`"sort (x#xs) = insert x (sort xs)"`** for the first goal leaving only `insert_nil` highlighted as "probably incorrect" giving further clues as to the culprit.

In this particular proof, attempts to prove the second goal lead to further proof branches that result in False conclusions attributable to `insert_nil` but also several branches that are discharged – overall $good(insert\_nil) = 0$ in all states while in general $bad(sort\_cons) = good(sort\_cons) + 1$. This suggests that the user may need access to further information about a rule's good and bad integers. Although it is unclear how such information can be conveyed by syntax highlighting alone, it would certainly be possible to introduce a further highlight for the "worst" rules (see §5) and/or to allow optional display of the good and bad values alongside the rules in which case `insert_nil` would be singled out in this example.

4.3 Getting Stuck

Assuming that `insert_nil` has been fixed, the last example we will consider picks up the main verification at a later stage. We will now assume that `insert` and `sort` have been removed from the suspect list. Two new functions and a new lemma have been introduced. `minl` returns the minimum element of a list of naturals and `-minl` returns a list with one occurrence of its minimum element removed. Among other things the following lemma has been established:

$$l \neq [] \Longrightarrow (sort\ l) = (minl\ l) \# sort(-minl\ l)$$

which when used in the proof leads to the goal

```
1. count_list x (Once (minl (a#list) #
                       sort (-minl (a#list)))) = 0
```

and the highlighted rules:

- *"Once [] = []"*
- "Once (x1#x2#xs) = (if x1=x2 then Once (x2#xs)
  else x1#x2#Once xs)"

A proof by cases follows on whether $-minl(a\#list)) = []$ simplification of the first goal leaves two subgoals of which the first:

```
1. [| a ≠ x & ¬ x ∈ list;
      (if a = minl (a # list) then list
                              else a # -minl list) = [];
       xs = a # list |]
   ⟹ count_list x (Once [minl (a # list)]) = 0
```

is associated with the following highlighted slices:

- *"Once [] = []"*
- "Once (x1#x2#xs) = (if x1=x2 then Once (x2#xs)
  else x1#x2#Once xs)"

While this doesn't directly highlight an error, the juxtaposition of the goal and the relevant rules, particularly with neither highlighted as used directly in the goal should prompt a user to recognise the omission of the relevant information.

5 Supportive Results

The ideas behind the interface design proposed here arise from work on the automated detection and repair of such errors within the proof planning framework [4]. Program slice tracking has been implemented in the $\lambda Clam$ [13] proof planning system. In the absence of an implementation in a theorem prover interface we report some results on the success of the heuristics within this system. We used a variation on the system reported in [4] [4]. That system attempts to repair erroneous rewrite rules. The system reported here simply terminated false branches and concluded the proof attempt by, for each rule, r, reporting $good(r)$ and $bad(r)$. Unfortunately

[4] Relevant code is available from the author on request.

some errors, especially those appearing in the recursive cases of definitions caused the system to be non-terminating, therefore an additional heuristic was used to close branches if the step case of an inductive proof could not be solved by appeal to the induction hypothesis [5]. We ran two experiments. In Experiment 1 closed step case branches did not contribute to the good/bad scores (ie. strictly adopting the conventions proposed in this paper). In Experiment 2 such closed branches increased the bad scores (arguably in a human proof attempt these branches would eventually have led to a False goal rather than the non-termination caused in $\lambda Clam$).

The table 2 shows the results for both sets of runs. The experiments involved 24 non-theorems based around errors in the definitions of list append, list membership and the insert and sort programs already covered in this paper. The theorems were selected from the $\lambda Clam$ benchmark set rather than being actual specifications for these functions. As such these results should be considered indicative only. The tables report, for each experiment, whether the "incorrect" rewrite rule was underlined (ie. whether its good score was greater than its bad score) and the average number of rules underlined. This is the average number when at least one rule is underlined – in several cases no rule had a larger bad integer than a good integer. The intention in presenting this average is to provide evidence of the extent to which the heuristics help focus attention on an erroneous rule – after all it is not much help if *all* the rules are underlined. Including cases where no rule is underlined reduces this average and tends to suggest better discrimination than is actually the case. To follow this up we provide a percentage of the rules excluded. This is the percentage of the rules involved with definitions actually used in the proof which were not underlined. Again this only refers to situations where at least one rule was highlighted to give an impression of the extent to which the choices were narrowed down. False positives reports the number of situations where some rule was highlighted but the incorrect rule was not. We also computed an overall score for each rewrite rule as a tuple of the bad score and the good score. These tuples were then ordered according to $\succ$ where

$$(b_1, g_1) \succ (b_2, g_2) \iff (b_1 < b_2) \vee ((b_1 = b_2) \wedge (g_1 > g_2))$$

and $<$ and $>$ are the standard order on natural numbers. We report on the percentage of cases where the intended error was picked out by this heuristic and when it was the only rule with the highest score.

The results show that it would be useful if the interface could also flag those rules which are scoring most highly under $\succ$ even where all rules are being used in more good branches than bad since this is clearly giving the best information about the location of errors.

The use of bad scoring for "stuck" goals (Experiment 2) is problematic – it improves the rate at which incorrect rules are identified, and the rate at which bad rules are highlighted as "worst" at the cost of losing discrimination (see Rules Excluded). Since the stuck heuristic is a crude attempt to mimic human "getting

[5] with the exception of a few special cases, for instance where the step case proof had branched following a case split

	Exp 1.	Exp 2.
Incorrect Rewrite Underlined	50%	66%
Average No. Rules underlined	1.62	2.11
Rules Excluded	52%	38%
False Positives	0	1
Incorrect Rewrite has Highest Score	62.5%	79.17%
Incorrect Rewrite has Unique Highest Score	29.17%	54.17%

Table 2
Summary of Experimental Results in $\lambda Clam$

stuck" behaviour it is perhaps not surprising the effects are equivocal. At any rate it is clear that, to a certain extent, this heuristic is too eager and prevents (in the first case) the proof from progressing to false branches that would (hopefully) later get scored if pursued by a human prover and, in the second case, generates too many false positives. Improving the heuristic is well outside the scope of this paper but interpretation of the above results need to bear its limitations in mind. The heuristic does suggest that there may be some benefit in allowing a human prover intervene in the scoring process and mark some branches as "bad" even where a False conclusion has not been reached.

Obviously these results are only indicative of how the heuristics might serve human users as opposed to an automated system but they do suggest that profitable use can be made of the information contained in program slices attached to proof branches. In particular the "worst" score looks particularly promising in terms of directing a user's attention to errors.

6 Related Work

The HAT tool [2] uses a mixture of algorithmic debugging and program slicing to direct a user's attention to relevant parts of a program's source. HAT creates an Evaluation Dependency Tree (EDT) tracing the execution sequence of function calls on a sample input. The nodes in this tree can be associated with their "call site" in the program. This allows the system to use a syntax highlighting mechanism to relate debugging traces back to specific parts of code. The tool works by identifying slices in the EDT and relating these back to the relevant portions of the code. This has recently been extended [3] to use a very similar polling system to that described above based on superimposing "correct" EDTs and "incorrect" EDTs to generate heuristic scores by which a "worst" slice can be identified.

In general the HAT tool only displays the most immediate redex rather than all those involved in a slice in order to reduce information overload – while it may be desirable to do something similar in proof-directed debugging it isn't at all obvious that the last rule to be used will generally prove to be the one at fault.

This is the first work I'm aware of that considers the use of proof tree branches as a slicing criterion or considers integrating the syntax highlighting interface of a debugging tool such as HAT into a Theorem Prover.

7 Further Work

7.1 Implementation

Clearly the most pressing and important piece of further work is providing an implementation of verification based program slicing to allow experimental evaluations of the extent to which it genuinely helps locate errors.

Out intention is to provide an implementation in Isabelle/Isar using the Proof General interface. This allows there to be a clean separation between the information used by the interface and that used by the underlying theorem prover. Such an approach also creates some challenges however, since the necessary properties of goals and proof states will have to be inferred. On the whole it should be relatively straightforward to identify goals and key constants within goals although it there will be some challenges involved in keeping track of proof states, in particular the relationships between parent and child goals needed to make updates correctly. In Isabelle successfully discharged goals are dropped from the proof state presented to the interface which again is likely to raise some challenges in the tracking of information.

Although no examples have been shown here where a rule is used directly with a tactic (e.g. the `rule` method in Isar) this also needs to trigger updates of tracking information. In general this should be relatively straightforward based on simple analysis of tactic calls.

Simplification is the major step where the exact rules used by the system are effectively concealed from the user. It is also the most important tactic which can be used across multiple goals discharging some but not others (so leading to ambiguities about successful proof branches) and can generate and discharge new branches within its own application invisible to the user. Fortunately Isabelle's simplifier provides a tracing mechanism from which is it possible to infer rule usage and determine when a proof branch has been discharged, from which it should be possible to infer the necessary information. It may also be possible to use the proof object (of the top theorem) to track program slice information [6].

We have not considered how backtracking should interact with program slicing. At present the design assumes that proof states are generated in sequence and implicitly assumes that they can only be backtracked in that sequence. However many theorem provers allow backtracking on any open goal not just the those most recently derived. In this case it may be necessary for the interface to store additional information about the relationships between goals and their parents from proof state to proof state. This problem may also mean that ultimately it is cleaner to store program slice information in the prover's proof state rather than in the interface.

[6] My thanks to an anonymous referee for this suggestion.

7.2 More Detailed Program Slices

So far we have considered program slices whose nodes are identifiable with the simple case structure of function definitions however there are further advantages to be gained if more sophisticated slicing is used in which function calls/sub-expressions are considered as nodes (as is common when applying program slicing to functional programs).

In the following example, again genuine, a student has been asked to provide a function, `removeAll`, which removes all occurrences of its first argument from its second. They appear to have programmed by analogy from a previous function, `removeOne`, where only one occurrence was to be removed and have forgotten to replace one call to this program. The code is expressed in Isabelle as:

```
primrec
   removeAll_nil: "removeAll x [] = []"
   removeAll_cons: "removeAll x (h#t) = (if x = h
                    then removeAll x t else h#removeOne x t)"
```

Consider an attempt to establish that

$$\neg x \in removeAll\ x\ l$$

The proof proceeds by induction on `l` followed by simplification of all goals automatically discharging the base case and leaving the step case goal:

```
1. !!a l. ¬ x ∈ removeAll x l
          ⟹ (x = a ⟶ ¬ a ∈ removeAll a l) &
               (x ≠ a ⟶ a ≠ x & ¬ x ∈ removeOne x l)
```

and highlighted rules.

- *"removeAll x [] = []"*
- **"removeAll x (h#t) = (if x = h then removeAll x t**
  else h#removeOne x t)"

Use of some introduction rules (`impI` and `conjI`) and more simplification gives three subgoals which are based around a case split on whether `x = h` and then (following from a lemma about $\in$) on the values in the head and tail of `h#removeOne x t`. The first of these (where `x = h`) is automatically discharged leaving two subgoals, the first of which is

```
1. [| x ≠ a; ¬ x ∈ removeAll x l |] ⟹ a ≠ x
```

Ideally we would like to highlight the rules associated with this goal as follows:

- *"removeAll x [] = []"*
- **"removeAll x (h#t) = (if x = h** *then removeAll x t*
  else h #removeOne x t)"

showing that `removeAll x t` is probably correct and that this goal is based on the value of `h` in `h#removeOne x t`.

This goal is easily discharged leaving only the goal:

```
2. [| x ≠ a; ¬ x ∈ removeAll x l |] ⟹ ¬ x ∈ removeOne x l
```

Again ideally we would like to highlight parts of the second program slice dif-

ferently:

- `"removeAll x [] = []"`
- `"removeAll x (h#t) = (if x = h then removeAll x t`
  `else h# removeOne x t)"`

Focusing attention on the problematic part of the rule which will eventually lead to False goals.

It should be easy enough to represent these slices within a system, for instance a simple list of integers can be used to indicate the position of a sub-expression within a rule and all sub-expressions of suspect rules stored in for use program slices. However it is much harder to see how information about which slice is relevant to a goal can be inferred without help by an interface such as Proof General. Indeed in order to supply the necessary information a theorem prover's internals may need modification in order to track the unifications performed when rules are applied in a meaningful way.

7.3 Imperative Programs

Obviously a long term objective is to extend this work to imperative programs. In these cases we lose the correspondence between program locations and rewrite rules. We would therefore need to adapt the concept of "used in a proof branch" to, for instance, identify individual program statements that had been involved in an instantiation of the assignment axiom in this branch of the proof.

8 Conclusion

This paper has discussed the use of verification as a program slicing tool. It has discussed how proof branches can be used to build up program slices based around equational rewrite rules and described a simple mechanism for deriving a heuristic score for how likely a given rule is to be correct. It has then discussed how such information might be presented to a user.

The mechanism proposed relies on a user identifying "suspect" rules. In the case study these all related to program function cases however there is no reason, in principle, why any definition or theorem in a theory could not be treated in the same way, allowing suspect specifications and definitions in general (non-verification based) proofs to be handled in the same way. The general mechanism can almost certainly be used in any situation where a reason is being sought for a proof failure.

Considerable further work, including an implementation, is required.

References

[1] D. Aspinall. Proof general: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems, Proc TACAS 2000*, volume 1785 of *LNCS*. Springer, 2000.

[2] O. Chitil. Source-based trace exploration. In C. Grelck, F. Huch, G. J. Michaelson, and P. Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004*, LNCS 3474, pages 126–141. Springer, March 2005.

[3] T. Davie and O. Chitil. One right does make a wrong. In H. Nilsson and M. van Eekelen, editors, *Seventh Symposium on Trends in Functional Programming*, pages 27–40, 2006.

[4] L. A. Dennis. Program slicing and middle-out reasoning for error location and repair. In W. Ahrendt, P. Baumgartner, and H. de Nivelle, editors, *IJCAR 2006 Workshop onDisproving - Non-Theorems, Non-Validity, Non-Provability*, 2006. To Appear.

[5] L. A. Dennis. The use of program slicing and middle-out reasoning to identify and repair program errors. Technical report, University of Nottingham, 2006. Submitted to IJCAR-2006 Workshop on Disproving - Non-Theorems, Non-Validity, Non-Provability.

[6] L. A. Dennis, R. Monroy, and P. Nogueira. Proof-directed debugging and repair. In H. Nilsson and M. van Eekelen, editors, *Seventh Symposium on Trends in Functional Programming*, pages 131–140, 2006.

[7] 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.

[8] L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In F. Baader, editor, *CADE19*, volume 2741 of *LNCS*, pages 279–283. Springer, 2003.

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

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

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

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

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

[14] N. Rodrigues and L. Barbosa. Slicing functional programs by calculation. In *Proceedings of the Dagstuhl Seminar on Beyond Program Slicing*, 2005.

[15] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[16] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

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

[18] M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99*, number 1690 in LNCS. Springer, 1999.