Citation for published version


DOI

Link to record in KAR

http://kar.kent.ac.uk/14245/

Document Version

UNSPECIFIED

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version. Users are advised to check http://kar.kent.ac.uk for the status of the paper. Users should always cite the published version of record.

Enquiries
For any further enquiries regarding the licence status of this document, please contact: researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at http://kar.kent.ac.uk/contact.html
Hat-Delta — One Right Does Make a Wrong

Thomas Davie & Olaf Chitil, University of Kent

Abstract: We outline two methods for locating bugs in a program. This is done by comparing computations of the same program with different input. At least one of these computations must produce a correct result, while exactly one must exhibit some erroneous behaviour. Firstly reductions that are thought highly likely to be correct are eliminated from the search for the bug. Secondly, a program slicing technique is used to identify areas of code that are likely to be correct. Both methods have been implemented. In combination with algorithmic debugging they provide a system that quickly and accurately identifies bugs.

1 Introduction

Program bugs often do not manifest themselves immediately. A user will often execute a program several times with correct results, only to later find a specific input that produces an erroneous behaviour. We aim to use the information that can be gathered from correct computations of the program to diagnose bugs in an erroneous computation.

We describe two new ways of exploiting information from correct computations when debugging the program. The extra information narrows the position of a bug and thus improves on earlier methods. The first method, based on finding repeated reductions, eliminates small sections of program computation from the search for bugs. After eliminating sections of computation a second method based on a program slicing can be used. Both methods provide heuristics which guide the debugger. The slicing method is less reliable in its predictions, but can remove larger numbers of questions when it is correct.

We use the two methods to locate bugs in Haskell programs. However, the technique can be applied in any situation where algorithmic debugging can be applied.
Figure 1: Buggy program used in Figure 2

```
sort [] = []
sort (x:xs) = insert x (sort xs)
```

```
insert x [] = [x]
insert x (y:ys)
  | x < y = x:y:ys
  | otherwise = insert x ys
```

Figure 2: Computation graphs of program in Figure 1. The upper computation is correct, but the lower one is erroneous.

## 2 Correct Subcomputations

When a computation produces a correct result, it is reasonable to assume that all its subcomputations produce correct results. This statement is not always true – it is possible that two bugs cancelled each other out. It is, however, remarkably hard to come up with non-trivial examples in which the statement is incorrect, and as such there is a very low chance of marking incorrect reductions correct. With this in mind, it is immediately possible to eliminate an area of computation graph from the search for a bug.

If a subcomputation appears in a correct computation graph, as well as that of an erroneous computation, it is possible to label it as a correct computation. The debugger can therefore eliminate it from the search for a bug. Not only the root reduction is said to be correct, but also all subcomputations involved in the reduction. In Figure 2 we can see that \texttt{sort [1,2]} computing the correct result has caused the computation of \texttt{sort []}, \texttt{insert 2 []}, \texttt{sort [2]} and \texttt{insert 1 [2]} to be considered correct as well. At present we consider a computation having a correct result to be a good indicator that all reductions within it are correct, and we label them as such (although we will allow the user to disable this behaviour). Experimental results will later tell us if such reductions should be trusted less.

Figure 2 shows both a correct and an erroneous computation of the program shown in Figure 1. Computation that occurred in the correct trace is
1  sort [] = []
2  sort (x:xs) = insert x (sort xs)

1  insert x [] = [x]
1  insert x (y:ys)
1  | x < y    = x:y:ys
0  | otherwise = insert x ys

Figure 3: The program shown in Figure 1, labeled.

marked as coloured text. While this technique finds a few correct subcompu-
tations, and hence reduces the number of questions asked by an algorithmic
decoder, it does not give very much extra information. In the above ex-
ample, the number of questions asked by an algorithmic debugger is cut by
only one.

Finding extra correct reductions gains relatively little – each correct re-
duction found in the erroneous computation graph cuts the number of ques-
tions asked by at most one. Often finding a correct reduction does not cut
the number of questions at all.

3 Correct Program Slices

The ‘correct subcomputation’ method looks only at the computation graph,
and ignores a large amount of data available from looking at the program as
well. The source code used in executions that ran correctly is more likely to
be correct than code that has never been executed. If some slice of the code
was executed 50,000 times during the computation of the correct program,
then it is a good guess that this slice is correct. This heuristic allows a
significant narrowing of the area of the program in which the bug is likely to
be.

Each reduction in the computation arises from a program slice. If a
program slice is correct, then all reductions arising from this slice must be.
Algorithmic debugging is based on this property – if a reduction is erroneous,
then the program slice is incorrect. Our method uses the case where a
reduction is known to be correct – in this case a slice from which it aries is
likely though not certain to be correct. This method allows the debugger to
find a new set of reductions that are likely to be correct.

In the example used before (Figure 1), we can apply this process, and
arrive at the result shown in Figure 3. This figure adds labels indicating
the number of times each line has been executed in the correct computation.
This labeling suggests a buggy line, but a debugging session is needed to
confirm it. An algorithmic debugger may now order its questions differently
in the hope of finding the bug faster. Instead of traversing the graph in
a breadth first manner (looking only at children of erroneous nodes), the
new algorithmic debugger will look at nodes in order of their likely erroneousness (based on the likely bugginess of the part of the program being executed). If a node is found to be definitely erroneous, the system reapplys the heuristic within that node’s children, rather than continuing in order of likely erroneousness. This method results in this new debugging session:

\[
\text{insert } 2 \ [1] \ \text{is } [2] \\
\text{No}
\]

Bug identified in ‘insert x (y:ys)’:
\[
\text{| otherwise = insert x ys}
\]

First, the most likely erroneous reduction is the computation of \text{insert} 2 \ [1] \ \text{to } [2], as there is no record that this section of the program has ever been used to produce a correct result. Hence the first question the algorithmic debugger asks. Second, the reduction of \text{insert 2 [] to [2]} is known to be correct from the technique described in Section 2, so now the bug can be identified. Compared to ordinary algorithmic debugging, there is a clear advantage in using this technique. In this example, the total number of questions asked is cut from 4 to only 1 (the normal ordering would cause the algorithmic debugger to ask 4 questions). In more complex examples the number of questions asked can be cut by an even greater factor.

An extension to this method would refine the estimates of program correctness as the debugger proceeded. To implement this method, each time the user answers the algorithmic debugger with a ‘yes’, the system would gain a new correct sub-program. The new correct sub-program in turn gives it a new reduction that it knows to be correct, and adds to the total information about correctly executing parts of the program.

4 Combination of Methods

Finding correct subcomputations is not very effective at cutting the number of questions asked, however it can be used to greatly improve upon finding correct program slices. In finding correct subcomputations we are able to identify several reductions that are very likely to be correct. These reductions can then be used to provide further program slices that have been executed correctly, and provide more data for our second method to work with.

5 Related Work

\textit{Delta Debugging} has been developed for imperative languages by Zeller and Cleve [9]. Their approach uses comparisons of two execution states at different points in time. The approach is hard to transfer directly to functional languages as it relies on comparing program state. Delta debugging was
however the inspiration for our comparative debugger. In *Scalable Statistical Bug Isolation* [15], the authors describe a method of performing statistical analysis on multiple program runs to identify the causes of program failure. The approach looks at the computation of predicates within programs and isolates predicates which appear to be good indicators of a certain bug occurring. This in turn allows them to isolate control flow patterns that cause erroneous behaviour.

### 6 Discussion and Future Work

**Initial Implementation** The two methods described in this paper have been implemented on top of the HAT tracer. First the HAT-DETECT debugger was re-written, allowing it to work correctly with the current HAT trace file format. After this, the new HAT-DETECT was used as a basis for the HAT-Delta debugger. Experiments comparing HAT-Delta with the original HAT-DETECT have so far shown that the number of questions asked is reduced by a vastly varying amount depending on the program, and the executions of that program. In some experiments, the number of questions is not cut at all, while in others the number is cut by a factor of ten. So although an initial implementation has been completed, there is still significant work to do in determining the best heuristics to use in order to provide a short search path in as many cases as possible.

**Combination With Other Views** The algorithmic debugging process consistently finds bugs, however it can often ask large numbers of questions, or questions the user finds difficult to answer. We have presented two methods for reducing the number of questions asked by an algorithmic debugger. However, the information gathered from the traces is independent of the algorithm used to view it. The information can be combined with other views, and provide the user with more information. Olaf Chitil has described a method of improving algorithmic debugging by allowing the user to navigate freely (and thus choose the most likely position of the bug themselves) [5]. The information gathered by comparative debugging could be combined with this view to provide hints to the user about what is buggy. In a similar way, the information can be combined with an observation based view. This view mechanism presents a listing of function applications and their results. An extension could highlight the likely erroneousness of these applications.

### 7 Conclusion

Large amounts of extra information can be gained by examining traces of programs that evaluate correctly. This information can be used to lower the number of questions asked by an algorithmic debugger using the methods
described in this paper. The proportion of questions removed from the debugging session can be as great as 90%, but it can be as little as 0%. There have been no observations made of what conditions are needed for delta debugging using this technique to be effective.
Bibliography


