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.

```
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 1: Buggy program used in Figure 2

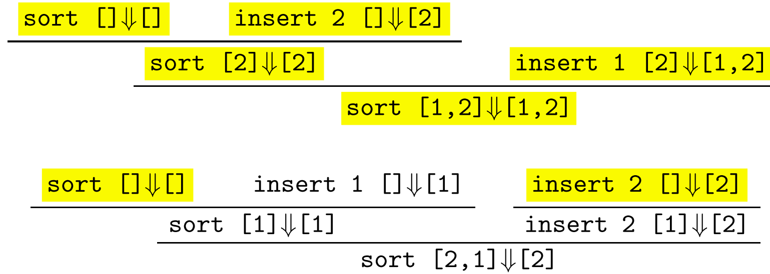

Figure 2: Computation graphs of program in Figure 1. The upper computation is correct, but the lower one is erroneous

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 `sort [1,2]` computing the correct result has caused the computation of `sort []`, `insert 2 []`, `sort [2]` and `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 subcomputations, and hence reduces the number of questions asked by an algorithmic debugger, it does not give very much extra information. In the above example, the number of questions asked by an algorithmic debugger is cut by only one.

Finding extra correct reductions gains relatively little – each correct reduction found in the erroneous computation graph cuts the number of questions 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 reapplies the heuristic within that node's children, rather than continuing in order of likely erroneousness. This method results in this new debugging session:

```
insert 2 [1] is [2]
> No

Bug identified in 'insert x (y:ys)':
  | otherwise = insert x ys
```

First, the most likely erroneous reduction is the computation of `insert 2 [1]` 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 `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

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

[1] Stephen R. Adams. Efficient sets — a balancing act. *Journal of Functional Programming*, 3(4):553–562, 1993.

[2] Krasimir Angelov and Simon Marlow. Visual Haskell: a full-featured Haskell development environment. In *Haskell'05: Proc. 2005 ACM SIGPLAN Haskell Workshop*, pages 5–16, Tallinn, Estonia, 2005. ACM Press.

[3] Thomas Böttcher and Frank Huch. A Debugger for Concurrent Haskell. In *Draft Proc. 14th Intl. Workshop on Implementation of Functional Languages (IFL'2002)*, pages 129–141, Madrid, Spain, 2002. Tech. Report 127-02, Dept. de Sistemas Informaticos y Programacion, Universidad Complutense de Madrid.

[4] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless G-Machine. In *Proc. 1988 ACM Conference on LISP and Functional Programming*, pages 244–258, Snowbird, Utah, USA, 1988. ACM Press.

[5] Olaf Chitil. Source-based trace exploration. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004*, LNCS 3474, pages 126–141. Springer, 2005.

[6] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming Haskell for tracing. In *Implementation of Functional Languages, 14th Intl. Workshop, IFL 2002, Revised Selected Papers*, pages 165–181. Springer LNCS 2670, 2003.

[7] K. Claessen and R. J. M. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. 5th Intl. ACM Conference on Functional Programming*, pages 268–279. ACM Press, 2000.

[8] K. Claessen, C. Runciman, O. Chitil, J. Hughes, and M. Wallace. Testing and tracing lazy functional programs using QuickCheck and Hat. In *Advanced Functional Programming, 4th International School (AFP 2002)*, pages 59–99. Springer LNCS 2638, 2002.

[9] Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. In *Proc. 4th Intl. Workshop on Automated Debugging (AADEBUG 2000)*, Munich, Germany, 2000. http://xxx.lanl.gov/abs/cs.SE/0012009.

[10] Tom Davie. Animation of lazy evaluation in Haskell using Hat traces. BSc project dissertation, Dept. of Computer Science, University of York, 2004.

[11] Mike Dodds. Using trace data to diagnose non-termination errors. MEng project dissertation, Dept. of Computer Science, University of York, 2004.

[12] Keith Hanna. Interactive visual functional programming. In *Proc. 7th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP'02)*, pages 145–156, Pittsburgh, USA, 2002. ACM Press.

[13] T. Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, June 1984.

[14] Daan Leijen. wxHaskell – a portable and concise GUI library for Haskell. In *Proc. ACM SIGPLAN 2004 Haskell Workshop*, pages 57–68, Snowbird, Utah, September 2004. ACM Press.

[15] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'2005)*, pages 15–26, Chicago, Illinois, June 2005. ACM Press.

[16] H. Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping University, April 1998.

[17] Henrik Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *J. Funct. Program.*, 11(6):629–671, 2001.

[18] Bernard Pope and Lee Naish. Practical aspects of declarative debugging in Haskell 98. In *Proc. 5th ACM SIGPLAN Intl. Conf. on Principles and Practice of Declarative Programming (PPDP'03)*, pages 230–240, Uppsala, Sweden, 2003. ACM Press.

[19] Niklas Röjemo. Highlights from nhc: a space-efficient Haskell compiler. In *FPCA '95: Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture*, pages 282–292, La Jolla, USA, 1995. ACM Press.

[20] Colin Runciman. TIP in Haskell — another exercise in functional programming. In Rogardt Heldal, Carsten Kehler Holst, and Philip

Wadler, editors, *Proc. Glasgow Workshop on Functional Programming 1991*, pages 278–292. Springer Verlag BCS Workshops in Computing, 1992.

[21] Tom Shackell and Colin Runciman. Faster production of redex trails: The Hat G-Machine. In Marko van Eekelen, editor, *Proc. 6th Symposium on Trends in Functional Programming (TFP 2005)*, pages 135–150. Tartu University Press, Estonia, 2005.

[22] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In *Proc. 9th Intl. Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97)*, pages 291–308, London, UK, 1997. Springer-Verlag.

[23] Andrew Peter Tolmach. *Debugging standard ML*. PhD thesis, Princeton University, Princeton, NJ, USA, 1992.

[24] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In Ralf Hinze, editor, *Proc. 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, Firenze, Italy, September 2001. Universiteit Utrecht UU-CS-2001-23. Final version in ENTCS 59(2).