Hat-delta — One Right Does Make a Wrong

Thomas Davie and Olaf Chitil

Computing Laboratory, University of Kent, CT2 7NF, UK
{tatd2, o.chitil}@kent.ac.uk,
WWW home page: http://www.cs.kent.ac.uk/people/{rpg/tatd2, staff/oc}

Abstract. We outline two heuristics for improving the localisation of 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 behavior. First, reductions that are thought highly likely to be correct are eliminated from the search for the bug. Second, a program slicing technique is used to identify areas of code that are likely to be correct. These techniques are used in combination with algorithmic debugging to create a debugger that quickly and accurately locates bugs. The implementation of a prototype system is now complete.

1 Introduction

Features of Haskell [6] such as strong type checking eliminate several simple classes of bugs in programs. These features do not however stop all bugs from occurring. Instead, it simply means that bugs are less commonly found at run time and that those that get past the compiler are often more subtle and difficult to comprehend. These 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 erroneous behavior. We aim to use the information that can be gathered from correct computations of the program to diagnose bugs in an erroneous computation.

In this paper, we describe two new ways of exploiting information from correct computations whilst debugging the program. These methods help to identify the location of a bug and thus improve on earlier methods. The first method, based on finding repeated reductions eliminates sections of program computation from the search for bugs. This 'computation comparison' method can be combined with a second method based on a program slicing. Both methods provide heuristics which guide the debugger. The 'program slicing' method is less reliable in its predictions, but can reduce the search space more when it is correct.

We use the two methods to locate bugs in Haskell programs. We have implemented the system based on the Hat tracer and are evaluating the results. Although our implementation is based on Haskell and Hat, 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
```

Fig. 1. Buggy program used in Figure 2

2 Algorithmic Debugging

We initially describe our comparative debugging methods as improvements over algorithmic debugging [7, 5, 4]. The methods are also applicable to other methods of debugging, as we describe in Section 7. Algorithmic debugging is a technique developed to deal with finding logical errors in declarative programs. The technique was later applied to functional languages. The process works by continuously narrowing down the part of a computation a bug has occurred in. At each stage in the process a question is asked: "should the application $fa_1 \ldots a_n$ reduce to x?". If the user answers yes, the reduction is marked as 'correct', meaning that no bug manifests itself at this level. If the user answers no, the reduction is marked as 'erroneous', meaning that either the definition of the function f, or one of its subcomputations is buggy. Given a no answer the debugger proceeds to examine all subcomputations. A reduction is identified as 'buggy' iff it is marked as erroneous and all subcomputations are correct. The program slice from which this reduction arose (the parts of the definition of the applied function that have been used in this reduction) contains a bug.

Figure 2 shows the *Evaluation Dependancy Tree* for the buggy code in Figure 1. The EDT shows what sub-computations were needed in order to compute an expression's value. If the evaluation of $fa_1 \ldots a_n$ depended on evaluating $gb_1 \ldots b_m$, then the node for $gb_1 \ldots b_m$ appears as a child of the node for $fa_1 \ldots a_n$ in the EDT. This EDT shows that to evaluate `sort [2,1]` (incorrectly), the program had to evaluate `sort [1]` and `insert 2 [1]`. To debug this program, an algorithmic debugger would ask the following questions:

```
sort [2,1] = [2]    The system asks about the top level node.
> no                The user says it is incorrect.
sort [1] = [1]      The system asks about an erroneous node's child.
> yes
insert 2 [1] = [2]
> no                One of the children is erroneous.
insert 2 [] = [2]   The system investigates that node's children.
> yes               All those children are correct, so the node is buggy.

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

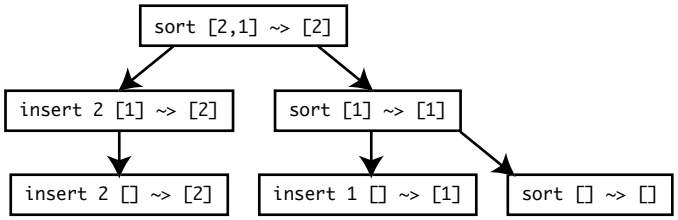


Fig. 2. Computation graph of program in Figure 1

As shown in this example, the program EDT is traditionally traversed in a top down, left to right manner. It is important to realise however that this traversal order is not mandatory. Questions may be asked in any order.

Algorithmic debugging has proven to be extremely useful for short computations. However, for large computations it can ask a large number of questions, or ask questions that the user either cannot answer, or knows to be irrelevant. We aim to reduce the number of questions asked and eliminate irrelevant questions.

3 Correct Subcomputations

If 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. Using this assumption, we can produce our first heuristic — with a high degree of certainty, the bug is not in the areas of the EDT that have been identified.

If the debugger is told that a reduction is correct, not only the root reduction is said to be correct, but also all subcomputations involved in the reduction. In Figure 3 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. It should be noted that thanks to Haskell being a lazy-evaluated language, we can easily identify which parts of the computation are actually used in this reduction — if they appear in the trace, then they have been evaluated and hence are used in the computation.

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. Experimental results so far show that this is reasonable, however there will be a mechanism in the final debugger to regard these reductions merely as highly likely to be correct.

Figure 3 shows both a correct and an erroneous computation of the program shown in Figure 1. Computation that occurred in the correct trace is marked in grey. While this technique finds a few correct subcomputations and hence

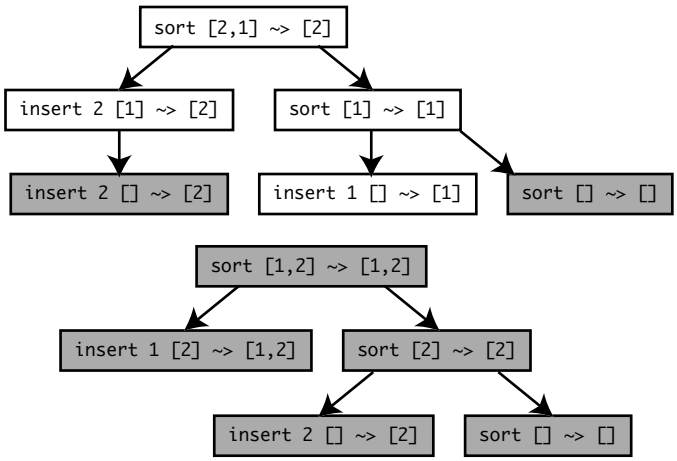


Fig. 3. Computation graphs of program in Figure 1. The upper computation is correct, while the lower is erroneous.

reduces the number of questions asked by an algorithmic debugger, it does not give very much extra information. In our 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 EDT cuts the number of questions asked by at most one. Often finding a correct reduction does not cut the number of questions at all. Finding an erroneous reduction is much more useful — when one is found, we can remove all of its siblings from the search space and thus remove a significant number of questions. Our next heuristic attempts to do this.

4 Correct Program Slices

The 'correct subcomputation' method looks only at the EDT and ignores the program. The source code used in executions that ran correctly is more likely to be correct than code that has never been correctly executed. If some slice of the code is 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 slice 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 is likely to be correct (although not necessarily). This method allows the debugger to find a new set of reductions that are likely to be correct.

```
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
```

Fig. 4. The program shown in Figure 1, labeled.

In the example used before (Figure 1), we can apply this process and arrive at the result shown in Figure 4. 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 single-step 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. This results in a new debugging session:

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

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

First, the most likely erroneous reduction is the reduction of `insert 2 [1]` to `[2]`, as this section of the program has never been executed, so the algorithmic debugger first asks this question. Second, the reduction of `insert 2 []` to `[2]` is known to be correct from the technique described in Section 3 and hence, the bug is 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.

We use a combination of both the correct subcomputation and the program slicing methods. 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 program slicing method to work with.

An extension of this method refines the estimates of program correctness as the debugger proceeded. To implement this extension, each time the user answers the algorithmic debugger with a 'yes', the system would gain a new correct subprogram. These answers in turn give it a new reduction that it knows to be correct and add to the total information about correctly executing parts of the program.

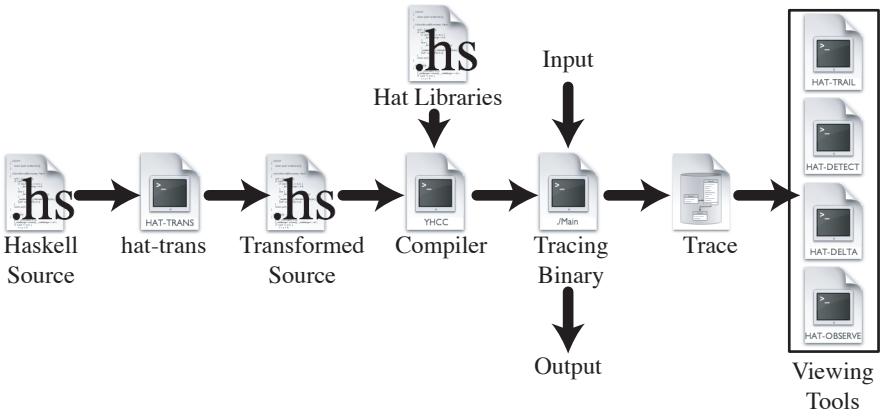
5 Implementation

The two heuristics described in this paper have been implemented on top of the HAT [1][8] tracer. HAT is a tracer for real Haskell programs, which provides a framework on which our debugger can be built.

5.1 The Hat Tracer

Tracing a computation with Hat consists of two phases: trace generation and trace viewing. These two phases are performed entirely separately. This modularity allows HAT to trace exactly the same computation multiple times.

First, HAT-TRANS converts a Haskell program into a self tracing version. This program, in addition to its normal input/output behavior writes a trace into a file. After this program has terminated, the programmer studies the trace with a collection of viewing tools. In this case, we have implemented an additional viewing tool — HAT-DELTA— that performs algorithmic debugging, guided by our heuristics.



5.2 Hat-Detect

HAT-DETECT implements standard algorithmic debugging. The program first constructs an evaluation dependancy tree from the HAT trace file. During the debugging process HAT-DETECT maintains a list of EDTs in which it has yet to search for the bug, we will call this list the 'candidate list'. The candidate list initially contains only the EDT for the entire computation. If a user answers a question with 'yes', then the bug is not in the EDT for that reduction and hence we discard it and move onto the next EDT in our list. If the user answers a question 'no', then a bug does manifest itself in this EDT. In this case we discard the contents of the candidate list and replace them with the children of this node. When we run out of EDTs to look at, we have found our bug and we

[1] http://www.haskell.org/hat

display a message. We keep track of the last question the user answered 'no' to and display a message indicating that the bug is in the definition of the function involved in this reduction.

5.3 Hat-Delta

To implement the computation comparison method, HAT-DELTA must do some extra work when the user answers 'yes' — HAT-DELTA gains correct computations from the user answering a question 'yes', as well as from multiple input traces. HAT-DELTA must gather all the sub-computations and then search all other trees eliminating these computations (as we no longer need the user to tell us that they are correct).

To implement the program slicing method HAT-DELTA must again do more work. With this heuristic, HAT-DELTA must maintain information about program slices and how often they have been executed. When a user answers 'yes', HAT-DELTA examines all child computations, to determine the program slice used in their execution. These are then unioned with the existing information. In each debugging step, HAT-DELTA will scan the EDTs in the candidate list looking for the computation most likely to be buggy (according to the heuristic) and ask that question. If it is found to be erroneous, HAT-DELTA will replace the contents of the candidate list with that nodes children. If not, HAT-DELTA will prune out that node, regenerate its heuristics and continue with the next most likely node.

HAT-DELTA currently looks ahead by up to three levels in the candidate list, however this number is arbitrarily assigned and a different number of levels might be more appropriate. The purpose of stopping HAT-DELTA looking too far ahead is simply that searching an entire candidate list takes a long time. We want to maximise the size of the jumps that HAT-DELTA can make, while minimising the time it takes to search.

HAT-DELTA currently uses a coarse-grained slicing mechanism, finding the function that has been evaluated, rather than the specific parts of that function. This slicing will be improved in future versions of HAT-DELTA, which will hopefully lead to an improvement in the heuristic's accuracy. There is still significant work to do in determining the best heuristics to use to provide a short search path in as many cases as possible.

6 Experimental Evaluation

Experiments on the initial implementation 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 actually rises slightly, while in others, the number is cut by a factor of ten.

Table 1 compares the number of questions asked by HAT-DETECT with the number asked by HAT-DELTA. We gave each program an input that caused it to exhibit erroneous behavior and ran the two tools on the trace file produced.

In these experiments we rely solely on 'yes' answers to give HAT-DELTA correct computations. In some cases HAT-DELTA provided significant reductions in the number of questions asked. The last row of the table shows that our heuristics can be wrong — we hypothesise that in this case, the heuristic did not have enough data to work on and hence made an error.

The program `addLists` takes a list of numbers, generates all sublists and pairs each sublist with the sum of its elements. `Fac . rev . take . fibs` generates the infinite list of fibonacci numbers, takes 5 of them, reverses the list and then takes the factorial of each element. `reverse . sort` first quick sorts a list, then reverses the result. `tautology` tests if a propositional logic statement is a tautology by partial evaluation and case analysis. `treeSort` performs a tree sort. `natInt` implements arithmetic in church numerals and conversion to and from integers. `primes` generates a list of prime numbers. `iSort` performs an insertion sort. We introduced a bug into each program.

Program	No of funs	HAT-DETECT	HAT-DELTA
`addLists`	5	11	6
`Fac . rev . take . fibs`	5	15	9
`reverse . sort`	4	6	5
`tautology`	6	11	10
`treeSort`	3	8	8
`natInt`	6	4	4
`primes`	3	3	3
`iSort`	2	6	7

Table 1. Number of questions asked using normal input

The results shown in Table 1 show HAT-DELTA operating in its worst possible environment — with only an erroneous input given. HAT-DELTA has no information to work with initially and indeed in the `primes` and `natInt` examples, the user never answers a question with yes. The lack of any yes answers means that HAT-DELTA never gets any additional data and as such makes no improvement over HAT-DETECT.

Program	No of funs	HAT-DETECT	HAT-DELTA
`addLists`	5	11	1
`iSort`	2	6	2
`Fac . rev . take . fibs`	5	13	6
`tautology`	6	11	5
`treeSort`	3	8	5
`reverse . sort`	4	8	5

Table 2. Number of questions asked where a correct input was given

HAT-DELTA was designed to work with more information than these results show — it was expected to have at least two inputs (at least one of which exhibits a correct behavior and one of which exhibits an erroneous behavior). Thus, we carried out a second series of experiments, on the same programs, giving them the two inputs required. The results of these experiments can be seen in Table 2. Not all the programs are present in this table, this is because we could not construct inputs for which the program worked in these cases. The results show a significant improvement in all cases, as the heuristics can now work better with more information.

We conjecture that with this technique, the number of questions asked is directly proportional to the number of functions in the program, rather than to the logarithm of the size of the trace (as with algorithmic debugging). This claim can be backed up by considering how the program slicing heuristic works. Once a function has been evaluated correctly, it is effectively removed from the debugging session, until other functions become less likely to be buggy. Thus, the debugger will eliminate functions from its search for the bug rather than eliminate specific evaluations as in algorithmic debugging.

We would expect that an improvement in the slicing technique will improve the accuracy of the slicing heuristic significantly. There are however several problems to consider with this approach. First, if the slices become too accurate, then effectively only the same application will match the same slice. This would lead to the heuristic never being applied. A balance must therefore be struck in how accurate slices become. Second, as slices become more accurate, it will become increasingly common that slices overlap, but do not match. Careful consideration must be given to what this means in terms of how the heuristic should be evaluated.

Further evaluation must be carried out. Each heuristic must be tested on its own and compared. We must compare multiple slicing methods and we must compare counting slices with simply marking slices as having been executed.

7 Combination With Other Views

We have looked at two heuristics that can improve 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) [1]. 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.

8 Related Work

Delta Debugging has been developed for imperative languages by Zeller and Cleve [2]. 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 is however the inspiration for our comparative debugger. In *Scalable Statistical Bug Isolation* [3], 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 behavior. There are clearly several methods of extracting information from multiple traces, more study is needed in this area to gain a clearer picture of how the methods fit together and what information is not yet being exploited.

9 Conclusion

Extra information can be gained from examining traces of programs that evaluate correctly. This information can be used to decrease the number of questions asked by an algorithmic debugger using the methods described in this paper. The gains provided by the heuristics can be extremely large, but in some situations can lead to an increase in the number of questions asked by a debugger. Implementation and Evaluation still needs to be done in order to find the conditions necessary for delta debugging to reduce the number of questions asked in a debugging session and to find the best heuristics to apply.

Acknowledgements

This work relies on previous work on the Haskell tracer Hat by Colin Runciman, Malcolm Wallace and Thorsten Brehm.

References

1. Olaf Chitil. Source-based trace exploration. In *Proceedings of the 16th International Workshop on Implementation of Functional Languages, IFL 2004*, LNCS 3474, pages 126–141. Technical Report 0408, University of Kiel, 2005.
2. Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. In *Proceedings of the Fourth International Workshop on Automated Debugging*, 2000.
3. Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005.
4. Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

5. Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Department of Computer and Information Science, Linköpings universitet, S-581 83, Linköping, Sweden, May 1998.
6. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, Cambridge, UK, April 2003.
7. Ehud Yehuda Shapiro. *Algorithmic program debugging.* MIT Press, 1982.
8. Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, UU-CS-2001-23. Universiteit Utrecht, 2001. Final proceedings to appear in ENTCS 59(2).