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

Algorithmic debugging is a semi-automatic method for locating bugs in programs. An algorithmic debugger asks a user a series of questions about the intended behaviour of the program. Here we present two new methods that reduces the number of questions a user must answer to locate a bug.
First, we describe a heuristic based on comparing computations of the same program with different inputs. Besides a computation that exhibits some erroneous behaviour, we use information from computations that produce correct results. The heuristic uses program slices to identify areas of code that are likely to be correct.
Secondly, we describe a method of compressing the search tree that guides the questions of an algorithmic debugger. This compression is particularly successful when used in combination with our heuristic.
Both heuristic and tree-compression are applicable to algorithmic debugging in general. We have implemented it for locating bugs in Haskell programs.

1 INTRODUCTION

Algorithmic debugging [5, 6, 7] is a semi-automatic method for locating bugs in programs that is based on the representation of a computation as a compositional tree, the *Evaluation Dependency Tree (EDT)*. Figure 2 shows the EDT for a computation of the Haskell program in Figure 1. Each node of the tree is labelled with a computation, that is, a big-step reduction of a redex to its value. The computation of a node is fully determined by the computations of all its children and the use of a small slice of the program.

For example, the computation `sort [2,1]` $\rightsquigarrow$ `[2]` is composed of the computations `insert 2 [1]` $\rightsquigarrow$ `[2]` and `sort [1]` $\rightsquigarrow$ `[1]` and the single-step reduction `sort [2,1]` $\rightsquigarrow$ `insert 2 (sort [1])`:

`sort [2,1]` $\rightsquigarrow$ `insert 2 (sort [1])` $\rightsquigarrow$ `insert 2 [1]` $\rightsquigarrow$ `[2]`

The composition of the computations of the child nodes and one single-step reduction disregards the actual evaluation order of expressions at run-time, but it follows the structure of the program.

The single-step reduction uses an instance of the slice

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

```
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 insertion sort program

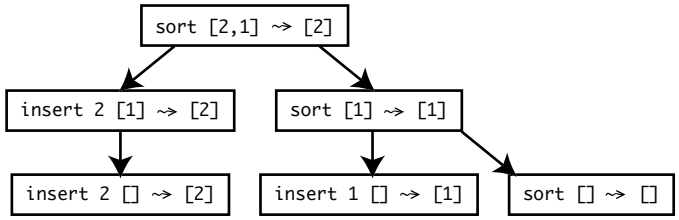

FIGURE 2. EDT of a computation of the buggy insertion sort program in Figure 1

of the program.

Another example is the computation `insert 1 [] ⇝ [1]`. The node has no children. The single step reduction only uses an instance of the program slice

```
insert x [] = [x]
```

In the same way every node in the EDT can be associated with the slice that the single step reduction is an instance of.

An algorithmic debugger asks the user whether computations of tree nodes agree with the intentions of the user. If the computation of a node agrees with the user's intentions, then the node is called *correct*, otherwise it is called *erroneous*. A node is *buggy*, if it is erroneous and all its children are correct. Because of the compositional definition of the EDT, the program slice associated with the buggy node contains a bug. That buggy program slice has to be modified to make the computation of the buggy node agree with the user's intentions.

An example session with an algorithmic debugger:

```
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
> yes                   node's child.
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 and its buggy slice is shown.

Bug identified:
  insert x (y:ys)
    | x < y     = ...
    | otherwise = insert x ys
```

The example shows that the slice may be only part of an equation. In this case the right hand side of the `x < y` guard is not included in the slice as it was not used in the single step reduction `insert 2 [1]` $\rightsquigarrow$ `[1]`.

The principle of algorithmic debugging does not specify any particular order in which questions are asked. Traditionally the EDT is traversed in a top down, left to right manner. Such a traversal establishes a path of erroneous nodes from the top node to a buggy node.

Algorithmic debugging has proven to work well in practice. However, for large computations it can ask a large number of questions. Many questions are difficult to answer and often questions are intuitively irrelevant. Hence here we present two methods for reducing the number of questions.

Both our methods are based on the fact that whereas the number of nodes of an EDT is proportional to the length of the computation, the number of possible slices associated with nodes is only proportional to the size of the program. Hence many nodes are associated with the same program slice. Our first method, a heuristic, determines a likelihood for the correctness or erroneousness of each program slice, and thus of each node of the tree. The second method is based on the observation that in the end we are interested in a buggy program slice, not a buggy node; so we can often collapse nodes with the same program slice into one node. Both methods modify the traditional traversal of an EDT such that a buggy program slice is usually reached with fewer questions.

Both methods are applicable to algorithmic debugging in general. We have implemented the methods in a debugger for Haskell programs. This implementation is based on the existing HAT tracing system.

2 HEURISTIC

Even programs that contain bugs often produce correct results for many inputs. Let us consider sorting the list `[1,2]` with our buggy program. The computation `sort [1,2]` $\rightsquigarrow$ `[1,2]` is correct. Let us combine the program slices of all

nodes of the EDT of this correct computation. Then let us separately combine the program slices of all nodes of the EDT of the erroneous computation `sort [2,1]` $\rightsquigarrow$ `[2]`. Finally we subtract the first slice from the latter, obtaining the following highlighted slice of the program:

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

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

This small program slice contains the bug. In general, localising a bug is not that easy, but the example highlights the importance of using program slices of correct computations as sources of information.

With each program slice we associate a likelihood that it is correct. Furthermore, every node in an EDT associated with this program slice then shall also have this likelihood to be correct.

2.1 Gathering Information

Using a second trace of a correct execution is one of three methods of retrieving data relating to program slices:

- The user may provide more than one trace to the debugger specifying for each whether it was successful or not, as shown in the example.
- The user may answer questions in the debugger, we then gather more information than a normal algorithmic debugger from this.
- The user may test their program with an automated testing tool and provide the trace of the testing session to the debugger.

We can treat each of these three cases in the same way. Each scenario provides EDTs associated with an indication of whether each EDT is correct, we can use this information to form our heuristics.

Each slice is associated with two values: the number of times a correct single step reduction has been an instance of the slice, and conversely the number of times an incorrect single step reduction has been an instance of the slice. When the user marks a computation as incorrect we calculate the slice that the top most single step reduction is an instance of, then increment its count of incorrect reductions.

Similarly, when the user marks a computation as correct, the count of correct reductions for the top level reduction's slice is incremented. Furthermore we calculate slices for all of the descendants of this reduction, and increment the counts

associated with those reductions. This can be done because it is likely that the entire reduction sequence of small step reductions was correct if the one big step was correct.

It is not always the case that all reductions in the subtree are correct. There are two situations where this property does not hold:

- When the program contains two buggy nodes, which cause the correct answer to be computed by fluke.
- When the program contains sharing, or is evaluated strictly. For example, the computation of the length of a list may evaluate correctly while the contents of the list are computed incorrectly. When the program is computed strictly, or the list is shared and demanded elsewhere in the program, the program trace will record this evaluation.

Although we may find occasional incorrect computations marked as correct, it is not a significant problem. We are using this data to calculate a heuristic, not make an authoritative decision. In the majority of cases the data helps us to find the bug faster. By including all descendants of correct reductions in our data we create an asymmetry. We cannot gather data from the descendants of erroneous reductions because the reduction being erroneous tells us nothing about the erroneousness of its children.

The process of calculating slices for every reduction in a computation can be time consuming. As such we limit the amount of data gathered. We currently simply take the first n reductions encountered in a top down, left to right depth first traversal of the EDT.

2.2 Using Information

When performing algorithmic debugging, the debugger follows a path of "no" answers until it finds a buggy node. The "yes" answers are used only to direct the debugger towards erroneous nodes. As such, we use the extra information the debugger gathers to guide it towards "no" answers. To do this we calculate a heuristic value. We then alter the order in which the debugger asks questions. Instead of taking a top-down left to right approach, we now jump to the node we think is most likely to be marked as erroneous. This gives us the ability to make jumps to nodes not in the direct children of a node, but in its grandchildren or even below.

This process requires calculating a heuristic, to provide a concrete value to base comparisons on. We have selected three simple heuristics which are compared in section 5:

- Negating the number of "yes" answers relating to the slice.
  If a slice has evaluated correctly before, then it will likely evaluate correctly again, so we should avoid the question.

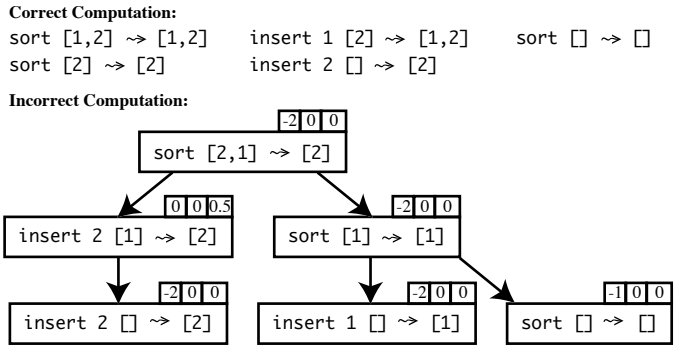

FIGURE 3. Sort algorithm after the user answers one question

- Counting the number of "no" answers relating to the slice.
  If part of the program has failed once, it is likely to fail again, we should keep asking about this slice.

- The proportion of questions about a slice that have been answered "no".
  This combines the other two heuristics, aiming to ask about slices that have evaluated correctly very few times and incorrectly many times. When we have no information about a slice, we regard it as equally likely to be erroneous or correct. Thus, the heuristic value becomes $\frac{1}{2}$.

Figure 3 shows the `sort` example examined earlier, along with the single step reductions from a correct computation of the same program. We have annotated the nodes of the EDT with the values of each heuristic we have studied: The left most negates the number of "yes" answers relating to the slice; the middle counts the number of "no" answers, and the rightmost counts the proportion of questions answered "no".

For example, the top most node is an instance of the slice `sort (x:xs) = insert x (sort xs)`, as are the correct reductions `sort [1,2]` $\rightsquigarrow$ `[1,2]` and `sort [2]` $\rightsquigarrow$ `[2]`. Because of this the node has the heuristic values: -2 for negating the number of correct computations of the slice; 0 for tracking the number of incorrect computations; and finally $0/2$ reductions have been incorrect. When we track correct reductions, the next question asked would be `insert 2 [1] = [2]?`, as it would be if we tracked the proportion of questions answered "no". The heuristic that tracks the number of "no" answers has not got any data to go on yet, and so would default to taking the top most node.

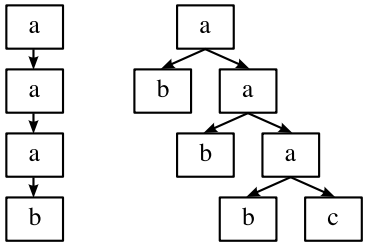

FIGURE 4. Common EDT structures for recursive programs

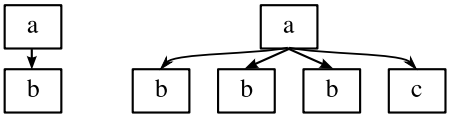

FIGURE 5. Compressed EDTs

3 TREE COMPRESSION

Recursive functions produce repetative patterns in EDTs. We present a method of compressing these frequently occurring patterns. Figure 4 shows two of the most common patterns. On the left is the pattern produced by a simple function that makes a recursive call. Each call uses the same program slice *a*. Finally, the function will call a base case in program slice *b*. When looking at this EDT we see that only two questions actually matter. If the user answers yes to the top level question about *a*, there is no bug, if the user answers "no", there is a bug in either slice *a*, or slice *b*. We can quickly find out if the bug is in slice *b* by asking the question for the bottom node in the tree. Thus we can collapse the tree, and remove the interleaved extra questions about *a*.

The second example shown in Figure 4 demonstrates the case where each call to the recursive function also calls a secondary function. In this case we may again collapse the spine of questions relating to slice *a*. However, no pattern can be discerned in the questions related to slice *b* and thus each of these questions must still be asked. These nodes in the EDT must be added to the top level question regarding slice *a*.

Applying this process in these two examples results in the two EDTs shown in Figure 5.

Using this second example, a normal algorithmic debugger may take a significant amount of time to identify a bug in slice *c* as it must ask several questions about slice *b* beforehand. This problem is neatly avoided however as our heuristic will cause the debugger to lower the likelihood of slice *b* being buggy as it asks

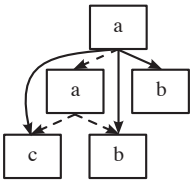

FIGURE 6. General Tree Compression

questions about it and gets yes answers. Thus this compression method works particularly well when combined with our heuristic driven search. As always the approaches effectiveness relies on having a good heuristic. Counting "no" answers for example will not allow us to skip to the question related to slice *c* in this example.

Trace compression can be applied in general to an EDT. Figure 6 shows how it is applied in the general case. For each node in the EDT we calculate a program slice. We then compare the node's slice with those of the node's children. We replace each child with a matching slice with its children (which are again recursively collapsed). This method often produces a large number nodes with similar questions as siblings. For example, if we collapse the EDT for a quick sort algorithm, we note that the slices for the recursive cases match, and hence collapse them to one node. This node has several children, each with the same evaluation `sort []` $\rightsquigarrow$ `[]`. As these questions are all literally the same we may combine them into one node. Thus, the EDT becomes two nodes — a parent asking about the recursive call, and a child asking about the base case.

In the context of a debugging session we can further generalise this algorithm. Each time the user says "yes" to a question, we can remove all occurrences of the question from the rest of the EDT. This further reduces the size of the EDT and hopefully the number of questions asked. This is a commonly used technique for reducing EDT size [6], but proves particularly effective in this situation.

4 IMPLEMENTATION

The two heuristics described in this paper have been implemented on top of the Haskell tracer HAT [1][9, 3]. HAT provides a framework on which our debugger is built.

4.1 Hat

Tracing a computation with HAT consists of two phases: trace generation and trace viewing. First, a special version of the program runs. In addition to its normal

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

input/output behaviour it writes a trace into a file. Second, after the program has terminated, the programmer studies the trace with a collection of viewing tools.

The trace as concrete data structure liberates the views from the time arrow of the computation. Most important for us is that we only have to implement another viewing tool and can just use the generated trace. An EDT can be reconstructed from the trace [9].

4.2 Algorithmic Debugging

HAT-DELTA was first implemented to perform standard algorithmic debugging before being extended to perform our heuristic and EDT compression. We implemented a three pass algorithm to get from a HAT trace file to our final output: The first pass reads the trace from file, and constructs a representation in memory with cyclic structures tagged. This is then converted into an EDT data structure. This EDT has several filters applied to remove unnecessary questions . For example, all nodes related to the IO monad are replaced with their children, as these questions ask the user no more than whether a function should produce some IO action. Questions relating to the prelude are removed, as are unevaluated computations. As our debugger is written in Haskell, the EDT is generated lazily, and so we only build those parts necessary to identify a bug, or ask the next question.

To implement algorithmic debugging HAT-DELTA 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 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.

Haskell is not a simple language and has features that can cause problems for an algorithmic debugger. Local function definitions cause a significant problem, as the user does not know the value of free variables. Given the following program an algorithmic debugger may ask the question `f 5 = 9?`.

```
g x = f 5
      where f y = x + y
```

In this situation, the user does not have enough information to answer the question. When local functions are asked about, our system provides a context with which the user can work, thus the above question would become:

```
f 5 = 9?
Within: g 4
```

Constant expressions also provide a problem, as they can be defined cyclically. This can create cycles in the EDT, and so cause the debugger to ask an infinite sequence of questions. We add an extra phase to the EDT's generation in which we detect cycles, and tag them. We then discover when we have reached a cycle, and notify the user of the position of the bug, along with an error message stating that it should perhaps not loop infinitely. When we tag cycles we keep track of all EDT nodes involved in the cycle, and thus can highlight errors in mutually recursive functions that loop infinitely.

4.3 Heuristic Based Debugging

To implement our heuristic based system, we have extended HAT-DELTA to do some extra work after the user answers a question. When the user answers 'no,' the slice associated with the computation is counted as a 'likely' buggy slices. When the user answers 'yes', HAT-DELTA must traverse the EDT collecting all sub-computations. All slices associated with the subtree are counted as 'unlikely' to be buggy.

The EDT candidate list must be maintained as a priority list based on the value of the heuristic associated with each EDT. When the debugger asks about the first item on the candidate list it will ask about the computation it thinks is most likely to be buggy.

This initial implementation allows HAT-DELTA to choose a child based on it's heuristic value, but does not allow the tool to make jumps through the EDT. An extended implementation maintains a priority list of nodes within all EDTs in the candidate list; this allows HAT-DELTA to jump to anywhere in the computation that is likely to be buggy. Calculating the heuristic value for every node in the EDT is however computationally expensive, and as such a cut-off depth is assigned, whereby HAT-DELTA will only investigate heuristic values n levels into the EDTs. The best value for this cut-off depth is still subject to experimental evaluation.

4.4 Quickcheck

Our implementation of HAT-DELTA supports using the QuickCheck [2, 3] test suite to provide test cases for debugging. QuickCheck automatically generates random test data to test 'properties' of the algorithm that the user has specified. This gives very wide ranging inputs and hence is likely to throw up both correct and incorrect computations. Using this data requires two alterations.

The QuickCheck library was modified so that rather than stopping as soon as it finds a counter example, the testing library runs its full compliment of tests. Carrying out all tests allows QuickCheck to provide much more data to HAT-DELTA and thus makes the heuristic more accurate.

HAT-DELTA required only minor modifications to allow it to recognise QuickCheck tests. The debugger then simply checks the value of the test result. If the value is False, the test is added to a list of erroneous evaluations to debug, and the fact that

this slice has been evaluated incorrectly is added to HAT-DELTA's pool of data. If the value is True, all the sub-computations are found and added to the list of likely correct slices. Once this process has been completed, HAT-DELTA may start debugging an erroneous test. In order to try and limit the number of questions asked, the smallest counter example is used.

5 EXPERIMENTAL EVALUATION

We have tested our implementation on a number of programs. Choosing tests for our evaluation has proven to be difficult, because of the level of familiarity the user must have with the program code. To effectively use an algorithmic debugger you must know the intended semantics of each function in the program.

We have evaluated the debugger using two middle sized programs. Our first program is a simple recursive decent parser for XML like structures (250 lines of code). Our second program provides conversion between types of data and units, by making traversals of a hyper-graph (363 lines of code). We also tested two slightly smaller programs: A program for predicting the winner of naughts and crosses games (103 lines of code, by Colin Runciman); and a program for converting to and from roman numerals (100 lines of code, by Malcolm Wallace). Finally we tested some trivial examples that demonstrate some common programming patterns. These include: Quick sorting a list and then reversing it; insertion sorting a list; testing if a predicate logic formula is a tautology; and calculating an approximation for the golden section.

We chose the larger examples because they were programs we both had a high degree of familiarity with. The middle sized programs were taken from the HAT team's database of programs for testing debuggers, they are used to test the effectiveness of other debugging tools, and as such test some more obscure cases. We chose the smaller programs simply to demonstrate the tools effectiveness at dealing with common programing patterns e.g. identifying bugs in recursive functions, or evaluators.

We introduced bugs into each of the programs, and performed tests to expose the bugs. In the two larger programs we have tried introducing different bugs in several places. We then debugged the programs using the tool in various modes:

- Detect mode — A mode that acts like a standard algorithmic debugger with none of our improvements active.
- Tracking yes answers — Avoiding questions related to slices the user has marked as correct.
- Tracking no answers — Seeking out questions related to slices the user has marked as incorrect.
- Tracking the proportion of no answers — Combining the above two strategies.

Test	Detect	Counting Yes		Counting No		Proportion	
Compression:		Off	On	Off	On	Off	On
XML – Bug 1	17	15	15	13	13	13	13
XML – Bug 2	17	9	9	13	13	13	13
Convert – Bug 1	9	9	9	9	9	9	9
Convert – Bug 2	11	9	8	13	12	9	8
Adjoxo	9	7	7	7	7	7	7
ToRoman	7	4	4	4	4	4	4
QSort & reverse	5	4	2	4	2	4	2
ISort	5	2	2	5	4	2	2
Tautology	7	6	5	6	5	8	7
Golden	7	7	4	7	4	7	4

TABLE 1. Number of questions asked by different debuggers

Test	Counting Yes	Counting No	Proportion
XML – Bug 1	10	9	7
XML – Bug 2	9	12	12

TABLE 2. Number of questions asked when using QuickCheck to test programs

- Each of the above heuristics using trace compression as well.
- Using each of the above strategies with QuickCheck to gather as much data as possible.

For each program mode we have counted the total number of questions asked by the debugger as a measure of the effectiveness of the method. Table 1 shows the number of questions each debugger asks to identify each bug. For each of the debuggers using a heuristic, two columns are shown. The left hand column shows the number of questions asked when trace compression is not used, while the right hand column shows trace compression enabled.

Table 2 shows the number of questions asked by each debugger when used in combination with QuickCheck. It is difficult to write suitable QuickCheck properties for many of our programs. As such this test was simply run on the XML parser.

From the results we can see clearly that using a heuristic can reduce the number of questions asked by the debugger, sometimes significantly. The choice of heuristic is significant. Counting the number of “no” answers is worse than both other heuristics in all but one case, and in one case is even worse than standard algorithmic debugging. More data would be needed to decide which of counting “yes” answers, and keeping track of the proportion of “no” answers is more effective. Current data seems to point to counting “yes” answers being more efficient,

but the results are not conclusive. Using testing suites to provide large amounts of test data improves the quality of all the heuristics.

6 RELATED WORK

Program slicing [8] is a generic term for extracting a section of a program that had an effect on a 'slicing criterion'. The first set of program slicing algorithms took a backward slice, i.e. the criterion set were of the form "what parts of the program had an effect on the value of this variable at this point?". The analysis was then working backwards from the variable to the beginning of the program to generate a slice. Forward slicing sets a criterion of the form "what parts of the program are effected by this variable". The analysis then proceeds forwards and collects all parts of the program the variable has an effect on. The first slicing algorithms were static, that is, they considered all possible computations of a program. A slicing algorithm is called dynamic, if it considers only a particular computation. So a dynamic slicing algorithm yields less general but more precise slices.

Program slicing based on algorithmic debugging is a non-traditional form of dynamic slicing. Every node of the EDT is associated with a small slice, and the combined slices of all nodes that are candidates for being buggy form the program slice which must contain a bug. HAT's source-based trace explorer performs this kind of program slicing [1].

Other strategies for traversing an EDT have been proposed. For example the divide and conquer approach attempts to ask questions that divide the tree evenly between what will be left to search if the user answers yes, or no. These strategies often work well for small examples, but struggle when large computations are involved. The divide and conquer technique for example requires the program to bring the entire EDT into memory, and count the number of nodes in it just to ask its first question.

In Section 3 we noted that when a user answers "yes" to a question, we may remove all other occurrences of the question from the EDT. In [4] we outlined a method of removing significant numbers of questions from an EDT by also removing nodes that matched any of the descendants of the correct node. Our premise for doing this was that situations where this is not correct occur vary rarely, however after experimental evaluation it was found that this could often lead to the debugger producing an incorrect result.

7 CONCLUSION

Using heuristics to guide an algorithmic debugger we can lower the number of questions it asks. We have tested three intuitively reasonable heuristics, each of which provided an improvement in most cases. In some cases the improvements were very significant. None of the heuristics showed a significant advantage over the others in general, although counting "no" answers does not appear to be very

effective. In specific cases however individual heuristics performed well. An investigation into what situations they perform well in should be carried out in the future.

Secondly we have presented a method of reducing the size of an EDT for debugging purposes. This works well when the bug is located in a simple recursive algorithm. The technique however provides little benefit when the bug is in program control code, or in a set of mutually recursive functions. The technique works particularly well in combination with our heuristic based approach as it needs a method of choosing a buggy child reasonably reliably.

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] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[3] Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. Testing and Tracing Lazy Functional Programs using QuickCheck and Hat. In *4th Summer School in Advanced Functional Programming*, number 2638 in LNCS, pages 59–99, Oxford, August 2003.

[4] Thomas Davie and Olaf Chitil. Correct computations direct debugging. In *Draft Proceedings of the 17th International Workshop on Implementation of Functional Languages, IFL 2005*, Dublin, Ireland, September 2005.

[5] Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[6] 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.

[7] Ehud Yehuda Shapiro. *Algorithmic program debugging*. MIT Press, 1982.

[8] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

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