DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN
UNIVERSIDAD POLITÉCNICA DE VALENCIA

P.O. Box: 22012 E-46071 Valencia (SPAIN)

Informe Técnico / Technical Report

Ref. No.: DSIC-II/09/06
Pages: 51

Title: Strategies for Algorithmic Debugging

Author(s): Josep Silva

Date: April 19, 2006
Keywords: Algorithmic Debugging, Strategies

Vº Bº
Leader of research Group Author(s)

Strategies for Algorithmic Debugging

1 Algorithmic Debugging

During the algorithmic debugging process, an oracle (typically the user) is prompted with equations and asked about their correctness; he answers "YES" when the result is correct, or "NO" when the result is wrong. Some algorithmic debuggers also accept the answer "I don't know" when the user cannot give an answer (maybe because the question is too complicated). After every question, some nodes of the ET leave the *suspicious area* which contains those nodes that can be buggy. When there is only one node in the suspicious area, the process finishes reporting this node as buggy. It should be clear that algorithmic debugging finds one bug at a time. In order to find different bugs, the process should be restarted once for each different bug.

Example 1. Consider the buggy program in Figure 1 (a). This program sums a list of integers ([1,2]) and computes the square of the result with three different methods. If the three methods compute the same result the program returns $True$, if not, it returns $False$. Here, one of the three methods—the one adding the partial sums of its input number—contains a bug. From this program, an algorithmic debugger can automatically generate the ET of Figure 1 (c) (for the time being, the reader can ignore the distinction between different shapes and white and dark nodes) which, in turn, can be used to produce a debugging session as depicted in Figure 1 (b). During the debugging session, the system asks the programmer about the correctness of some ET nodes w.r.t. the intended semantics. At the end of the debugging session, the algorithmic debugger determines that the bug of the program was located in function "$sum2$" (node 26). The definition of function "$sum2$" should be: "$sum2\ x = div\ (x * (decr\ x))\ 2$".

2 Strategies

2.1 Single Stepping:

The first algorithmic debugging strategy to be proposed was *Single Stepping* [9]. In essence, this strategy performs a bottom-up search because it proceeds by doing a post-order traversal of the ET. It asks first about all the children of a given node, and then (if they are correct) about the node itself. If the equation of this node is wrong then this is the buggy node; if it is correct, then the post-order traversal continues. In this strategy, all the nodes without children (the leafs) are asked first, then the nodes without grandchildren, and so on. In consequence, the first node answered "NO" is identified as buggy (because all its children have already been answered "YES").

For instance, the sequence of 16 questions asked for the ET in Figure 1 would be: 3-YES, 6-YES, 11-YES, 13-YES, 15-YES, 17-YES, 18-YES, 22-YES, 25-YES, 27-YES, 30-YES, 10-YES, 16-YES, 21-YES, 24-YES, 26-NO.

Note that in this strategy questions are semantically unconnected.

```
main = sqrtest [1,2]

sqrtest x = test (computs (arrsum x))

test (x,y,z) = (x==y) && (y==z)

arrsum [] = 0
arrsum (x:xs) = x + (arrsum xs)

computs x = ((comput1 x),(comput2 x),(comput3 x))

comput1 x = square x

square x = x*x

comput2 x = arrsum (list x x)

list x y | y==0      = []
         | otherwise = x:list x (y-1)

comput3 x = arrsum (partialsums x)

partialsums x = [(sum1 x),(sum2 x)]

sum1 x = div (x * (incr x)) 2
sum2 x = div (x + (decr x)) 2

incr x = x + 1
decr x = x - 1
```

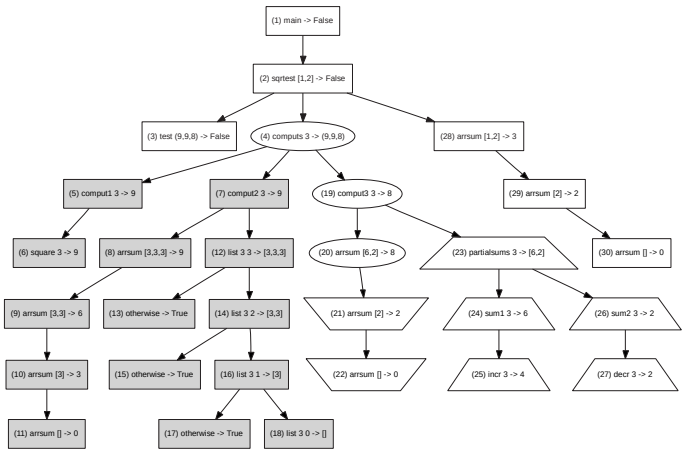
(a) Example program

```
Starting Debugging Session...

(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(3)  test [9,9,8] = False? YES
(4)  computs 3 = [9,9,8]? NO
(5)  comput1 3 = 9? YES
(7)  comput2 3 = 9? YES
(19) comput3 3 = 8? NO
(20) arrsum [6,2] = 8? YES
(23) partialsums 3 = [6,2]? NO
(24) sum1 3 = 6? YES
(26) sum2 3 = 2? NO
(27) decr 3 = 2? YES

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

(c) Debugging session for program (a)



(b) Execution tree of program (a)

Fig. 1. Example program (a) with its associated ET (b) and debugging session (c)

2.2 Top-Down Search:

Top-Down Search [5] is the strategy that has been traditionally used in many articles (see, e.g., [2]) to measure the performance of different debugging tools

and methods. It basically consists in a top-down, left-to-right traversal of the ET.

For instance, the sequence of 12 questions asked for the ET in Figure 1 (b) is shown in Figure 1 (c).

This strategy significantly improves Single Stepping because it cuts a branch of the ET every time it gets a "YES" answer. However, it is still very naive, since it does not take into account the structure of the tree (i.e. how balanced is it). For this reason, a number of variants trying to improve it can be found in the literature:

Top-Down Zooming: During the search in previous strategies, the rule or indeed the function definition may change query by query. If the oracle is human, this continuous change of function definitions slows down the answers of the user because he has to switch thinking once and again about different function definitions. This can be avoided by changing the order of the questions: In this strategy [7], recursive child calls are preferred.

The sequence of questions asked for the ET in Figure 1 (b) is exactly the same than with Top-Down Search (Figure 1 (c)) because no recursive calls are found. A difference between both strategies would appear when asking the children of node 12; while Top-Down Search would ask node 13 first, Top-Down Zooming would ask node 14, thus favoring the recursive call.

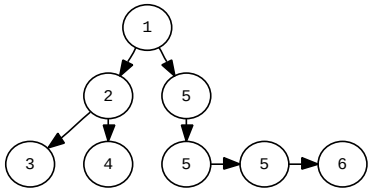
Another variant of this strategy called *Exception Zooming* recently introduced by Ian MacLarty [6] selects first those nodes that produced an exception at runtime.

Heaviest First: Selecting always the left most child does not take into account the size of the subtrees that can be explored. Blinks proposed in [1] a variant of the Top-Down Search in order to consider this information when selecting a child. This variant is called *Heaviest First* because it always selects the child with a bigger subtree. The objective is to avoid selecting small subtrees which have a lower probability to contain the bug.

For instance, the sequence of 9 questions asked for the ET in Figure 1 would be: 1-NO, 2-NO, 4-NO, 7-YES, 19-NO, 23-NO, 24-YES, 26-NO, 27-YES.

More Rules First: We introduce a new variant of Top-Down Search which tries to further improve Heaviest First. It is based on the fact that every equation in the ET is associated with a rule of the source code (i.e. the rule that the debugger identifies as buggy when it finds a buggy node in the ET). Taking into account that the final objective of the process is to find the program's rule which contains the bug—rather than a node in the ET—and considering that there is not a relation one-to-one between nodes and rules since several nodes can refer to the same rule, it seems to be logical to explore first those subtrees with a higher number of associated rules (instead of exploring those subtrees with a higher number of nodes).

Example 2. Consider the next ET:



where each node is labeled with its associated rule and where the oracle answered *NO* to the question in the root of the tree. While 'Heaviest First' would select the right child because this subtree has four nodes instead of three, 'More Rules First' would select the left child because this subtree contains three different rules instead of two.

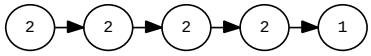
This strategy relays on the idea that all the rules have the same probability to contain the bug (rather than all the nodes). Another possibility could be to associate a different probability to contain the bug to each rule (e.g. depending on its structure: Is it recursive? Does it contain higher order calls? etc.).

Independently of the strategy followed, considering the rule associated to each node can help to avoid many unnecessary questions. Consider the following example:

Example 3. The tree associated to the '`append`' function definition:

```
(1)  append [] y = y (2)  append (x:xs) y = x:append xs y
```

for the call "`append [1,2,3,4] [5,6]`" is the following:



where each node is labeled with its associated rule. Here, the bug can be found with only one question to the node labeled "1". If the answer is "NO" the bug is in rule (1), if the answer is "YES" then we can stop the search because the rest of the nodes has the same associated rule (2) and, thus, the bug must necessarily be in this rule. Note that using this information can significantly improve the search: with a list of n numbers as the first argument, other strategies could need $n + 1$ questions to reach the bug.

With this strategy, the sequence of 8 questions asked for the ET in Figure 1 would be: 1-NO, 2-NO, 4-NO, 19-NO, 23-NO, 24-YES, 26-NO, 27-YES.

2.3 Divide & Query

Together with Single Stepping, Shapiro proposed another strategy which is optimal in the worst case: Divide & Query [9]. The idea of Divide & Query is to ask in every step a question which divides the remaining nodes in the ET by two. The main inconvenience of this strategy is that the sequence of questions are totally unconnected from a semantic point of view.

For instance, the sequence of 6 questions asked for the ET in Figure 1 would be: 7-YES, 19-NO, 23-NO, 24-YES, 26-NO, 27-YES.

Hirunkitti's Divide & Query: In [4], Hirunkitti noted that Shapiro's algorithm does not always choose the node closest to the halfway point in the tree and addressed this problem slightly modifying the original Divide & Query algorithm.

Divide by Rules & Query: The same idea discussed in the strategy *More Rules First* can be applied in order to improve D&Q. Instead of dividing the ET into two subtrees with similar number of nodes, we can divide it into two subtrees with similar number of rules. The objective is to eliminate from the suspicious area as much rules as possible with every question. With this strategy, the sequence of 5 questions asked for the ET in Figure 1 would be: 19-NO, 23-NO, 24-YES, 26-NO, 27-YES.

Biased Weighting Divide & Query: MacLarty proposed in his PhD thesis [6] that not all the nodes should be considered equally while dividing the tree. His variant to Divide & Query divides the tree by only considering some kinds of nodes and/or by associating a different weight to every kind of node.

In particular, their algorithmic debugger has been implemented for the functional logic language Mercury [3] which distinguishes between 13 different node types.

2.4 Hat-Delta

Hat [10] is a tracer for Haskell. Recently, Davie and Chitil introduced a declarative debugger tool based on the Hat traces that includes a new strategy called Hat-Delta. The main idea of this strategy is to use previous answers of the user in order to compute which node is more likely to be wrong. The debugger associates to each rule of the program the number of times it has been executed in correct computations based on previous answers.

Example 4. Consider this program:

```
(4)  sort [] = []
(8)  sort (x:xs) = insert x (sort xs)
(4)  insert x [] = [x]
(4)  insert x (y:ys)
(4)    | x<y       = x:y:ys
(0)    | otherwise = insert x ys
```

where the numbers at the left indicates the number of times rules have been executed correctly. Clearly, with this information, "otherwise = insert x ys" is more likely to be wrong.

2.5 Subterm Dependency Tracking:

In 1986, Pereira [8] noted that the answers "YES", "NO" and "I don't know" were insufficient; and he pointed out another possible answer of the user: "*Inadmissible*". An equation, or, more concretely, some of its arguments, are inadmissible if they violate the preconditions of its function definition. For instance,

consider the equation $insert\ 'b'$ "cc" = "bcc", where function $insert$ inserts the first argument in a list of mutually different characters (the second argument). This equation is not wrong but inadmissible, since the argument "cc" has repeated characters.

However, with only these four possible answers the system fails to get fundamental information from the user about *why* the equation is wrong or inadmissible. In particular, the user could specify which exact (sub)term in the result or the arguments is wrong or inadmissible respectively.

Consider for instance the equation from the previous example $insert\ 'b'$ "cc" = "bcc". Here, the programmer could detect that the second argument should not have been computed. In this case, the programmer could mark the second argument ("cc") as inadmissible. This information is essential because it allows the debugger to avoid questions related to the correct parts of the equation and concentrate on the wrong parts.

Based on this idea, MacLarty et al. [6] proposed a new strategy called *Subterm Dependency Tracking*. Essentially, once the user selected a particular wrong subterm, this strategy searches backwards in the computation for the node that introduced the wrong subterm. All the nodes traversed during the search define a *dependency chain* of nodes between the node that produced the wrong subterm and the node where the user identified it. The sequence of questions defined in this strategy follows the dependency chain from the origin of the wrong subterm.

For instance, if the user is asked question 3 from the ET in Figure 1 (b), his answer would be "YES" but he could also mark subexpression "8" as inadmissible. Then, the system would compute the chain of nodes which passed this subexpression from the node which computed it until question 3. This chain is formed by nodes 4, 19 and 20 which have an ellipse shape. The system would ask first 20, then 19, and finally 4 following the computed chain.

In our example, the sequence of 8 questions asked for the ET in Figure 1 (b), combining this strategy with Top-Down Search, would be: 1-NO, 2-NO, 3-YES (user marks "8"), 20-YES, 19-NO, 23-NO (user marks "2"), 26-NO, 27-YES.

2.6 Dynamic Weighting Search

Here, we propose a new strategy which takes advantage of the knowledge acquired from previous answers in order to formulate the next question. The main idea is that every node in the ET has a weight associated (representing the probability to be buggy), after every question, the debugger gets information that changes the weights and the debugger asks for the node with a bigger weight. When the associated weight of a node is 0, then this node leaves the suspicious area of the ET. The probabilities are modified based on the idea that those nodes of the tree which produced or manipulated a wrong (sub)term are more likely to be wrong than those that did not.

Computing weights from subterms: Firstly, as it happens in *Subterm Dependency Tracking* we allow the oracle to mark a subterm from an equation as wrong (instead of the whole node). This provides information about *why* an

equation is wrong (i.e. is the result incorrect? is one argument inadmissible?). Let us assume that the oracle marks a subterm "s" of a node "A" as wrong. Then, the suspicious area is automatically divided in four sets: The first set contains the node, say "B", that introduced "s" into the computation and all the nodes needed to evaluate the expression in node "B". The second set contains the nodes that, during the computation, passed the wrong subterm from equation to equation until node "A". The third set contains all the nodes which could have influenced the expression "s" in node "B" from the beginning of the computation. Finally, the rest of the nodes form the fourth set. Each subset has a different probability to contain the bug.

Example 5. Consider the ET in Figure 1 (b), where the oracle was asked about the correctness of equation 3 and he pointed out the computed subterm "`8`" as inadmissible. Then, the four sets are denoted in the figure by using different shapes and colors: (1) Those nodes which evaluated the equation 20 to produce the wrong subterm are denoted by an inverted trapezium. (2) Those nodes that passed the wrong subterm until the oracle detected it in the equation 3 are denoted by an ellipse. (3) Those nodes that produced the equation 20 which computed the wrong subterm are denoted by a white rectangle or a trapezium. (4) The rest of nodes are denoted by a grey rectangle.

The source of the wrong subterm is the equation which computed it. From our experience, all the nodes involved in the evaluation of this equation are more likely to contain the bug. However, it is also possible that the functions that passed this wrong term during the computation should have modify it and they didn't. In consequence, they could also contain the bug. Finally, it is also possible (but indeed less probable) that the equation that computed the wrong subterm had a wrong argument and it was the reason why it produced a wrong subterm. In this case, this inadmissible argument should be further inspected. In the example, the wrong term "`8`" was computed because equation 20 had a wrong argument "`[6,2]`" which should be "`[6,3]`"; the nodes which computed this wrong argument have a trapezium shape.

Consequently, in the previous example, after the oracle marked "`8`" as wrong in equation 3, we could increase the weight of the nodes in subset (1) with 3, the nodes in in subset (2) with 2, and the nodes in subset (3) with 1. The nodes in subset (4) can be extracted from the suspicious area because they couldn't influence the value of the wrong subterm, and, consequently, their probability to contain the bug is zero.

References

1. D. Binks. *Declarative Debugging in Gödel.* PhD thesis, University of Bristol, 1995.
2. R. Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming*, pages 8–13, New York, NY, USA, 2005. ACM Press.
3. T. Conway, F. Henderson, and Z. Somogyi. Code Generation for Mercury. In *International Logic Programming Symposium*, pages 242–256, 1995.

4. V. Hirunkitti and C. J. Hogger. A Generalised Query Minimisation for Program Debugging. In *AADEBUG, Springer Lecture Notes in Computer Science, vol 749*, pages 153–170, 1993.
5. J. W. Lloyd. Declarative error diagnosis. *New Gen. Comput.*, 5(2):133–154, 1987.
6. I. MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.
7. M. Maeji and T. Kanamori. Top-Down Zooming Diagnosis of Loic Programs. Technical Report TR-290, ICOT, 1987.
8. L. M. Pereira. Rational Debugging in Logic Programming. In *Proceedings on Third international conference on logic programming*, pages 203–210, New York, NY, USA, 1986. Springer-Verlag New York, Inc.
9. E.Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.
10. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-View Tracing for Haskell: a New Hat. In *Proc. of the 2001 ACM SIGPLAN Haskell Workshop*. Universiteit Utrecht UU-CS-2001-23, 2001.