A Survey on Algorithmic Debugging Strategies[☆]

Josep Silva

Universidad Politécnica de Valencia, DSIC
Camino de Vera s/n, CP 46022, Valencia, Spain

Abstract

Algorithmic debugging is a debugging technique that has been extended to practically all programming paradigms. Roughly speaking, the technique constructs an internal representation of all (sub)computations performed during the execution of a buggy program; and then, it asks the programmer about the correctness of such computations. The answers of the programmer guide the search for the bug until it is isolated by discarding correct parts of the program. After twenty years of research in algorithmic debugging many different techniques have appeared to improve the original proposal. Surprisingly, no study exists that joins together all these techniques and compares their advantages and their performance. This article presents a study that compares all current algorithmic debugging techniques and analyzes their differences and their costs. The research identifies the dimensions on which each strategy relies. This information allows us to combine the strong points of different strategies.

Keywords: Algorithmic debugging, software engineering

1. Introduction

In 1982, Ehud Saphiro presented in his masters thesis [1] a new debugging technique called *algorithmic debugging*. Algorithmic debugging was originally

[☆]This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant GVPRE/2008/001, and by the *Universidad Politécnica de Valencia* (Programs PAID-05-08 and PAID-06-08).

Email address: jsilva@dsic.upv.es (Josep Silva)
URL: http://www.dsic.upv.es/~jsilva (Josep Silva)

defined in the context of the logic paradigm, but it has been later extended and applied to practically all programming paradigms, and different implementations have evolved to produce mature debuggers for several languages. When a program's execution reveals a bug, the technique can automatically isolate a buggy portion of the source code by asking a series of questions to the programmer about computations performed during this execution. The answers of the programmer are used to discard those parts of the program that executed correctly, and thus, they do not caused the bug.

This article surveys more than two decades of work on algorithmic debugging, mainly focussing on the different search strategies used to improve the performance of algorithmic debugging. This work extends the conference paper "A Comparative Study of Algorithmic Debugging Strategies" [2]. This paper was limited in length, and this improved version contains more examples and explanations with a deeper analysis, and new material including an empirical evaluation of all the strategies. The article starts with an introduction to algorithmic debugging which is explained in a paradigm-independent way through the use of different examples. Then, each algorithmic debugging-based technique contained in the article is studied in a different section. In this way, algorithmic debugging experts can concentrate just on the discussions of the techniques they are interested in. All the techniques are explained with a common running example. This allows us to easily compare one technique with the others by observing the differences they produce with the same example and bug.

The article has been structured as follows: First, in Section 2 we introduce algorithmic debugging and the nomenclature that is used in the rest of the article. Next, in Section 3, we present an in-depth study of all search strategies for algorithmic debugging that has appeared in the literature so far. In this study, for the sake of clarity, we use a common example, and we produce an algorithmic debugging session to debug the same buggy program with each strategy. This illustrates the main differences between the strategies, and their strong and weak points. In Section 4 we compare all strategies according to the information they use and to their performance. In Section 5 we present an empirical evaluation of the strategies with a collection of benchmarks. Finally, Section 6 concludes.

2. Algorithmic Debugging

Algorithmic debugging (also called declarative debugging) is a semi-automatic debugging technique which is based on the answers of an oracle (typically the programmer) to a series of questions generated automatically by the algorithmic debugger. The questions are always whether a given result of an activation of a subcomputation with given input values is actually correct. The answers provide the debugger with information about the correctness of some (sub)computations of a given program; and the debugger uses them to guide the search for the bug until a buggy portion of code is isolated.

Example 2.1. *Consider this simple Haskell program inspired in a similar example by [3]. It wrongly (it has a bug) implements the sorting algorithm* Insertion Sort*:*

```
main = insort [2,1,3]

insort [] = []
insort (x:xs) = insert x (insort xs)

insert x [] = [x]
insert x (y:ys) = if x>=y then (x:y:ys)
                          else (y:(insert x ys))
```

An algorithmic debugging session for this program is the following (`YES` *and* `NO` *answers are provided by the oracle):*

```
Starting Debugging Session...
(1)  main = [2,3,1]? NO
(2)  insort [2,1,3] = [2,3,1]? NO
(3)  insort [1,3] = [3,1]? NO
(4)  insort [3] = [3]? YES
(5)  insert 1 [3] = [3,1]? NO
(6)  insert 1 [] = [1]? YES

Bug found in rule:
insert x (y:ys) = if x>=y then (x:y:ys)
                          else (y:(insert x ys))
```

The debugger points out the part of the code which contains the bug. In this case, `x>=y` *should be* `x<=y`*. Note that, to debug the program, the programmer only has to answer questions. It is not even necessary to see the code.*

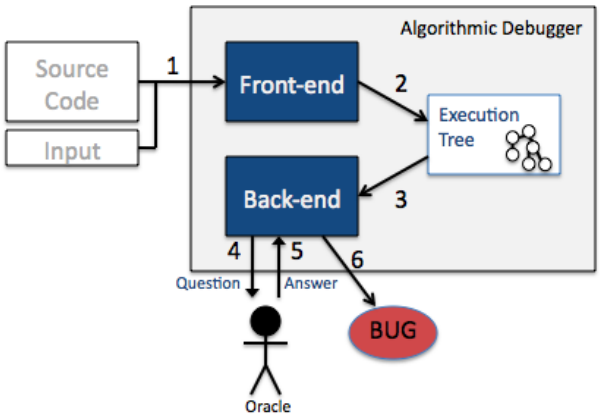


Figure 1: Architecture of an algorithmic debugger

Typically, algorithmic debuggers have a front-end which produces a data structure representing a program execution—the so-called *execution tree* (ET)[1] [4]—; and a back-end which uses the ET to ask questions and process the oracle's answers to locate the bug.

In Figure 1 we can observe the architecture of an algorithmic debugger. Clearly, the schema is very similar to the architecture of a compiler where (i) only the front-end handles the source code; (ii) the front-end produces an intermediate data structure (the ET) which is the input of the back-end; and (iii) the final result is only produced by the back-end. These properties allow one to use the same back-end to debug different languages by only replacing the front-end. An important difference between a compiler and an algorithmic debugger is that compilation is static while algorithmic debugging is a dynamic technique, i.e., the ET is generated for a particular execution with a given input (while a compiler is for source code without any execution). Sometimes, the front-end and the back-end are not independent (e.g., in Buddha [5] where the ET is generated in main memory when executing the back-end), or they are intertwiningly executed (e.g., in Freja [4] where the ET is built lazily while the back-end is running).

[1]Depending on the programming paradigm, the execution tree is called differently, e.g., *Proof Tree*, *Computation Tree*, *Evaluation Dependence Tree*, etc. We use ET to refer to any of them.

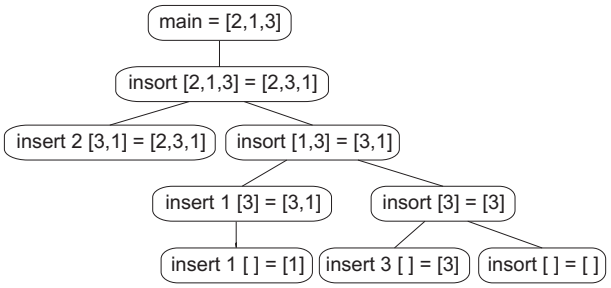


Figure 2: ET of the program in Example 2.1

Depending on the programming paradigm used (i.e., logic, functional, imperative...) the nodes of the ET contain different information: an atom or predicate that was proved in the computation (logic paradigm); or an equation which consists of a function call (functional or imperative paradigm), procedure call (imperative paradigm) or method invocation (object-oriented paradigm) with completely evaluated arguments and results[2], etc. The nodes can also contain additional information about the context of the question. For instance, they can contain constraints (constraint paradigm), attributes values (object-oriented paradigm), or global variables values (imperative paradigm). As we want to study debuggers from heterogeneous paradigms—in order to be general enough—we will simply refer to all the information in an ET's node as a *question*, and will avoid the atom/predicate/function/method/procedure distinction unless necessary.

Essentially, algorithmic debugging is a two-phase process: During the first phase, the ET is built. For instance, in the functional paradigm, the ET is constructed as follows: The root node is the *main* function of the program; for each node n with associated function f, and for each function call in the right-hand side of the definition of f which has been evaluated, a new node is recursively added to the ET as the child of n. As an example, the ET of the program in Example 2.1 is depicted in Figure 2.

This notion of ET is valid for functional languages but it is insufficient for other paradigms such as the imperative programming paradigm. In general, the information included in the nodes of the ET contains all the data needed to answer the questions. For instance, in the imperative programming paradigm, nodes include the function (or procedure) call, the output

[2]Or as much as needed if we consider a lazy language.

of this call, and the values of all parameters and global variables before and after the function was called. Similarly, in object-oriented languages, every node with a method invocation includes the values of the attributes of the object owner of this method (see, e.g., [6]).

Obviously, the information inside an ET node can be represented—and shown to the user—in many different ways. The most common way to represent the information inside the ET's nodes is through the use of equations whose left-hand side represents the call, and whose right-hand side shows the results produced by this call (see, e.g., Figure 2).

Example 2.2. *Consider the following Java program:*

```
Class Circle{
Point center;
float radius;

void init(){
   center = new Point(0,0);
   radius = 2;
}

float getArea(){
   float area = PI * (radius^2);
   return area;
}}
```

Consider also a circle object `C` *(*`Circle C = new Circle();`*) and the invocation of methods* `C.init(); C.getArea()`*. Then, the ET nodes that respectively represent the execution of* `C.init()` *and* `C.getArea()` *contain:*

```
{C.center = ⊥, C.radius = ⊥}
 C.init() -> ⊥
{C.center = (0,0), C.radius = 2}

{C.center = (0,0), C.radius = 2}
 C.getArea() -> 12.57
{C.center = (0,0), C.radius = 2}
```

where the first row represents the input values, the third row represents the output values, the result produced by the call is at the right of the arrow, and $\perp$ *represents an undefined value.*

Once the ET is built, in the second phase, the debugger uses a strategy to traverse the ET asking an oracle to answer each question. At the beginning, the *suspicious area* which contains those nodes that can be buggy (a buggy node is associated with a buggy source code of the program) is the whole ET; but, after every answer, some nodes of the ET leave the suspicious area. When all the children of a node with a wrong equation (if any) are correct, the node becomes buggy and the debugger locates the bug in the part of the program associated with this node [7]. If a bug symptom is detected then algorithmic debugging is complete [1]. It is important to say that, once the ET is built, the problem of traversing it and selecting a node to generate a question is mostly independent of the language used (a clear exception are those strategies which use information about the syntax of the program); hence many algorithmic debugging strategies can theoretically work for any language.

Unfortunately, in practice—for real programs—algorithmic debugging can produce long series of questions which are semantically unconnected (i.e., consecutive questions which refer to different and independent parts of the computation) making the process of debugging too complex for human users.

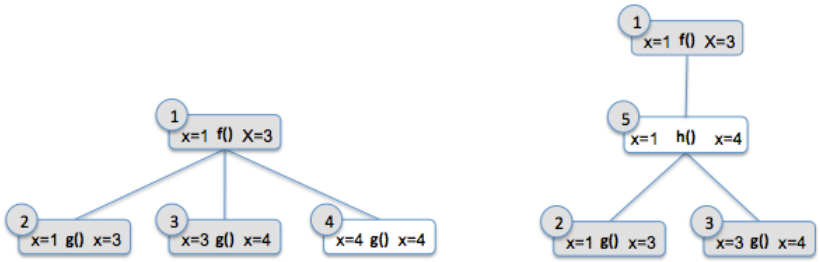
Furthermore, questions can also be very complex. For instance, during a debugging session with a compiler, the algorithmic debugger of the Mercury language [8] asked a question of more than 1400 lines.

Therefore, efficient techniques and strategies to reduce the number of questions, to simplify them and to improve the order in which they are asked have been proposed to make algorithmic debuggers usable in practice.

As shown in Example 2.1, during the algorithmic debugging process, an oracle is prompted with equations and asked about their correctness; it 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 oracle cannot give an answer (e.g., because the question is too complex). Answers are used to prune the ET. When there is only one node in the suspicious area, the process finishes reporting this node as buggy. It should be clear that, given a program with a wrong behavior, this technique guarantees that, whenever the oracle answers all the questions, a bug will eventually be found. It is important to highlight that soundness of algorithmic debugging ensures that a buggy portion of code is found; however, this portion of code could not be responsible for the buggy behavior. In particular, the buggy portion of code which is reported, is the source code associated with the method of the buggy node found. However, the buggy node found could not

be responsible of the buggy symptom of the program. Let us explain it with an example.

Example 2.3. *Consider the following ETs:*



If we assume the following intended semantics:

$x = 1$	$f()$	$x = 2$	$x = 1$	$g()$	$x = 4$
$x = 3$	$g()$	$x = 3$	$x = 4$	$g()$	$x = 4$
$x = 1$	$h()$	$x = 4$			

then grey nodes are wrong and white nodes are right.

These two ETs show two interesting (and rare) cases that could happen in a debugging session. In the ET at the left, we observe that two buggy nodes exists (nodes 2 and 3). In this ET, no matter what strategy we use, an algorithmic debugger will report node 2 or node 3 as buggy (the first one found by the used strategy). Therefore, the debugger will report the source code of function g as buggy. However, it could be possible that the wrong behavior observed in the root of the ET was caused by function f (even though g is clearly wrong). This phenomenon can be easily seen in the ET of the right, where nodes 2 and 3 are buggy, but their wrong behaviors in combination produced a correct global behavior by a fluke.

Algorithmic debugging finds one bug at a time. In order to find different bugs, the process should be restarted again for each different bug. Of course, once the first bug is removed, algorithmic debugging may be applied again in order to find another bug; and modern algorithmic debuggers keep a record of previous answers of the oracle to avoid the repetition of questions.

Let us illustrate the process with a slightly more complex example that we will use throughout the paper as our running example[3].

[3]While almost all the techniques and algorithmic debugging strategies presented here are independent of the programming paradigm used, in order to be concrete and w.l.o.g. we will base our examples on the functional programming paradigm.

```
main = sqrtest [1,2]

sqrtest x = test (computs (listsum x))

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

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

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

comput1 x = square x

square x = x*x

comput2 x = listsum (list x x)

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

comput3 x = listsum (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
```

Figure 3: Example program

Example 2.4. *Consider the buggy program in Figure 3 adapted to Haskell from [3]. 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* $True$*; otherwise, 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 5 (for the time being, the reader can ignore the distinction between different shapes and white and dark nodes) which, in*

```
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
(16) comput3 3 = 8? NO
(17) listsum [6,2] = 8? YES
(20) partialsums 3 = [6,2]? NO
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

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

Figure 4: Debugging session for the program in Figure 3

turn, can be used to produce a debugging session as depicted in Figure 4.[4] *During the debugging session, the system asks the oracle about the correctness of some ET's nodes w.r.t. the intended semantics. At the end of the debugging session, the algorithmic debugger determines that the bug of the program is located in function "*`sum2`*" (node 23). The definition of function "*`sum2`*" should be:* `sum2 x = div (x*(decr x)) 2`*.*

A debugging technique that is very related to algorithmic debugging is *abstract diagnosis* [9, 10, 11]. This technique relies on the same idea as algorithmic debugging but there are two important differences: (1) Abstract diagnosis assumes the existence of a (correct) specification of the program that is being debugged. This specification is used to decide the correctness of (sub)computations with a process of abstract interpretation that compares the program with the correct specification. (2) Abstract diagnosis does not need a bug symptom to detect errors in the program because they can be automatically detected by comparing the semantics of the program with the semantics of the reference specification. Even though they are different, abstract diagnosis can also benefit from the study of this article. In the case that

[4]The first question is often skipped by debuggers because it is associated with the root of the ET and thus it is assumed to be wrong.

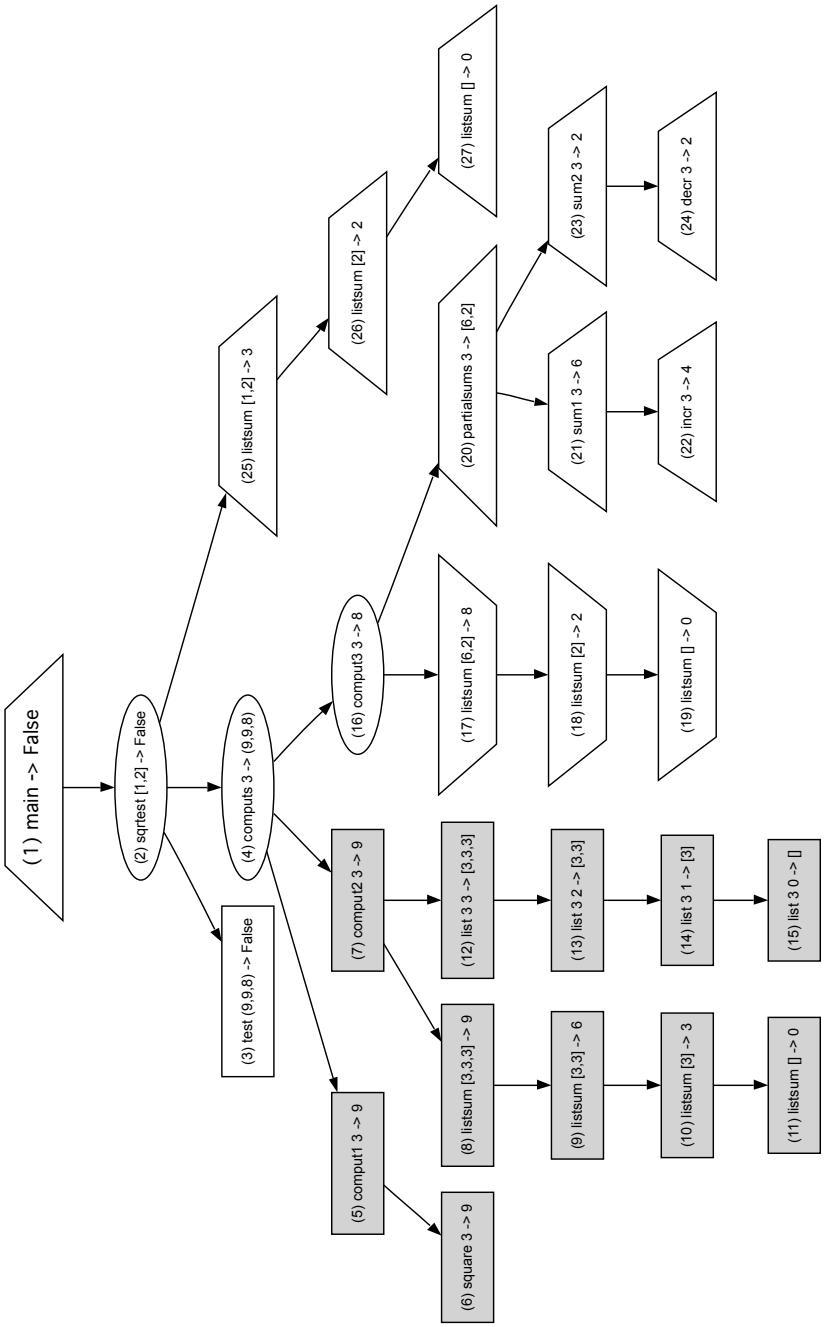


Figure 5: Execution tree of the program in Figure 3

we have available a correct specification, we could use the abstract diagnosis technique to (automatically) answer the questions selected by algorithmic debugging strategies, thus guiding abstract diagnosis by algorithmic debugging strategies.

3. Search Strategies for Algorithmic Debugging

One of the most important metrics to measure the performance of a debugger is the time spent to find the bug. In the case of algorithmic debuggers it is $q * t$ where q is the number of questions asked and t is the average time spent by the oracle to answer a question [2].

An algorithmic debugging strategy determines how to traverse the ET in order to select the next question to ask. Algorithmic debugging strategies are based on the fact that the ET can be pruned using the information provided by the oracle. Given a question associated with a node n of the ET, a NO answer prunes all the nodes of the ET except the subtree rooted at n; and a YES answer prunes the subtree rooted at n. Each strategy takes advantage of this property in a different manner.

A correct equation in the tree does not guarantee that the subtree rooted at this equation is free of errors. It can be the case that two buggy nodes caused the correct answer by chance [12]. In contrast, an incorrect equation does guarantee that the subtree rooted at this equation does contain a buggy node [13]. Therefore, if a program produced a wrong result, then the equation in the root of the ET is wrong and thus there must be at least one buggy node in the ET. We will assume in the following that the debugging session has been started after discovering a bug symptom in the output of the program, and thus the root of the tree contains a wrong equation. Hence, we know that there is at least one bug in the program. We will also assume that the oracle is able to answer all the questions. Then, all the strategies will find a bug.

Different strategies have arisen to minimize both the number of questions and the time needed to answer the questions. First, the number of questions can be reduced by pruning the ET (e.g., the strategy Divide and Query [1] prunes near half of the ET after every answer). Secondly, the time needed to answer the questions can be reduced by avoiding complex questions, or by producing a series of questions which are semantically related. For instance, the strategy Top-Down Zooming [14] tries to ask questions related to the same recursive (sub)computation.

Therefore, the effectiveness of the debugger is strongly dependent on the number, order, and complexity of the questions asked. Surprisingly, some algorithmic debuggers do not implement more than one strategy thus the programmer is forced to follow a rigid and predefined order of questions.

In the following, we enumerate and describe all the strategies that have appeared in the literature and we compare their cost, and study how to combine them.

3.0.1. Single Stepping (Shapiro, 1982)

The first algorithmic debugging strategy to be proposed was *single stepping* [1]. 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. Therefore, in this strategy, all the nodes without children (the leaves) are asked first, then their parents, and so on. Hence, the first node answered NO is identified as buggy (because all its children have already been answered YES).

For instance, the sequence of 19 questions asked for the ET in Figure 5 is:

```
Starting Debugging Session...

(3)  test [9,9,8] = False? YES        (7)  comput2 3 = 9? YES
(6)  square 3 = 9? YES                (19) listsum [] = 0? YES
(5)  comput1 3 = 9? YES               (18) listsum [2] = 2? YES
(11) listsum [] = 0? YES              (17) listsum [6,2] = 8? YES
(10) listsum [3] = 3? YES             (22) incr 3 = 4? YES
(9)  listsum [3,3] = 6? YES           (21) sum1 3 = 6? YES
(8)  listsum [3,3,3] = 9? YES         (24) decr 3 = 2? YES
(15) list 3 0 = []? YES               (23) sum2 3 = 2? NO
(14) list 3 1 = [3]? YES
(13) list 3 2 = [3,3]? YES             Bug found in rule:
(12) list 3 3 = [3,3,3]? YES           sum2 x = div (x + (decr x)) 2
```

Note that in this strategy questions are semantically unconnected (i.e., consecutive questions refer to independent parts of the computation).

3.0.2. Top-Down Search (Av-Ron, 1984)

Due to the fact that questions are asked in a logical order, *top-down search* [15] is the strategy that has been traditionally used (see, e.g., [16, 17]) to measure the performance of different debugging tools and methods. It basically performs a top-down, left-to-right traversal of the ET. In particular, when a node is answered NO, one of its children is asked; if it is answered YES, one of its siblings is, thus, the node asked is always a child or a sibling of the previous question node. Therefore, the idea is to follow the path of wrong equations from the root of the tree to the buggy node.

For instance, the sequence of 12 questions asked for the ET in Figure 5 is shown in Figure 4.

This strategy significantly improves single stepping because it prunes a part of the ET after every answer. However, it is still very naive, since it does not take into account the structure of the tree (e.g., how balanced it is). For this reason, a number of variants aiming at improving it can be found in the literature.

3.0.3. Top-Down Zooming (Maeji and Kanamori, 1987)

During the ET exploration of the previously presented strategies, the rule or indeed the function definition may change from one query to the next. If the oracle is human, this continuous change of function definitions slows down the answers of the programmer because she has to switch thinking once and again from one function definition to another. This drawback can be partially overcome by changing the order of the questions: In this version of top-down search, recursive child calls are preferred [14].

The sequence of questions asked for the ET in Figure 5 is exactly the same as with top-down search (Figure 4) because no recursive calls are found. A difference between both strategies can be observed in the ET of Figure 2. If the node with the question `insort [1,3] = [3,1]` is answered NO, then while top-down search would ask first the left child node (with question `insert 1 [3] = [3,1]`), top-down zooming would ask the right child node (with question `insort [3] = [3]`), thus favoring the recursive call.

Another variant of this strategy called *exception zooming*, introduced by Ian MacLarty [8], selects first those nodes that produced an exception at runtime.

3.0.4. Heaviest First (Binks, 1995)

Selecting always the left-most child does not take into account the size of the subtrees that can be explored. Binks proposed in [18] a variant of 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 of containing a bug.

For instance, the sequence of 9 questions asked for the ET in Figure 5 would be[5]:

```
Starting Debugging Session...

(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(4)  computs 3 = (9,9,8)? NO
(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(20) partialsums 3 = [6,2]? NO
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

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

3.0.5. Divide & Query (Shapiro, 1982)

In 1982, together with single stepping, Shapiro proposed another strategy: the so-called divide & query (D&Q) [1]. The idea of D&Q is to ask in every step a question which divides the remaining nodes in the ET by two, or, if this is not possible, into two parts with a weight as similar as possible. In particular, the original algorithm by Shapiro always chooses the heaviest node whose weight is less than or equal to $w/2$ where w is the weight of the suspicious area in the ET. This strategy has a worst case query complexity

[5]Here, and in the following, we will break the indeterminism by selecting the left-most node in the figures. For instance, the fourth question could be either `(7)` or `(16)` because both have a weight of 9. We selected `(7)` because it is on the left.

of order $b\ log_2\ n$ where b is the average branching factor of the tree and n its number of nodes.

This strategy works well with a large search space—this is normally the case of realistic programs—because its query complexity is proportional to the logarithm of the number of nodes in the tree. If the ET is big and unbalanced this strategy is better than top-down search [16]; however, the main drawback of this strategy is that successive questions may have no connection, from a semantic point of view, with each other. This is due to the fact that the questions asked are often related to different and independent parts of the computation; and thus, the programmer requires more time for answering the questions.

The performance of algorithmic debugging is strongly influenced by the structure of the ET. If the ET is wide (i.e., with a big branching factor), then algorithmic debugging needs much more questions than if the ET is deep. Compare for instance the extreme case of an ET with 100 nodes with a branching factor of 1, and an ET with 100 nodes and a branching factor of 100. In the first case D&Q cannot prune any node, while in the second case, half of the tree is pruned after every answer.

For instance, the sequence of 6 questions asked by D&Q for the ET in Figure 5 is:

```
Starting Debugging Session...

(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(17) listsum [6,2] = 8? YES
(21) sum1 3 = 6? YES
(24) decr 3 = 2? YES
(23) sum2 3 = 2? NO

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

First, question (7) is asked because initially $w = 27$, thus $w/2 = 13.5$; and nodes (7) and (16) are the heaviest nodes whose weight, 9, is less than or equal to 13.5. Node (7) was selected because it is on the left. Therefore, in the second step, $w = 18$, $w/2 = 9$ and node (16) is selected. Then, $w = 9$ and $w/2 = 4.5$. Therefore, node (17) with weight 3 is selected. Finally, nodes (21), (24) and (23) are asked following the same reasoning.

3.0.6. Hirunkitti's Divide & Query (Hirunkitti and Hogger, 1993)

In [19], Hirunkitti and Hogger 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 D&Q algorithm. Their version of D&Q is the same as the one by Shapiro except that their version always chooses a node which produces a least difference between:

- $w/2$ and the heaviest node whose weight is less than or equal to $w/2$
- $w/2$ and the lightest node whose weight is bigger than or equal to $w/2$

where w is the weight of the suspicious area in the execution tree.

For instance, the sequence of 6 questions asked for the ET in Figure 5 is:

```
Starting Debugging Session...

(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(20) partialsums 3 = [6,2]? NO
(21) sum1 3 = 6? YES
(24) decr 3 = 2? YES
(23) sum2 3 = 2? NO

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

Note that, contrarily to Divide & Query, after node (16) is asked, $w/2 = 9/2 = 4.5$, and the next node selected is (20) whose weight is 5 (closer to the midpoint than (17) that was selected by Divide & Query).

3.0.7. Biased Weighting Divide & Query (MacLarty, 2005)

MacLarty proposed in his Masters thesis [8] that not all the nodes should be considered equally while dividing the ET. His variant of D&Q divides the ET by only considering some kinds of nodes and/or by associating a different weight to every kind of node.

In particular, his algorithmic debugger was implemented for the functional logic language Mercury [20] which distinguishes between 13 different node types. This is due to the nature of the language. As it is a logic language, a call can produce one single solution, many different solutions, or

just fail. Hence, many different subtrees can be produced for a call node and the result of each subtree is indicated with special event nodes (*exit*, *fail*, etc.).

This idea could be also applied to, e.g., functional or imperative languages by considering a categorization of the nodes. For instance, to compute the size of a subtree, a recursive call could be associated with a weight 1.1, a higher-order operation could be associated with a weight 1.2, etc. This is equivalent to the assumption that these nodes have a higher probability to be buggy.

3.0.8. Hat Delta (Davie and Chitil, 2005)

Hat [21] is a tracer for Haskell. Davie and Chitil introduced a declarative debugger tool based on the Hat's traces that includes a new strategy called Hat Delta [22]. Initially, Hat Delta is identical to top-down search but it becomes different as the number of questions asked increases. The main idea of this strategy is to use previous answers of the oracle in order to compute which node has an associated rule that is more likely to be wrong (e.g., because it has been answered NO more times than the others).

Consider for instance that after 20 questions, nodes associated with rule "A" has been answered 10 times NO, while nodes associated with rule "B" has been answered 10 times YES. With this information, rule "A" is more likely to be wrong than rule "B".

During a debugging session, a sequence of questions, each of them related to a particular rule, is asked. In general, after every question, it is possible to compute the total number of questions asked for each rule, the total number of answers YES/NO, and the total number of nodes associated with this rule. Moreover, when a node is set correct or wrong, Hat Delta marks all the rules of its descendants as correctly or incorrectly executed respectively. This strategy uses all this information to select the next question. In particular, three different heuristics have been proposed based on this idea [12]:

- *Counting the number of YES answers.* If a rule has been executed correctly before, then it will likely execute correctly again. The debugger associates to each rule of the program the number of times it has been executed in correct computations based on previous answers.

- *Counting the number of NO answers.* This is analogous to the previous heuristic but collecting wrong computations.

- *Calculating the proportion of NO answers.* This is derived from the previous two heuristics. For a node with associated rule r we have:

$$\frac{number\ of\ answers\ NO\ for\ r}{number\ of\ answers\ NO/YES\ for\ r}$$

If r has not been asked before a value of $\frac{1}{2}$ is assigned.

Example 3.1. *Consider the following program where the numbers at the left indicate respectively the number of times each rule has been executed correctly, the number of times each rule has failed and the proportion of NO answers for this rule.*

```
4|0|0     sort [] = []
8|4|1/3   sort (x:xs) = insert x (sort xs)
4|0|0     insert x [] = [x]
          insert x (y:ys)
4|0|0       | x<y = x:y:ys
0|0|1/2     | otherwise = insert x ys
```

With this information, `otherwise = insert x ys` *is more likely to be wrong (it has a probability of* $\frac{1}{2}$ *to be wrong).*

3.0.9. Subterm Dependency Tracking (MacLarty et al., 2005)

In 1986, Pereira [23] noted that the answers "Yes", "No" and "I Don't Know" were insufficient; and he pointed out another possible answer of the programmer: *Inadmissible* (see also [24]). An equation, or more precisely, some of its arguments, are inadmissible if they violate the preconditions of its function definition. Hence, inadmissibility implies the assumption that all functions are partial with respect to all their arguments; thus, only some values of the domain of the arguments are valid. From a technical point of view, this problem could be solved with the use of (often complex) user-defined data types or with assertions with the preconditions of the functions. This would allow the debugger to automatically identify the inadmissible arguments. Nevertheless, this information is often missing in programs, thus, only the oracle can decide whether the preconditions of a function have been violated in a particular function call.

Example 3.2. *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, because the argument "*`cc`*" has repeated characters. From this equation we may conjecture (but not prove) that the type of the second argument is a string. Hence, "*`cc`*" is of the correct type but it does not satisfy the precondition of not having repeated characters. Therefore, inadmissibility allows us to identify errors in left-hand sides of equations.*

Inadmissibility allows the programmer to specify that some argument in the atom/predicate or function/method/procedure call associated with the question should not have been computed (i.e., it violates the preconditions of the atom/predicate/function/ method/procedure). Answering "Inadmissible" to a question redirects the search for the bug in a new direction related to the nodes which are responsible for the inadmissibility. That is, those nodes that could have influenced the inadmissible argument [25].

The concept of inadmissibility allows us to specify that the question itself is wrong. However, it does not allow us to say *why* the equation is wrong or inadmissible. In particular, the programmer could specify which exact (sub)term in the result or the arguments is wrong or inadmissible respectively. This provides specific information about *why* an equation is wrong (i.e., what part of the result is incorrect? is one particular argument inadmissible?).

Consider again the equation `insert 'b' "cc" = "bcc"` of Example 3.2. Here, the programmer could detect that the second argument should not have been computed; she could then 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 [8] proposed a new strategy called subterm dependency tracking. In this strategy, the programmer can mark subterms in the result and in the arguments—not necessarily inadmissible arguments—as wrong. Essentially, once the programmer selects 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 programmer 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 programmer is asked question 3 from the ET in Figure 5, his answer would be YES but she could also mark subexpression "8" as wrong because it should be the square of an integer number. 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, 16 and 17. The system would ask first 17, then 16, and finally 4 following the computed chain.

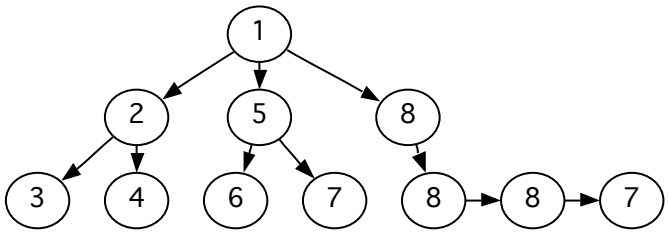
In our running example, the sequence of 8 questions asked for the ET in Figure 5, combining this strategy with top-down search, is:

```
Starting Debugging Session...

(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(3)  test (9,9,8) = False? YES  (the programmer marks ‘‘8")
(17) listsum [6,2] = 8? YES
(16) comput3 3 = 8? NO
(20) partialsums 3 = [6,2]? NO  (the programmer marks ‘‘2")
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

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

3.0.10. Less YES First (Silva, 2006)

Less YES First [26] is a variant of top-down search which further improves 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 a bug—rather than a node in the ET—and considering that there is not a one-to-one relation between nodes and rules because several nodes can refer to the same rule, it is important to also consider the node's rules during the search. A first idea could be to explore first those subtrees with a higher number of associated rules (instead of exploring those subtrees with a higher number of nodes).

Example 3.3. *Consider the following 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 selects the right-most child because this subtree has four nodes instead of three, less YES first selects the left-most child because this subtree contains three different rules instead of two.

Clearly, this approach relies on the idea that all the rules have the same probability of containing a bug (rather than all the nodes). Another possibility could be to associate a different probability of containing a bug to each rule, e.g., depending on its structure: Is it recursive? Does it contain higher-order calls?.

The probability of a node to be buggy is $p \cdot q$ where p is the probability that the node is wrong, and q is the probability that all the children of this node are correct. Therefore, the probability P of a branch b to contain a bug is

$$P = 1 - \prod_{i=1}^{n}(1 - (p_i \cdot q_i))$$

where n is the number of nodes in b.

Observe that we could think that $P = p_{root_b}$, where $root_b$ is the root of b; because if a node is wrong, then it or at least one of its descendants is buggy. But this is not true because a correct node can have incorrect descendants (see the discussion in Example 2.3). If we were able to approximate P we would be able to produce the most efficient variant of top-down search, because we would always select the subtree with the higher probability of containing the bug. However, approximating P is not easy. One could think that $p = r$ where r is the probability of the rule associated to the node of being wrong. But this is clearly insufficient because a wrong rule does not always executes incorrectly. Thus, we could extend the equation to $p = r \cdot s$ being s the probability that the buggy code of the rule is executed. Unfortunately, this

is still insufficient. The reason is that one rule could have two bugs and they both together could compensate their effects.

If we assume that a wrong rule always produces a wrong result and that all the rules have the same probability of being wrong, given a bug in a program, the probability of being in branch b is

$$\frac{r}{R}$$

where R is the number of rules in the program.

Hence, under the previous assumptions this strategy is (on average) better than heaviest first. For instance, in Example 3.3 the left-most branch has a probability of $\frac{3}{8}$ to contain a buggy node, while the right-most branch has a probability of $\frac{2}{8}$ even though it has more nodes.

However, in general, a wrong rule can produce a correct result, and thus it is important to also consider the probability of a wrong rule to return a wrong answer. This probability has been approximated by the debugger Hat Delta (see Section 3.0.8) by using previous answers of the oracle. The main idea is that a rule answered NO n times out of m is more likely to be wrong than a rule answered NO n' times out of m if $n' < n \leqslant m$.

Less YES First uses this idea in order to compute the probability of a branch to contain a buggy node. Hence, this strategy is a combination of the ideas from both heaviest first and Hat Delta. However, while heaviest first considers the structure of the tree and does not take into account previous answers of the oracle, Hat Delta does the opposite; thus, the advantage of less YES first over them is the use of more information (both the structure of the tree and previous answers of the oracle).

A direct generalization of Hat Delta for branches would result in counting the number of YES answers of a given branch; but this approach would not take into account the number of rules in the branch. In contrast, Less YES First proceeds as follows:

When a node is set correct, its associated rule and all the rules of its descendants are marked as correctly executed. If a rule has been executed correctly before, then it will likely execute correctly again. The debugger associates to each rule of the program the number of times it has been executed in correct subcomputations based on previous answers. Then, when we have to select a child to ask, we can compute the total number of rules in the subtrees rooted at the children, and the total number of answers YES for every rule.

This strategy selects the child whose subtree is less likely to be correct (and thus more likely to be wrong). To compute this probability we calculate for every branch b a weight w_b with the following equation:

$$w_b = \sum_{i=1}^{n} \frac{1}{r_i^{(YES)}}$$

where n is the number of nodes in b and $r_i^{(YES)}$ is the number of answers YES for the rule r of the node i.

As with heaviest first, we select the branch with the biggest weight, the difference is that this equation to compute the weight takes into account previous answers of the oracle. Moreover, we assume that initially all the rules have been answered YES once, and thus, at the beginning, this strategy asks those branches with more nodes, but it becomes different as the number of questions asked increases.

With this strategy, the sequence of 9 questions asked for the ET in Figure 5 is:

```
 Starting Debugging Session...

(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(4)  computs 3 = (9,9,8)? NO
(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(20) partialsums 3 = [6,2]? NO
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

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

3.0.11. Divide by YES & Query (Silva, 2006)

The same idea used in less YES first can be applied in order to improve D&Q. Instead of dividing the ET into two subtrees with a similar number of nodes, we can divide it into two subtrees with a similar weight. The problem that this strategy tries to address is the D&Q's assumption that all the nodes

have the same probability of containing a bug. In contrast, this strategy tries to compute this probability.

By using the equation to compute the weight of a branch (see Section 3.0.10), this strategy computes the weight associated with the subtree rooted at each node. Then, the node which divides the tree into two subtrees with a more similar weight is selected. In particular, the node selected is the node that produces a least difference between:

- $w/2$ and the heaviest node whose weight is less than or equal to $w/2$
- $w/2$ and the lightest node whose weight is bigger than or equal to $w/2$

where w is the weight of the suspicious area in the ET.

As with D&Q, different nodes could divide the ET into two subtrees with a similar weights; in this case, another strategy (e.g., Hirunkitti) can be used in order to select one of them.

We assume again that initially all the rules have been answered YES once. Therefore, at the beginning this strategy is similar to D&Q, but the differences appear as the number of answers increases.

Example 3.4. *Consider again the ET in Example 3.3. Similarly to D&Q, the first node selected is the top-most "8" because only structural information is available. Let us assume that the answer is YES. Then, we mark all the nodes in this branch as correctly executed. Therefore, the next node selected is "2"; because, despite the subtrees rooted at "2" and "5" have the same number of nodes and rules, we now have more information which allows us to know that the subtree rooted at "5" is more likely to be correct since node "7" has been correctly executed before.*

The main difference with respect to D&Q is that divide by YES & query not only takes into account the structure of the tree (i.e., the distribution of the program rules between its nodes), but also previous answers of the oracle.

With this strategy, the sequence of 5 questions asked for the ET in Figure 5 is:

```
Starting Debugging Session...

(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
```

```
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

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

3.0.12. Dynamic Weighting Search (Silva, 2006)

Subterm dependency tracking (see Section 3.0.9) relies on the idea that if a subterm is marked, then the error will likely be in the sequence of functions that produced and passed the incorrect subterm until the function where the programmer found it. However, the error could also be in any other equation previous to the origin of the dependency chain.

Silva [27] proposed a new strategy which is a generalization of subterm dependency tracking and which can integrate the knowledge acquired by other strategies in order to formulate the next question.

The main idea is that every node in the ET has an associated weight (representing the probability of being buggy). After every question, the debugger gets information that changes the weights and it asks for the node with a higher weight. When the associated weight of a node is 0, then this node leaves the suspicious area of the ET. Weights are modified based on the assumption 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. Here, we compute weights instead of probabilities without loss of generality.

Computing Weights from Subterms

First, as with subterm dependency tracking, the oracle is allowed to mark a subterm from an equation as wrong (instead of the whole equation). Let us assume that the programmer is being asked about the correctness of the equation in a node n_1, and she marks a subterm s as wrong (or inadmissible). Then, the suspicious area is automatically divided into four sets. The first set contains the node, say n_2, that introduced s into the computation and all the nodes needed to execute the equation in node n_2. The second set contains the nodes that, during the computation, passed the wrong subterm from equation to equation until node n_1. The third set contains all the nodes which could have influenced the expression s in node n_2 from the beginning of the computation. Finally, the rest of the nodes form the fourth set. Since these nodes could not produce the wrong subterm (because they could not have influenced it), the nodes in the fourth set are extracted from

the suspicious area and, thus, the new suspicious area is formed by the sets 1, 2 and 3.

Each subset can be assigned a different probability of containing a bug. Let us show it with an example.

Example 3.5. *Consider the ET in Figure 5, where the oracle was asked about the correctness of equation 3 and it pointed out the computed subterm "*`8`*" as inadmissible. Then, the four sets are denoted in the figure by using different shapes and colors:*

- ***Set 1:*** *those nodes which evaluated the equation 17 to produce the wrong subterm are denoted by an inverted trapezoid.*
- ***Set 2:*** *those nodes that passed the wrong subterm until the programmer detected it in the equation 3 are denoted by an ellipse.*
- ***Set 3:*** *those nodes that could influence the wrong subterm are denoted by a trapezoid.*
- ***Set 4:*** *the rest of nodes are denoted by a grey rectangle.*

The source of a 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 a bug. However, it is also possible that the functions that passed this wrong term during the computation should have modified it and they did not. Therefore, they could also contain a bug. Finally, it is also possible (but indeed less likely) that the equation that computed the wrong subterm had a wrong argument and this 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 17 had a wrong argument "`[6,2]`" which should be "`[6,3]`"; the nodes which computed this wrong argument have a trapezoid 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 the first subset with 3, the nodes in the second subset with 2, and the nodes in the third subset with 1. The nodes in the fourth subset can be extracted from the suspicious area because they could not influence the value of the wrong subterm and, consequently, their probability of containing a bug is zero.

Given an ET, with a node containing an expression e marked as wrong, roughly, each of the four subsets is computed as follows:

- The equations in ET which belong to the Set 1 are those which are in the subtree rooted at the node that produced e (the origin of e).

- Given two nodes $n, n' \in$ ET where the expression e appears in n, e does not appear in the parent of n and node n' is the origin of e, then, the nodes in ET which belong to the Set 2 are all the nodes which are in the path between n and n' except n and n'.

- The Set 4 contains all the nodes that could not influence the expression e, and thus, it can be computed following the approach of [28]. In particular, Silva and Chitil introduced an algorithm which extracts a slice with respect to the expression e which contains all the nodes that could be the cause of e. Therefore, all the nodes that do not belong to the slice form Set 4.

- Finally, Set 3 contains all the equations in ET that do not belong to the Sets 1, 2 and 4.

Computing Weights From Rules

The information about the correctness of nodes which is provided by the oracle during the debugging session can be used to dynamically change weights.

As in Hat Delta, the main idea is that a rule answered NO n times out of m is more likely to be wrong than a rule answered NO n' times out of m if $n' < n \leqslant m$. Given a node n with associated rule r, its weight w is computed from the probability p of n to be wrong—based on the questions history—and the length of its 95% confidence interval:

$$w = p - (\frac{1.96}{\sqrt{n}}\sqrt{p\ (1-p)})^2$$

$$p = \frac{number\ of\ answers\ NO\ for\ r}{n = number\ of\ answers\ NO/YES\ for\ r}$$

For instance, if the debugger has to select a child A, B or C; where A is related to a rule that has been answered three times YES out of three questions, B is related to a rule that has been answered three times NO out of three questions, and C is related to a rule that has been answered three times YES and three times NO out of six questions; then B is more likely to be wrong than C (because $w_B > w_C$) and, in turn, C than A (because $w_C > w_A$).

Hence, the debugger would ask first B and, with the new information, would recompute weights.

This strategy can integrate information used by other strategies in order to modify nodes' weights.

4. A Comparison of Search Strategies

A summary of the information used by all strategies is shown in Figure 6. The meaning of each column is the following:

- '*(Str)ucture*' is marked if the strategy takes into account the distribution of nodes (or rules) in the tree;
- '*(Rul)es*' is marked if the strategy considers the rules associated with nodes;
- '*(Sem)antics*' is marked if the strategy follows an order of semantically related questions. The more marks the more strong relation between questions;
- '*(Sub)expressions*' is marked if the strategy allows the user to mark subexpressions as wrong or inadmissible;
- '*(His)tory*' is marked if the strategy considers previous answers in order to select the next node to ask (besides cutting the tree);
- '*(Div)isible*' is marked if the strategy can work with a subset of the whole ET. ETs can be huge and thus, it is desirable not to explore the whole tree after every question. Some strategies allow us to only load a part of the tree at a time, thus significatively speeding up the internal processing of the ET; and hence, being much more scalable than other strategies that need to explore the whole ET before every question. For instance, top-down can load the nodes whose depth is less than d, and ask d questions before loading another part of the tree. Note, however, that some of the non-marked strategies could work with a subset of the whole ET if they where restricted. For instance, heaviest first could be restricted by simply limiting the search for the heaviest branch to the loaded nodes of the ET. Other strategies need more simplifications: less YES first or Hat Delta could be restricted by only marking as correctly

Strategy	Str.	Rul.	Sem.	Sub.	His.	Div.	Cost
Single Stepping	-	-	-	-	-	✓	n
Top-Down Search	-	-	✓	-	-	✓	$b \cdot d$
Top-Down Zooming	-	-	✓✓	-	-	✓	$b \cdot d$
Heaviest First	✓	-	✓	-	-	-	$b \cdot d$
Less YES First	✓	✓	✓	-	✓	-	$b \cdot d$
Divide & Query	✓	-	-	-	-	-	$b \cdot log_2 n$
Biased Weighting D&Q	✓	-	-	-	-	-	$b \cdot log_2 n$
Hirunkitti's D&Q	✓	-	-	-	-	-	$b \cdot log_2 n$
Divide by YES & Query	✓	✓	-	-	✓	-	$b \cdot d$
Hat Delta	-	✓	-	-	✓	-	n
Subterm Dependency Track.	-	-	✓✓✓	✓	-	-	n
Dynamic Weighting Search	✓	✓	-	✓	✓	-	n

Figure 6: Comparing algorithmic debugging strategies

executed the first d levels of descendants of a node answered YES; and then restricting the search for the heaviest branch (respectively node) to the loaded nodes of the ET. Finally,

- '*Cost*' represents the worst case query complexity of the strategy. Here, n represents the number of nodes in the ET, d its maximum depth and b its average branching factor.

If we analyze the information in Figure 6, we see that in seven strategies (single stepping, top-down, top-down zooming, heaviest first, divide & query, biased weighting D&Q and Hirunkitti's D&Q) selecting the next question strictly depends on the structure of the ET (i.e., it can be determined by only analyzing the ET). In contrast, the cost of less YES first, divide by YES & query, Hat Delta, subterm dependency tracking and dynamic weighting search also depends on the information provided by the user (previous answers and/or subterms marked by the user).

The cost of single stepping is too expensive. Its worst case query complexity is order n, and its average cost is $n/2$.

Top-down and its variants have a cost of $b \cdot d$ which is significantly lower than the one of single stepping. The improvement of top-down zooming over top-down is based on the time needed by the programmer to answer the questions; their query complexity is the same.

In contrast, while in the worst case the costs of top-down and heaviest first are equal, in the mean case heaviest first shows an improvement over top-down. In particular, on average, for each wrong node with b children $s_i, 1 \leq i \leq b$:

- Top-down asks $\frac{b+1}{2}$ of the children.
- Heaviest first asks $\frac{\sum_{i=1}^{b} weight(s_i) \cdot pos(s_i)}{\sum_{i=1}^{b} weight(s_i)}$ of the children.

  where function *weight* computes the weight of a node and function *pos* computes the position of a node in a list containing it and all its brothers which is in descending order by their weights.

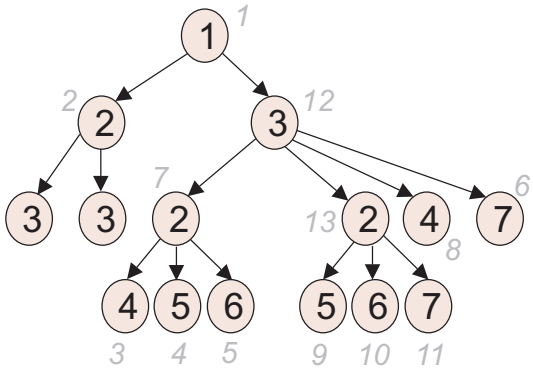
In the case of less YES first, the improvement is based on the fact that the heaviest branch is not always the branch with a higher probability of containing the buggy node.

D&Q has a cost order of $b \cdot (log_2\ n)$ in the worst case.

The cost of the rest of the strategies is highly influenced by the answers of the oracle.

The worst case of Hat Delta happens when the branching factor is 1 and the buggy node is in the leaf of the ET. In this case the cost is n. However, in normal situations, when the ET is wide, the worst case is still close to n; and it occurs when the first branch explored is answered NO, and the information of NO answers obtained makes the algorithmic debugger explore the rest of the ET bottom up. It can be seen in Example 4.1.

Example 4.1. *Consider the following ET where each node is labeled with its associated rule. If we use the version of Hat Delta which counts correct computations, then, when the oracle answers YES to a node, all its subnodes are marked as correctly executed once. This information could make the debugger ask more questions than with top-down. In particular, a possible list of questions asked is specified by marking each node with its position in the list. This produces the following session: 1.- NO, 2.- YES, 3.- YES, 4.- YES, 5.- YES, 6.- YES, 7.- YES, 8.- YES, 9.- YES, 10.- YES, 11.- YES, 12.- NO, 13.- YES.*

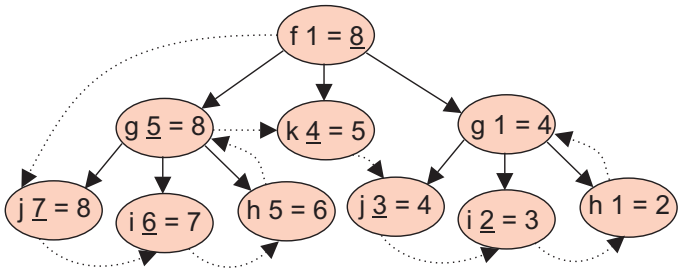


Even though subterm dependency tracking is a top-down search version enriched with additional information provided by the oracle, this information—that we assume correct here to compute the costs—could make the algorithmic debugger ask more questions than with the standard top-down search. In fact, this strategy—and also dynamic weighting search if we assume that top-down search is used by default—has a worst case query complexity of n because the expressions marked by the programmer can make the algorithmic debugger explore the whole ET. The next example shows the path of nodes asked by the algorithmic debugger in the worst case.

Example 4.2. *Consider the following program:*

```
f x = g (k (g x))

g x = j (i (h x))

h x = x+1

i x = x+1

j x = x+1

k x = x+1
```

whose ET w.r.t. the call `f 1` *is:*

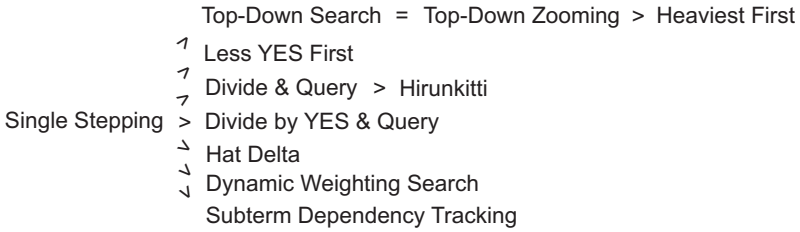


Here, the sequence of nodes asked with subterm dependency tracking is depicted with dotted arrows and the expressions marked by the programmer as wrong or inadmissible are underlined. Then, the debugging session is:

```
Starting Debugging Session...

f 1 = 8? NO  (The user selects "8")
j 7 = 8? YES (The user selects "7")
i 6 = 7? YES (The user selects "6")
h 5 = 6? YES
g 5 = 8? YES (The user selects "5")
k 4 = 5? YES (The user selects "4")
j 3 = 4? YES (The user selects "3")
i 2 = 3? YES (The user selects "2")
h 1 = 2? YES
g 1 = 4? NO

Bug found in rule:
g x = j (i (h x))
```

And, thus, all the ET's nodes are asked.

As a summary, the following diagram shows the relation between the costs of the strategies discussed so far:



This diagram together with the costs associated with every strategy in Figure 6 allows algorithmic debuggers to automatically determine which strategy to use depending on the structure of the ET. This means that the algorithmic debugger can dynamically change the strategy used during a debugging session.

Following this idea we have defined a hybrid strategy that combines heaviest first and Hirunkitti's D&Q. The new strategy is implemented by Algorithm 1, where some functions are used whose name is self-explaining. Functions *size* and *depth* compute respectively the number of nodes and the maximum depth of an ET. Functions $HeaviestFirst$ and $Hirunkitti$ select a node of an ET using respectively the strategies heaviest first and D&Q by Hirunkitti. Function $askNode$ ask to the oracle the question associated with a given node, and it returns the answer given by the oracle. Finally, function *prune* uses the answer of the oracle to remove from the ET all those nodes that cannot be buggy.

Algorithm 1 An hybrid strategy for algorithmic debugging

Input: An ET T whose root is a wrong node
Output: A buggy node

begin

(1) **repeat**
(2) n = size(T)
(3) d = depth(T)
(4) node = **if** $d < log_2 n$
(5) **then** HeaviestFirst(T)
(6) **else** Hirunkitti(T)
(7) answer = askNode(node)
(8) T = prune(T,answer)
(9) **until** T reveals a buggy node b
(10) **return** b

end

In the algorithm, after every question the new maximum depth (d) of the ET and its remaining number of nodes (n) is computed; if $d < log_2 n$ then heaviest first is used in order to select a new node, else Hirunkitti is applied.

5. Empirical Evaluation of Algorithmic Debugging Strategies

The comparison of algorithmic debugging strategies presented in the previous section, must be empirically validated in order to determine precisely how the strategies work in practice. For instance, the comparison concluded that Hirunkitti's version of D&Q is better than Shapiro's version. But the question is *How much better?* Similarly, after having read Section 4, we now know that Heaviest First is better than Top-Down; but, again, *How much better? Is the difference significant in practice?*

We conducted several experiments to answer these questions. Basically, the experiments were done by debugging several programs with all the strategies in order to know which strategies work better in practice (as an average). The main problem was that no debugger implemented all the strategies—in fact, the most advanced debugger was Mercury's debugger[6] [8] which only implemented four strategies—; and also that there was no maintained implementation of some strategies such as Divide by YES & Query.

Therefore, we had to implement all the strategies in order to ensure that all of them can be applied to the same target language, and hence the results are comparable. We implemented the strategies in the JDD debugger[7] [29] which is a Java debugger; and we selected 29 Java benchmarks to perform the experiments. Some benchmarks are Java programs implemented by university students; others are small to medium Java benchmarks including games, mathematical transformations, etc; and others are real Java applications such as a parser or a library. All the information related to this experiment, together with the source code of the benchmarks is publicly available at:

http://users.dsic.upv.es/~jsilva/DDJ/#Experiments

The results of the experiment are shown in Figure 7, where the first column contains the names of the benchmarks; column nodes shows the number of nodes in the ET associated with each benchmark; and the other columns represent algorithmic debugging strategies. The strategies are ordered according to their performance: The hybrid strategy between Heaviest First and Divide & Query by Hirunkitti implemented by Algorithm 1 (HF+D&QH), Divide & Query by Hirunkitti (D&QH), Divide & Query by

[6] http://www.mercury.csse.unimelb.edu.au
[7] http://users.dsic.upv.es/~jsilva/DDJ

Benchmark	Nodes	HF+D&QH	D&QH	D&QS	DR&Q	HF	MRF	TD	HD-P	HD-Y	HD-N	SS	Average
NumReader	*12*	28,99	28,99	31,36	29,59	44,38	44,38	49,70	49,70	49,70	49,70	53,25	41,80
Orderings	*46*	12,09	12,09	12,63	14,40	17,16	17,29	20,82	21,05	20,60	19,60	51,02	19,89
Factoricer	*62*	9,83	9,83	9,93	20,03	12,55	12,55	12,55	15,04	15,04	18,29	50,77	16,94
Sedgewick	*12*	30,77	30,77	33,14	30,77	34,91	34,91	43,20	43,79	43,79	43,79	53,25	38,46
Clasifier	*23*	20,31	20,31	22,40	21,88	22,92	23,26	31,94	32,12	32,12	34,55	51,91	28,52
LegendGame	*71*	8,87	8,87	8,95	16,72	11,15	11,23	13,37	14,68	14,68	16,94	50,68	16,01
Cues	*18*	32,69	32,41	32,41	32,41	33,24	34,63	42,11	39,06	39,06	44,32	52,35	37,70
Romanic	*123*	9,85	10,84	11,23	13,56	7,44	11,88	13,41	13,29	13,29	13,30	50,40	15,32
FibRecursive	*4.619*	0,27	0,27	0,28	1,20	0,33	0,41	0,46	3,92	3,92	0,48	50,01	5,59
Risk	*33*	16,78	16,78	18,08	19,38	18,69	18,69	31,14	24,31	24,31	32,79	51,38	24,76
FactTrans	*198*	3,89	3,89	3,93	6,22	6,58	6,58	7,16	7,37	7,24	7,50	50,25	10,06
RndQuicksort	*72*	8,73	8,73	8,73	11,41	12,03	12,23	13,51	13,62	12,93	14,54	50,67	15,19
BinaryArrays	*128*	5,52	5,52	5,71	7,13	7,75	7,94	8,59	7,90	8,15	8,71	50,38	11,21
FibFactAna	*351*	2,45	2,44	2,45	5,38	7,61	7,71	8,57	6,40	7,39	5,99	50,14	9,68
NewtonPol	*7*	39,06	39,06	43,75	39,06	43,75	43,75	45,31	45,31	45,31	45,31	54,69	44,03
RegresionTest	*18*	23,27	23,27	25,21	25,21	26,87	26,87	32,96	32,96	32,96	32,96	52,35	30,45
BoubleFibArrays	*171*	4,41	4,41	4,57	11,40	5,95	6,96	6,96	24,50	24,87	6,96	50,29	13,75
ComplexNumbers	*60*	10,02	10,02	10,32	11,31	11,39	11,39	15,75	15,78	15,80	19,19	50,79	16,53
StatsMeanFib	*68*	9,05	9,05	9,41	11,05	13,04	13,38	16,15	22,01	20,84	17,27	50,70	17,45
Integral	*5*	44,44	44,44	47,22	44,44	50,00	50,00	50,00	50,00	50,00	50,00	55,56	48,74
TestMath	*48*	11,91	11,91	12,16	12,99	15,95	16,28	24,20	22,41	23,87	22,37	50,98	20,46
TestMath2	*228*	3,51	3,51	3,51	9,73	10,55	10,81	28,56	12,29	13,24	14,37	50,22	14,57
Figures	*113*	6,76	6,75	6,79	8,09	7,68	7,79	10,60	10,17	10,16	10,76	50,43	12,36
FactCalc	*59*	10,14	10,14	10,42	11,53	13,69	14,22	18,50	20,47	20,47	20,69	50,81	18,28
SpaceLimits	*127*	16,00	16,07	19,15	21,74	13,68	16,80	22,78	22,87	22,86	26,15	50,38	22,59
Argparser	*129*	12,34	12,10	13,08	20,48	13,07	13,32	15,98	15,98	15,98	15,98	50,38	18,06
Cglib	*1.216*	1,94	1,93	2,33	2,12	2,52	2,65	6,61	6,14	5,73	7,32	50,04	8,12
Kxml2	*1.172*	2,91	2,86	3,01	3,56	3,06	3,48	6,79	8,58	6,97	7,77	50,04	9,00
Javassist	*1.357*	4,34	4,34	5,44	4,49	4,74	4,75	5,86	6,20	9,26	6,06	50,04	9,59
Average	**363,66**	**13,49**	**13,50**	**14,40**	**16,11**	**16,30**	**16,76**	**20,81**	**20,96**	**21,05**	**21,16**	**51,18**	**20,52**

Figure 7: Performance of algorithmic debugging strategies

Shapiro (D&QS), Divide by Rules & Query (DR&Q), Heaviest First (HF), More Rules First (MRF), Top-Down (TD), Hat Delta Proportion (HD-P), Hat Delta YES (HD-Y), Hat Delta NO (HD-N), Single Stepping (SS).

For each benchmark, we produced its associated ET and assumed that the buggy node could be any node of the ET (i.e., any subcomputation in the execution of the program could be buggy). Therefore, we performed a different experiment for each possible case and, hence, each row of the table summarizes a number of experiments. In particular, benchmark *Factoricer* has been debugged 62 times with each strategy; each time, the buggy node was a different node, and the results shown are the average proportion of nodes asked by each strategy (i.e., results have been normalized with respect to the size of the ETs). Similarly, benchmark *Cglib* has been debugged 1216 times with each strategy, and so on.

The results of the comparison provide interesting information related to the strategies which confirms the previous analysis; but they also offer additional information that complements it. First, the experiments confirm that single stepping is not usable in practice. It needs many more questions than any other strategy for larger programs. Top-Down, which is the most

used strategy is also not a good option. For instance, in the experiments it needs almost 7% more questions than Divide & Query by Hirunkitti. The experiments confirm that hirunkitti's Divide & Query is slightly better than Shapiro's Divide & Query: On average, Shapiro's version needs 1% more questions than Hirunkitti's version. Observe that in many experiments they are equivalent. These means that the situation where Hirunkitti's version behaves better is not very frequent. Not all variants of Top-Down improved it. In the case of Hat Delta heuristics, we see that it is better to prune the tree by using the proportion, probably because it takes into account both YES and NO answers. In addition, we see that counting YES answers is better than counting NO answers. This result confirms the same result obtained by Davie and Chitil (2006) when they compared these Hat Delta heuristics. Heaviest First performs around 4,5% less questions than Top-Down. The idea of using rules to prune the ET turns out to be useful when it is combined with Top-Down (4% less questions). However, it is better to divide the tree by nodes than by rules (D&Q by rules performs on average 3% more questions than D&Q by Hirunkitti). The new hybrid strategy presents the best results, but they are almost equivalent to the results obtained by D&Q by Hirunkitti. The reason is that in almost all cases $d >= log_2 n$, and thus, the normal situation is that the hybrid strategy behaves as D&Q by Hirunkitti.

6. Conclusions and Future Work

This work offers an overview of more than twenty years of research in algorithmic debugging. Since the first technique introduced by Shapiro, many new search strategies and improvements have appeared in the literature. This work has established the relations between all these works, and has compared and ordered them from a theoretical perspective.

The classification and comparison of algorithmic debugging strategies has shown the dimensions which are taken into account by each strategy. This is a useful information for the definition of new strategies; and also for their combination in order to join together the strong points of different strategies.

Each strategy proposed in the literature or implemented in mature debuggers has been studied separately by analyzing its strong and weak points. In addition, all of them have been used to debug the same example with the same bug. This is very useful to be able to see their behavior in the same context; because each strategy comes from a different language or even

paradigm, and this is the first time that all of them are studied and compared together.

The study of costs is a useful tool for the implementation of future strategies. It allows us to define dynamic strategies that change during a debugging session according to the changes in the structure of the ET. Thanks to the parameterized costs, it is possible to determine what strategy is (in the worst case) more efficient given the size, depth, and other parameters of the ET. With this information, we will be able to define not only combined strategies that use information of various strategies, but it also allows us to define dynamic strategies that change their behavior according to the user answers.

Acknowledgements
The author greatly thanks Bernd Braßel, Sebastian Fisher, Wolfgang Lux, Lee Naish, Henrik Nilsson and Bernie Pope for providing detailed and useful information about their debuggers. He also thanks Diego Cheda and David Insa for their valuable help in the experiments conducted to test all the strategies. Finally, he thanks Olaf Chitil for planting the seed of this study.

Appendix A. Vocabulary Index

References

[1] E. Shapiro, Algorithmic Program Debugging, MIT Press, 1982.

[2] J. Silva, A Comparative Study of Algorithmic Debugging Strategies, in: Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06), Springer LNCS 4407, 2007, pp. 143–159.

[3] P. Fritzson, N. Shahmehri, M. Kamkar, T. Gyimóthy, Generalized Algorithmic Debugging and Testing, ACM Letters on Programming Languages and Systems (LOPLAS) 1 (4) (1992) 303–322.

[4] H. Nilsson, Declarative Debugging for Lazy Functional Languages, Ph.D. thesis, Linköping, Sweden (1998).

[5] B. Pope, A Declarative Debugger for Haskell, Ph.D. thesis, The University of Melbourne, Australia (2006).

[6] R. Caballero, C. Hermanns, H. Kuchen, Algorithmic Debugging of Java Programs, Electronic Notes in Theoretical Computer Science 177 (2007) 75–89. doi:10.1016/j.entcs.2007.01.005.
URL `http://portal.acm.org/citation.cfm?id=1247751.1248158`

[7] H. Nilsson, P. Fritzson, Algorithmic Debugging for Lazy Functional Languages, Journal of Functional Programming 4 (3) (1994) 337–370.

[8] I. MacLarty, Practical Declarative Debugging of Mercury Programs, Ph.D. thesis, Department of Computer Science and Software Engineering, The University of Melbourne, Australia (2005).

[9] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, S. Lucas, Abstract Diagnosis of Function Programs, in: Proceedings of the 12th International Conference on Logic based Program Synthesis and Transformation (LOPSTR'02), Springer-Verlag, Berlin, Heidelberg, 2003, pp. 1–16.

[10] M. Comini, G. Levi, G. Vitiello, Abstract Debugging of Logic Programs, in: L. Fribourg, F. Turini (Eds.), Proceedings Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994, Vol. 883 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1994, pp. 440–450.

[11] F. C. M. Alpuente, D. Ballis, M. Falaschi, An Integrated Framework for the Diagnosis and Correction of Rule-Based Programs, Theoretical Computer Science 411 (47) (2010) 4055–4101.

[12] T. Davie, O. Chitil, Hat-delta: One Right Does Make a Wrong, in: Seventh Symposium on Trends in Functional Programming (TFP'06), 2006.

[13] L. Naish, A Declarative Debugging Scheme, Journal of Functional and Logic Programming 1997 (3).
URL citeseer.ist.psu.edu/article/naish97declarative.html

[14] M. Maeji, T. Kanamori, Top-Down Zooming Diagnosis of Logic Programs, Tech. Rep. TR-290, ICOT, Japan (1987).

[15] E. Av-Ron, Top-Down Diagnosis of Prolog Programs, Ph.D. thesis, Weizmanm Institute, Israel (1984).

[16] R. Caballero, A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs, in: Proceedings of the 14th Workshop on Curry and Functional Logic Programming (WCFLP'05), ACM Press, New York, USA, 2005, pp. 8–13. doi:http://doi.acm.org/10.1145/1085099.1085102.

[17] G. Kokai, J. Nilson, C. Niss, GIDTS: A Graphical Programming Environment for Prolog, in: Workshop on Program Analysis For Software Tools and Engineering (PASTE'99), ACM Press, 1999, pp. 95–104.
URL citeseer.ist.psu.edu/248106.html

[18] D. Binks, Declarative Debugging in Gödel, Ph.D. thesis, University of Bristol (1995).

[19] V. Hirunkitti, C. J. Hogger, A Generalised Query Minimisation for Program Debugging., in: Proceedings of International Workshop of Automated and Algorithmic Debugging (AADEBUG'93), Springer LNCS 749, 1993, pp. 153–170.

[20] T. Conway, F. Henderson, Z. Somogyi, Code Generation for Mercury, in: In Proceedings of the 12th International Conference on Logic Programming (ICLP'95), 1995, pp. 242–256.
URL citeseer.ist.psu.edu/conway94code.html

[21] M. Wallace, O. Chitil, T. Brehm, C. Runciman, Multiple-View Tracing for Haskell: a New Hat, in: Proceedings of the 2001 ACM SIGPLAN Haskell Workshop, Universiteit Utrecht UU-CS-2001-23, 2001, pp. 151–170.

[22] T. Davie, O. Chitil, Hat-delta: One Right Does Make a Wrong, in: A. Butterfield (Ed.), Draft Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages (IFL'05), Tech. Report No: TCD-CS-2005-60, University of Dublin, Ireland, 2005, p. 11.

[23] L. M. Pereira, Rational Debugging in Logic Programming, in: Proceedings on 3rd International Conference on Logic Programming (ICLP'86), Springer-Verlag LNCS 225, New York, USA, 1986, pp. 203–210.

[24] L. Naish, A Three-Valued Declarative Debugging Scheme, in: Proceedings of Workshop on Logic Programming Environments (LPE'97), 1997, pp. 1–12.
URL `citeseer.ist.psu.edu/naish97threevalued.html`

[25] J. Silva, O. Chitil, Combining Algorithmic Debugging and Program Slicing, in: Proceedings of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06), ACM Press, 2006, pp. 157–166.

[26] J. Silva, Algorithmic debugging strategies, in: Proceedings of International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06), 2006, pp. 134–140.

[27] J. Silva, Three New Algorithmic Debugging Strategies, in: Proceedings of VI Jornadas de Programación y Lenguajes (PROLE'06), 2006, pp. 243–252.

[28] J. Silva, O. Chitil, Combining Algorithmic Debugging and Program Slicing, Tech. Rep. DSIC-II/04/06, UPV, available from URL: `http://www.dsic.upv.es/~jsilva/research.htm#techs` (2006).

[29] D. Insa, J. Silva, An Algorithmic Debugger for Java, in: Proceedings of the 26th IEEE International Conference on Software Maintenance September 12-18, Timisoara, Romania, 2010.