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/12/06
Pages: 51

Title: A Classification of Algorithmic Debugging Strategies

Author(s): Josep Silva

Date: December 04, 2006
Keywords: Algorithmic Debugging, Debugging

Vº Bº
Leader of research Group Author(s)

A Classification of Algorithmic Debugging Strategies

Josep Silva

DSIC, Technical University of Valencia
Camino de Vera s/n, E-46022 Valencia, Spain
jsilva@dsic.upv.es

1 Introduction

Algorithmic debugging is a debugging technique which relies on the programmer having an *intended interpretation* of the program. In other words, some computations of the program are correct and others are wrong with respect to the programmer's intended semantics. Therefore, algorithmic debuggers compare the results of sub-computations with what the programmer intended. By asking the programmer questions or using a formal specification the system can identify precisely the location of a program's bug.

Essentially, algorithmic debugging is a two-phase process: An *execution tree* (see, e.g., [13]), ET for short, is built during the first phase. Each node in this ET corresponds to an equation which consists of a function call with completely evaluated arguments and results[1]. Roughly speaking, 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, a new node is recursively added to the ET as the child of n. This notion of ET is valid for functional languages but it is insufficient for other paradigms as the imperative programming paradigm. In general, the information included in the nodes of the ET incudes all the data needed to determine if the equations are correct. For instance, in the imperative programming paradigm, with the function (or procedure) of each node it is included the value of all global variables when 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., [4]). In the second phase, the debugger traverses the ET asking an oracle (typically the programmer) whether each equation is correct or wrong. At the beginning, the *suspicious area* which contains those nodes that can be buggy (a buggy node is associated with a buggy rule of the program) is empty; but, after every question, 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 function definition of this node [16]. If a bug symptom is detected then algorithmic debugging is complete [18]. It is important to say that, once the execution tree is built, the problem of traversing it and selecting a node is

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

independent of the language used; hence 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.

Furthermore, questions can also be very complex. For instance, during a debugging session with a compiler, the algorithmic debugger of the Mercury language [11]—currently, one of the most advanced algorithmic debuggers—asked a question of more than 1400 lines.

Therefore, new techniques and strategies to reduce the number of questions, to simplify them and to improve the order in which they are asked are a necessity to make algorithmic debuggers usable in practice.

In this paper we review and compare the current algorithmic debugging strategies and propose three new strategies (less YES first, divide by YES and query, and dynamic weighting search) that can further reduce the number of questions asked during an algorithmic debugging session.

The rest of the paper is organized as follows. The next section shows an example of algorithmic debugging session that will be used along the paper. Section 3 reviews current algorithmic debugging strategies and proposes three new strategies. In Section 4 an algorithm to combine different strategies is introduced. Then, in Section 5 we present a comparison of all techniques and we study their costs. Finally, Section 6 concludes.

2 An Algorithmic Debugging Session

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 programmer cannot give an answer (e.g., because the question is too complex). After every question, some nodes of the ET leave the suspicious area. 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 again for each different bug.

Let us illustrate the process with an example[2].

Example 1. Consider the buggy program in Fig. 1 adapted to Haskell from [8]. 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.

[2] While almost all the 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
```

Fig. 1. Example program

From this program, an algorithmic debugger can automatically generate the ET of Fig. 2 (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 Fig. 3. During the debugging session, the system asks the oracle 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 is located in function "`sum2`" (node 23). The definition of function "`sum2`" should be: `sum2 x = div (x*(decr x)) 2`

3 Algorithmic Debugging Strategies

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

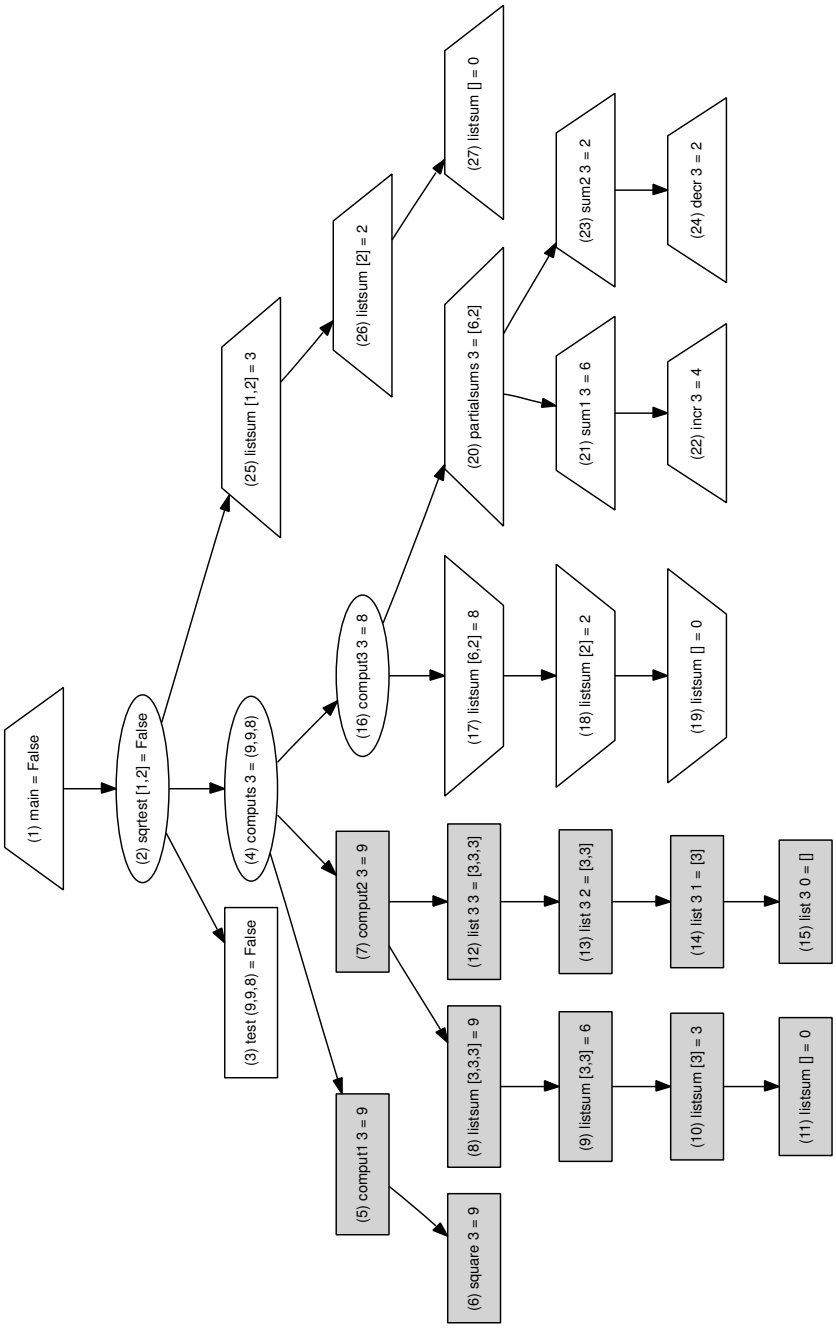


Fig. 2. Execution tree of the program in Fig. 1

```
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
```

Fig. 3. Debugging session for the program in Fig. 1

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 fluke [7]. 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 the bug.

3.1 Single Stepping *(Shapiro, 1982)*

The first algorithmic debugging strategy to be proposed was *single stepping* [18]. 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, 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 Fig. 2 would be: 3-YES, 6-YES, 5-YES, 11-YES, 10-YES, 9-YES, 8-YES, 15-YES, 14-YES, 13-YES, 12-YES, 7-YES, 19-YES, 18-YES, 17-YES, 22-YES, 21-YES, 24-YES, 23-NO.

Note that in this strategy questions are semantically unconnected.

3.2 Top-Down Search *(Av-Ron, 1984)*

Due to the fact that questions are asked in a logical order, *top-down search* [1] is the strategy that has been traditionally used (see, e.g., [3, 10]) to measure the performance of different debugging tools and methods. It basically consists in a top-down, left-to-right traversal of the ET and, thus, the node asked is always a child or a sibling of the previous question node. When a node is answered NO, one of its children is asked; if it is answered YES, one of its siblings is. 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 Fig. 2 is shown in Fig. 3.

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:

Top-Down Zooming *(Maeji and Kanamori, 1987)* During the search of previous 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 he 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 strategy [12], recursive child calls are preferred.

The sequence of questions asked for the ET in Fig. 2 is exactly the same as with top-down search (Fig. 3) because no recursive calls are found.

Another variant of this strategy called *exception zooming*, introduced by Ian MacLarty [11], selects first those nodes that produced an exception at runtime.

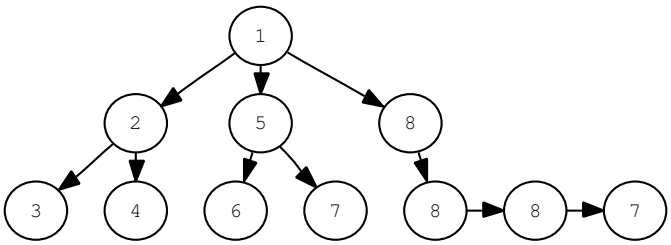
Heaviest First *(Binks, 1995)* Selecting always the left-most child does not take into account the size of the subtrees that can be explored. Binks proposed in [2] a variant of top-down search in order to consider this information when selecting a child. This variant is called *heaviest first* because it always selects the child with a bigger subtree. The objective is to avoid selecting small subtrees which have a lower probability of containing the bug.

For instance, the sequence of 9 questions asked for the ET in Fig. 2 would be[3]: 1-NO, 2-NO, 4-NO, 7-YES, 16-NO, 20-NO, 21-YES, 23-NO, 24-YES.

[3] 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.

Less YES First ***(Silva, 2006)*** This section introduces a new 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 the bug—rather than a node in the ET—and considering that there is not a relation one-to-one 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 2. 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 the bug (rather than all the nodes). Another possibility could be to associate a different probability of containing the 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 $q \cdot p$ where q is the probability that the rule associated to this node is wrong, and p is the probability of this rule to execute incorrectly. Therefore, under the assumption that all the rules have the same probability of being wrong, the probability P of a branch b to contain the bug is:

$$P = \frac{\sum_{i=1}^{n} p_i}{R}$$

where n is the number of nodes in b, R is the number of rules in the program, and p_i is the probability of the rule in node i to produce a wrong result if it is incorrect.

Clearly, if we assume that a wrong rule always produces a wrong result[4] we have that

$$P = \frac{\sum_{i=1}^{r} p_i}{R} \text{ and } \forall i.p_i = 1,$$

thus the probability is:

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

where r is the number of rules in b, and thus, this strategy is (on average) better than heaviest first. For instance, in Example 2 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 $\frac{2}{8}$ despite it has more nodes.

However, in general, a wrong rule can produce a correct result, and thus we need to consider the probability of a wrong rule to return a wrong answer. This probability has been approximated by the debugger Hat-delta (see Section 3.4) 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 $n' < n \leqslant m$.

Here, we use 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 user, 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 user).

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, we proceed as follows: When a node is set correct, we mark its associated rule and all the rules of its descendants 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 computations 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:

[4] This assumption is valid for instance in those flattened functional languages where all the conditions in the right-hand side of function definitions have been distributed between its rules. This is relatively frequent in internal languages of compilers, but not in source languages.

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

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.

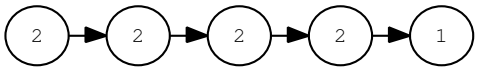
As with heaviest first, we select the branch with the biggest weight, the difference is that this equation to compute the weight takes into account previous answers of the user. Moreover, we assume that initially all the rules have been answered YES once, and thus, at the beginning, this strategy asks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

As pointed out by Davie and Chitil [7], independently of the considered strategy, taking into account the rule associated with each node can help to avoid many unnecessary questions. Consider the following example:

Example 3. The ET associated with function `append`:

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

w.r.t. 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 have the same associated rule (2) and, thus, the bug must necessarily be in rule 2. Note that the use of 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.

Davie and Chitil have generalized this idea and introduced the tree compression technique [7]. This technique proceeds as follows: For any node n_1 with associated rule r_1 and child n_2 with associated rule r_1, it replaces node n_2 with its children (i.e., the children of n_2 become the children of n_1). Therefore, the tree compression technique prunes ETs even before starting to ask any question.

With this strategy, the sequence of 9 questions asked for the ET in Fig. 2 is: 1-NO, 2-NO, 4-NO, 7-YES, 16-NO, 20-NO, 21-YES, 23-NO, 24-YES.

3.3 Divide & Query *(Shapiro, 1982)*

In 1982, together with single stepping, Shapiro proposed another strategy: the so-called divide & query (D&Q) [18]. 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 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 [3]; however, the main drawback of this strategy is that successive questions may have no connection, from a semantic point of view, with each other; requiring the programmer more time for answering the questions.

For instance, the sequence of 6 questions asked for the ET in Fig. 2 is: 7-YES, 16-NO, 17-YES, 21-YES, 24-YES, 23-NO.

Hirunkitti's Divide & Query ***(Hirunkitti and Hogger, 1993)*** In [9], 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 divide & query algorithm. Their version of divide & query is the same as the one of 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 greater than or equal to $w/2$

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

For instance, the sequence of 6 questions asked for the ET in Fig. 2 is: 7-YES, 16-NO, 17-YES, 21-YES, 24-YES, 23-NO.

Biased Weighting Divide & Query ***(MacLarty, 2005)*** MacLarty proposed in his PhD thesis [11] that not all the nodes should be considered equally while dividing the tree. His variant of D&Q divides the tree 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 [5] which distinguishes between 13 different node types.

Divide by YES & Query ***(Silva, 2006)*** The same idea used in less YES first can be applied in order to improve divide & query. 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 the bug. In contrast, this strategy tries to compute this probability.

By using the equation to compute the weight of a branch, this strategy computes the weight associated to 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 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 greater 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, we could follow another strategy (e.g., Hirunkitti) 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 4. Consider again the ET in Example 2. 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 user.

With this strategy, the sequence of 5 questions asked for the ET in Fig. 2 is: 7-YES, 16-NO, 21-YES, 23-NO, 24-YES.

3.4 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 [6]. 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).

This strategy assumes 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 $n' < n \leqslant m$. 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 [7]:

– *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 5. Consider this program:

```
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
```

where the left numbers 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.
With this information, `otherwise = insert x ys` is more likely to be wrong.

3.5 Subterm Dependency Tracking *(MacLarty et al., 2005)*

In 1986, Pereira [17] 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 [14]). An equation or, more precisely, 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. Hence, inadmissibility allows us to identify errors in left-hand sides of equations.

However, with only these four possible answers the system fails to get fundamental information from the programmer about *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., which part of the result is incorrect? is one particular argument inadmissible?).

Consider again the equation `insert 'b' "cc" = "bcc"`. Here, the programmer could detect that the second argument should not have been computed; he

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 et al. [11] proposed a new strategy called subterm dependency tracking. 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 Fig. 2, 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 up to question 3. This chain is formed by nodes 2, 4, 16 and 17. The system would ask first 17, then 16, and finally 4 following the computed chain.

In our example, the sequence of 8 questions asked for the ET in Fig. 2, combining this strategy with top-down search, is: 1-NO, 2-NO, 3-YES (the programmer marks "8"), 17-YES, 16-NO, 20-NO (the programmer marks "2"), 23-NO, 24-YES.

3.6 Dynamic Weighting Search *(Silva, 2006)*

Subterm dependency tracking 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 up to 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.

Here, we propose 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, w.l.o.g., we compute weights instead of probabilities and we assume initially that all the nodes have a weight 1 and that a weight 0 means "*out of the suspicious area*".

Computing Weights from Subterms

Firstly, as with subterm dependency tracking, we allow the oracle 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 he 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 up to 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 the bug. Let us show it with an example.

Example 6. Consider the ET in Fig. 2, 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:

- **Set 1:** those nodes which evaluated the equation 20 to produce the wrong subterm are denoted by an inverted trapezium.
- **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 trapezium.
- **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 the 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 the 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 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 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 the bug is zero[5].

These subsets of the ET are in fact slices of different parts of the computation. In order to extract these slices, we use the formalization of Augmented Redex

[5] A proof can be found in [19].

Trails of the Appendix. We use ARTs because they can represent the underlying trace from which ETs are produced.

Lemma 1 (Origin of an expression in ETs). *Given an ET $\mathcal{E}$ for an ART $\mathcal{G}$ and an expression $e \in \mathcal{E}$, $e \neq main$ associated to $n \in \mathcal{G}$ marked as wrong or inadmissible during an algorithmic debugging session, then, the equation in $\mathcal{E}$ that introduced e into the computation is:*

$$equ(\mathcal{G}, m) = mef(\mathcal{G}, mt) \qquad \text{where } m = parent(n)$$

We often call this equation "the origin of e".

Proof. Firstly, e cannot be further evaluated because it appears in an equation of $\mathcal{E}$, and from Definition 3 holds that n represents e in $\mathcal{G}$. Let us assume that $l = r$ is the equation in $\mathcal{E}$ that introduced e into the computation (the origin of e). Then, either $e \in l$ or $e \in r$.

Since we assume that the computation starts in the distinguished function *main* which have no arguments, and, because new expressions are only introduced in right-hand sides, if $e \in l$ then $e = main$. But one of the premises of the lemma is that $e \neq main$, and thus, $e \in r$.

Let n' be the node associated to r. Since e cannot be further evaluated then $parent(n) = parent(n')$. Therefore, $m = parent(n)$ is the node associated to l, and $mt = n'$ is the node associated to r; then, following Definition 2, $equ(\mathcal{G}, m) = mef(\mathcal{G}, mt)$ is $l = r$.

Note that e does not necessarily appear in the equation $l = r \in \mathcal{E}$ which is the origin of e, because e could be a subexpression of l' in the equation $l' = r' \in \mathcal{E}$ where the oracle identified e. For instance, consider the question "$3 - 1 = 2$?" produced from the ART in Fig. 6. If the oracle marks 1 as inadmissible, then the equation which introduced 1 into the computation is "*decr* $3 = 2$?" which not includes 1 due to the normalization process.

Each of the four subsets is computed as follows:

- The equations in $\mathcal{E}$ which belong to the Set 1 are those which are in the subtree rooted at the origin of e.
- Given an ET $\mathcal{E}$ and two equations $eq, eq' \in \mathcal{E}$ where the expression e appears in eq and the equation eq' is the origin of e, then, if the oracle marks e as wrong or inadmissible during an algorithmic debugging session, the equations in $\mathcal{E}$ which belong to the Set 2 are all the equations which are in the path between eq and eq' except eq and eq'.
- The Set 4 contains all the nodes that could not influence the expression e, and thus, it can be computed following the approach of [19]. In particular, by using the algorithm defined in [19] a slice w.r.t. the expression e can be computed. All the equations $equ(\mathcal{G}, n) = mef(\mathcal{G}, nt)$ in $\mathcal{E}$ associated to reductions n to nt in $\mathcal{G}$ that do not belong to the slice form Set 4.
- Finally, Set 3 contains all the equations in $\mathcal{E}$ that do not belong to Sets 1, 2 and 4.

Input: an execution tree (ET)
Output: a node
Initialization: Nodes' weight set to "1"; info = $\varnothing$
Repeat
    $node = computeWeights(info, ET)$;
    $info = info \cup askQuestion(node)$;
Until $bugFound(info, ET)$
Return: buggy node

Fig. 4. Algorithmic debugging procedure

This strategy can integrate information used by other strategies (e.g., previous answers of the oracle) in order to modify nodes' weights. In the next section we introduce an algorithm to combine information from different strategies. This algorithm can help dynamic weighting search to integrate information used by other strategies (e.g., previous answers of the oracle) in order to modify nodes' weights.

4 Combining Algorithmic Debugging Strategies

In Fig. 5 we can see that each strategy uses different information during the search. Hence a combination of strategies can produce more efficient new strategies. The algorithm in Fig. 4 is a simple generic procedure for algorithmic debugging. For simplicity, it assumes that the information provided by the oracle is stored in a data structure '*info*' which is combined by means of the operator $\cup$. Essentially, *info* stores all the answers and expressions marked by the oracle, the information about which nodes belong to the suspicious area and, in general, all the information needed for the strategies being used as, for instance, the number of YES answers for every node. It uses the following three functions:

bugFound(info,ET): It is a simple test to check if the bug can be found with the information provided by the oracle so far.
askQuestion(node): It prompts the oracle with the question of a node and gets new information.
computeWeights(info,ET): This function determines the new weights of the nodes and selects the heaviest node w.r.t. a particular strategy.

Therefore, the strategy used in the algorithm is only implemented by *computeWeights*. Now, we propose a combination of strategies to implement this function.

Function *computeWeights* proceeds by computing the weight of every node, and then returning the node with the maximum weight. We compute the weight of a node from the sum $a+b+c+d+e+f$ where each component is the result of applying a different strategy plus an addend. For instance:

$a = a' +$ Optimizations
$b = b' +$ Computing Weights from Subterms
$c = c' +$ Divide by YES & Query
$d = d' +$ Less YES First
$e = e' +$ Hirunkitti
$f = f' +$ Heaviest First

where $a' \ldots f'$ are the addends and every strategy produces a weight between 0 and a bound C. It is not difficult to modify a given strategy to produce a weight for each node. For instance, heaviest first can be modified in order to assign a higher weight to the nodes with more descendants. Hence, in Example 2, nodes labeled 5 and 2 would be assigned a weight of three, whilst nodes 3, 4, 6 and 7 would be assigned a weight of two. In addition, we must ensure that weights are not greater than the bound C, thus, in order to assign a weight to node n, we can use the equation:

$$\frac{C \cdot weight(n)}{weight(r)}$$

where r is the root node of the ET. Other strategies can be modified in a similar way, e.g., Less YES first assigns a higher weight to the nodes whose subtree is more likely to contain the buggy node, and D&Q assigns a higher weight to the nodes that are closer to the center of the ET.

Here, we consider *Optimizations* as another strategy because, independently of the strategy used, it is always possible to apply some optimizations during the debugging process. Some of them are obvious, as for instance excluding the first question (`main`) from the suspicious area (because its answer is always NO), or propagating answers to duplicate questions. For instance, in the ET of Fig. 2 questions 11, 19 and 27 are identical and, thus, if one of them is answered with YES or NO, so are the others. Other optimizations, on the other hand, are less obvious and they can influence the order of the questions as it was shown in Example 3. Optimizations should be used to select a particular node by assigning to it a weight C; and to eliminate nodes which cannot be buggy (i.e., nodes equal to a node answered YES) by assigning them a weight zero. Therefore, the addend of optimizations should be the greatest addend.

For instance, the previous combination of strategies with the addends $a' = 47C$, $b' = 23C$, $c' = 11C$, $d' = 5C$, $e' = 2C$ and $f' = 0$ would apply the optimizations first (if possible), then, whenever two nodes have the same weight, computing weights from subterms would be applied to break the tie, and so on. This is due to the fact that these addends guarantee the following relations:

$a > b + c + d + e + f$,
$b > c + d + e + f$,
$c > d + e + f$,
$d > e + f$,
$e > f$

Note the importance of C—but not of the value of C—to ensure the previous relations. In contrast, with the addends $a' = b' = ... = f'$ all the strategies would be applied at the same time.

The best combination of strategies depends in each case on the current context (i.e., branching factor of the tree, previous answers of the oracle, etc) thus the combination of strategies used by the debugger should also be dynamically modified during the algorithmic debugging process. Therefore, the debugger itself should state the addends dynamically after every question. In the next section we compare the cost of a set of strategies in order to theoretically determine their addends.

5 Comparing Strategies

A summary of the information used by every strategy is shown in Fig. 5. The meaning of each column is the following:

- '*(Struct)ure*' is marked if the strategy takes into account the distribution of nodes (or rules) in the tree;
- '*Rules*' 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;
- '*(Inadm)issibility*' is marked if the strategy accepts "inadmissible" answers;
- '*History*' 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 tree 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 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 branching factor.

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

Strategy	Struct.	Rules	Sem.	Inadm.	History	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 Tracking	-	-	✓✓✓	✓	-	-	n
Dynamic Weighting Search	✓	✓	-	✓	✓	-	n

Fig. 5. Comparing algorithmic debugging strategies

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. Note that, after the tree compression described in Section 3.2, no zoom is possible and, thus, both strategies are identical.

In contrast, while in the worst case the costs of top-down and heaviest first are equal, in the mean case heaviest first performs 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 $\dfrac{b+1}{2}$ of the children.
- Heaviest first asks $\dfrac{\sum_{i=1}^{b} weight(s_i) \cdot pos(s_i)}{\sum_{i=1}^{b} weight(s_i)}$ 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 ordered by their weights.

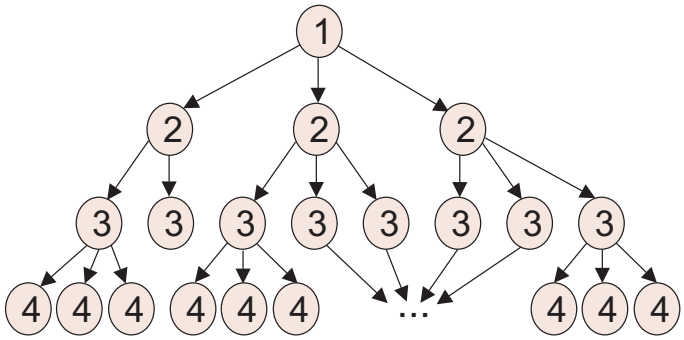
In the case of less YES first, the improvement is based on the fact that the heaviest branch is not always the branch with a higher probability of containing the buggy node. While heaviest first and less YES first have the same worst case query complexity, their average cost must be compared empirically.

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

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

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 YES, and the information of YES answers obtained makes the algorithmic debugger explore the rest of the ET bottom up. It can be seen in Example 7.

Example 7. Consider the following ET:



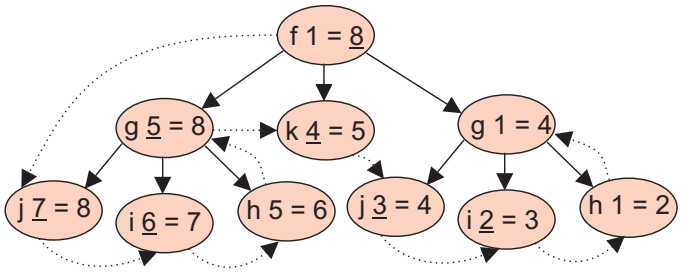
If we use the version of Hat-delta which counts correct computations, then the first node asked would be the left-most 2. If it is answered YES, then all the nodes in this branch are marked as correctly executed, and thus, rule 2 has been correctly executed once, rule 3 has been correctly executed twice and rule 4 has been correctly executed three times. Therefore, the rest of the nodes are asked bottom up. In this case, the cost is $n - \frac{(d-1)d}{2}$.

Despite subterm dependency tracking is a top-down 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. In fact, this strategy—and also dynamic weighting search if we assume that top-down 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 8. 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:

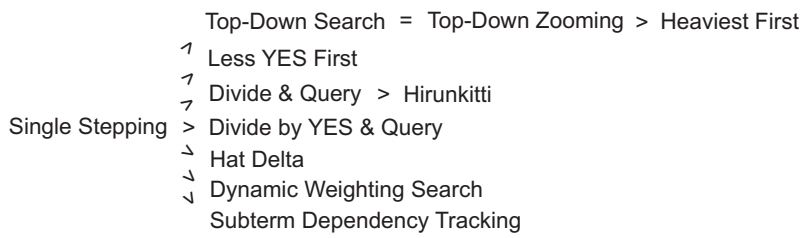
```
Starting Debugging Session...

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 to every strategy in Fig. 5 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. For instance, 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. If we use this result in the combination of strategies proposed in Section 4 we can conclude that if $d < log_2 n$ then $f' < e'$.

6 Conclusions

This article introduces three new strategies and some optimizations for algorithmic debugging. Less YES first tries to improve heaviest first and divide by YES

& query tries to improve D&Q by considering previous answers of the oracle during the search. Dynamic weighting search allows the user to specify the exact part of an equation which is wrong. This extra information can produce a much more accurate debugging session.

We have introduced an algorithm to combine different strategies, and we have compared the most important algorithmic debugging strategies from a theoretical perspective. The comparison has been done according to seven dimensions including their worst case query complexity; and have produced some objective criteria to determine which strategy is better depending on the context.

We have implemented all the strategies and incorporated them in the algorithmic debugger DDT [3]. As future work, we plan to perform an empirical comparison of all the strategies in order to determine a weighting for their combination. With the knowledge acquired from the experiment we will be able to approximate the strategies' weights and to determine how they should change and on which factors this change depends.

7 Acknowledgements

I greatly thank Olaf Chitil, Thomas Davie and Yong Luo for many discussions about the contents of this paper. I also want to thank the anonymous referees of LOPSTR'06 for their helpful comments.

References

1. E. Av-Ron. *Top-Down Diagnosis of Prolog Programs.* PhD thesis, Weizmanm Institute, 1984.
2. D. Binks. *Declarative Debugging in Gödel.* PhD thesis, University of Bristol, 1995.
3. R. Caballero. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proc. of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*, pages 8–13, New York, USA, 2005. ACM Press.
4. R. Caballero. Algorithmic Debugging of Java Programs. In *Proc. of the 2006 Workshop on Functional Logic Programming (WFLP'06)*, pages 63–76. Electronic Notes in Theoretical Computer Science, 2006.
5. T. Conway, F. Henderson, and Z. Somogyi. Code Generation for Mercury. In *In Proc. of the International Logic Programming Symposium*, pages 242–256, 1995.
6. T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In Andrew Butterfield, editor, *Draft Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages (IFL'05)*, page 11. Tech. Report No: TCD-CS-2005-60, University of Dublin, Ireland, September 2005.
7. T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 06*, April 2006.
8. P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimóthy. Generalized Algorithmic Debugging and Testing. *LOPLAS*, 1(4):303–322, 1992.
9. V. Hirunkitti and C. J. Hogger. A Generalised Query Minimisation for Program Debugging. In *Proc. of International Workshop of Automated and Algorithmic Debugging (AADEBUG'93)*, pages 153–170. Springer LNCS 749, 1993.

10. G. Kokai, J. Nilson, and C. Niss. GIDTS: A Graphical Programming Environment for Prolog. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*, pages 95–104. ACM Press, 1999.
11. I. MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.
12. M. Maeji and T. Kanamori. Top-Down Zooming Diagnosis of Logic Programs. Technical Report TR-290, ICOT, Japan, 1987.
13. L. Naish. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.
14. L. Naish. A Three-Valued Declarative Debugging Scheme. In *Proc. of Workshop on Logic Programming Environments (LPE'97)*, pages 1–12, 1997.
15. H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.
16. H. Nilsson and P. Fritzson. Algorithmic Debugging for Lazy Functional Languages. *Journal of Functional Programming*, 4(3):337–370, 1994.
17. L. M. Pereira. Rational Debugging in Logic Programming. In *Proc. on Third International Conference on Logic Programming*, pages 203–210, New York, USA, 1986. Springer-Verlag LNCS 225.
18. E.Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.
19. J. Silva and O. Chitil. Combining Algorithmic Debugging and Program Slicing. In *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 157–166. ACM Press, 2006.
20. J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. of the 9th Int'l Symp. on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS 1292, 1997.
21. 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*, pages 151–170. Universiteit Utrecht UU-CS-2001-23, 2001.

APPENDIX[6]

This appendix formalizes a trace structure that can be used to produce execution trees. The *augmented redex trail (ART)* [20, 21] is such a trace structure. The ART is used in the Haskell tracer Hat [20, 21] and has similarities with the trace structure constructed by the earlier algorithmic debugger Freja [15]. The ART has been designed so that it can be constructed efficiently and after its complete construction the execution tree needed for algorithmic debugging can be constructed from it easily. In addition, an ART contains more information than an execution tree. An execution tree can be constructed from an ART but not vice versa. The ART supports several other views of a computation besides the execution tree. Nonetheless an ART is not larger than an execution tree with sharing. An ART is a complex graph of expression components that can represent both eager or lazy computations.

Definition 1 (Augmented redex trail). *A **node** n is a sequence of the letters l, r and t; which stands respectively for left, right and top. There is also a special node $\perp$ (bottom).*

*A **node expression** $\mathcal{T}$ is a function symbol, a data constructor, a node or an application of two nodes.*

*A **trace graph** is a partial function $\mathcal{G} : n \mapsto \mathcal{T}$ such that its domain is prefix-closed (i.e., if $ni \in dom(\mathcal{G})$, then $n \in dom(\mathcal{G})$), but $\perp \notin \mathcal{D}om(\mathcal{G})$. We often regard a trace graph as a set of tuples $\{(n, \mathcal{G}(n)) \mid n \in dom(\mathcal{G})\}$*

*A node can **represent** several terms, because any nodes n and nt are considered equal. A node **approximates** a term if it represents this term, except that some subterms may be replaced by $\perp$.*

*A node n plus a node nt represent a **reduction step** of the program, if node n represents the redex and node nt approximates the reduct, and for each variable of the corresponding program rule there is one shared node.*

*A trace graph $\mathcal{G}$ where node ϵ (empty sequence) approximates the start term M and every pair of nodes n and nt represents a reduction step with respect to the program P, is an **augmented redex trail (ART)** for start term M and program P.*

The sequence of the letters l, r and t of every node is used as identifier, thus every node has a unique sequence of these letters. Any application of two terms t_1 and t_2 represented with the sequence x, has an arc to these terms whose nodes are identified respectively with the sequences xl and xr. When a term t_1 represented by a node x is reduced to another term t_2 through a single reduction step, t_2 is identified with the sequence xt. We often call this kind of nodes ended with t as reductions. Finally, a node labeled with $\perp$ represents a non-evaluated term.

For example, consider the graph depicted in Fig. 6. This graph is part of the ART generated from the program in Fig. 1. In particular, it corresponds to the function call "$sum2$ 3" producing the result "2". As specified in Definition 1, each

[6] Part of the material of this appendix comes from [19].

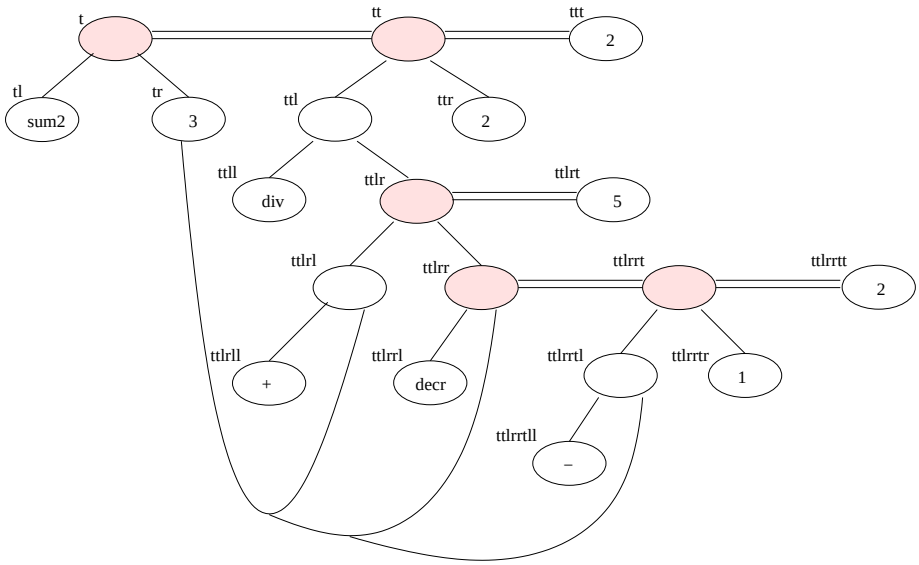


Fig. 6. Example of ART

node is identified by a sequence of letters 't', 'l' and 'r'[7]. Every node contains the expression it represents, except for a node which represents an application of two expressions connected to it by simple lines. A double line represents a single reduction step. For instance, node *ttlrrt* represents the function call "$3-1$" that has been reduced to "2".

In the following, a (atom) denotes a function symbol or constructor, and n, m and o denote nodes. We usually write $n \in \mathcal{G}$ instead of $n \in \mathcal{D}om(\mathcal{G})$ when it is clear from the context. We also say that a node $n \in \mathcal{G}$ is reduced iff $nt \in \mathcal{G}$. Given a node $n \in \mathcal{G}$ where $\mathcal{G}(n)$ is an application of nodes $m \cdot o$, we respectively refer to m and o as $\mathit{left}(n)$ and $\mathit{right}(n)$.

The parent of a node is the node which created it, that is, given a reduction step $t \to s$, and a node n representing a subterm of s, then, the parent of n is the node which represents t. Formally:

$$\begin{aligned} \mathit{parent}(nl) &= \mathit{parent}(n) \\ \mathit{parent}(nr) &= \mathit{parent}(n) \\ \mathit{parent}(nt) &= n \end{aligned}$$

The recursive application of function *parent* produces the ancestors of a node:

$$\mathit{ancestors}(n) = \begin{cases} \{m\} \cup \mathit{ancestors}(m) & \text{if } m = \mathit{parent}(n) \text{ defined} \\ \emptyset & \text{otherwise} \end{cases}$$

[7] For easier reading we here call the first node t; within the ART of the complete computation the node is actually *trtrtrtrlr*.

Finally, to extract the most evaluated form represented by a node, function *mef* is defined by:

$$mef(\mathcal{G}, n) = \begin{cases} a & \text{if } \mathcal{G}(n) = a \\ mef(\mathcal{G}, m) & \text{if } \mathcal{G}(n) = m \\ mef(\mathcal{G}, m) \cdot mef(\mathcal{G}, o) & \text{if } \mathcal{G}(n) = m \cdot o \text{ and } nt \notin \mathcal{G} \\ mef(\mathcal{G}, nt) & \text{if } nt \in \mathcal{G} \end{cases}$$

Intuitively, function *mef* traverses the graph $\mathcal{G}$ following reduction edges (n to nt) as far as possible. Note that this function is similar to function *val* introduced in Definition **??**, but *mef* is defined for a higher-order language.

Now we see how easily the execution tree is obtained from the ART. We first formally define execution tree.

Definition 2 (execution tree). *Let $\mathcal{G}$ be an ART. The execution tree $\mathcal{E}$ associated to $\mathcal{G}$ is a tree whose nodes are labeled with equations of the form $l = r$ such that:*

- *the root node is $equ(\mathcal{G}, \epsilon) = mef(\mathcal{G}, t)$, if $t \in \mathcal{G}$,*
- *for each reduction step n to nt in $\mathcal{G}$ there is a node $equ(\mathcal{G}, n) = mef(\mathcal{G}, nt)$ in $\mathcal{E}$, and*
- *for each pair of nodes $n'_1, n'_2 \in \mathcal{E}$, with n'_1 labeled $equ(\mathcal{G}, n_1) = r_1$ and n'_2 labeled $equ(\mathcal{G}, n_2) = r_2$; n'_1 is the parent of n'_2 iff $n_2 = n_1 to$ where $o \in \{r, l\}^*$.*

Function equ is defined as follows:

$$equ(\mathcal{G}, n) = \begin{cases} a & \text{if } \mathcal{G}(n) = a \\ mef(\mathcal{G}, m) \cdot mef(\mathcal{G}, o) & \text{if } \mathcal{G}(n) = m \cdot o \end{cases}$$

The reduced nodes of the ART become the nodes of the execution tree. The function *parent* expresses the child–parent relationship between these nodes in the execution tree. The equation of a node n can be reconstructed by applying the function *mef* to $left(n)$ and $right(n)$ to form the left-hand side, and to nt to form the right-hand side. For instance, for the ART in Fig. 6 the algorithmic debugger produces the execution tree in Fig. 7. It contains five questions, one for each reduction step (reduced nodes have been shaded). For example, the question asked for node *ttlr* is "$3 + 2 = 5$?".

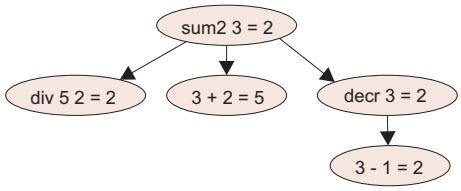


Fig. 7. Execution tree of the ART in Fig. 6

Definition 3 (nodes associated to expressions). *Let $\mathcal{G}$ be an ART and $\mathcal{E}$ the execution tree associated to $\mathcal{G}$. A node $n \in \mathcal{G}$ is associated to an expression $e \in \mathcal{E}$ iff n represents e and $\nexists\, n' \in \mathcal{G} \mid n' = no$, $o \in \{r, l, t\}^*$ and n' represents e.*