An Evaluation of Algorithmic Debugger Implementations[1]

Diego Cheda and Josep Silva[2]

DSIC, Technical University of Valencia
Camino de Vera s/n, E-46022 Valencia, Spain

Abstract

Algorithmic debugging is a semi-automatic debugging technique which is based on the answers of an oracle (usually the programmer) to a series of questions generated automatically by the algorithmic debugger. The technique typically traverses a record of the execution—the so-called *execution tree*—which only captures the declarative aspects of the execution and hides operational details. In this work we review and compare the most important algorithmic debuggers of different programming paradigms. In the study we analyze the features incorporated by current algorithmic debuggers, and we identify some features not supported yet by any debugger. We then compare all the debuggers giving rise to a map of the state of the practice in algorithmic debugging.

Keywords: Debugging, Algorithmic Debugging

1 Introduction

Algorithmic debugging [19] (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. These 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 the portion of code responsible for the buggy behavior is isolated.

Typically, algorithmic debuggers have a front-end which produces a data structure representing a program execution—the so-called *execution tree* (ET)[3] [16]—; and a back-end which uses the ET to ask questions and process the oracle's answers to locate the bug. Sometimes, the front-end and the back-end are not independent

[1] This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acción Integrada* HA2006-0008.

[2] Email:{dcheda,jsilva}@dsic.upv.es

[3] 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.

(e.g., in Buddha where the ET is generated to main memory when executing the back-end), or they are intertwiningly executed (e.g., in Freja where the ET is built lazily while the back-end is running).

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, 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 compare debuggers from heterogeneous paradigms—in order to be general enough—we will simply refer to all the information in an ET's node as *question*, and will avoid the atom/predicate/function/method/procedure distinction unless necessary.

```
(0) main = sort [3,1,2]

(1) sort [] = []
(2) sort (x:xs) = insert x (sort xs)

(3) insert x [] = [x]
    insert x (y:ys)
(4)     | x <= y = (x:ys)
(5)     | x > y  = (y:insert x ys)
```

Fig. 1. Buggy definition of insort.

Once the ET is built, the debugger basically traverses it by using some search strategies [19,10,14,1,13], and asking the oracle whether each question found during the traversal of the ET is correct or not. Given a program with a wrong behavior, this technique guarantees that, whenever the oracle answers all the questions, the bug will eventually be found. If there exists more than one bug in the program, only one of them will be found with each debugging session. Of course, once the first bug is removed, algorithmic debugging may be applied again in order to find another bug. Let us illustrate the process with an example.

```
Starting algorithmic debugging session

main = [1,3] ? NO
sort [3,1,2] = [1,3] ? NO
sort [1,2] = [1] ? NO
sort [2] = [2] ? YES
insert 1 [2] = [1] ? NO

Bug found in function "insert", rule (4)
```

Fig. 2. Debugging session for insort.

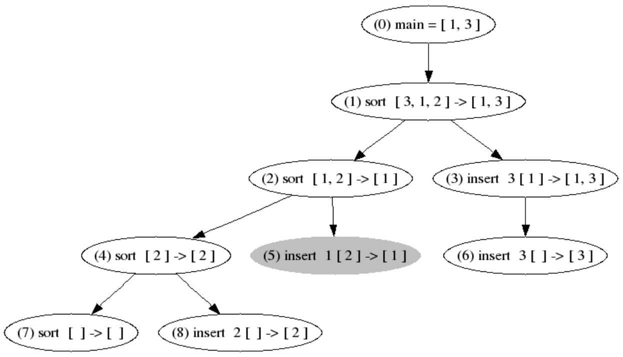
(0) main = [1, 3]
(1) sort [3, 1, 2] -> [1, 3]
(2) sort [1, 2] -> [1]
(3) insert 3 [1] -> [1, 3]
(4) sort [2] -> [2]
(5) insert 1 [2] -> [1]
(6) insert 3 [ ] -> [3]
(7) sort [ ] -> [ ]
(8) insert 2 [ ] -> [2]

Fig. 3. ET of the program in Fig. 1.

Example 1.1 Consider the erroneous definition of the insertion sort algorithm shown in Fig. 1. The bug is located in function `insert` at rule `(4)`. The correct right-hand side of the rule `(4)` should be `(x:y:ys)`. In Fig. 2 we can see an algorithmic debugging session for the program 1. First, the program is run and the ET—shown in Fig. 3, with the buggy node colored in gray—is generated by the debugger. Then, the debugging session starts at the root node of the tree. Following a top-down, left-to-right traversal of the ET, the oracle answers all the questions. Because the result of function `insert` at node `(5)` is incorrect and this node does not have any children, then the bug is located at this node.

The main contributions of this paper are the following: (i) We have identified and defined several functional requirements that an algorithmic debugger should implement. The collection of requirements presented is language-independent. Therefore, it constitutes a functional requirements specification to implement an algorithmic debugger. (ii) We have compared all current algorithmic debuggers taking into account their current functionality. Until now there was not a survey on algorithmic debugging implementations. The evaluation of the debuggers with respect to the requirements has shown what features each current algorithmic debugger offers. (iii) We have empirically evaluated the scalability of some debuggers by testing them with a set of selected benchmarks. This evaluation allowed us to also compare the average of the memory usage of each debugger.

2 A Comparison of Algorithmic Debuggers

2.1 Algorithmic Debuggers

The first part of our study consisted in the selection of the debuggers that were going to participate in the study. We found thirteen algorithmic debuggers, and we selected all of them except those immature enough to be used in practice (see Section 4.1). Our objective was not to compare algorithmic debugging-based techniques, but to compare mature and usable implementations. Therefore, we have evaluated each debugger according to its last implementation, not to its last report/article/thesis description.

The debuggers included in the study are the following:

Buddha: Buddha[4] [18] is an algorithmic debugger for Haskell 98 programs.

DDT: DDT[5] [2] is part of the multiparadigm language TOY's distribution.It uses a*graphical user interface* (GUI), implemented in Java (i.e., it needs the Java Runtime Environment to work).

Freja: Freja[6] [16] is a debugger for Haskell, result of Henrik Nilsson's thesis. It is able to debug a subset of Haskell; unfortunately, it is not maintained anymore.

Hat-Delta: Hat-Delta[7] [4] belongs to Hat, a set of debugging tools for Haskell. In particular, it is the successor of the algorithmic debugger Hat-Detect.

B.i.O.: B.i.O. (Believe in Oracles)[8] is a debugger integrated in the Curry compiler KICS, and can work as an algorithmic debugger.

Mercury's Algorithmic Debugger: Mercury's compiler[9] has a debugger integrated with both a procedural debugger and an algorithmic debugger [13].

Münster Curry Debugger: The Münster Curry compiler[10] [12] includes an algorithmic debugger.

Nude: The NU-Prolog Debugging Environment (Nude)[11] [15] integrates a collection of debugging tools for NU-Prolog programs.

2.2 Usability Features

After selecting the debuggers to be compared, we wanted to identify discriminating features and dimensions to use in the comparison. To do so, we extensively tested the debuggers with a set of benchmarks from the *Nofib-buggy* benchmarks collection [22] in order to check all the possibilities offered by the debuggers. We also identified some desirable properties of declarative debuggers that are not implemented by any debugger. Some of them have been proposed in related bibliography and others are introduced here. The desirable features of an algorithmic debugger are:

2.2.1 Multiple Search Strategies

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 [21].

Different strategies (see, e.g., Single Stepping [19], Divide & Query [19,8], Top-Down [10], Top-Down Zooming [14], Heaviest First [1], Subterm Dependency Tracking [13] and Dynamic Weighting Search [20]) have arisen to minimize both the number of questions and the time needed to answer the questions. Firstly, the number of questions can be reduced by pruning the ET (e.g., the strategy Divide and Query [19] prunes near half of the ET after every answer). Secondly, the time needed to

[4] http://www.cs.mu.oz.au/∼bjpop/buddha

[5] http://toy.sourceforge.net

[6] http://www.ida.liu.se/∼henni

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

[8] http://www.informatik.uni-kiel.de/prog/mitarbeiter/bernd-brassel/projects/

[9] http://www.cs.mu.oz.au/research/mercury

[10] http://danae.uni-muenster.de/∼lux/curry/

[11] http://www.cs.mu.oz.au/∼lee/papers/nude/

answer the questions can be reduced by avoiding complex questions, or by producing a series of questions which are semantically related (i.e., consecutive questions refer to related parts of the computation). For instance, the strategy Top-Down Zooming tries to ask questions related to the same recursive (sub)computation [14]. A survey of algorithmic debugging strategies can be found in [21].

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

2.2.2 Accepted Answers

Algorithmic debugging strategies are based on the fact that the ET can be pruned using information provided by the oracle. Given a question associated to a node n in the ET, the debugger should be able to accept the following answers from the oracle:

- "Yes" to indicate that the node is correct or valid. In this case, the debugger prunes the subtree rooted at n, and the search continues with the next node according to the selected strategy.
- "No" when the node is wrong or invalid. This answer prunes all the nodes of the ET except the subtree rooted at n, and the search strategy continues with a node in this subtree.
- "Inadmissible" [17] that allows the user to specify that some argument in the atom/predicate or function/method/procedure call associated to 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 [23].
- "Don't Know" to skip a question when the user cannot answer it (e.g., because it is too difficult).
- "Trusted" to indicate that some module, argument or predicate/function/method/ procedure in the program is trusted (e.g., because it has been already tested). All nodes related to a trusted module, argument or predicate/function/method/procedure should also be automatically trusted.

2.2.3 Tracing Subexpressions

A "No" or an "Inadmissible" answer is often too imprecise. When the programmer knows exactly which part of the question is wrong, she could mark a subexpression as wrong. This avoids many useless questions which are not related to the wrong subexpression. For instance, Mercury's debugger [13] uses subexpression information to enhance bug search.

Tracing subexpressions has two main advantages. First, it reduces the search space because the debugger only explores the part of the ET related to the wrong subexpression. Second, it makes the debugging process more understandable, because it gives the user some control over the bug search.

2.2.4 *Tree Compression*

Tree compression is a technique used to remove redundant nodes from the ET [4].

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

Fig. 4. Definition of `append` function.

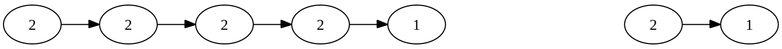

Fig. 5. ET of `append [1,2,3,4] [5,6]` and its associated compressed tree.

Let us illustrate the technique with an example. Consider the function definition in Fig. 4, and its associated ET for the call "`append [1,2,3,4] [5,6]`" in Fig. 5 (left). The function `append` makes three recursive calls to rule `(2)` and, finally, one call to the base case in rule `(1)`. Clearly, there are only two rules that can be buggy: rules 1 and 2. Therefore, the ET can be compressed as depicted in Fig. 5 (right). With this ET, the debugger will only ask at most two questions. Tree compression allows us to remove unnecessary questions before starting the debugging session, hence, it should be implemented by all the debuggers as a postprocess to the computation of the ET.

2.2.5 *Memoization*

The debugger should not ask the same question twice. This can be easily done by memoizing the answers of the programmer. Memoization can be done intra- or inter-session.

2.2.6 *Graphical User Interface*

A GUI can speed up the debugging session, because it allows the user to freely explore the ET and mark nodes independently of (or in parallel with) the running strategy. Graphical features such as collapsing subcomputations of the ET can be very useful.

2.2.7 *Undo Capabilities*

Not only the program can be buggy. Also the debugging session itself could be buggy, e.g., when the programmer answers a question incorrectly. Therefore, the debugger should allow the programmer to correct wrong answers. For instance, a desirable feature is allowing the user to undo the last answer and return to the previous state. Despite it seems to be easy to implement, surprisingly, most algorithmic debuggers lack of an *undo* command; and the programmer is forced to repeat the whole session when she does a mistake with an answer.

Some debuggers drive buggy debugging sessions in a different manner. For instance, Freja uses the answers *maybe yes* and *maybe no* in addition to *yes* and *no*. They are equal to their counterparts except that the debugger will remember that no definitive answer has been given, and return to those questions later unless the bug has been found through other answers first.

2.2.8 ET Exploration

Algorithmic debugging can become too rigid when it is only limited to questions generation. Sometimes, the programmer has an intuition about the part of the ET where the bug can be. In this situation, letting the programmer to freely explore the ET could be the best strategy. It is desirable to provide the programmer with the control over the ET exploration when she wants to direct the search for the bug.

2.2.9 Trusting

Programs usually reuse code already tested (e.g., external functions, modules...). Therefore, when debugging a program, this code should be trusted. Trusting can be done at the level of modules, questions or arguments. And it can be done statically (by including annotations or flags when compiling) or dynamically (The programmer answers a question with “Trusted" and all the questions referring to the same atom/predicate/function/method/procedure are automatically set “Correct").

2.2.10 Scalability

One of the main problems of current algorithmic debuggers is their low scalability. The ETs produced by real programs can be huge (indeed gigabytes) and thus it does not fit in main memory.

Nevertheless, many debuggers store the ET in main memory; hence, they produce a “memory overflow" exception when applied to real programs. Current solutions include storing the ET in secondary memory (e.g., [4]) or producing the ET on demand (e.g., [16,13]).

A mixture between main and secondary memory would be desirable. It would be interesting to load a cluster of nodes in main memory and explore them until a new cluster is needed. This solution would take advantage of the speed acquired by working on main memory (e.g., keeping the possibility to apply strategies on the loaded cluster) while being able to store huge ETs in secondary memory.

A new approach developed in the debugger B.i.O. changes time by space: They do not need to store an ET because they reexecute the program once and again to generate the next question. As they consider a lazy language, they first execute the program and record a file with step counts specifying how much the subcomputations have been evaluated. This file is later used by the back-end to reexecute the program eagerly once and again in order to produce the questions as they are required.

3 Functionality Comparison

Table 3 presents a summary of the available functionalities of the studied algorithmic debuggers. Every column gathers the information of one algorithmic debugger. The meaning of the rows is the following:

- *Implementation Language*: Language used to implement the debugger.
- *Target Language*: Debugged language.

- *Strategies*: Algorithmic debugging strategies supported by the debugger: Top Down (TD), Divide & Query (DQ), Hat-Delta's Heuristics (HD), Mercury's Divide & Query (MD), and Subterm Dependency Tracking (SD).
- *DataBase/Memoization*: Is a database used to store answers for future debugging sessions (inter-session memory)? Are questions remembered during the same session (intra-session memory)?
- *Front-End*: Is it integrated into the compiler or is it standalone? Is it an interpreter/compiler or is it a program transformation?
- *Interface*: Interface used between the front-end and the back-end. If the front-end is a program transformation, then it specifies the data structure generated by the transformed program. Whenever the data structure is stored into the file system, brackets specify the format used. Here, DDT exports the ET in two formats: XML or a TXT. Hat-Delta uses an ART (Augmented Redex Trail) with a native format. B.i.O. generates a list of step counts (see Section 2.2.10) which is stored in a plain text file.
- *Execution Tree*: When the back-end is executed, is the ET stored in the file system or in main memory? This row also specifies which debuggers produce the ET on demand.
- *Accepted Answers*: Yes (YE), No (NO), Don't Know (DK), Inadmissible (IN), Maybe Yes (MY), Maybe Not (MN), and Trusted (TR).
- *Tracing Subexpressions*: Is it possible to specify that a (sub)expression is wrong?
- *ET Exploration*: Is it possible to explore the ET freely?
- *Tree Compression*: Does the debugger implement the tree compression technique?
- *Undo*: Is it possible to undo an answer?
- *Trusting*: Is it possible to trust modules (Mo), functions (Fu) and/or arguments (Ar)?
- *GUI*: Has the debugger a graphical user interface?
- *Version*: Evaluated version of the debugger.

4 Efficiency Comparison

We studied the growing rate of the internal data structure stored by the debuggers. This information is useful to know the scalability of each debugger and, in particular, their limitations with respect to the ET's size.

The study has proved that several debuggers are not usable with real programs because they run out of memory with long running computations. The main reason is that many of them store their ET in memory, and the size of the ET produces a memory overflow as soon as the computation is not medium-size. While the Münster Curry Debugger has the lowest growing rate in the size of its ET, the most scalable debugger is Hat-Delta which successfully passed more benchmarks. The main advantages of the ART used by Hat-Delta is that it is stored in secondary memory, and it shares data structures between different nodes.

In this experiment we have only considered those debuggers which already have a

Feature / Debugger	**B.i.O.**	**DDT**	**Freja**	**Hat-Delta**	**Buddha**	**Mercury Debugger**	**Münster Curry Debugger**	**Nue-Prolog**
Implementation Language	Curry Haskell	Toy (front-end) Java (back-end)	Haskell	Haskell	Haskell	Mercury	Haskell (front-end) Curry (back-end)	Prolog
Target Language	Curry	Toy	Haskell subset	Haskell	Haskell	Mercury	Curry	Prolog
Strategies	TD	TD DQ	TD	HD	TD	TD DQ SD MD	TD	TD
DataBase / Memoization	NO/NO	NO/YES	NO/NO	NO/YES	NO/YES	NO/YES	NO/NO	YES/YES
Front-end	Independent Prog. Tran.	Integrated Prog. Tran.	Integrated Compiler	Independent Prog. Tran.	Independent. Prog. Tran.	Independent Compiler	Integrated Compiler	Independent Compiler
Interface	Steps count (Plain text)	ET (XML/TXT)	ET	ART (Native)	ET	ET on Demand	ET	ET on Demand
Execution Tree	Main Memory on Demand	Main Memory	Main Memory	File System	Main Memory on Demand	Main Memory on Demand	Main Memory	Main Memory on Demand
Accepted answers?	YE NO DN	YE NO DN TR	YE NO MY MN	YE NO	YE NO DN IN TR	YE NO DN IN TR	YE NO	YE NO
Tracing subexpressions?	NO	NO	NO	NO	NO	YES	NO	NO
ET exploration?	YES	YES	YES	YES	YES	YES	NO	NO
Tree compression?	NO	NO	NO	YES	NO	NO	NO	NO
Undo	YES	NO	YES	NO	NO	YES	NO	NO
Trusting	Mo/Fu/Ar	Fu	Mo/Fu	Mo	Mo/Fu	Mo/Fu	Mo/Fu	Fu
GUI	NO	YES	NO	NO	NO	NO	NO	NO
Version	Kics 0.81893 (13.5.2008)	1.1 (2003)	1999-2000	2.05 (22.10.2006)	1.2.1 (01.12.2006)	Mercury 0.13.1 (01.12.2006)	0.9.10 (10.5.2006)	NU-Prolog 1.7.2 (13.07.2004)

stable version and which are still maintained: Buddha, DDT, Hat-Delta and Münster Curry debugger. We have omitted Mercury‘s debugger from this study because the collection of benchmarks used in the study are pure functional programs; and their translation to Mercury produced significant differences in the size and the structure of the ET which made it incomparable to the others.

In order to compare the growing rate of the ET's size of each debugger, we selected some benchmarks from the nofib-buggy suite [22] and we created other benchmarks which are particularly useful for algorithmic debugging because they allow us to produce algorithmic debugging sessions with series of both huge and tiny questions. They also allow us to compare the debuggers with broad and long ETs. All these benchmarks together with the experiment results are publicly available at:

```
http://www.dsic.upv.es/~jsilva/algdeb
```

It is important to note that in this part of the study our objective was not to compare the debuggers against real programs. This is useless, because with the same ET, independently of its size, all the debuggers find the bug with the same number of questions (if they use the same strategy). Our objective was to study the behavior of the debuggers when handling different kinds of ETs. In particular, we produced deep, broad, balanced and unbalanced ETs with different sizes of nodes. With the experiment we were able to know how efficient the debuggers are when storing the ET in memory, and how scalable they are when the size of this ET grows up.

Once we collected the benchmarks for the experiment, we reprogrammed them for all the targeted languages of the debuggers. Finally, we tested the debuggers with a set of increasing input data in order to be able to graphically represent the growing rate of the size of the ET.

Since each debugger handles the ET in a different way, we had to use a different method in each case to measure their size:

- Hat-Delta stores the ART into a file and traverses it during the debugging process. Then, we considered the size of the whole ART rather than the size of the implicit ET. Since the ART is used for other purposes than algorithmic debugging (e.g., tracing) it contains more information than needed.
- DDT saves ETs in two formats: TXT and XML. For our purpose, we used the size of the XML because this is the file loaded by the tool to make the debugging process.
- Münster Curry Debugger generates the ET in main memory. But we applied a patch—provided by Wolfgang Lux—that allows us to store the ET in a text file.
- Buddha also generates the whole ET in main memory. In this case, we used a shell script to measure the physical memory used by the data structures handled by the debugger. It might produce a slightly unfair advantage in that in-memory representation of the ET is likely to be more compact than any other representation stored on disk.

Figure 6 shows the results of one of the experiments. It shows the size of the ET produced by four debuggers when debugging the program *merge-sort*. X-axis represents the number of nodes in the ETs; Y-axis represents the ET's size in Kb; and Z-axis shows the debuggers. In this example we can see that Hat and Buddha supported big ETs without problems. However, DDT and the Münster Curry Debugger were not able to manage ETs with more than (approximately) two and ten thousand nodes respectively. DDT was out of memory and crashed. The Münster Curry Debugger needed a lot of time to generate big ETs, and we stopped it after 6 hours running.

It is important to note that we selected on purpose some benchmarks that process big data structures. The reason is that the size of the nodes is strongly dependent on the size of its data structures, and we wanted to study the growing rate with both small and big input data.

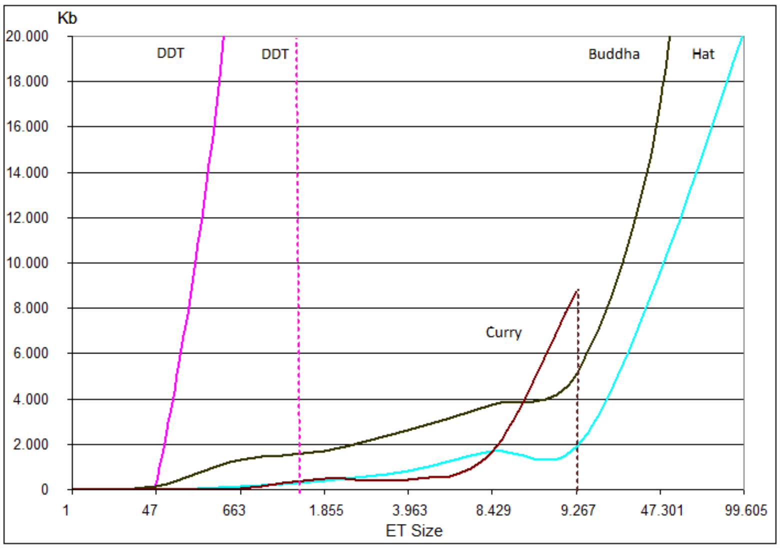

Fig. 6. Growing rate of declarative debuggers for merge-sort.

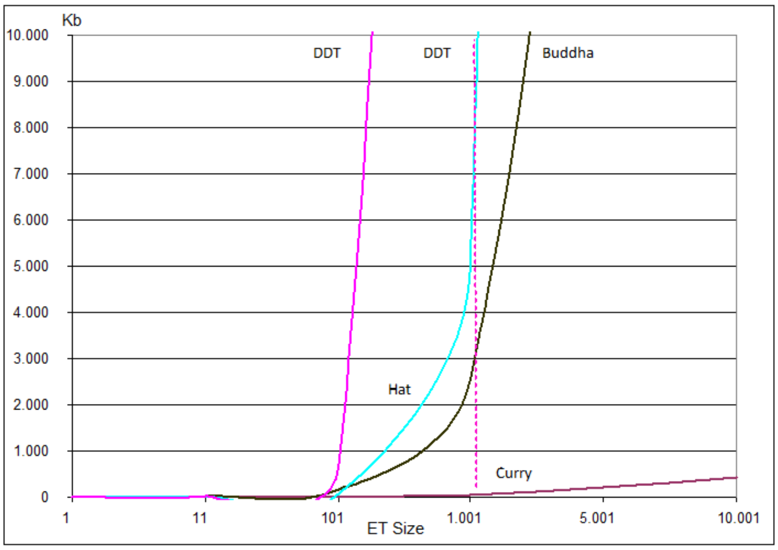

Fig. 7. Growing rate of declarative debuggers for factorial.

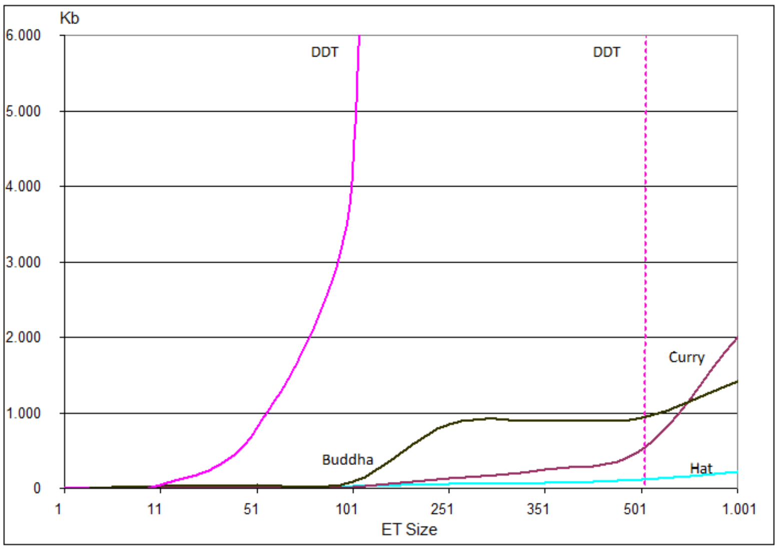

Fig. 8. Growing rate of declarative debuggers for length.

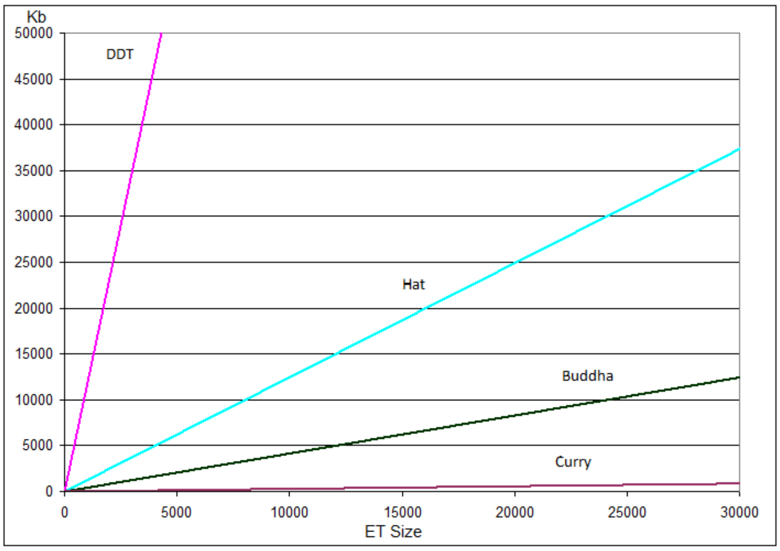

Fig. 9. Linear tendency graph of the growing rate of the ET's size.

Figures 7 and 8 show respectively the results obtained with the program *factorial* and *length*. They show two extreme cases: while, in *factorial* the impact of the input data over the size of the ET's nodes is not significant, in *length* it is very significant (we used lists up to one thousand elements).

Combining all the experiments, we computed the average linear tendency of the ET's size growing rate. It is depicted in Fig. 9.

4.1 Other Debuggers

We did not include in our study those debuggers which do not have a stable version yet. This is the case, for instance, of the declarative debuggers for Java described in [3] and [7]. We contacted with the developers of the Java Interactive Visualization

Environment (JIVE) [6] and the next release will integrate an algorithmic debugger called JavaDD [7]. This tool uses the Java Platform Debugger Architecture to examine the events log of the execution and produce the ET.

We neither considered the debugger GADT [5] (which stands for Generalized Algorithmic Debugging and Testing). Despite it was quite promising because it included a testing and a program slicing phases, its implementation was abandoned in 1992.

The Prolog's debugger GIDTS (Graphical Interactive Diagnosis, Testing and Slicing System) [9] integrated different debugging methods under a unique GUI. Unfortunately, it is not maintained anymore, and thus, it was discarded for the study.

Other debuggers that we discarded are the declarative debugger of the language Escher [11], because it is not maintained since 1998; and the declarative debugger GraDE [1] for the logic programming language Gödel which was abandoned.

We are aware of new versions of the studied debuggers which are currently under development. We did not include them in the study because they are not being distributed yet, and thus, they are not part of the current state of the practice. There are, however, two cases that we want to note:

(i) Hat-Delta: The old algorithmic debugger of Hat was Hat-Detect. Hat-Delta has replaced Hat-Detect because it includes new features such as tree compression and also improved strategies to explore the ET. Nevertheless, some of the old functionalities offered by Hat-Detect have not been integrated in Hat-Delta yet. Since these functionalities were already implemented by Hat-Detect, surely, the next release of Hat-Delta will include them. These functionalities are the following:
   - Strategy Top-Down,
   - undo capabilities,
   - new accepted answers: Maybe Yes, Maybe No, and Trusted, and
   - trusting of functions.

(ii) DDT: There is a new β-version of DDT which includes many new features. These features are the following:
   - New Strategies: Heaviest First, Hirunkitti's Divide & Query, Single Stepping, Hat-Delta's Heuristics, More Rules First, and Divide by Rules & Query,
   - tree compression, and
   - a database for inter-session memoization.

5 Conclusions and Future Work

The main conclusion of the study is that many techniques that have been studied and developed on a theoretical level, have not been implemented and/or integrated into usable algorithmic debuggers.

The functionality comparison has produced a precise description of what features are implemented by each debugger (hence, for each language). From the description, it is easy to see that many features which should be implemented by any algorithmic debugger are only included in one of them. For instance, only the Mercury debugger is able to trace subexpressions, only Hat-Delta implements tree compression, only DDT has a GUI and only it allows the user to graphically explore the ET.

Another important conclusion is that none of the debuggers implement all the ET exploration strategies that appear in the literature. For instance, Hirunkitti's divide and query which is supposed to be the most efficient strategy has not been implemented yet.

Regarding the efficiency comparison, one conclusion is that the main problem of current algorithmic debuggers is not time but space. Their scalability is very low because they are out of memory with long running computations due to the huge growing rate of the ET. In this respect, the more scalable debugger of the four we compared are Hat-Delta (i.e., it supported more debugging sessions than the others) and Buddha. However, the Münster Curry Debugger presents the lower ET's growing rate. It is surprising that much of the debuggers use a file to store the ET, and no debugger uses a database. The use of a database would probably solve the problem of the ET's size. Another technology that is not currently used and needs to be further investigated is the use of a clustering mechanism to allow the debugger exploring (and loading) only the part of the ET (the cluster of nodes) needed at each stage of the debugging session. Currently, no algorithmic debugger uses this technology that could both speed up the debugging session and increase the scalability of the debuggers.

Despite the big amount of problems we have identified, we can also conclude that there is a continuous progression in the development of algorithmic debuggers. This can be easily checked by comparing the dates of the releases studied and their implemented features. Also by comparing different versions of the same debuggers (for instance, from Hat-Detect to Hat-Delta many new functionalities have been added such as tree compression and new search strategies).

The comparison presented here only took into account objective criteria that can be validated. Subjective criteria that can be interpreted were omitted from the study. For instance, we did not talk about how easy to use or to install the debuggers are. This is another quirk of maturity, but we let this kind of comparison for future work.

This paper not only compares current implementations, but it also constitutes a guide to implement a usable algorithmic debugger, because it has established the main functionality and scalability requirements of a real program-oriented debugger.

6 Acknowledgements

The authors greatly thank Bernd Braßel, Sebastian Fisher, Wolfgang Lux, Lee Naish, Henrik Nilsson, and Bernie Pope for providing detailed and useful information about their debuggers.

References

[1] D. Binks. *Declarative Debugging in Gödel.* PhD thesis, University of Bristol, 1995.

[2] R. Caballero. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proc. of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*, pages 8–13, New York, USA, 2005. ACM Press.

[3] 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.

[4] 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.

[5] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimóthy. Generalized Algorithmic Debugging and Testing. *LOPLAS*, 1(4):303–322, 1992.

[6] P.V. Gestwicki and B. Jayaraman. Jive: Java interactive visualization environment. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 226–228, New York, NY, USA, 2004. ACM Press.

[7] H. Girgis and B. Jayaraman. JavaDD: a Declarative Debugger for Java. Technical Report 2006-07, University at Buffalo, March 2006.

[8] 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.

[9] 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.

[10] J. W. Lloyd. Declarative error diagnosis. *New Gen. Comput.*, 5(2):133–154, 1987.

[11] J.W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Computer Science Department, University of Bristol, 1995.

[12] W. Lux. Mï£¡nster curry user's guide (release 0.9.10 of may 10, 2006). Available at: `http://danae.uni-muenster.de/~lux/curry/user.pdf`, 2006.

[13] I. MacLarty. *Practical Declarative Debugging of Mercury Programs.* PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.

[14] M. Maeji and T. Kanamori. Top-Down Zooming Diagnosis of Logic Programs. Technical Report TR-290, ICOT, Japan, 1987.

[15] L. Naish, P. W. Dart, and J. Zobel. The NU-Prolog debugging environment. In Antonio Porto, editor, *Proceedings of the Sixth International Conference on Logic Programming*, pages 521–536, Lisboa, Portugal, June 1989.

[16] H. Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping, Sweden, May 1998.

[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] B. Pope. *A Declarative Debugger for Haskell.* PhD thesis, The University of Melbourne, Australia, 2006.

[19] E.Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1982.

[20] J. Silva. Three New Algorithmic Debugging Strategies. In *Proc. of VI Jornadas de Programación y Lenguajes (PROLE'06)*, pages 243–252, 2006.

[21] J. Silva. A Comparative Study of Algorithmic Debugging Strategies. In *Proc. of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 143–159. Springer LNCS 4407, 2007.

[22] J. Silva. Nofib-buggy: The buggy benchmarks collection of haskell programs). Available at: `http://einstein.dsic.upv.es/darcs/nofib-buggy/`, 2007.

[23] 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.