A Comparative of Algorithmic Debuggers *

Diego Cheda and Josep Silva
DSIC, Technical University of Valencia
Camino de Vera S/N, 46022 Valencia, Spain
{dcheda,jsilva}@dsic.upv.es

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 overview and compare the most important algorithmic debuggers of different programming paradigms. In the study we analyze the most important features incorporated by current algorithmic debuggers, and we identify some features not supported yet by any debugger. We then compare all the debuggers given rise to a map of the state of the practice in algorithmic debugging.

1 Introduction

Algorithmic debugging [18] (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 of 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)[1] [15]—; 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 (e.g., in Buddha where the ET is generated to main memory when executing the back-end), or they are interlazedly 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 which 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, it 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

*This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02, by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054, by LERNet AML/19.0902/97/0666/II-0472-FA and by the *Vicerrectorado de Innovación y Desarrollo de la UPV* under project TAMAT ref. 5771.

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

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.

Once the ET is built, the debugger basically traverses it by using some search strategies [18, 9, 13, 1, 12], 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.

```
(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)
```

Figure 1: Buggy definition of insertion sort

Example 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 found at this node.*

In this work, we first identify a set of desirable features that an algorithmic debugger should implement, and then, we compare the most important algorithmic debuggers to study the state of the practice in algorithmic debugging.

The rest of the paper is organized as follows: In the next section, we first overview the debuggers which participate in the study; and then, we identify some of the desirable features of an algorithmic debugger. Later, in Section 3 we compare the debuggers by studying which features are implemented in which debuggers. This gives rise to a functionality comparison. In Section 4 we compare the debuggers from a different perspective. Our aim here is to study the efficiency of the debuggers by comparing their resources consumption. Section 5 gives some details about other debuggers not included in the study. Finally, Section 6 concludes.

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 eleven algorithmic debuggers, and we selected all of them except those that where not mature enough as to be used in practice (see Section 5). Our objective was not to compare algorithmic debugging-based techniques, but to compare real 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[2] [17] is an algorithmic debugger for Haskell 98 programs.

DDT: DDT[3] [2] is part of the multiparadigm language TOY's distribution [10]. It uses a *graphical user interface* (GUI), imple-

[2] http://www.cs.mu.oz.au/~bjpop/buddha
[3] http://toy.sourceforge.net

```
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)
```

Figure 2: Debugging session for insertion sort

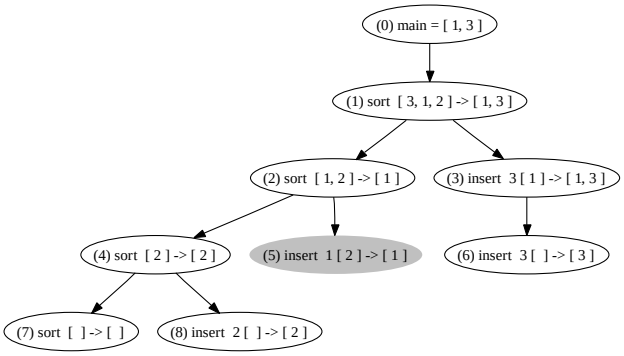

Figure 3: Execution tree of the program in Fig. 1.

mented in Java (i.e., it needs the Java Runtime Environment to work).

Freja: Freja[4] [15] is a debugger implemented in Haskell result of Henric Nilsson's thesis. It is able to debug a subset of Haskell; unfortunately, it is not maintained anymore.

Hat-Delta: Hat-Delta[5] [3] belongs to Hat [23], a set of debugging tools for Haskell. In particular, it is the successor of the algorithmic debugger Hat-Detect.

Kiel's Algorithmic Debugger: Curry [7] is a functional logic language. Very recently, researchers from the University of Kiel have developed an algorithmic debugger for Curry. This algorithmic debugger has the peculiarity that it does not need to store the ET.

Mercury's Algorithmic Debugger: Mercury[6] is a purely declarative logic and functional programming language [12]. Its compiler has a debugger integrated with both a procedural debugger and an algorithmic debugger.

Münster Curry Debugger: One of the most important compilers for Curry is the Münster Curry compiler [11] which includes an algorithmic debugger.

[4] http://www.ida.liu.se/~henni
[5] http://Haskell.org/Hat
[6] http://www.cs.mu.oz.au/research/mercury

Nude: The NU-Prolog Debugging Environment (Nude) [14] integrates a collection of debugging tools for NU-Prolog programs [22].

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 [20] 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 following features are some of the desirable characteristics of an algorithmic debugger.

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 this time 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 [19].

Different strategies ([18, 9, 13, 1, 12]) have arisen to minimize both the number of questions and the time needed to answer the

questions. On the one hand, the number of questions can be reduced by pruning the ET (e.g., the strategy Divide and Query [18] prunes near half of the ET after every answer). On the other hand, the time needed to answer the questions can be reduced by avoiding complex questions, or by producing a series of questions which are semantically related (i.e. consecutive questions refer to related parts of the computation). For instance, the strategy Top-Down Zooming always tries to ask questions related to the same recursive (sub)computation [13]. A survey of algorithmic debugging strategies can be found in [19].

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; and 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 non-valid. 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" [16] 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 [21].
- "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 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 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 unnecessary questions which are not related to the wrong subexpression. For instance, Mercury's debugger [12] allows to trace subexpressions marked as wrong and use this information to guide the search for the bug.

Tracing subexpressions has two main advantages. On the one hand, it reduces the search space because the debugger only explores the part of the ET related to the wrong subexpression. On the other hand, 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 [3].

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

Figure 4: Definition of `append` function.

Figure 5: ET of `append [1,2,3,4] [5,6]`

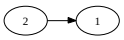

Figure 6: Compressed tree of the program in Fig. 5

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. 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. 6. With this ET, the debugger will only ask at most two questions. The tree compression technique allows 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 avoid the programmer to answer the same question twice. This can be easily done by memoizing the answers of the programmer. The memoization can be done intra- or inter-session.

Before debugging, programs often contain more than one bug. However, algorithmic debuggers only find one bug with each session. Therefore, in order to find more than one bug, the programmer must execute the debugger several times, and thus, the same questions can be repeated in different sessions.

To avoid this situation, the debugger should also store the user's answers in a database and reuse them in future sessions. Hence, in successive debugging sessions of the same program, the same question is not asked once and again.

2.2.6 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.

2.2.7 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.

2.2.8 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 questions or modules. 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").

2.2.9 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., [3]) or producing the ET on demand (e.g., [15, 12]).

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 University of Kiel 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 Comparative

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 independent? 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 with a native format. Kiel's debugger generates a list of step counts (see Section 2.2.9) 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 implements the tree compression technique?
- *Undo*: Is it possible to undo an answer?

- *Trusting*: Is it possible to trust modules (Mod) and/or functions (Fun)?
- *GUI*: Has the debugger a graphical user interface?
- *Version*: Evaluated version of the debugger.

4 Efficiency Comparative

We wanted to study 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, its limitations w.r.t. the ET's size. This study proved that several debuggers are not applicable to real programs because they are out of memory with medium-size programs.

To compare the debuggers, we selected ten benchmarks from the nofib-buggy suite [20]. Then, we reprogrammed the benchmarks 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.

The result of the empirical study produced a very surprising result: most of the debuggers cannot be applicable to real programs. The main reason is that many of them store their ET in main memory, and the size of the ET exceeds the memory size as soon as the size of the computation is not small.

Fig. 7 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.

5 Other Debuggers

We did not include in our study those debuggers which do not have an stable version yet. This is the case, for instance, of JavaDD; a declarative debugger for Java. We contacted with the developers of the Java Interactive Visualization Environment (JIVE) [5]. The next release will integrate an algorithmic debugger for Java programs called JavaDD [6]. This tool uses the Java Platform Debugger Architecture to examine the events log of the execution and produce the ET. This tool has not been included in the study because there is not a stable version yet.

We neither considered the debugger GADT [4] (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) [8] integrated different debugging methods under a unique GUI. Unfortunately, it is not maintained anymore, and thus, it was discarded for the study.

On the other hand, 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 notable cases that we want to note:

1. 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.
   - Trusting of functions.
2. 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,

Feature/Debugger	Buddha	DDT	Freja	Hat Delta	Kiel Debugger	Mercury Debugger	Munster Curry Debugger	Nue-Prolog
Implementation Language	Haskell	Toy (front-end) Java (back-end)	Haskell	Haskell	Curry Prolog Haskell	Mercury	Haskell (front-end) Curry (back-end)	Prolog
Target Language	Haskell	Toy	Haskell subset	Haskell	Curry	Mercury	Curry	Prolog
Strategies	TD	TD DQ	TD	HD	TD	TD DQ SD MD	TD	TD
DataBase / Memoization	NO/YES	NO/YES	NO/NO	NO/YES	NO/NO	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	ET	ET (XML/TXT)	ET	ART (Native)	Steps count (Plain text)	ET on Demand	ET	ET on Demand
Execution Tree	Main Memory	Main Memory	Main Memory on Demand	File System	Main Memory on Demand	Main Memory on Demand	Main Memory	Main Memory on Demand
Accepted answers?	YE NO DN IN TR	YE NO DN TR	YE NO MY MN	YE NO	YE NO DN	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	NO	YES	NO	NO
Tree compression?	NO	NO	NO	YES	NO	NO	NO	NO
Undo	NO	NO	YES	NO	YES	YES	NO	NO
Trusting	Mod/Fun	Fun	Mod/Fun	Mod	NO	Mod/Fun	Mod/Fun	Fun
GUI	NO	YES	NO	NO	NO	NO	NO	NO
Version	1.2.1 (01.12.2006)	1.1 (2003)	1999-2000	2.05 (22.10.2006)	0.1 (10.7.2007)	Mercury 0.13.1 (01.12.2006)	0.9.10 (10.5.2006)	NU-Prolog 1.7.2 (13.07.2004)

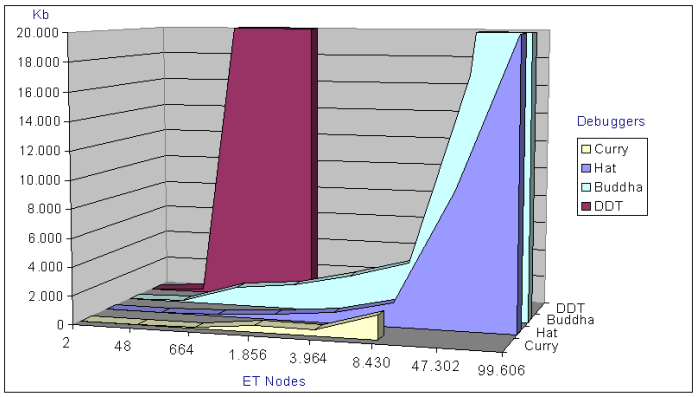

Figure 7: Growing rate of declarative debuggers

More Rules First, and Divide by Rules & Query.

- Tree compression.
- A database for inter-session memoization.

6 Conclusions and Future Work

The main conclusion of the study is that the state of the theory in algorithmic debugging is one step forward than the state of the practice. In general, algorithmic debuggers are maintained by researchers doing a thesis or as a part of their research in a university. We are not aware of any commercial algorithmic debugger.

As a result, researchers implement in their debuggers techniques developed in their university to prove their usefulness, but the general objective of the debuggers is not the debugging of real programs as it is proved by the fact that most of them do not have any mechanism to control the growing size of the ET (e.g., they store the whole ET in main memory).

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

7 Acknowledgements

The authors greatly thank Lee Naish, Henrik Nilsson, Wolfgang Lux, Bernie Pope and Sebastian Fisher 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] 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.

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

[5] 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.

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

[7] M. Hanus. Curry: An Integrated Functional Logic Language (version 0.8.2 of march 28, 2006). Available at: `http://www.informatik.uni-kiel.de/~curry/`, 2006.

[8] 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.

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

[10] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

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

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

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

[14] 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.

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

[16] 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.

[17] B. Pope. Buddha: A declarative debugger for haskell, 1998.

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

[19] J. Silva. Algorithmic debugging strategies. In *Proc. of International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006)*, pages 134–140, 2006.

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

[21] 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.

[22] J. Thom and J. Zobel. Nu-prolog reference manual, version 1.3. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1988.

[23] 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.