A Declarative Debugger for Maude[⋆]

Adrian Riesco, Alberto Verdejo, Narciso Martí-Oliet, and Rafael Caballero

Facultad de Informática, Universidad Complutense de Madrid, Spain

Abstract. Declarative debugging has been applied to many declarative programming paradigms; in this paper, a declarative debugger for rewriting logic specifications, embodied in the Maude language, is presented. Starting from an incorrect computation (a reduction, a type inference, or a rewrite), the debugger builds a tree representing this computation and guides the user through it to find a wrong statement. We present the debugger's main features, such as support for functional and system modules, two possible constructions of the debugging tree, two different strategies to traverse it, use of a correct module to reduce the number of questions asked to the user, selection of trusted vs. suspicious statements, and trusting of statements "on the fly".

1 Introduction

Declarative debugging, introduced by E. Y. Shapiro [8], is a semi-automatic technique that starts from a computation considered incorrect by the user (error symptom) and locates a program fragment responsible for the error. It has been widely employed in the logic [6], functional [7], and multiparadigm [3] programming languages. The declarative debugging scheme uses a *debugging tree* as a logical representation of the computation. Each node in the tree represents the result of a computation step, which must follow from the results of its child nodes by some logical inference. Diagnosis proceeds by traversing the debugging tree, asking questions to an external oracle (generally the user) until a so-called *buggy node* is found. Any buggy node represents a wrong computation step, and the debugger can display the program fragment responsible for it.

Maude [4] is a declarative language based on both equational and rewriting logic for the specification and implementation of a whole range of models and systems. Here we present a declarative debugger for *Maude functional and system modules*. Functional modules define data types and operations on them by means of *membership equational logic* theories that support multiple sorts, subsort relations, equations, and assertions of membership in a sort. Declarative debugging of functional modules has been presented in [2,1]. System modules specify rewrite theories that also support *rules*, defining local concurrent transitions that can take place in a system.

The debugging process starts with an incorrect computation from an initial term. Our debugger, after building a proof tree for that inference, will present to the user questions about the computation. Moreover, since the questions are located in the proof tree, the

[⋆] Research supported by MEC Spanish projects *DESAFIOS* (TIN2006-15660-C02-01) and *MERIT-FORMS* (TIN2005-09027-C03-03), and Comunidad de Madrid program *PROMESAS* (S-0505/TIC/0407).

answer allows the debugger to discard a subset of the questions, leading and shortening the debugging process. The current version of the tool supports all kinds of modules (except for the attribute `strat`), different ways of trusting statements, two possible constructions of the debugging tree for rewritings, and two strategies for traversing it. The debugger is implemented on top of Full Maude [4, Chap. 18]—allowing to debug the different modules provided by it, such as object-oriented and parameterized ones—and exploiting the reflective capabilities of Maude. Complete explanations about the fundamentals and novelties of our debugging approach can be found in the technical report [10], which, together with the source files for the debugger, examples, and related papers, is available from the webpage `http://maude.sip.ucm.es/debugging`.

2 Using the Debugger

We make explicit first what is assumed about the modules introduced by the user; then we present the available commands.

Assumptions. A rewrite theory has an underlying equational theory, containing equations and memberships, which is expected to satisfy the appropriate executability requirements, namely, it has to be terminating, confluent, and sort decreasing. Rules are assumed to be coherent with respect to the equations; for details, see [4].

In our debugger, unlabeled statements are assumed to be correct. Moreover, the user can trust more statements or introduce a correct module to check the inferences. In order to obtain a nonempty abbreviated proof tree, at least the buggy statement must be suspicious; the user is responsible for the correctness of these decisions.

Commands. The debugger is initiated in Maude by loading the file `dd.maude`, which starts an input/output loop that allows the user to interact with the tool. Since the debugger is implemented on top of Full Maude, all modules must be introduced enclosed in parentheses. If a module with correct definitions is used to reduce the number of questions, it must be indicated before starting the debugging with the command `(correct module MODULE-NAME .)`. Since rewriting with rules is not assumed to terminate, a bound, which is 42 by default although can be `unbounded`, is used when searching in the correct module and can be set with the command `(set bound BOUND .)`. The user can debug with only a subset of the labeled statements by using the command `(set debug select on .)`. Once this mode is activated, the user can select and deselect statements by using `(debug [de]select LABELS .)`. Moreover, all the labeled statements of a flattened module can be selected or deselected with the commands `(debug include/exclude MODULES .)`. When debugging rewrites, two different trees can be built: one whose questions are related to one-step rewrites and another one whose questions are related to several steps. The user can switch between these trees with the commands `(one-step tree .)`, which is the default one, and `(many-steps tree .)`, taking into account that the many-steps debugging tree usually leads to shorter debugging sessions (in terms of the number of questions) but with likely more complicated questions. The proof tree can be navigated by using two different strategies: the more intuitive *top-down* strategy, that traverses the tree from the root asking each time for the correctness of all the children of the current node, and then continues with one of the incorrect children; and the more efficient *divide and query* strategy, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, the latter being the default one. The user can switch between them with the commands (top-down strategy .) and (divide-query strategy .). Debugging is started with the following commands for wrong reductions, memberships, and rewrites.[1]

```
(debug [in MODULE-NAME :] INITIAL-TERM -> WRONG-TERM .)
(debug [in MODULE-NAME :] INITIAL-TERM : WRONG-SORT .)
(debug [in MODULE-NAME :] INITIAL-TERM =>* WRONG-TERM .)
```

How the process continues depends on the selected strategy. In case the top-down strategy is selected, several nodes will be displayed in each question. If there is an invalid node, we must select one of them with the command (node N .). If all the nodes are correct, we answer (all valid .). In the divide and query strategy, each question refers to one inference that can be either correct or wrong. The different answers are transmitted with the commands (yes .) and (no .). Instead of just answering yes, we can also *trust* some statements on the fly if, once the process has started, we decide the bug is not there. To trust the current statement we type the command (trust .). Finally, we can return to the previous state by using the command (undo .).

We show in the next sections how to use these commands to debug several examples.

3 Functional Module Example: Multisets

We use sets and multisets to illustrate how to debug functional modules. We describe sets by means of a membership that asserts that a set is a multiset without repetitions. However, the equation mt2 is wrong, because it should add 1 to mult(N, S):

```
    cmb [set] : N S : Set if S : Set /\ mult(N, S) = 0 .
    eq [mt2] : mult(N, N S) = mult(N, S) .
```

If we check now the type of 1 1 2 3 we obtain it is Set! We debug this wrong behavior with the command

```
Maude> (debug 1 1 2 3 : Set .)
```

that builds the associated debugging tree, and selects a node using divide and query:

```
Is this membership (associated with the membership set) correct?
1 2 3  : Set
Maude> (yes .)
```

The debugger continues asking the questions below, now associated to equations:

```
Is this reduction (associated with the equation mt3) correct?
mult(1, 2 3) -> 0
Maude> (yes .)
Is this reduction (associated with the equation mt2) correct?
mult(1, 1 2 3) -> 0
Maude> (no .)
```

[1] If no module name is given, the current module is used by default.

With this information, the debugger finds the wrong statement:

```
The buggy node is: mult(1, 1 2 3) -> 0
With the associated equation: mt2
```

4 System Module Example: Operational Semantics

We illustrate in this section how to debug system modules by means of the semantics of the WhileL language, a simple imperative language described in [5] and represented in Maude in [9]. The syntax of the language includes skip, assignment, composition, conditional statement, and while loop. The state of the execution is kept in the *store*, a set of pairs of variables and values.

Evaluation semantics. The evaluation semantics takes a pair consisting of a command and a store and returns a store.[2] However, we have committed an error in the while loop:

```
 crl [WhileR2] : < While be Do C, st > => < skip, st' >
              if < be, st > => T /\ < C, st > => < skip, st' > .
```

That is, if the condition is true, the body is evaluated only once. Thus, if we execute the program below to multiply x and y and keep the result in z

```
Maude> (rew < z := 0 ; (While Not Equal(x, 0) Do
                        z := z +. y ; x := x -. 1), x = 2 y = 3 z = 1 > .)
result Statement : < skip, y = 3 z = 3 x = 1 >
```

we obtain `z = 3`, while we expected to obtain `z = 6`. We debug this behavior with the top-down strategy and the default one-step tree by typing the commands

```
Maude> (top-down strategy .)
Maude> (debug < z := 0 ; (While Not Equal(x, 0) Do
                           z := z +. y ; x := x -. 1), x = 2 y = 3 z = 1 >
        =>* < skip, y = 3 z = 3 x = 1 > .)
```

The debugger computes the tree and asks about the validity of the root's children:

```
Please, choose a wrong node:
Node 0 : < z := 0, x = 2 y = 3 z = 1 > =>1 < skip, x = 2 y = 3 z = 0 >
Node 1 : < While Not Equal(x,0) Do z := z +. y ; x := x -. 1, x = 2 y = 3 z = 0 >
         =>1 < skip, y = 3 z = 3 x = 1 >
Maude> (node 1 .)
```

The second node is erroneous, because x has not reached 0, so the user selects this node to continue the debugging, and the following question is related to its children:

```
Please, choose a wrong node:
Node 0 : < Not Equal(x,0), x = 2 y = 3 z = 0 > =>1 T
Node 1 : < z := z +. y ; x := x -. 1, x = 2 y = 3 z = 0 >
         =>1 < skip, y = 3 z = 3 x = 1 >
Maude> (all valid .)
```

[2] In order to reuse this module later, the returned result is a pair `< skip, st >`.

Since both nodes are right, the debugger determines that the current node is buggy:

```
The buggy node is:
< While Not Equal(x,0) Do z := z +. y ; x := x -. 1, x = 2 y = 3 z = 0 >
=>1 < skip, y = 3 z = 3 x = 1 >
With the associated rule: WhileR2
```

Computation semantics. In contrast to the evaluation semantics, the computation semantics describes the behavior of programs in terms of small steps. In order to illustrate this, we make a mistake in the rule describing the semantics of the composition, keeping the initial state instead of the new one computed in the condition:

```
 crl [ComRc1] : < C ; C', st > => < C'' ; C', st >
             if < C, st > => < C'', st' > /\ C =/= C'' .
```

If we rewrite now a program to swap the values of two variables, their values are not exchanged. We use the many-steps tree to debug this wrong behavior:

```
Maude> (many-steps tree .)
Maude> (debug < x := x -. y ; y := x +. y ; x := y -. x, x = 5 y = 2 >
           =>* < skip, y = 2 x = 0 > .)
Is this rewrite correct?
< y := x +. y ; x := y -. x, x = 5 y = 2 > =>+ < skip, y = 2 x = 0 >
Maude> (no .)
```

The transition is wrong because the variables have not been properly updated.

```
Is this rewrite (associated with the rule ComRc1) correct?
< y := x +. y ; x := y -. x,x = 5 y = 2 > =>1 < x := y -. x,x = 5 y = 2 >
Maude> (no .)
Is this rewrite (associated with the rule OpR) correct?
< x +. y, x = 5 y = 2 > =>1 7
Maude> (trust .)
```

We consider that the application of a primitive operation is simple enough to be trusted. The next question is related to the application of an equation to update the store

```
Is this reduction (associated with the equation st1) correct?
x = 5 y = 2[7 / y] -> x = 5 y = 7
Maude> (yes .)
```

Finally, a question about assignment is posed:

```
Is this rewrite (associated with the rule AsRc) correct?
< y := x +. y, x = 5 y = 2 > =>1 < skip, x = 5 y = 7 >
Maude> (yes .)
```

With this information, the debugger is able to find the bug. However, since we have the evaluation semantics of the language already specified and debugged, we can use that module as correct module to reduce the number of questions to only one.

```
Maude> (correct module EVALUATION-WHILE .)
Maude> (debug ... .)
Is this rewrite correct?
< y := x +. y ; x := y -. x, x = 5 y = 2 > =>+ < skip, y = 2 x = 0 >
Maude> (no .)
```

5 Conclusions

We have implemented a declarative debugger for Maude modules that allows to debug wrong reductions, type inferences, and rewrites. Although the complexity of the debugging process increases with the size of the proof tree, it does not depend on the total number of statements but on the number of applications of suspicious statements involved in the wrong inference. Moreover, bugs found when reducing complex initial terms can, in general, be reproduced with simpler terms which give rise to smaller proof trees. We plan to improve the interaction with the user by providing a complementary graphical interface that allows the user to navigate the tree with more freedom. This interaction could also be improved by allowing the user to give the answer "don't know," that would postpone the answer to the question by asking alternative questions. We are also studying how to handle the `strat` operator attribute, that allows the specifier to define an evaluation strategy. This can be used to represent some kind of laziness.

References

1. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: A declarative debugger for Maude functional modules. In: Proceedings Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008. Elsevier, Amsterdam (to appear, 2008)
2. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: Declarative debugging of membership equational logic specifications. In: Degano, P., Nicola, R.D., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 174–193. Springer, Heidelberg (2008)
3. Caballero, R., Rodríguez-Artalejo, M.: DDT: A declarative debugging tool for functional-logic languages. In: Proc. 7th International Symposium on Functional and Logic Programming (FLOPS 2004). LNCS, vol. 2998, pp. 70–84. Springer, Heidelberg (2004)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude: A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
5. Hennessy, M.: The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics. John Wiley & Sons, Chichester (1990)
6. Lloyd, J.W.: Declarative error diagnosis. New Generation Computing 5(2), 133–154 (1987)
7. Nilsson, H., Fritzson, P.: Algorithmic debugging of lazy functional languages. Journal of Functional Programming 4(3), 337–370 (1994)
8. Shapiro, E.Y.: Algorithmic Program Debugging. ACM Distinguished Dissertation. MIT Press, Cambridge (1983)
9. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. Journal of Logic and Algebraic Programming 67, 226–293 (2006)
10. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: Declarative debugging of Maude modules. Technical Report SIC-6/08, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2008), `http://maude.sip.ucm.es/debugging`