A declarative debugger for Maude functional modules

R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo

Facultad de Informática, Universidad Complutense de Madrid, Spain

Abstract

A declarative debugger for Maude functional modules, which correspond to executable specifications in membership equational logic, is presented. Starting from an incorrect computation, declarative debugging builds a debugging tree as a logical representation of the computation, that then is traversed by asking questions to an external oracle until the error is found. We summarize the construction of appropriate debugging trees for oriented equational and membership inferences, where all the nodes whose correctness does not need any justification have been collapsed. The reflective features of Maude allow us to generate and navigate the debugging tree of a Maude computation using operations in Maude itself; even the user interface of the declarative debugger can be specified in this way. We present the debugger's main features, such as two different strategies to traverse the debugging tree, use of a correct module to reduce the number of questions asked to the user, selection of trusted vs. suspicious statements by means of labels, and trusting of statements "on the fly."

Keywords: declarative debugging, membership equational logic, Maude, functional modules, metalevel implementation

1 Introduction

Maude is a high-level language and high-performance system supporting both equational and rewriting logic [12] computation for a wide range of applications. In particular, Maude functional modules correspond to specifications in *membership equational logic* (*MEL*) [2,13], which, in addition to equations, allows the statement of *membership assertions* characterizing the elements of a sort. In this way, Maude makes possible the faithful specification of data types (like sorted lists or search trees) whose data are defined not only by means of constructors, but also by the satisfaction of additional properties.

The Maude system supports several approaches for debugging Maude programs: tracing, term coloring, and using an internal debugger [7, Chap. 22]. The tracing facilities allow us to follow the execution on a specification, that is, the sequence of rewrites that take place. Term coloring consists in printing with different colors

⋆ 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).

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

the operators used to build a term that does not fully reduce. The Maude debugger allows the user to define break points in the execution by selecting some operators or statements. When a break point is found the debugger is entered. There, we can see the current term and execute the next rewrite with tracing turned on.

The Maude debugger has as a disadvantage that, since it is based on the trace, it shows to the user every small step obtained by using a single statement. Thus the user can loose the general view of the *proof* of the incorrect inference that produced the wrong result. That is, when the user detects an unexpected statement application it is difficult to know where the incorrect inference started.

Different debugging approaches based on the language's semantics have been introduced in the field of declarative languages, such as *abstract diagnosis*, which formulates a debugging methodology based on abstract interpretation [9,1], or *declarative debugging*, also known as *algorithmic debugging*, which was first introduced by E. Y. Shapiro [19] and that constitutes the framework of this work. Declarative debugging has been widely employed in the logic [10,14,22], functional [21,17,16,18], and multiparadigm programming [5,3,11] languages. Declarative debugging 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. The declarative debugging scheme [15] 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. A buggy node is a node containing an erroneous result, but whose children all have correct results. Hence, a buggy node has produced an erroneous output from correct inputs and corresponds to an erroneous fragment of code, which is pointed out as an error.

During the debugging process, the user does not need to understand the computation operationally. Any buggy node represents an erroneous computation step, and the debugger can display the program fragment responsible for it. From an explanatory point of view, declarative debugging can be described as consisting of two stages, namely the debugging tree *generation* and its *navigation* following some suitable strategy [20].

Here we present a declarative debugger for *Maude functional modules* [7, Chap. 4]. The debugging process starts with an incorrect transition from the initial term to a fully reduced unexpected one. Our debugger, after building a proof tree for that inference, will present to the user questions of the following form: "Is it correct that T fully reduces to T'?", which in general are easier to answer. Moreover, since the questions are located in the proof tree, the answer allows the debugger to discard a subset of the questions, leading and shortening the debugging process.

The current version of the tool has the following characteristics:

- It supports all kinds of functional modules: operators can be declared with any combination of axiom attributes (except for the attribute `strat`, that allows to specify an evaluation strategy); equations can be defined with the `otherwise` attribute; and modules can be parameterized.
- It provides two strategies to traverse the debugging tree: *top-down*, 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 *divide and query*, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise.

- Before starting the debugging process, the user can select a module containing only correct statements. By checking the correctness of the inferences with respect to this module (i.e., using this module as oracle) the debugger can reduce the number of questions asked to the user.
- It allows the user to debug Maude functional modules where some equations and memberships are suspicious and have been labeled (each one with a different label). Only these labeled statements generate nodes in the proof tree, while the unlabeled ones are considered correct. The user is in charge of this labeling. Moreover, the user can answer that he trusts the statement associated with the currently questioned inference; that is, statements can be trusted "on the fly."

Exploiting the fact that rewriting logic is reflective [6,8], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [7, Chap. 14], a feature that makes Maude remarkably extensible and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. Therefore, we are able to generate and navigate the debugging tree of a Maude computation using operations in Maude itself. In addition, the Maude system provides another module, `LOOP-MODE` [7, Chap. 17], which can be used to specify input/output interactions with the user. Thus, our declarative debugger for Maude functional modules, including its user interactions, is implemented in Maude itself. As far as we know, this is the first declarative debugger implemented using such reflective techniques.

Complete explanations about the fundamentals of our declarative debugging approach, additional examples, and more information about the implementation can be found in the technical report [4], which, together with the Maude source files for the debugger, is available from the webpage `http://maude.sip.ucm.es/debugging`.

2 Declarative debugging of Maude functional modules

As mentioned in the introduction, Maude uses membership equational logic [2,13], a very expressive version of equational logic which, in addition to equations, allows the statement of membership assertions characterizing the elements of a sort. We present below how its specifications are represented as Maude functional modules and a brief description of the theoretical basics of our debugger.

2.1 Membership equational logic

A *signature* in *MEL* is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1 \ldots k_n, k}\}_{(k_1 \ldots k_n, k) \in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k \in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. Intuitively, terms with a kind

but without a sort represent undefined or error elements. *MEL* atomic formulas are either *equations* $t = t'$, where t and t' are Σ-terms of the same kind, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. *Sentences* are universally-quantified Horn clauses of the form $(\forall X)\, A_0 \Leftarrow A_1 \wedge \ldots \wedge A_n$, where each A_i is either an equation or a membership assertion. Order-sorted notation $s_1 < s_2$ (with $s_1, s_2 \in S_k$ for some kind k) can be used to abbreviate the conditional membership $(\forall x : k)\, x : s_2 \Leftarrow x : s_1$. A *specification* is a pair (Σ, E), where E is a set of sentences over the signature Σ.

Models of *MEL* specifications are called algebras. A Σ-*algebra* $\mathcal{A}$ consists of a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1 \ldots k_n, k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. The meaning $[\![t]\!]_{\mathcal{A}}$ of a term t in an algebra $\mathcal{A}$ is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies an equation $t = t'$ (or the equation holds in the algebra), denoted $\mathcal{A} \models t = t'$, when both terms have the same meaning: $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. In the same way, satisfaction of a membership is defined as: $\mathcal{A} \models t : s$ when $[\![t]\!]_{\mathcal{A}} \in A_s$. A specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of terms $[t]$. We refer to [2,13] for a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, initial and free algebras, and specification morphisms.

Since the *MEL* specifications that we consider are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness, their equations $t = t'$ can be oriented from left to right, $t \to t'$. Such a statement holds in an algebra, denoted $\mathcal{A} \models t \to t'$, exactly when $\mathcal{A} \models t = t'$, i.e., when $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. Moreover, under those assumptions an equational condition $u = v$ in a conditional equation can be checked by finding a common term t such that $u \to t$ and $v \to t$, that is, $u \downarrow v$. This is the notation we will use in the inference rules and debugging trees studied in Sect. 2.3.

2.2 Representation in Maude

Maude functional modules, introduced with syntax `fmod...endfm`, are executable *MEL* specifications and their semantics is given by the corresponding initial membership algebra in the class of algebras satisfying the specification.

In a functional module we can declare sorts (by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`).

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. Kinds are *not* declared explicitly, and correspond to the connected components of the subsort relation. The kind corresponding to a sort `s` is denoted `[s]`. For example, if we have sorts `Nat` for natural numbers and `NzNat` for nonzero natural numbers with a subsort `NzNat < Nat`, then `[NzNat] = [Nat]`.

An operator declaration like

```
op _div_ : Nat NzNat -> Nat .
```

is logically understood as a declaration at the kind level

```
op _div_ : [Nat] [Nat] -> [Nat] .
```

together with the conditional membership axiom

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

A subsort declaration `NzNat < Nat` is logically understood as the conditional membership axiom

```
cmb N : Nat if N : NzNat .
```

2.2.1 An example: binary search trees

As an example of Maude functional modules, we show how to specify binary search trees without repeated elements, whose nodes contain elements that satisfy the theory `STOSET` (defining a strict total order on them) [7, Sect. 8.3].

```
fmod SEARCH-TREE{X :: STOSET} is
 sorts NeSearchTree{X} SearchTree{X} Tree{X} .
 subsorts NeSearchTree{X} < SearchTree{X} < Tree{X} .
 op empty : -> SearchTree{X} [ctor] .
 op ___ : Tree{X} X$Elt Tree{X} -> Tree{X} [ctor] .
```

where the operation for building non-empty search trees uses juxtaposition and `X$Elt` denotes the sort `Elt` from the theory `STOSET`.

A tree is a search tree when its root is bigger than all the elements in the left subtree and smaller than all the elements in the right subtree; this requirement is specified by means of memberships. Assuming that the subtrees are search trees, instead of comparing with all their elements, it is enough to compare with the minimum or maximum of the appropriate subtree.

```
 vars E E' : X$Elt .
 vars L R : SearchTree{X} .
 vars L' R' : NeSearchTree{X} .
 mb [leaf]     : empty E empty : NeSearchTree{X} .
 cmb [1ch1] : L' E empty    : NeSearchTree{X} if max(L') < E .
 cmb [1ch2] : empty E R'    : NeSearchTree{X} if E < min(R') .
 cmb [2ch]  : L' E R'       : NeSearchTree{X}
  if max(L') < E /\ E < max(R') .
 ops min max : NeSearchTree{X} -> X$Elt .
 ceq [mn1] : min(empty E R) = E if empty E R : NeSearchTree{X} .
 ceq [mn2] : min(L' E R)    = min(L') if L' E R : NeSearchTree{X} .
 ceq [mx1] : max(L E empty) = E if L E empty : NeSearchTree{X} .
 ceq [mx2] : max(L E R')    = max(R') if L E R' : NeSearchTree{X} .
```

The `delete` operation is specified as usual by structural induction, and in the non-empty case by comparing the element to be deleted with the root of the tree and distinguishing the three cases according to whether the former is smaller than, equal to, or bigger than the latter.

```
 op delete : SearchTree{X} X$Elt -> SearchTree{X} .
 eq  [dl1] : delete(empty, E)     = empty .
 ceq [dl2] : delete(L E R, E')    = delete(L, E') E R
             if E' < E /\ L E R : NeSearchTree{X} .
 ceq [dl3] : delete(L E R, E')    = L E delete(R, E')
             if E < E' /\ L E R : NeSearchTree{X} .
 ceq [dl4] : delete(empty E R, E) = R if empty E R : NeSearchTree{X} .
 ceq [dl5] : delete(L E empty, E) = L if L E empty : NeSearchTree{X} .
 ceq [dl6] : delete(L' E R', E)   = L' E' delete(R', E)
             if E' := min(R') /\ L' E R' : NeSearchTree{X} .
```

(**Reflexivity**) $$\frac{}{e \to e}\ (Rf)$$

(**Transitivity**) $$\frac{e_1 \to e' \quad e' \to e_2}{e_1 \to e_2}\ (Tr)$$

(**Congruence**) $$\frac{e_1 \to e_1' \quad \dots \quad e_n \to e_n'}{f(e_1, \dots, e_n) \to f(e_1', \dots, e_n')}\ (Cong)$$

(**Subject Reduction**) $$\frac{e \to e' \quad e' : s}{e : s}\ (SRed)$$

(**Membership**) $$\frac{\{\theta(u_i) \downarrow \theta(u_i')\}_{1 \le i \le n} \quad \{\theta(v_j) : s_j\}_{1 \le j \le m}}{\theta(e) : s}\ (Mb)$$
if $e : s \Leftarrow u_1 = u_1' \wedge \cdots \wedge u_n = u_n' \wedge v_1 : s_1 \wedge \cdots \wedge v_m : s_m$

(**Replacement**) $$\frac{\{\theta(u_i) \downarrow \theta(u_i')\}_{1 \le i \le n} \quad \{\theta(v_j) : s_j\}_{1 \le j \le m}}{\theta(e) \to \theta(e')}\ (Rep)$$
if $e \to e' \Leftarrow u_1 = u_1' \wedge \cdots \wedge u_n = u_n' \wedge v_1 : s_1 \wedge \cdots \wedge v_m : s_m$

Fig. 1. Semantic calculus for Maude functional modules

```
endfm
```

This specification could be completed with other operations for insertion and look up.

Now we can instantiate this module with the predefined module `INT` of integer numbers, and reduce the following term

```
Maude> red delete((empty 1 empty) 2 ((empty 4 empty) 5 (empty 6 (empty 7 empty))), 5) .
result NeSearchTree{Int}:
   (empty 1 empty) 2 ((empty 4 empty) 6 (empty 6 (empty 7 empty)))
```

We obtain a *tree with repetitions*. Moreover, Maude infers that *it is a search tree!* Did you notice the bugs? We will show in Sect. 3.3 how to use the debugger to detect them.

2.3 Declarative debugging

The inference rules of the calculus defining the operational semantics can be found in Fig. 1. They are an adaptation to the case of Maude functional modules of the deduction rules for *MEL* presented in [2]. We assume the existence of an *intended interpretation* $\mathcal{I}$ of the specification, which is a Σ-algebra corresponding to the model that the user had in mind while writing the statements E. The user expects that $\mathcal{I} \models e \to e', \mathcal{I} \models e : s$ for each reduction $e \to e'$ and membership $e : s$ computed w.r.t. the specification (Σ, E).

We will say that $e \to e'$ (respectively $e : s$) is *valid* when it holds in $\mathcal{I}$, and *invalid* otherwise. Declarative debuggers rely on some external oracle, normally the user, in order to obtain information about the validity of some nodes in the debugging tree. The concept of validity can be extended to distinguish *wrong equations* and *wrong membership axioms*, which are those specification pieces that can deduce something invalid from valid information.

It will be convenient to represent deductions in the calculus as *proof trees*, where the premises are the child nodes of the conclusion at each inference step. In declarative debugging we are specially interested in *buggy nodes* which are invalid nodes

with all its children valid. Our goal is to find a buggy node in any proof tree T rooted by the initial error symptom detected by the user. This could be done simply by asking questions to the user about the validity of the nodes in the tree according to the following *top-down* strategy:

Input: A tree T with an invalid root.
Output: A buggy node in T.
Description: Consider the root N of T. There are two possibilities:
- If all the children of N are valid, then finish identifying N as buggy.
- Otherwise, select the subtree rooted by any invalid child and use recursively the same strategy to find the buggy node.

Proving that this strategy is complete is straightforward by using induction on the height of T. As a consequence, if T is a proof tree with an invalid root, then there exists a buggy node $N \in T$ such that all the ancestors of N are invalid.

However, we will not use the proof tree T as debugging tree, but a suitable abbreviation which we denote by $APT(T)$ (from *Abbreviated Proof Tree*). In order to simplify the proof trees we take advantage of a property that every Σ-algebra $\mathcal{A}$ satisfies: if $e \to e'$ (respectively $e : s$) can be deduced by any of the first four inference rules of the calculus using premises that hold in $\mathcal{A}$, then $\mathcal{A} \models e \to e'$ (respectively $\mathcal{A} \models e : s$). This property cannot be extended to the *membership* and *replacement* inference rules, where the correctness of the conclusion depends not only on the calculus but also on the associated specification statement, which could be wrong. Therefore the only inferences that can obtain an invalid conclusion from valid premises, i.e., the only possible buggy nodes, correspond to the *replacement* and *membership* inferences. The $APT(T)$ keeps only the nodes corresponding to these inferences. Fig. 2 shows the definition of $APT(T)$, where the T_i represent proof trees corresponding to the premises in some inferences.

The rule APT_1 keeps the root unaltered and employs the auxiliary function APT' to produce the child subtrees. APT' is defined in rules $APT_2 \ldots APT_8$. It takes a proof tree as input parameter and returns a forest $\{T_1, \ldots, T_n\}$ of APTs as result. The rules for APT' are assumed to be tried top-down, in particular APT_4 must not be applied if APT_3 is also applicable. It is easy to check that every node $N \in T$ conclusion of a *replacement* or *membership* inference has its corresponding node $N' \in APT(T)$ labeled with the same abbreviation, and conversely, that for each $N' \in APT(T)$ different from the root, there is a node $N \in T$, which is the conclusion of a *replacement* or *membership* inference. In particular the node associated to $e_1 \to e_2$ in the righthand side of APT_3 is the node $e_1 \to e'$ of the proof tree T, which is not included in the $APT(T)$. We have chosen to introduce $e_1 \to e_2$ instead of simply $e_1 \to e'$ in the $APT(T)$ as a pragmatic way of simplifying the structure of the APTs, since e_2 is obtained from e' and hence likely simpler (the root of the tree T_{n+1} in APT_3 must be necessarily of the form $e' \to e_2$ by the structure of the inference rule for transitivity in Fig. 1).

Although $APT(T)$ is no longer a proof tree we keep the inference labels (*Rep*) and (*Mb*), assuming implicitly that they contain a reference to the equation or membership axiom used at the corresponding step in the original proof trees. This information is used by the debugger in order to single out the incorrect fragment of

$$(\mathbf{APT}_1)\ APT\left(\frac{T_1 \dots T_n}{af}(R)\right) = \frac{APT'\left(\frac{T_1 \dots T_n}{af}(R)\right)}{af} \text{ (with } (R) \text{ any inference rule)}$$

$$(\mathbf{APT}_2)\ APT'\left(\frac{}{e \to e}(Rf)\right) = \emptyset$$

$$(\mathbf{APT}_3)\ APT'\left(\frac{\frac{T_1 \dots T_n}{e_1 \to e'}(Rep) \quad T_{n+1}}{e_1 \to e_2}(Tr)\right) = \left\{\frac{APT'(T_1) \dots APT'(T_n)\ APT'(T_{n+1})}{e_1 \to e_2}(Rep)\right\}$$

$$(\mathbf{APT}_4)\ APT'\left(\frac{T_1 \quad T_2}{e_1 \to e_2}(Tr)\right) = \{APT'(T_1),\ APT'(T_2)\}$$

$$(\mathbf{APT}_5)\ APT'\left(\frac{T_1 \dots T_n}{e_1 \to e_2}(Cong)\right) = \{APT'(T_1), \dots, APT'(T_n)\}$$

$$(\mathbf{APT}_6)\ APT'\left(\frac{T_1 \quad T_2}{e : s}(SRed)\right) = \{APT'(T_1),\ APT'(T_2)\}$$

$$(\mathbf{APT}_7)\ APT'\left(\frac{T_1 \dots T_n}{e : s}(Mb)\right) = \left\{\frac{APT'(T_1) \dots APT'(T_n)}{e : s}(Mb)\right\}$$

$$(\mathbf{APT}_8)\ APT'\left(\frac{T_1 \dots T_n}{e_1 \to e_2}(Rep)\right) = \left\{\frac{APT'(T_1) \dots APT'(T_n)}{e_1 \to e_2}(Rep)\right\}$$

Fig. 2. Transformation rules for obtaining abbreviated proof trees

specification code. The abbreviation of the tree reduces and simplifies the questions that will be asked to the user while keeping the soundness and completeness of the technique, as the following theorem (proved in [4]) guarantees:

Theorem 2.1 *Let S be a specification, $\mathcal{I}$ its intended interpretation, and T a finite proof tree with invalid root. Then:*

- *$APT(T)$ contains at least one buggy node (completeness).*
- *Any buggy node in $APT(T)$ has an associated wrong statement in S (soundness).*

The theorem states that we can safely employ the abbreviated proof tree as a basis for the declarative debugging of Maude functional modules: the technique will find a buggy node starting from any initial symptom detected by the user. Of course, these results assume that the user answers correctly all the questions about the validity of the APT nodes asked by the debugger (see Sect. 3.1).

The APT for the wrong membership inference of Sect. 2.2.1 is shown in Fig. 3, where ▲ denotes the `empty` search tree and ▽ represents the already depicted proof subtree with root `max(▲ 6 (▲ 7 ▲))` → `7`.

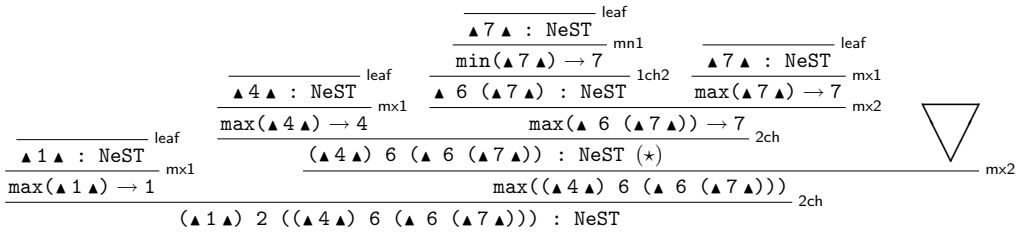

Fig. 3. Abbreviated proof tree

3 Using the debugger

Before describing the basics of the user interaction with the debugger, we make explicit what is assumed about the modules introduced by the user; then we present the available commands and how to use them to debug the buggy example introduced in Sect. 2.2.1.

3.1 Assumptions

Since we are debugging Maude functional modules, they are expected to satisfy the appropriate executability requirements, namely, the specifications have to be terminating, confluent, and sort decreasing.

One interesting feature of our tool is that the user is allowed to trust some statements, by means of labels applied to the suspicious statements. This means that the unlabeled statements are assumed to be correct. A trusted statement is treated in the implementation as the first four rules in Fig. 1 are treated in the *APT* transformation; more specifically, an instance of the *membership* or *replacement* inference rules corresponding to a trusted statement is collapsed in the abbreviated proof tree. In order to obtain a nonempty abbreviated proof tree, the user must have labeled some statements (all with different labels); otherwise, everything is assumed to be correct. In particular, the buggy statement must be labeled in order to be found. When not all the statements are labeled, the correctness and completeness results shown in Sect. 2.3 are conditioned by the goodness of the labeling for which the user is responsible.

Although the user can introduce a module importing other modules, the debugging process takes place in the *flattened* module. However, the debugger allows the user to trust a whole imported module.

As already mentioned, navigation of the debugging tree takes place by asking questions to an external oracle, which in our case is either the user or another module introduced by the user. In both cases the answers are assumed to be correct. If either the module is not really correct or the user provides an incorrect answer, the result is unpredictable. Notice that the information provided by the correct module need not be complete, in the sense that some functions can be only partially defined.

3.2 Commands

The debugger is initiated in Maude by loading the file `fdd.maude`, which starts an input/output loop that allows the user to interact with the tool.

As we said in the introduction, the generated proof tree can be navigated by

using two different strategies, namely, *top-down* and *divide and query*, being the latter the default one. The user can switch between them by using the commands `(top-down strategy .)` and `(divide-query strategy .)`. 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 .)`. Moreover, the user can trust all the statements in several modules with the command `(trust[*] MODULE-NAMES-LIST .)` where `*` means that modules are considered flattened.

Once we have selected the strategy and, optionally, the module above, we start the debugging process with the command [1]

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

If we want to debug only with a subset of the labeled statements, we use the command

```
(debug [in MODULE-NAME :] INITIAL-TERM -> WRONG-TERM with LABELS .)
```

where `LABELS` is the set of suspicious equation and membership labels that must be taken into account when computing the debugging tree. In the same way, we can debug a membership inference with the commands

```
(debug [in MODULE-NAME :] INITIAL-TERM : WRONG-SORT .)
(debug [in MODULE-NAME :] INITIAL-TERM : WRONG-SORT with LABELS .)
```

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 .)`, where `N` is the identifier of that wrong node. If all the nodes are correct, we type `(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 to the debugger 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 in both strategies by using the command `(undo .)`.

3.3 Binary search trees revisited

We recall from Sect. 2.2.1 that the deletion in our binary search trees specification is incorrect. In particular Maude assigns the sort `NeSearchTree{Int}` to a tree with repetitions. We can debug the inference of this membership with the command

```
Maude> (debug in SEARCH-TREE-TEST :
(empty 1 empty) 2 ((empty 4 empty) 6 (empty 6 (empty 7 empty))) : NeSearchTree{Int} .)
```

that generates the debugging tree shown in Fig. 3 by considering all the labeled statements as suspicious. Since the default navigation strategy is divide and query, the debugger selects the node marked with $\star$ in the figure, and then asks the following question:

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

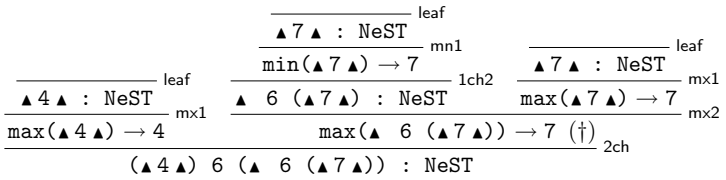

Fig. 4. Abbreviated proof tree after the first question

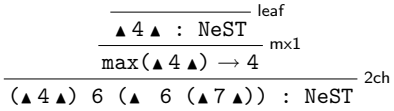

Fig. 5. Abbreviated proof tree after the second question

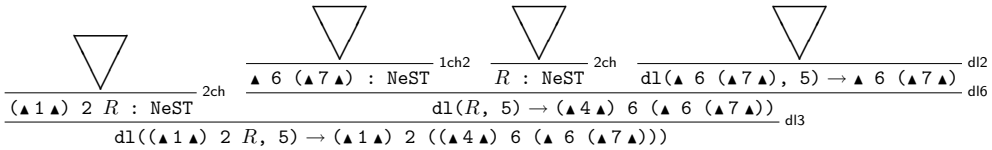

Fig. 6. Abbreviated proof tree for the top down strategy

```
Is this membership (associated with the membership 2ch) correct?
(empty 4 empty) 6 (empty 6 (empty 7 empty)) : NeSearchTree{Int}
Maude> (no .)
```

Since the answer is `no`, the debugger discards the rest of the tree and focuses in the subtree with this node as root (see Fig. 4). The next question corresponds to the node marked with † in the figure.

```
Is this transition (associated with the equation mx2) correct?
max(empty 6 (empty 7 empty)) -> 7
Maude> (yes .)
```

When the answer is `yes` the corresponding subtree is deleted, obtaining in this case the tree in Fig. 5. The next question is

```
Is this transition (associated with the equation mx1) correct?
max(empty 4 empty) -> empty 4 empty
Maude> (trust .)
```

In the last question, we realized that the equation applied is so simple that we can *trust* it. This answer has a behavior similar to `yes`: it deletes all the subtrees whose root is labeled as the current statement. With these answers, we obtain a tree with only one node and the debugger is able to conclude which is the buggy membership.

```
The buggy node is:
(empty 4 empty) 6 (empty 6 (empty 7 empty)) : NeSearchTree{Int}
With the associated membership: 2ch
```

In fact, if we check now this membership we notice that it compares the root with the biggest value of the right subtree, when it should be compared with the smallest one. After fixing this error, the `delete` function is still incorrect, so we debug this function (using the top-down strategy for illustration's sake) as follows:

```
Maude> (top-down strategy .)
Maude>
  (debug in SEARCH-TREE-TEST :
  delete((empty 1 empty) 2 ((empty 4 empty) 5 (empty 6 (empty 7 empty))), 5)
```

```
      -> (empty 1 empty) 2 ((empty 4 empty) 6 (empty 6 (empty 7 empty)))
 with leaf 1ch1 1ch2 2ch dl2 dl3 dl4 dl5 dl6 .)
```

where we have decided to mark as suspicious the memberships and the non-trivial equations of `delete`. In this case, the debugger builds the proof tree (partially) shown in Fig. 6 (where R denotes the search tree `(▲ 4 ▲) 5 (▲ 6 (▲ 7 ▲))`), so it asks the following questions:

```
Is any of these nodes wrong?
Node 0 : (empty 1 empty) 2 ((empty 4 empty) 5 (empty 6 (empty 7 empty))) :
       NeSearchTree{Int}
Node 1 : delete((empty 4 empty) 5 (empty 6 (empty 7 empty)), 5) ->
       (empty 4 empty) 6 (empty 6 (empty 7 empty))
Maude> (node 1 .)

Is any of these nodes wrong?
Node 0 : empty 6 (empty 7 empty) : NeSearchTree{Int}
Node 1 :(empty 4 empty) 5 (empty 6 (empty 7 empty)) : NeSearchTree{Int}
Node 2 : delete(empty 6 (empty 7 empty), 5) -> empty 6 (empty 7 empty)
Maude> (all valid .)

The buggy node is:
delete((empty 4 empty) 5 (empty 6 (empty 7 empty)), 5) ->
    (empty 4 empty) 6 (empty 6 (empty 7 empty))
With the associated equation: dl6
```

The debugger concludes that the problem is within the equation `dl6`. We leave to the interested reader the task of fixing it.

4 Implementation

As we said in the introduction, the reflective power of Maude allows us to generate and navigate the debugging tree of a computation in Maude itself. Since navigation is done by asking questions to the user, this stage has to handle the navigation strategy together with the input/output interaction with the user. Indeed, this interaction can also be implemented in Maude by using the predefined module `LOOP-MODE` [7, Chap. 17], that handles the input/output and maintains the persistent state of the tool.

Below we show the main functions involved in the implementation; the technical report [4] provides a full explanation of the complete implementation, including the user interaction.

4.1 Debugging tree construction

To build the debugging tree we use the facts that the equations defined in Maude functional modules are both *terminating* and *confluent*. Instead of creating the complete proof tree and then abbreviating it, we build the abbreviated proof tree directly. The information kept in each node corresponds to an inference, represented by a statement's label, its lefthand side (a term), and its righthand side (either a term or a sort).

The function `createTree` controls the construction of this tree (it implements the function APT from Fig. 2). It receives the module where a suspicious inference took place, a correct module (or the constant `maybe` when no such module is pro-

vided) to prune the tree, the term initially reduced, the (erroneous) result obtained, and the set of suspicious statement labels. It keeps the initial reduction as the root of the tree and uses an auxiliary function `createForest` (implementing the function APT' from Fig. 2) that, in addition to the arguments received by `createTree`, receives the module "cleaned" of suspicious statements (by using `transform`), and generates the forest of abbreviated trees corresponding to the reduction between the two terms given as arguments. This transformed module is used to improve the efficiency of the tree construction, because we can use it to check if a term reaches its final form by using only trusted equations, thus avoiding to build a tree that will be finally empty.

```
op createTree : Module Maybe{Module} Term Term QidSet -> Tree .
ceq createTree(M, CM, T, T', QS) = contract(
             tree(node('root : T -> T', getOffspring*(F) + 1), F))
 if M' := transform(M, QS) /\
    F := createForest(M, M', CM, normal(M, T), normal(M, T'), QS) .
```

where `contract` prunes the root of the tree if it is duplicated after the computation of the tree.

We use the function `createForest` to create a forest of abbreviated trees. This function checks if the terms can be reduced by using only trusted statements or if the correct module can compute this reduction; in such cases, there is no need to compute the forest. Otherwise, it works with the same innermost strategy as the Maude interpreter: It first tries to fully reduce the subterms (by means of the function `reduceSubterms`), and once all the subterms have been reduced, if the result is not the final one, it tries to reduce at the top (by using the function `applyEq`), to reach the final result by *transitivity*.

```
op createForest : Module Module Maybe{Module} Term Term QidSet -> Forest .
ceq createForest(OM, TM, CM, T, T', QS) = mtForest
 if reduce(TM, T) == T' or-else reduce(M, T) == reduce(M, T') .
ceq createForest(OM, TM, CM, T, T', QS) =
                          if T'' == T' then F
                          else F applyEq(OM, TM, CM, T'', T', QS)
                          fi
 if < T'', F > := reduceSubterms(OM, TM, CM, T, QS) [owise] .
```

The `reduceSubterms` function returns a pair consisting of the term with its subterms fully reduced (that is, this function mimics a specific behavior of the *congruence* rule in Fig. 1) and the forest of abbreviated trees generated by these reductions.

The function `applyEq` tries to apply (at the top) one equation,[2] by using the *replacement* rule from Fig. 1, with the constraint that we cannot apply equations with the `otherwise` attribute while other equations can be applied. To apply an equation we check if the term we are trying to reduce matches the lefthand side of the equation and its conditions are fulfilled. If this happens, we obtain a substitution (from both the matching with the lefthand side and the matching conditions) that we can apply to the righthand side of the equation. Note that if we can obtain the transition in the correct module, the forest is not calculated.

[2] Since the module is assumed to be confluent, we can choose any equation and the final result should be the same.

```
op applyEq : Module Module Maybe{Module} Term Term QidSet -> Forest .
op applyEq : Module Module Maybe{Module} Term Term QidSet EquationSet -> Forest .
ceq applyEq(OM, TM, M, T, T', QS) = mtForest
 if reduce(M, T) == reduce(M, T') .
eq applyEq(OM, TM, CM, T, T', QS) =
          applyEq(OM, TM, CM, T, T', QS, getEqs(OM)) [owise] .
```

First, we try to apply the equations without the `otherwise` attribute. Otherwise, we check the other equations.

```
ceq applyEq(OM, TM, CM, T, T', QS, Eq EqS) =
        if in?(AtS, QS)
        then tree(node(label(AtS) : T -> T', getOffspring*(F) + 1), F)
        else F
        fi
 if ceq L = R if C [AtS] . := generalEq(Eq) /\
    not owise?(AtS) /\
    SB := metaMatch(OM, L, T, C, 0) /\
    R' := normal(OM, substitute(R, SB)) /\
    F := conditionForest(substitute(C, SB), OM, TM, CM, QS)
         createForest(OM, TM, CM, R', T', QS) .
```

where the function `in?` checks if the equation `Eq` is suspicious. If this is the case, a new node corresponding to the applied equation is generated. The forest for the conditions is generated by the function `conditionForest`; since it is used after having checked that the condition is fulfilled (by the function `metaMatch` above), it does not check it again. It distinguishes between the different types of conditions. If the condition is an equation, trees of the reduction of the terms to their normal forms are generated.

```
op conditionForest : Condition Module Module Maybe{Module} QidSet -> Forest .
eq conditionForest(T = T' /\ COND, OM, TM, CM, QS) =
              createForest(OM, TM, CM, T, reduce(OM, T), QS)
              createForest(OM, TM, CM, T', reduce(OM, T), QS)
              conditionForest(COND, OM, TM, CM, QS) .
```

The case of the matching conditions is very similar. In the membership case, we use the version of `createForest` that builds a forest for a membership inference where the sort is the least one assignable to the term in the condition.

```
eq conditionForest(T : S /\ COND, OM, TM, CM, QS) =
              createForest(OM, TM, CM, T, type(OM, T), QS)
              conditionForest(COND, OM, TM, CM, QS) .
```

To generate the forest for memberships we use another version of the function `createForest`, that mimics the *subject reduction* rule from Fig. 1 by first computing the tree for the full reduction of the term (by means of `createForest`) and then computing the tree for the membership inference by using an auxiliary version of `createForest` that uses the operator declarations and the membership axioms. Note that if we can infer the type from the correct module, there is no need to calculate the forest.

```
op createForest : Module Module Maybe{Module} Term Sort QidSet -> Forest .
op createForest : Module Module Maybe{Module} Term Sort QidSet OpDeclSet
                  MembAxSet -> Forest .
ceq createForest(OM, TM, CM, T, S, QS) = mtForest
 if Ty := type(CM, T) /\ sortLeq(CM, Ty, S) .
ceq createForest(OM, TM, CM, T, S, QS) =
      createForest(OM, TM, CM, T, T', QS)
```

```
      createForest(OM, TM, CM, T', S, QS, getOps(OM), getMbs(OM))
  if T' := reduce(OM, T) [owise] .
```

The auxiliary `createForest` computes a forest for a membership inference of the least sort of a term previously fully reduced; this corresponds to a concrete application of the *membership* inference rule from Fig. 1. It first checks if the membership has been inferred by using the operator declarations. If the membership has not been computed by using these declarations, it checks the membership axioms.

To check the operators we examine that both the arity and co-arity of the term and the declaration fit (with function `checkTypes`) and recursively calculate the forest generated by the subterms (by using `createForest*`). Notice that we never generate a new node for the application of an operator, because we always trust the signature.

```
op applyOp : Module Module Maybe{Module} Term Sort QidSet OpDeclSet -> Maybe{Forest} .
ceq applyOp(OM, TM, CM, Q[TL], Ty, QS, op Q : TyL -> Ty [AtS] . ODS) =
             createForest*(OM, TM, CM, TL, QS)
 if checkTypes(TL, TyL, OM) .
ceq applyOp(OM, TM, CM, CONST, S, QS, op Q : nil -> Ty [AtS] . ODS) = mtForest
 if getName(CONST) = Q /\ getType(CONST) = Ty .
eq applyOp(OM, TM, CM, T, S, QS, ODS) = noProof [owise] .
```

We check the membership axioms in a similar fashion to the equation application, that is, we only generate a new root below the forest for the conditions if the membership is suspicious. The unconditional axioms generate leaves of the tree, while the conditional ones generate nodes with (possibly) non-empty forests.

```
op applyMb : Module Module Maybe{Module} Term Sort QidSet MembAxSet -> Forest .
ceq applyMb(OM, TM, CM, T', S, QS, MA MAS) =
        if in?(AtS, QS)
        then tree(node(label(AtS) : T' : S, getOffspring*(F) + 1), F)
        else F
        fi
 if cmb T : S if C [AtS] . := generalMb(MA) /\
    SB := metaMatch(OM, T, T', C, 0) /\
    F := conditionForest(substitute(C, SB), OM, TM, CM, QS) .
eq applyMb(OM, TM, CM, T, S, QS, MA) = mtForest [owise] .
```

4.2 Debugging tree navigation

Regarding the navigation of the debugging tree, we have implemented two strategies. In the top-down strategy the selection of the next node of the debugging tree is done by the user, thus we do not need any function to compute it. The divide and query strategy used to traverse the debugging tree selects each time the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise.

The function `searchBestNode` calculates the best node by searching for a subtree that minimizes the function `getDiff`, where the first argument is the size of the whole tree and the second one the size of the subtree.

```
op getDiff : Nat Nat -> Nat .
eq getDiff(N, N') = sd(N, 2 * N') .
```

Since we use the symmetric difference function, the difference between the size of the whole tree and the double of the size of the current subtree will initially

decrease (while the double of the size of the subtree is bigger than the size of the tree) and finally it will increase (when the size of the tree is bigger than the double of the size of the subtree). Thus, the function `searchBestNode` keeps the information about the last difference in order to stop searching in the subtree when the current difference is bigger than the last one. It uses an auxiliary function that receives the tree, the total number of nodes in the whole tree, the last and the best difference so far, the identifier of the best node, and the identifier of the root of the subtree it is currently traversing. The last and best difference are initialized with a value big enough (ten times the number of nodes), in order to avoid the selection of the initial root as the best node.

```
op searchBestNode : Tree -> NatList .
op searchBestNode : Tree Nat Nat Nat NatList NatList -> NPair .

eq searchBestNode(tree(node(I, NODES), F)) =
    first(searchBestNode(tree(node(I, NODES), F), NODES,
                         10 * NODES, 10 * NODES, nil, nil)) .

ceq searchBestNode(T, NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL) =
                                                  < BEST_NODE, BEST_DIFF >
 if LAST_DIFF <= getDiff(NODES, getOffspring(T)) .
```

If the new difference is better than the last one, the function recursively traverses the forest of the current node with the function `searchBestNode*`.

```
ceq searchBestNode(tree(ND, F), NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL) =
     if NEW_DIFF < BEST_DIFF then
       searchBestNode*(F, NODES, NEW_DIFF, NEW_DIFF, NL, NL, 0)
     else
       searchBestNode*(F, NODES, NEW_DIFF, BEST_DIFF, BEST_NODE, NL, 0)
     fi
 if NEW_DIFF := getDiff(NODES, offspring(ND)) /\
    LAST_DIFF > NEW_DIFF .
```

5 Conclusions and future work

In this paper we have presented how to use the Maude reflective capabilities to implement a debugger for Maude functional modules. It complements other debugging techniques for Maude, such as tracing and term coloring, by allowing to debug a large range of modules (only the `strat` attribute is forbidden, although we expect to allow it in a near future). An important advantage of this kind of debuggers is the help provided in locating the buggy statements, assuming the user answers correctly the corresponding questions. As far as we know, this is the first declarative debugger implemented in the same language it debugs.

From the theoretical point of view, the main novelty of our approach w.r.t. other proposals for declarative debugging of functional languages such as [21,17,18] is that our debugging tree (the *APT*) is obtained from a proof tree in a suitable semantic calculus, which allows us to prove the correctness and completeness of the debugging technique. Furthermore, our debugging of MEL specifications has required an appropriate treatment of memberships which do not appear in previous works.

The complexity of the debugging process increases with the size of the proof

tree. In the case of the top-down strategy the number of questions for a tree T is proportional to $depth(T) * degree(T)$. In the case of the divide and query strategy the number of questions is, on average, proportional to $\log size(T)$. Note that the size of the tree 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 be, in general, reproduced with simpler terms which give rise to smaller proof trees.

We can minimize the number of questions by trusting statements or keeping track of the questions already answered, in order to avoid asking the same question twice.

Since one of the requirements of this kind of debuggers is the interaction with an oracle, that typically is the user, one of the principal aspects that must be improved is the user interface. We plan to provide a complementary graphical interface that allows the user to navigate the tree with more freedom.

We plan to extend our framework by studying how to debug *system modules*, which correspond to rewriting logic specifications and have rules in addition to memberships and equations. These rules can be non-terminating and non-confluent, and thus behave very differently from the statements in the specifications we handle here. Indeed, if the rules of a system module are confluent and terminating, we can use the current version of our debugger by first translating the rewrite rules into equations.

In the context of general system modules, we also plan to study how to debug *missing answers* [14] in addition to the wrong answers we have treated thus far. That is, the non-determinism inherent to a system module implies that a term can be rewritten in several different ways. If the specification does not fulfill the intended model, it may be the case that not all the possible solutions are reached.

Acknowledgments

We thank Santiago Escobar for motivating this research and the anonymous referees for their helpful suggestions to improve this paper.

References

[1] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

[2] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[3] R. Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming*, pages 8–13, New York, NY, USA, 2005. ACM Press.

[4] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. Declarative debugging of Maude functional modules. Technical Report 4/07, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2007. http://maude.sip.ucm.es/debugging.

[5] R. Caballero and M. Rodríguez-Artalejo. DDT: A declarative debugging tool for functional-logic languages. In *Proc. 7th International Symposium on Functional and Logic Programming (FLOPS'04)*, volume 2998 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.

[6] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, Stanford University, 2000.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[8] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.

[9] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.

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

[11] I. MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, University of Melbourne, 2005.

[12] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[13] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[14] L. Naish. Declarative diagnosis of missing answers. *New Generation Comput.*, 10(3):255–286, 1992.

[15] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.

[16] H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *J. Funct. Program.*, 11(6):629–671, 2001.

[17] H. Nilsson and P. Fritzson. Algorithmic debugging of lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

[18] B. Pope. Declarative debugging with Buddha. In *Advanced Functional Programming - 5th International School, AFP 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 273–308. Springer, 2005.

[19] E. Y. Shapiro. *Algorithmic Program Debugging*. ACM Distinguished Dissertation. MIT Press, 1983.

[20] J. Silva. A comparative study of algorithmic debugging strategies. In *Logic-Based Program Synthesis and Transformation*, volume 4407 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2007.

[21] N. Takahashi and S. Ono. DDS: A declarative debugging system for functional programs. *Systems and Computers in Japan*, 21(11):21–32, 1990.

[22] A. Tessier and G. Ferrand. Declarative diagnosis in the CLP scheme. In *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, pages 151–174, London, UK. Springer, 2000.

Solution to the challenge in page 12: To fix the equation `dl6` we must delete the element `E'` instead of `E` in the subtree.

```
ceq [dl6] : delete(L' E R', E) = L' E' delete(R', E')
    if E' := min(R') /\ L' E R' : NeSearchTree{X} .
```