Declarative Debugging of Membership Equational Logic Specifications*

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

Facultad de Informática, Universidad Complutense de Madrid, Spain

Abstract. Algorithmic debugging has been applied to many declarative programming paradigms; in this paper, it is applied to the rewriting paradigm embodied in Maude. We introduce a declarative debugger for executable specifications in membership equational logic which correspond to Maude functional modules. Declarative debugging is based on the construction and navigation of a debugging tree which logically represents the computation steps. We describe the construction of appropriate debugging trees for oriented equational and membership inferences. These trees are obtained as the result of collapsing in proof trees all those nodes whose correctness does not need any justification. We use an extended example to illustrate the use of the declarative debugger and its 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, and selection of trusted vs. suspicious statements by means of labels. The reflective features of Maude have been extensively used to develop a prototype implementation of the declarative debugger for Maude functional modules using Maude itself.

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

1 Introduction

As argued in [23], the application of declarative languages out of the academic world is inhibited by the lack of convenient auxiliary tools such as *debuggers*. The traditional separation between the problem logic (defining *what* is expected to be computed) and control (*how* computations are carried out actually) is a major advantage of these languages, but it also becomes a severe complication when considering the task of debugging erroneous computations. Indeed, the involved execution mechanisms associated to the control make it difficult to apply the typical techniques employed in imperative languages based on step-by-step trace debuggers.

Consequently, new 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],

* 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).

and multiparadigm [5,3,11] programming 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].

In this paper we present a declarative debugger for *Maude functional modules* [7, Chap. 4]. Maude is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. It is a declarative language because Maude modules correspond in general to specifications in rewriting logic [12], a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. This logic is an extension of equational logic; in particular, Maude functional modules correspond to specifications in *membership equational logic* [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.

For a specification in rewriting or membership equational logic to be executable in Maude, it must satisfy some executability requirements. In particular, Maude functional modules are assumed to be confluent, terminating, and sort-decreasing[1] [7], so that, by orienting the equations from left to right, each term can be reduced to a unique canonical form, and semantic equality of two terms can be checked by reducing both of them to their respective canonical forms and checking that they coincide. Since we intend to debug functional modules, we will assume throughout the paper that our membership equational logic specifications satisfy these executability requirements.

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 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

[1] All these requirements must be understood *modulo* some axioms such as associativiy and commutativity that are associated to some binary operations.

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.

Here we present a different approach based on declarative debugging that solves this problem for functional modules. 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 easy 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.[2]
- 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."

Detailed proofs of the results, additional examples, and much 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] For the sake of simplicity, our running example will be unparameterized, but it can easily be parameterized, as shown in [4].

2 Maude functional modules

Maude uses a very expressive version of equational logic, namely membership equational logic (*MEL*) [2,13], which, in addition to equations, allows the statement of membership assertions characterizing the elements of a sort. Below we present the logic and how its specifications are represented as Maude functional modules.

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*. The kind of a sort s is denoted by $[s]$. We write $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ to denote respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of K-kinded variables. 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, and X is a set of K-kinded variables containing all the variables in the A_i. 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 in *MEL* 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 $s \in S_k$, with the meaning that the elements in sorts are well-defined, whereas elements in a kind not having a sort are undefined or error elements. 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 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, $t \rightarrow t'$. Such a statement holds in an algebra, denoted $\mathcal{A} \models t \rightarrow t'$, 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 $u \rightarrow t$ and $v \rightarrow t$. This is the notation we will use in the inference rules and debugging trees studied in Sect. 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[3]

```
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 .
```

2.3 A buggy example: non-empty sorted lists

Let us see a simple example showing how to specify sorted lists of natural numbers in Maude. The following module includes the predefined module `NAT` defining the natural numbers.

```
(fmod SORTED-NAT-LIST is
  pr NAT .
```

We introduce sorts for non-empty lists and sorted lists. We identify a natural number with a sorted list with a single element by means of a subsort declaration.

```
  sorts NatList SortedNatList .
  subsorts Nat < SortedNatList < NatList .
```

The lists that have more than one element are built by means of the associative juxtaposition operator `__`.

```
  op __ : NatList NatList -> NatList [ctor assoc] .
```

We define now when a list (with more than one element) is sorted by means of a membership assertion. It states that the first number must be smaller than the first of the rest of the list, and that the rest of the list must also be sorted.

[3] The underscores indicate the places where the arguments appear in mixfix syntax.

```
vars E E' : Nat .   var L : NatList .   var OL : SortedNatList .
cmb [olist] : E L : SortedNatList if E <= head(L) /\ L : SortedNatList .
```

The definition of the `head` function distinguishes between lists with a single element and longer ones.

```
op head : NatList -> Nat .
eq [hd1] : head(E) = E .
eq [hd2] : head(L E) = E .
```

We also define a sort function which sorts a list by successively inserting each natural number in the appropriate position in the sorted sublist formed by the numbers previously considered.

```
op insertion-sort : NatList -> SortedNatList .
op insert-list : SortedNatList Nat -> SortedNatList .
eq [is1] : insertion-sort(E) = E .
eq [is2] : insertion-sort(E L) = insert-list(insertion-sort(L), E) .
```

The function `insert-list` distinguishes several cases. If the list has only one number, the function checks if it is bigger than the number being inserted, and returns the sorted list. If the list has more than one element, the function checks that the list is previously sorted; if the number being inserted is smaller than the first of the list, it is located as the (new) first element, while if it is bigger we keep the first element and recursively insert the element in the rest of the list.

```
ceq [il1] : insert-list(E, E') = E' E if E' < E .
eq [il2] : insert-list(E, E') = E E' [owise] .
ceq [il3] : insert-list(E OL, E') = E E' OL
 if E' <= E /\ E OL : SortedNatList .
ceq [il4] : insert-list(E OL, E') = E insert-list(OL, E')
 if E OL : SortedNatList [owise] .
endfm)
```

Now, we can reduce a term in this module. For example, we can try to sort the list `3 4 7 6` with

```
Maude> (red insertion-sort(3 4 7 6) .)
result SortedNatList : 6 3 4 7
```

But... the list obtained *is not sorted!* Moreover, Maude infers that *it is sorted*. Did you notice the bugs? We will show how to use the debugger in Sect. 4.3 to detect them.

3 Declarative debugging of Maude functional modules

We describe how to build the debugging trees for *MEL* specifications. Detailed proofs can be found in [4].

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

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

(**Congruence**) $$\frac{e_1 \to e'_1 \quad \ldots \quad e_n \to e'_n}{f(e_1,\ldots,e_n) \to f(e'_1,\ldots,e'_n)}\ (Cong)$$

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

(**Membership**) $$\frac{\{\theta(u_i) \to t_i \leftarrow \theta(u'_i)\}_{1\le i\le n} \quad \{\theta(v_j) : s_j\}_{1\le j\le m}}{\theta(e) : s}\ (Mb)$$
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**) $$\frac{\{\theta(u_i) \to t_i \leftarrow \theta(u'_i)\}_{1\le i\le n} \quad \{\theta(v_j) : s_j\}_{1\le j\le m}}{\theta(e) \to \theta(e')}\ (Rep)$$
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

3.1 Proof trees

Before defining the debugging trees employed in our declarative debugging framework we need to introduce the semantic rules defining the specification semantics. The inference rules of the calculus can be found in Fig. 1, where θ denotes a substitution.

They are an adaptation to the case of Maude functional modules of the deduction rules for *MEL* presented in [13]. The notation $\theta(u_i) \to t_i \leftarrow \theta(u'_i)$ must be understood as a shortcut for $\theta(u_i) \to t_i$, $\theta(u'_i) \to t_i$. We assume the existence of an *intended interpretation* I of the specification, which is a Σ-algebra corresponding to the model that the user had in mind while writing the statements E, i.e., the user expects that $I \models e \to e'$, $I \models e : s$ for each reduction $e \to e'$ and membership $e : s$ computed w.r.t. the specification (Σ, E). As a Σ-algebra, I must satisfy the following proposition:

Proposition 1. *Let $S = (\Sigma, E)$ be a MEL specification and let $\mathcal{A}$ be any Σ-algebra. If $e \to e'$ (respectively $e : s$) can be deduced using the semantic calculus rules* reflexivity, transitivity, congruence, *or* subject reduction *using premises that hold in $\mathcal{A}$, then $\mathcal{A} \models e \to e'$ (respectively $\mathcal{A} \models e : s$).*

Observe that this proposition 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.

We will say that $e \to e'$ (respectively $e : s$) is *valid* when it holds in 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.

Definition 1. *Let* $R \equiv (af \Leftarrow u_1 = u'_1 \wedge \cdots \wedge u_n = u'_n \wedge v_1 : s_1 \wedge \cdots \wedge v_m : s_m)$*, where af denotes an atomic formula, that is, either an oriented equation or a membership axiom in a specification S. Then:*

- $\theta(R)$ *is a* wrong equation instance *(respectively, a* wrong membership axiom instance*) w.r.t. an intended interpretation I when*
  - *There exist* $t_1, \ldots, t_n$ *such that* $I \models \theta(u_i) \rightarrow t_i$, $I \models \theta(u'_i) \rightarrow t_i$ *for* $i = 1 \ldots n$.
  - $I \models \theta(v_j) : s_j$ *for* $j = 1 \ldots m$.
  - $\theta(af)$ *does not hold in I.*
- *R is a* wrong equation *(respectively, a* wrong membership axiom*) if it admits some wrong instance.*

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. For example, the proof tree depicted in Fig. 2 corresponds to the result of the reduction in the specification for sorted lists described at the end of Sect. 2.3. For obvious reasons, the operation names have been abbreviated in a self-explanatory way; furthermore, each node corresponding to an instance of the *replacement* or *membership* inference rules has been labelled with the label of the equation or membership statement which is being applied.

In declarative debugging we are specially interested in *buggy nodes* which are invalid nodes with all its children valid. The following proposition characterizes buggy nodes in our setting.

Proposition 2. *Let N by a buggy node in some proof tree in the calculus of Fig. 1 w.r.t. an intended interpretation I. Then:*

1. *N is the consequence of either a* membership *or a* replacement *inference step.*
2. *The equation associated to N is a wrong equation or a wrong membership axiom.*

3.2 Abbreviated proof trees

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 an easy consequence, the following result holds:

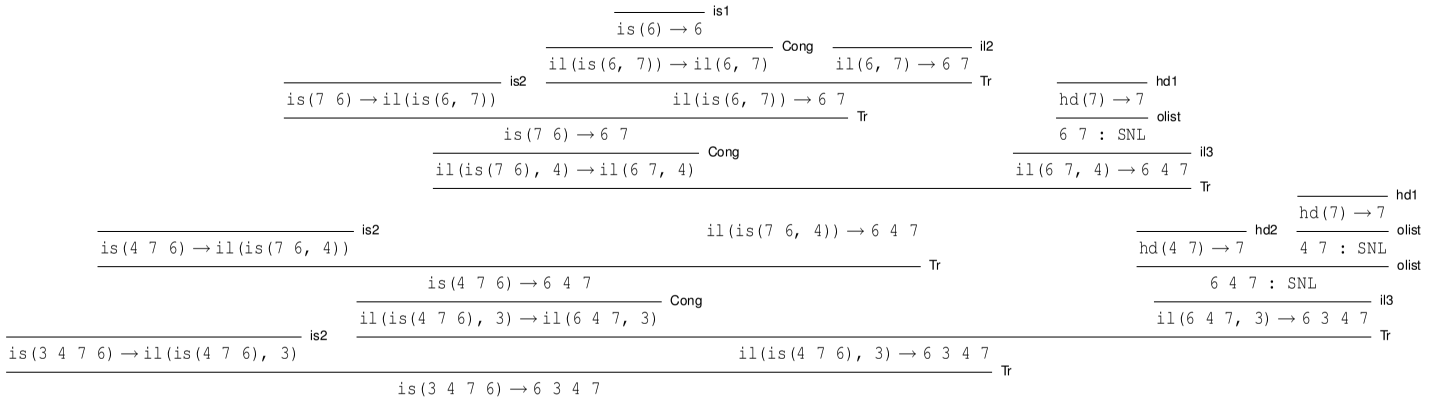

Fig. 2. Proof tree for the sorted lists example

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

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

$$(\mathbf{APT}_3)\ APT'\left(\frac{\frac{T_1 \ldots T_n}{e_1 \to e'}(Rep) \quad T'}{e_1 \to e_2}(Tr)\right) = \left\{\frac{APT'(T_1) \ldots APT'(T_n)\ APT'(T')}{e_1 \to e_2}(Rep)\right\}$$

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

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

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

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

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

Fig. 3. Transforming rules for obtaining abbreviated proof trees

Proposition 3. *Let T be 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*), or simply APT if the proof tree T is clear from the context. The reason for preferring the APT to the original proof tree is that it reduces and simplifies the questions that will be asked to the user while keeping the soundness and completeness of the technique. In particular the APT contains only nodes related to the *replacement* and *membership* inferences using statements included in the specification, which are the only possible buggy nodes as Proposition 2 indicates. Fig. 3 shows the definition of $APT(T)$. 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 children 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 $N \in T$ that is the 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 $e_1 \rightarrow e_2$ in the righthand side of APT_3 is the node $e_1 \rightarrow e'$ of the proof tree T, which is not included in the $APT(T)$. We have chosen to introduce $e_1 \rightarrow e_2$ instead of simply $e_1 \rightarrow e'$ 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' in APT_3 must be necessarily of the form $e' \rightarrow e_2$ by the structure of the inference rule for transitivity in Fig. 1). We will formally state below (Theorem 1) that skipping $e_1 \rightarrow e'$ and introducing instead $e_1 \rightarrow e_2$ is safe from the point of view of the debugger.

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. It will be used by the debugger in order to single out the incorrect fragment of specification code.

Before proving the correctness and completeness of the debugging technique we need some auxiliary results. The first one indicates that APT' transforms a tree with invalid root into a set of trees such that at least one has an invalid root. We denote the root of a tree T as $root(T)$.

Lemma 1. *Let T be a proof tree such that $root(T)$ is invalid w.r.t. an intended interpretation I. Then there is some $T' \in APT'(T)$ such that $root(T')$ is invalid w.r.t. I.*

An immediate consequence of this result is the following:

Lemma 2. *Let T be a proof tree and $APT(T)$ its abbreviated proof tree. Then the root of $APT(T)$ cannot be buggy.*

The next theorem guarantees the correctness and completeness of the debugging technique based on *APT*s:

Theorem 1. *Let S be a specification, 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 equation in S.*

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. 4.1).

The tree depicted in Fig. 4 is the abbreviated proof tree corresponding to the proof tree in Fig. 2, using the same conventions w.r.t. abbreviating the operation names. The debugging example described in Sect. 4.3 will be based on this abbreviated proof tree.

4 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. 2.3.

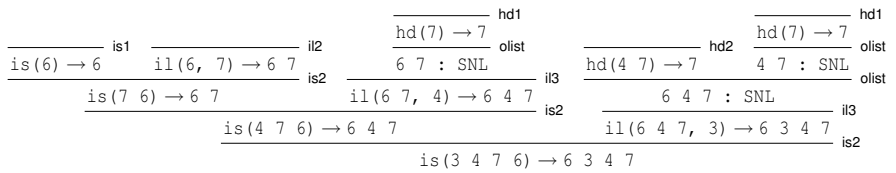

Fig. 4. Abbreviated proof tree for the sorted lists example

4.1 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 *reflexivity*, *transitivity*, and *congruence* rules are treated in the *APT* transformation described in Fig. 3; 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. 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 mentioned in the introduction, 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. In the same way, the signature of the correct module need not coincide with the signature of the module being debugged. If the correct module cannot help in answering a question, the user may have to answer it.

4.2 Commands

The debugger is initiated in Maude by loading the file `dd.maude` (available from `http://maude.sip.ucm.es/debugging`), 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*, the latter being 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 process 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[4]

```
(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 axiom labels that must be taken into account when generating 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 this 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 .)`.

4.3 Sorted lists revisited

We recall from Sect. 2.3 that if we try to sort the list `3 4 7 6`, we obtain the strange result

```
Maude> (red insertion-sort(3 4 7 6) .)
result SortedNatList : 6 3 4 7
```

That is, the function returns an unsorted list, but Maude infers it is sorted. We can debug the buggy specification by using the command

```
Maude> (debug in SORTED-NAT-LIST : insertion-sort(3 4 7 6) -> 6 3 4 7 .)
```

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

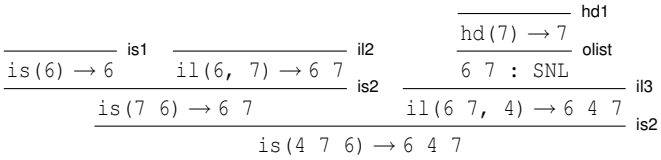

Fig. 5. Abbreviated proof tree after the first question

With this command the debugger computes the tree shown in Fig. 4. Since the default navigation strategy is divide and query, the first question is

```
Is this transition (associated with the equation is2) correct?
insertion-sort(4 7 6) -> 6 4 7
Maude> (no .)
```

We expect `insertion-sort` to order the list, so we answer negatively and the subtree in Fig. 5 is selected to continue the debugging. The next question is

```
Is this transition (associated with the equation is2) correct?
insertion-sort(7 6) -> 6 7
Maude> (yes .)
```

Since the list is sorted, we answer `yes`, so this subtree is deleted (Fig. 6 left). The debugger asks now the question

```
Is this membership (associated with the membership olist) correct?
6 7 : SortedNatList
Maude> (yes .)
```

This sort is correct, so this subtree is also deleted (Fig. 6 right) and the next question is prompted.

```
Is this transition (associated with the equation il3) correct?
insert-list(6 7, 4) -> 6 4 7
Maude> (no .)
```

With this information the debugger selects the subtree and, since it is a leaf, it concludes that the node is associated with the buggy equation.

```
The buggy node is:
insert-list(6 7, 4) -> 6 4 7
With the associated equation: il3
```

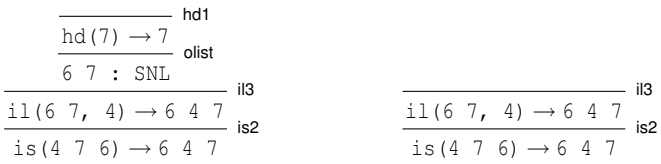

Fig. 6. Abbreviated proof trees after the second and third questions

That is, the debugger points to the equation `il3` as buggy. If we examine it

```
ceq [il3] : insert-list(E OL, E') = E E' OL
 if E' <= E /\ E OL : SortedNatList .
```

we can see that the order of `E` and `E'` in the righthand side is wrong and we can proceed to fix it appropriately.

We can check the fixed function by sorting again the list `3 4 7 6`. We obtain now the sorted list `3 4 6 7`. Then, we have solved one problem, but if we reduce the unsorted list `6 3 4 7`

```
Maude> (red 6 3 4 7 .)
result SortedNatList : 6 3 4 7
```

we can see that Maude continues assigning to it an incorrect sort.

We can check this inference by using the command

```
Maude> (debug 6 3 4 7 : SortedNatList .)
```

The first question the debugger prompts is

```
Is this membership (associated with the membership olist) correct?
3 4 7 : SortedNatList
Maude> (yes .)
```

Of course, this list is sorted. The following question is

```
Is this transition (associated with the equation hd2) correct?
head(3 4 7) -> 7
Maude> (no .)
```

But the head of a list should be the first element, not the last one, so we answer `no`. With only these two questions the debugger prints

```
The buggy node is:
head(3 4 7) -> 7
With the associated equation: hd2
```

If we check the equation `hd2`, we can see that we take the element from the wrong side.

```
eq [hd2] : head(L E) = E .
```

To debug this module we have used the default divide and query strategy. Let us check it now with the top-down strategy. We debug again the inference

```
insertion-sort(3 4 7 6) -> 6 3 4 7
```

in the initial module with the two errors. The first question asked in this case is

```
Is any of these nodes wrong?
Node 0 : insertion-sort(4 7 6) -> 6 4 7
Node 1 : insert-list(6 4 7, 3) -> 6 3 4 7
Maude> (node 0 .)
```

Both nodes are wrong, so we select, for example, the first one. The next question is

```
Is any of these nodes wrong?
Node 0 : insertion-sort(7 6) -> 6 7
Node 1 : insert-list(6 7, 4) -> 6 4 7
Maude> (node 1 .)
```

This time, only the second node is wrong, so we select it. The debugger prints now

```
Is any of these nodes wrong?
Node 0 : 6 7 : SortedNatList
Maude> (all valid .)
```

There is only one node, and it is correct, so we give this information to the debugger, and it detects the wrong equation.

```
The buggy node is:
insert-list(6 7, 4) -> 6 4 7
With the associated equation: il3
```

But remember that we chose a node randomly when the debugger showed two wrong nodes. What happens if we select the other one? The following question is printed.

```
Is any of these nodes wrong?
Node 0 : 6 4 7 : SortedNatList
Maude> (node 0 .)
```

Since this single node is wrong, we choose it and the debugger asks

```
Is any of these nodes wrong?
Node 0 : head(4 7) -> 7
Node 1 : 4 7 : SortedNatList
Maude> (node 0 .)
```

The first node is the only one erroneous, so we select it. With this information, the debugger prints

```
The buggy node is:
head(4 7) -> 7
With the associated equation: hd2
```

That is, the second path finds the other bug. In general, this strategy finds different bugs if the user selects different wrong nodes.

In order to prune the debugging tree, we can define a module specifying the sorting function `sort` in a correct, but inefficient, way. This module will define the functions `insertion-sort` and `insert-list` by means of `sort`.

```
(fmod CORRECT-SORTING is
  pr NAT .
  sorts NatList SortedNatList .
  subsorts Nat < SortedNatList < NatList .
  vars E E' : Nat .  vars L L' : NatList .  var OL : SortedNatList .
  op __ : NatList NatList -> NatList [ctor assoc] .
  cmb E E' : SortedNatList if E <= E' .
  cmb E E' L : SortedNatList if E <= E' /\ E' L : SortedNatList .
```

The `sort` function is defined by switching unsorted adjacent elements in all the possible cases for lists.

```
  op sort : NatList -> SortedNatList .
  ceq sort(L E E' L') = sort(L E' E L') if E' < E .
  ceq sort(L E E') = sort(L E' E) if E' < E .
  ceq sort(E E' L) = sort(E' E L) if E' < E .
  ceq sort(E E') = E' E if E' < E .
  eq sort(L) = L [owise] .
```

We now use `sort` to implement `insertion-sort` and `insert-list`.

```
  op insertion-sort : NatList -> SortedNatList .
  op insert-list : SortedNatList Nat -> SortedNatList .
  eq insertion-sort(L) = sort(L) .
  eq insert-list(OL, E) = sort(E OL) .
 endfm)
```

We can use this module to prune the debugging trees built by the `debug` commands if we previously introduce the command

```
Maude> (correct module CORRECT-SORTING .)
```

Now, we try to debug the initial module (with two errors) again. In this example, all the questions about correct inferences have been pruned, so all the answers are negative. In general, the correct module does not have to be complete, so some correct inferences could be presented to the user.

```
Maude> (debug in SORTED-NAT-LIST : insertion-sort(3 4 7 6) -> 6 3 4 7 .)

Is this transition (associated with the equation il3) correct?
insert-list(6 4 7, 3) -> 6 3 4 7
Maude> (no .)

Is this membership (associated with the membership olist) correct?
6 4 7  : SortedNatList
Maude> (no .)

Is this transition (associated with the equation hd2) correct?
head(4 7) -> 7
Maude> (no .)
```

```
The buggy node is:
head(4 7) -> 7
With the associated equation: hd2
```

The correct module also improves the debugging of the membership. With only one question we obtain the buggy equation.

```
Maude> (debug in SORTED-NAT-LIST : 6 3 4 7 : SortedNatList .)

Is this transition (associated with the equation hd2) correct?
head(3 4 7) -> 7
Maude> (no .)

The buggy node is:
head(3 4 7) -> 7
With the associated equation: hd2
```

4.4 Implementation

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 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 an extension of Full Maude [7, Chap. 18]. As far as we know, this is the first declarative debugger implemented using such reflective techniques.

The implementation takes care of the two stages of generating and navigating the debugging tree. 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.

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 main function in the implementation of the debugging tree generation is called `createTree`. It receives the module where a wrong inference took place, a correct module (or the constant `noModule` when no such module is provided) to prune the tree, the term initially reduced, the (erroneous) result obtained, and the set of suspicious statement labels. It keeps the initial inference as the root of the tree and uses an auxiliary function `createForest` that, in addition to the arguments received by `createTree`, receives the module "cleaned" of suspicious statements, and generates the abbreviated forest corresponding to the reduction between the two terms passed 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 only using trusted equations, thus avoiding to build a tree that will be finally empty.

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 this best node by searching for a subtree minimizing an appropriate function.

The technical report [4] provides a full explanation of this implementation, including the user interaction.

5 Conclusions and future work

In this paper we have developed the foundations of declarative debugging of executable *MEL* specifications, and we have applied them to implement a debugger for Maude functional modules. As far as we know, this is the first debugger implemented in the same language it debugs. This has been made possible by the reflective features of Maude. In our opinion, this debugger provides a complement to existing debugging techniques for Maude, such as tracing and term coloring. An important contribution of our debugger is the help provided by the tool in locating the buggy statements, assuming the user answers correctly the corresponding questions. The debugger keeps track of the questions already answered, in order to avoid asking the same question twice.

We want to improve the interaction with the user by providing a complementary graphical interface that allows the user to navigate the tree with more freedom. 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.

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. In this context, we also plan to study how to debug *missing answers* [14] in addition to the wrong answers we have treated thus far.

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*, LNCS 2664, 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)*, LNCS 2998, 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*, LNCS 4350. Springer, 2007.
8. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.
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 Generation Computing*, 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*, LNCS 1376, pages 18–61. Springer, 1998.
14. L. Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 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. *Journal of Functional Programming*, 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*, LNCS 3622, 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*, LNCS 4407, 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.
23. P. Wadler. Why no one uses functional languages. *SIGPLAN Not.*, 33(8):23–27, August 1998.