Declarative Debugging of Maude Functional Modules*

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

Technical Report SIC-4-07

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid

December, 2007

*Research supported by MEC Spanish projects *DESAFIOS* (TIN2006-15660-C02-01) and *MERIT-FORMS* (TIN2005-09027-C03-03), and Comunidad de Madrid program *PROMESAS* (S0505/TIC/0407).

Abstract

We introduce a declarative debugger for Maude functional modules, which correspond to executable specifications in membership equational logic. First 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 correction does not need any justification.

Since Maude supports the reflective features in its underlying logic, it includes a predefined `META-LEVEL` module providing access to metalevel concepts such as specifications or computations as usual data. This allows 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 for Maude can be specified in Maude itself. We describe in detail this metalevel implementation of our tool.

Finally, we include several extended examples 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, 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

Contents

1 Introduction

As argued in [21], 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 difficult the application of the typical techniques employed in imperative languages based on step-by-step trace debuggers.

Consequently, new debugging approaches based on the languages semantics have been introduced in the field of declarative languages, such as *abstract diagnosis*, which formulates a debugging methodology based on abstract interpretation [1], or *declarative debugging*, also known as *algorithmic debugging*, which was first introduced by E. Y. Shapiro [18]. Declarative debugging has been widely employed in the logic [9, 13, 20], functional [16, 15, 17], and multi-paradigm programming [4, 3, 10] languages. Declarative debugging starts from a computation considered incorrect by the user (error symptom) and locates a program fragment responsible for the error. The declarative debugging scheme [14] uses a *debugging tree* as 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 children 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 have all 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 [19].

In this paper we present a declarative debugger for *Maude functional modules* [6, Chapter 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 [11], 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, 12], 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. Moreover, exploiting the fact that rewriting logic is reflective [5, 7], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [6, Chapter 14], a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications.

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] [6], so that, by

[1] All these requirements must be understood *modulo* some axioms such as associativiy and commutativity

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 [6, Chapter 22]. The tracing facilities allow us to follow the execution on a specification, that is, the sequence of rewrites that take place. We can select which operators or statements (equations, memberships, or rules) are traced, and how much information is shown in each step. Term coloring consists in printing with different colors the operators used to build a term that does not fully reduce. It uses the `ctor` attribute that can be given to an operator indicating that it is a constructor. If an operator is colored, this means that the term contains nonconstructors, that is, that there is a "strangeness" in the term. The different colors indicate the source of the strangeness. The Maude debugger allows 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. We can also execute another Maude command, which in turn can enter the (fully re-entrant) debugger.

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 lose 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 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.

---

that are associated to some binary operations.

- It allows 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.

- When the debugger is started, it allows to select the list of labeled statements that will be used to generate the proof tree, thus restricting the debugged statements. 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." This produces that other nodes associated with the currently trusted statement are also deleted from the tree.

As mentioned before, the Maude system includes the predefined `META-LEVEL` module supporting reflection in rewriting logic [6, Chapter 14]. 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` [6, Chapter 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. The Maude source files for the debugging tool are available from the webpage `http://maude.sip.ucm.es/debugging`.

The rest of the paper is structured as follows. Section 2 provides a summary of the main concepts of membership equational logic and the syntax of Maude functional modules. Section 3 describes the theoretical foundations of the debugging proof trees for membership equational logic specifications. Section 4 shows how to use the debugging tool by means of several examples, while Section 5 describes in detail the Maude implementation of the tool. Finally, Section 6 concludes and mentions some future work, such as debugging Maude system modules, which correspond to rewriting logic specifications and have rules in addition to memberships and equations.

2 Maude functional modules

As mentioned in the introduction, Maude uses a very expressive version of equational logic, namely membership equational logic [2, 12], which, in addition to equations, allows the statement of membership assertions characterizing the elements of a sort. In the following sections we present the logic and how its specifications are represented as Maude functional modules.

2.1 Membership equational logic

A *signature* in membership equational logic 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.

The atomic formulas of membership equational logic 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. A *specification* is a pair (Σ, E), where E is a set of sentences in membership equational logic over the signature Σ.

Models of membership equational logic 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, 12] for a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, initial and free algebras, and specification morphisms.

Since the membership equational logic 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 Section 3.

2.2 Representation in Maude

Maude functional modules, introduced with syntax `fmod ... endfm`, are executable membership equational 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: sorted lists

Let us see a simple example showing how to specify in Maude sorted lists. We use a module parameterized by the theory `TOSET` [6, Section 8.3], that requires a sort `Elt` and a total order `_<=_` over the elements of this sort.

```
fmod SORTED-LIST{X :: TOSET} is
```

We introduce sorts for lists and sorted lists. We identify an element with a sorted list with a single element by means of a subsort declaration.

```
  sorts List{X} SortedList{X} .
  subsorts X$Elt < SortedList{X} < List{X} .
```

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

```
  op __ : List{X} List{X} -> List{X} [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 element must be smaller than the first of the rest of the list, and that the rest of the list must also be sorted.

```
  vars E E' : X$Elt .
  var L : List{X} .
  var OL : SortedList{X} .

  cmb [olist] : E L : SortedList{X}
   if E <= head(L) /\ L : SortedList{X} .
```

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

```
  op head : List{X} -> X$Elt .
  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 element in the appropriate position in the sorted sublist formed by the elements previously considered.

```
  op insertion-sort : List{X} -> SortedList{X} .
  op insert-list : SortedList{X} X$Elt -> SortedList{X} .

  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 element, the function checks if this element is bigger than the inserted element, and returns the sorted list. If the list has more than one element, the function checks that the list is previously sorted; if the element we want to insert 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 : SortedList{X} .
  ceq [il4] : insert-list(E OL, E') = E insert-list(OL, E')
   if E OL : SortedList{X} [owise] .
endfm
```

In order to be able to execute this module, we instantiate it with the view `NatAsToset`.

```
view NatAsToset from TOSET to NAT is
 sort Elt to Nat .
endv

fmod SORTED-LIST-TEST is
 protecting SORTED-LIST{NatAsToset} .
endfm
```

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

```
red insertion-sort(3 4 7 6) .
```

We obtain the result

```
result SortedList{NatAsToset}: 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 Section 4.3.1 to detect them.

3 Declarative debugging of Maude functional modules

We describe how to build the debugging trees for membership equational logic specifications.

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 Figure 1.

They are an adaptation to the case of Maude functional modules of the deduction rules for membership equational logic presented in [12]. The notation $\theta(u_i) \rightarrow t_i \leftarrow \theta(u_i')$ must be understood as a shortcut for $\theta(u_i) \rightarrow t_i$, $\theta(u_i') \rightarrow t_i$. 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, i.e., the user expects that $\mathcal{I} \models e \rightarrow e'$, $\mathcal{I} \models e : s$ for each reduction $e \rightarrow e'$ and membership $e : s$ computed w.r.t. the specification (Σ, E). As a Σ-algebra, $\mathcal{I}$ must verify the following proposition:

(**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 $e \to e' \Leftarrow u_1 = u'_1 \wedge \cdots \wedge u_n = u'_n \wedge v_1 : s_1 \wedge \cdots \wedge v_m : s_m$

Figure 1: Semantic calculus for Maude functional modules

Proposition 1 *Let $S = (\Sigma, E)$ be a membership equational logic specification and $\mathcal{A}$ a 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 $\mathcal{A} \models e : s$).*

Proof. The result is as direct consequence of the definition of satisfaction in an algebra. For instance, in the case of the *transitivity* rule: if $\mathcal{A} \models e_1 \to e'$ and $\mathcal{A} \models e' \to e_2$ then $[\![e_1]\!]_{\mathcal{A}} = [\![e']\!]_{\mathcal{A}}$ and $[\![e']\!]_{\mathcal{A}} = [\![e_2]\!]_{\mathcal{A}}$. Therefore $[\![e_1]\!]_{\mathcal{A}} = [\![e_2]\!]_{\mathcal{A}}$ and $\mathcal{A} \models e_1 \to e_2$. In the case of the *subject reduction* rule, if $\mathcal{A} \models e \to e'$ and $\mathcal{A} \models e' : s$, then $[\![e]\!]_{\mathcal{A}} = [\![e']\!]_{\mathcal{A}}$ and $[\![e']\!]_{\mathcal{A}} \in A_s$, and hence $[\![e]\!]_{\mathcal{A}} \in A_s$. The *reflexivity* and *congruence* rules are checked analogously. □

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 $\mathcal{I}$, 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.

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 $\mathcal{I}$ when*
  - *There exist $t_1, \ldots, t_n$ such that $\mathcal{I} \models u_i \to t_i$, $\mathcal{I} \models u'_i \to t_i$ for $i = 1 \ldots n$.*

- $\mathcal{I} \models \theta(v_j) : s_j$ *for* $j = 1 \ldots m$.
- *af does not hold in* $\mathcal{I}$.

- 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 children nodes of the conclusion at each inference step. For example, the proof tree depicted in Figure 2 corresponds to the result of the reduction in the specification for sorted lists described at the end of Section 2.2.1. 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 Figure 1 w.r.t. an intended interpretation $\mathcal{I}$. 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.*

Proof. The first item is a straightforward consequence of Proposition 1. This proposition applies to all the inference rules except *membership* and *replacement*, which are consequently the only inference rules that can produce a conclusion that does not hold in $\mathcal{I}$ (i.e., invalid) from premises that hold in $\mathcal{I}$ (i.e., valid). In order to prove the second item we observe that the premises in both the *membership* and *replacement* rules are of the form $(\theta(u_i) \rightarrow t_i \leftarrow \theta(u'_i))$ for $i = 1 \ldots n$ and $(\theta(v_j) : s_j)$ for $j = 1 \ldots m$, and that all of them are valid by the definition of buggy node. This, together with the invalid consequence at the inference step (the consequence is N, which is buggy and therefore invalid), matches the definition of wrong instance with substitution θ, which in turn means that the equation/membership axiom is wrong w.r.t. $\mathcal{I}$. □

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 pointing out at 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:

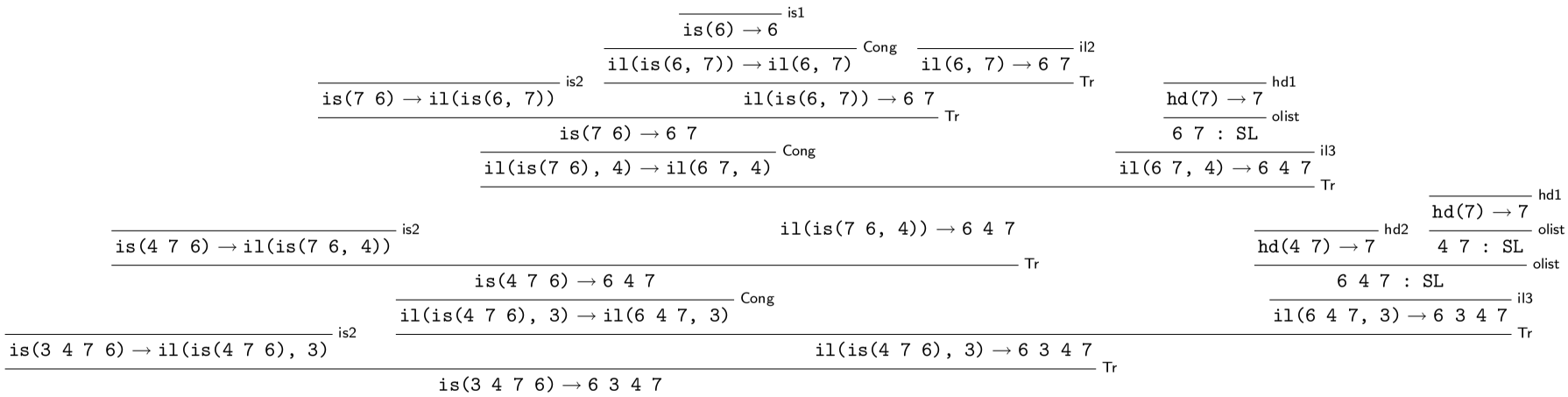

Figure 2: Proof tree for the sorted lists example

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

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

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

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

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

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

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

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

Figure 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.

Figure 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 $N \in T$ 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 Figure 1). We will prove formally below (Theorem 1) that skipping $e_1 \to e'$ and introducing instead $e_1 \to e_2$ 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. This information 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 previous 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 $\mathcal{I}$. Then there is some $T' \in APT'(T)$ such that $root(T')$ is invalid w.r.t. $\mathcal{I}$.*

Proof. By induction on $height(T)$. We distinguish cases based on the rule (APT_i), $2 \leq i \leq 8$, applicable at $root(T)$. In the case of (APT_3), (APT_7), and (APT_8) we have that $APT'(T)$ contains only one element T', and that $root(T) = root(T')$. Hence the result holds trivially. If the applicable rule is (APT_2), then T cannot be invalid by Proposition 1. The same proposition ensures that if (APT_4), (APT_5), or (APT_6) can be applied at $root(T)$ (i.e., the first inference of T is either a a transitivity, a congruence, or a subject reduction), and $root(T)$ is invalid, then there must be some subtree T', child of $root(T)$ in T, such that $root(T')$ is invalid. Then, by the inductive hypothesis ($height(T') < height(T)$, we have that $root(APT'(T'))$ is invalid, and from the form of (APT_4), (APT_5), (APT_6) it is easy to check that $T' \in APT'(T)$. □

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.*

Proof. Let *af* be the root of T. By (APT_1) we have that $APT(T)$ is

$$\frac{APT'(T)}{af}$$

Obviously if *af* is valid it cannot be buggy. If it is invalid then, by Proposition 1, $APT'(T)$ will contain some T' with invalid root, and therefore the root of $APT(T)$ will have an invalid child, and it cannot be buggy. □

The next theorem guarantees the correctness and completeness of the debugging technique based on APTs:

Theorem 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).*

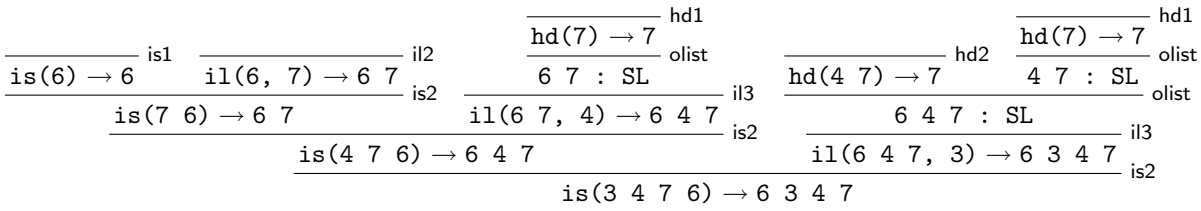

Figure 4: Abbreviated proof tree for the sorted lists example

- *Any buggy node in $APT(T)$ has an associated wrong equation in S.*

Proof. The completeness is trivial since $APT(T)$ is a finite tree with an invalid root (the root of T), and proving that such tree always contain a buggy node is an easy induction on the height of the tree.

In order to prove the correctness, it is enough to prove that any buggy node in $APT(T)$ corresponds to a buggy node in T. Then by the second item of Proposition 2 the associated equation/membership axiom is wrong. Let N be a buggy node $APT(T)$. By Lemma 2, N is not $root(APT(T))$. Therefore it was introduced by function APT' (see rule (APT_1)). We distinguish cases depending on the particular rule (APT_i), $2 \leq i \leq n$ that introduced N. In particular such rule must be either (APT_3), (APT_7) or (APT_8) because (APT_2), (APT_4), (APT_5), and (APT_6) do not explicitly introduce any new node in the APT.

- (APT_3). In this case the buggy node must be of the form $e_1 \rightarrow e_2$, while its associated node in T is of the form $e_1 \rightarrow e'$. We observe in the rule that $e_1 \rightarrow e_2$ is also part of T, with $e_1 \rightarrow e'$ and the root of some tree T_{n+1} its two children. Since $e_1 \rightarrow e_2$ is invalid and the consequence of a transitivity rule in T, the Proposition 1 implies that either $e_1 \rightarrow e'$ or $root(T_{n+1})$ must be invalid. But $root(T_{n+1})$ cannot be invalid, because by Lemma 1 then $root(APT'(T_{n+1}))$ would also be invalid and its parent node $e_1 \rightarrow e_2$ in $APT(T)$ would not be buggy. Therefore the associated node in T, $e_1 \rightarrow e'$ is invalid. Moreover, its children nodes in T: $root(T_1), \ldots, root(T_n)$, cannot be invalid because by Lemma 1 any invalid node $root(T_i)$, $1 \leq i \leq n$ would mean that $root(APT'(T_i))$ is also invalid, and that its parent node in $APT(T)$ $e_1 \rightarrow e_2$ is not buggy. Therefore $e_1 \rightarrow e'$ is buggy in T, and its associated equation wrong as the theorem states.

- (APT_7). The buggy node $e : s$ of $APT(T)$ is also a node in T. Moreover it is also buggy in T because the existence of any invalid children among $root(T_1), \ldots, root(T_n)$ would imply that $e : s$ is not buggy in $APT(T)$ by Lemma 1.

- (APT_8). Analogous to the previous case. □

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 Section 4.1).

The tree depicted in Figure 4 is the abbreviated proof tree corresponding to the proof tree in Figure 2, using the same conventions w.r.t. abbreviating the operation names. The debugging example described later in Section 4.3.1 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 several buggy examples.

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 Figure 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 Section 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. If the user wants to trust a whole imported module, either its statements are not labeled or the corresponding labels are not considered in the debugging command, as described below.

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.

4.2 Commands

The debugger is initiated in Maude by loading the file `fdd.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*, being the latter the default one. The user can switch between them by using the commands

```
(top-down strategy .)
(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 with MODULE-NAME .)
```

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

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

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

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

where `LABELS` is the set of suspicious equation and membership axiom 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 INITIAL-TERM : WRONG-SORT in MODULE-NAME .)
(debug INITIAL-TERM : WRONG-SORT in MODULE-NAME 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 must 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 .)
(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 .)
```

4.3 Examples

We show how the debugger works by means of several examples: the sorted lists specification already introduced in Section 2.2.1, binary search trees, and the operational semantics of a simple imperative language.

4.3.1 Sorted lists

We recall from Section 2.2.1 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 SortedList{NatAsToset} : 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

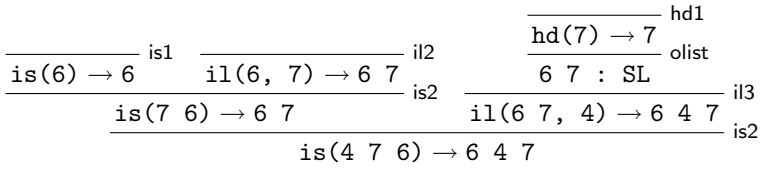

Figure 5: Abbreviated proof tree after the first question

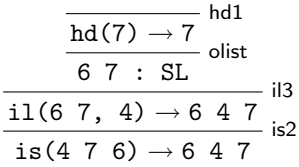

Figure 6: Abbreviated proof tree after the second question

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

With this command the debugger computes the tree shown in Figure 4. The first question asked by the debugger 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 Figure 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 (Figure 6). The debugger asks now the question

```
Is this membership (associated with the membership olist) correct?

6 7  : SortedList{NatAsToset}

Maude> (yes .)
```

This sort is correct, so this subtree is also deleted (Figure 7) and the next question is prompted.

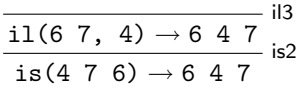

Figure 7: Abbreviated proof tree after the third question

```
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
```

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 : SortedList{X} .
```

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 SortedList{NatAsToset}: 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 : SortedList{NatAsToset} in SORTED-LIST-TEST .)
```

The first question the debugger prompts is

```
Is this membership (associated with the membership olist) correct?

3 4 7  : SortedList{NatAsToset}

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(2 5 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 occasion 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 one of the nodes is wrong, so we select it. The debugger prints now

```
Is any of these nodes wrong?
Node 0 : 6 7 : SortedList{NatAsToset}

Maude> (all valid .)
```

There is only a 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 : SortedList{NatAsToset}

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 : SortedList{NatAsToset}

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 defining 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{X :: TOSET} is

 sorts List{X} SortedList{X} .
 subsorts X$Elt < SortedList{X} < List{X} .

 vars E E' : X$Elt .
 vars L L' : List{X} .
 var OL : SortedList{X} .

 op __ : List{X} List{X} ~> List{X} [ctor assoc] .

 mb L L' : List{X} .

 cmb E E' : SortedList{X}
  if E < E' .
 cmb E E' L : SortedList{X}
  if E < E' /\ E' L : SortedList{X} .
```

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

```
 op sort : List{X} -> SortedList{X} .
 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 use now `sort` to implement `insertion-sort` and `insert-list`.

```
 op insertion-sort : List{X} -> SortedList{X} .
 op insert-list : SortedList{X} X$Elt -> SortedList{X} .

 eq insertion-sort(L) = sort(L) .
 eq insert-list(OL, E) = sort(E OL) .
endfm
```

We instantiate the module with the view `NatAsToset`.

```
fmod CORRECT-SORTED-LIST-TEST is
 protecting CORRECT-SORTING{NatAsToset} .
endfm
```

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

```
Maude> (correct with CORRECT-SORTED-LIST-TEST .)
```

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 has not to be complete, so some correct inferences could be presented to the user.

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

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  : SortedList{NatAsToset}

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 6 3 4 7 : SortedList{NatAsToset} in SORTED-LIST-TEST .)

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
```

4.3.2 Binary search trees

To implement binary search trees, we define a theory for the contents in the nodes with a `combine` function.

```
fth CONTENTS is
  sort Contents .
  op combine : Contents Contents -> Contents [assoc] .
endfth
```

```
view Contents from TRIV to CONTENTS is
  sort Elt to Contents .
endv
```

The nodes of our search trees will consist of a key that satisfies the theory `STOSET` (with strict total order) together a value fulfilling the theory above.

```
fmod SEARCH-TREE{X :: STOSET, Y :: CONTENTS} is
 protecting MAYBE{Contents}{Y} * (op maybe to not-found) .

 sorts NeSearchTree{X, Y} SearchTree{X, Y} Tree{X, Y} .

 subsorts NeSearchTree{X, Y} < SearchTree{X, Y} .

 op empty : -> Tree{X, Y} [ctor] .
 op _<_,_>_ : Tree{X, Y} X$Elt Y$Contents Tree{X, Y} -> Tree{X, Y} [ctor] .
```

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. We specify this relation by means of memberships.

```
 vars K K' : X$Elt .
 vars C C' : Y$Contents .
 vars L R : SearchTree{X, Y} .
 vars L' R' : NeSearchTree{X, Y} .

 mb [leaf] : empty < K,C > empty : NeSearchTree{X, Y} .
 cmb [1child1] : L' < K,C > empty : NeSearchTree{X, Y}
  if key(max(L')) < K .
 cmb [1child2] : empty < K,C > R' : NeSearchTree{X, Y}
  if K < key(min(R')) .
 cmb [2child] : L' < K,C > R' : NeSearchTree{X, Y}
  if key(max(L')) < K /\ K < key(max(R')) .
```

In these search trees we define the `delete` function as:

```
 op delete : SearchTree{X, Y} X$Elt -> SearchTree{X, Y} .

 eq [dl1] : delete(empty, K) = empty .
 ceq [dl2] : delete(L < K,C > R, K') = delete(L, K') < K,C > R
  if K' < K .
 ceq [dl3] : delete(L < K,C > R, K') = L < K,C > delete(R, K')
  if K < K' .
 eq [dl4] : delete(empty < K,C > R, K) = R .
 eq [dl5] : delete(L < K,C > empty, K) = L .
 ceq [dl6] : delete(L' < K,C > R', K) = L' < K', C' > delete(R', K')
  if L < K', C' > R := max(R') .   *** It should be min(R') !!
```

The function `max` returns a tree whose root is the maximal element of the tree.

```
 op max : NeSearchTree{X, Y} -> NeSearchTree{X, Y} .

 eq [max1] : max(L < K,C > empty) = L < K,C > empty .
 eq [max2] : max(L < K,C > R') = max(R') .
endfm
```

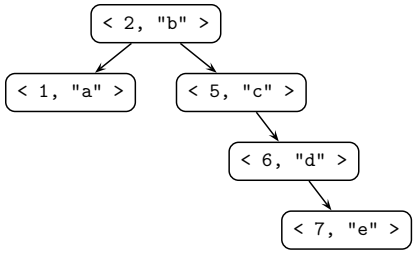

Figure 8: Initial search tree

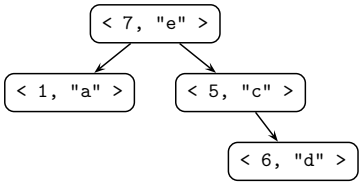

Figure 9: Wrong search tree

We instantiate the keys with integers and the contents with strings, where the `combine` function is mapped to string concatenation.

```
view IntAsStoset from STOSET to INT is
  sort Elt to Int .
endv

view StringAsContents from CONTENTS to STRING is
 sort Contents to String .
 op combine to _+_ .
endv

fmod SEARCH-TREE-TEST is
 protecting SEARCH-TREE{IntAsStoset, StringAsContents} .
endfm
```

Suppose now that we have the search tree shown in Figure 8, where the first component in each node is the key of the node and the second one is the associated value. We want to delete the node with key `2`, but when we apply the function `delete` we obtain the tree shown in Figure 9, that is not a search tree.

If the initial tree is denoted by `initial` we debug the execution with the command

```
Maude> (debug delete(initial, 2) ->
 (empty < 1,"a" > empty) < 7,"e" >(empty < 5,"c" >(empty < 6,"d" > empty))
 in SEARCH-TREE-TEST with dl2 dl3 dl4 dl5 dl6 leaf .)
```

Note that we have specified the equations where we consider the error should be. The equations for `max` and the first equation for `delete` are simple enough to judge them as correct.

The debugger computes the debugging tree and (by using the default divide and query strategy) it prompts the following question:

```
Is this transition (associated with the equation dl3) correct?

delete(empty < 6,"d" > (empty < 7,"e" > empty), 7) -> empty < 6,"d" > empty

Maude> (yes .)
```

The node has been correctly deleted, so we answer affirmatively and the next question is:

```
Is this transition (associated with the equation dl3) correct?

delete(empty < 5,"c" > (empty < 6,"d" > (empty < 7,"e" > empty)), 7) ->
  empty < 5,"c" > (empty < 6,"d" > empty)

Maude> (yes .)
```

The answer is `yes` again, and the debugger is able now of identifying the buggy node, that in this case is the root of the tree.

```
The buggy node is:
delete(intial, 2) -> wrong
With the associated equation: dl6
```

4.3.3 WhileL

We show in this section how to describe the evaluation semantics of a very simple programming language with arithmetic and Boolean expressions, assignments, composition, conditionals, and loops [8].

First, we define the syntax of our language. We define sorts for the arithmetic and Boolean variables, operators, and expressions; the commands; and the programs.

```
fmod WHILEL-SYNTAX is
 pr QID .

 sorts Var BVar Num Boolean Op BOp Exp BExp Com Prog .
```

The arithmetic variables are defined by means of the operator `V` and a quoted identifier. Variables and numbers (implemented from the constant `0` and the successor function) are concrete cases of expression.

```
 op V : Qid -> Var .
 subsort Var < Exp .
 subsort Num < Exp .

 op 0 : -> Num .
 op s : Num -> Num [iter] .

 ops + - * : -> Op .

 op ___ : Exp Op Exp -> Exp [prec 20] .
```

In the same way, Boolean variables are created with the operator `BV`.

```
 op BV : Qid -> BVar .
 subsort BVar < BExp .
 subsort Boolean < BExp .

 ops T F : -> Boolean .
 ops And Or : -> BOp .
 op ___ : BExp BOp BExp -> BExp [prec 20] .
 op Not_ : BExp -> BExp [prec 15] .
 op Equal : Exp Exp -> BExp .
```

We define the following commands: `skip`, assignment, concatenation of commands, if statements, and while statements. We consider a command a concrete case of program.

```
 op skip : -> Com .
 op _:=_ : Var Exp -> Com [prec 30] .
 op _;_ : Com Com -> Com [assoc prec 40] .
 op If_Then_Else_ : BExp Com Com -> Com [prec 50] .
 op While_Do_ : BExp Com -> Com [prec 60] .

 subsort Com < Prog .
endfm
```

The evaluation of arithmetic and Boolean expression is defined for each operator.

```
fmod AP is
 pr WHILEL-SYNTAX .

 vars n n' : Num .
 var bv bv' : Boolean .

 op Ap : Op Num Num -> Num .
 eq Ap(+, 0, n) = n .
 eq Ap(+, s(n), n') = s(Ap(+, n, n')) .
 eq Ap(*, 0, n) = 0 .
 eq Ap(*, s(n), n') = Ap(+, n', Ap(*, n, n')) .
 eq Ap(-, 0, n) = 0 .
 eq Ap(-, s(n), 0) = s(n) .
 eq Ap(-, s(n), s(n')) = Ap(-, n, n') .

 op Ap : BOp Boolean Boolean -> Boolean .
 eq Ap(And, T, bv) = bv .
 eq Ap(And, F, bv) = F .
 eq Ap(Or, T, bv) = T .
 eq Ap(Or, F, bv) = bv .
endfm
```

The environment, defined in the module `ENV` below, keeps the value associated to each variable.

```
fmod ENV is
 inc WHILEL-SYNTAX .

 sorts Value Variable .

 subsorts Num Boolean < Value .
 subsorts Var BVar < Variable .

 sort ENV .
```

The empty environment is represented by `mt`, the assignment of a value to a variable is defined by `_=_`, and bigger environments are created by juxtaposition.

```
  op mt : -> ENV .
  op _=_ : Variable Value -> ENV [prec 20] .
  op __ : ENV ENV -> ENV [assoc id: mt prec 30] .
```

The module also defines look up and update functions. In case a variable has assigned more than one value, the leftmost is the newest and all the others can be deleted.

```
  op _[_] : ENV Var -> Num .
  op _[_] : ENV BVar -> Boolean .
  op _[_/_] : ENV Value Variable -> ENV [prec 35] .

  vars X X' : Variable .
  vars V W : Value .
  var rho : ENV .
  var n : Num .
  var x : Var .

  eq rho [V / X] = (X = V) rho .
  eq (X = V rho)[X'] = if X == X' then V else rho[X'] fi .
  eq (X = V) rho (X = W) = (X = V) rho .
endfm
```

The evaluation semantics of the language is described in the module `EVALUATION`. We use pairs of expressions or commands with environments (called statements) to obtain the result of the execution of a program.

```
fmod EVALUATION is
 pr ENV .
 pr AP .

 sort Statement .
 subsorts Num Boolean ENV < Statement .

 op <_,_> : Exp ENV -> Statement .
 op <_,_> : BExp ENV -> Statement .
 op <_,_> : Com ENV -> Statement .
```

The evaluation of expressions uses the function `Ap` above.

```
  var x : Var .
  vars st st' st'' : ENV .
  vars e e' : Exp .
  var op : Op .
  vars v v' : Num .
  var bx : BVar .
  var bv bv' : Boolean .
  var bop : BOp .
  var be be' : BExp .
  vars C C' : Com .

  eq [CR] : < n, st > = n .
```

```
eq [VarR] : < x, st > = st[x] .

ceq [OpR] : < e op e', st > = Ap(op,v,v')
         if v := < e, st >  /\
            v' := < e', st > .

eq [BCR1] : < T, st > = T .
eq [BCR2] : < F, st > = F .

eq [BVarR] : < bx, st > =  st[bx] .

ceq [BOpR] : < be bop be', st > = Ap(bop,bv,bv')
         if bv := < be, st >  /\
            bv' := < be', st > .

ceq [EqR1] : < Equal(e,e'), st > = T
         if v := < e, st > /\
            < e', st > = v .
ceq [EqR2] : < Equal(e,e'), st > = F
         if v := < e, st > /\
            v' := < e', st > /\
            v =/= v' .

ceq [Not1] : < Not be, st > = F
         if < be, st > = T .
ceq [Not2] : < Not be, st > = T
         if < be, st > = F .
```

The execution of the `skip` command does not change the state.

```
eq [SkipR] : < skip, st > = st .
```

The assignment updates the environment with a new value for the variable.

```
ceq [AsR] : < x := e, st > = st[v / x]
         if v := < e, st >  .
```

The concatenation of statements evaluates the first command, and uses the result to evaluate the rest.

```
ceq [ComR] : < C ; C', st > = st'  ***  <-- ERROR: st' should be st''
          if st' := < C, st > /\
             st'' := < C', st' > .
```

The if statement evaluates the condition and then selects the branch that must be evaluated.

```
ceq [IfR1] : < If be Then C Else C', st > = st'
         if < be, st > = T /\
            st' := < C, st >  .
ceq [IfR2] : < If be Then C Else C', st > = st'
         if < be, st > = F /\
            st' := < C', st > .
```

The while statement also distinguish if the condition is false or not. In the first case, it returns the same environment. In the second one it evaluates the body of the loop and the tries to evaluate the whole statement again.

```
 ceq [WhileR1] : < While be Do C, st > = st
             if < be, st > = F .
 ceq [WhileR2] : < While be Do C, st > = st'
             if < be, st > = T /\
                st' := < C ; (While be Do C), st > .
endfm
```

Now, we can execute the following program, that computes on variable `V('z)` the result of `V('x)` times `V('y)`.

```
< V('z) := 0 ;
     (While Not Equal(V('x),0) Do
       V('z) := V('z) + V('y) ;
       V('x) := V('x) - s(0)),
     V('x) = s^2(0) V('y) = s^3(0) V('z) = 0 >
```

But we obtain the environment

```
V('z) = 0 V('x) = s^2(0) V('y) = s^3(0)
```

which is not correct since the final value of `V('z)` should be `s^6(0)`.

We start the debugging of this reduction with the command

```
Maude> (debug < V('z) := 0 ;
     (While Not Equal(V('x),0) Do
       V('z) := V('z) + V('y) ;
       V('x) := V('x) - s(0)),
     V('x) = s^2(0) V('y) = s^3(0) V('z) = 0 >
 ->
 V('z) = 0 V('x) = s^2(0) V('y) = s^3(0) in EVALUATION
 with AsR ComR WhileR1 WhileR2 .)
```

We have decided to suspect only of the assignment, concatenation, and while equations, that are the ones involved in this program. The first question the debugger prints is

```
Is this transition (associated with the equation WhileR2) correct?

Maude> < While Not Equal(V('x),0) Do
  V('z):= V('z)+ V('y);
  V('x):= V('x)- s(0),
 V('x)= s(0) V('z)= s^3(0) V('y)= s^3(0) >  ->
V('z) = s^6(0) V('x) = s(0) V('y) = s^3(0)

Maude> (no .)
```

This transition is incorrect, because the value of `V('x)` is `s(0)` and it should be `0` (because the loop has finished). The next question is

```
Is this transition (associated with the equation ComR) correct?

< V('x):= V('x) - s(0);
  (While Not Equal(V('x),0) Do
    V('z):= V('z) + V('y);
    V('x):= V('x) - s(0)),
 V('z)= s^6(0) V('x)= s(0) V('y)= s^3(0) > ->
   V('x) = 0 V('z) = s^6(0) V('y) = s^3(0)

Maude> (yes .)
```

This inference is correct, because it returns the intended final result. The following question is

```
Is this transition (associated with the equation ComR) correct?

< V('z):= V('z) + V('y);
  V('x):= V('x) - s(0);
  (While Not Equal(V('x),0) Do
     V('z):= V('z) + V('y);
     V('x):= V('x) - s(0)),
 V('x) = s(0) V('z) = s^3(0) V('y) = s^3(0) > ->
  V('z)= s^6(0) V('x)= s(0) V('y)= s^3(0)

Maude> (no .)
```

Again, the final value of `V('x)` is `0` when it should be `s(0)`, so the reduction is incorrect. The final question is

```
Is this transition (associated with the equation AsR) correct?

< V('z) := V('z) + V('y),
  V('x) = s(0) V('z) = s^3(0) V('y) = s^3(0) > ->
 V('z) = s^6(0) V('x) = s(0) V('y) = s^3(0)

Maude> (yes .)
```

This is an assignment where the value of `V('z)` has been updated. Thus, the inference is correct and the debugger prompts the message

```
The buggy node is:
< V('z) := V('z) + V('y);
  V('x):= V('x)- s(0);
  (While Not Equal(V('x),0) Do
     V('z):= V('z) + V('y);
     V('x):= V('x)- s(0)),
  V('x)= s(0) V('z)= s^3(0) V('y)= s^3(0) > ->
  V('z)= s^6(0) V('x)= s(0) V('y)= s^3(0)
With the associated equation: ComR
```

5 Implementation

We show in this section how the ideas described in the previous sections are implemented. This implementation can be done in Maude itself by means of its reflection capabilities, that allows to use Maude terms and modules as data. Sections 5.1 to 5.3 describe the tree construction stage, where the abbreviated proof tree is constructed by means of a function `createTree` that receives the initial symptom (a wrong inference), the module where it took place, a correct module (possibly empty), and a set of suspicious labels. The tree navigation is explained in Sections 5.4 and 5.5, where the two strategies and the interaction with the user are implemented.

5.1 Proof tree definition

In this section we show how to represent in Maude the proof trees needed by the debugging process. First, we implement parametric general trees with generic data in each node. Then, we instantiate them by defining the concrete data for building our proof trees.

The parameterized module that describes the tree behavior receives the theory `TRIV` (that simply requires a sort `Elt`) as parameter. We use lists of natural numbers to identify (the position of) each node.

```
fmod TREE{X :: TRIV} is
 pr NAT-LIST .
```

General trees are defined by means of the constructor `tree`, composed by some contents (received from the theory) and a `Forest`, which in turn is a list of trees.

```
  sorts Tree Forest .
  subsort Tree < Forest .

  op tree : X$Elt Forest -> Tree .

  op mtForest : -> Forest .
  op __ : Forest Forest -> Forest [assoc id: mtForest] .
```

We define now some operations over trees. The function `find` extracts the nth tree from a forest. Notice the use of `~>` in the operator declaration, stating that the function is partial, that is, if the number received as argument is bigger than the size of the forest, the function is not defined.

```
  var  T : Tree .
  vars F F' F'' : Forest .
  var  N : Nat .
  var  C : X$Elt .
  var  NL : NatList .

  op find : Forest Nat ~> Tree .
  eq find(T F, 0) = T .
  eq find(T F, s(N)) = find(F, N) .
```

The function `size` calculates the length of a forest.

```
  op size : Forest -> Nat .
  eq size(mtForest) = 0 .
  eq size(T F) = s(size(F)) .
```

Given a tree and a list identifying a node, the function `getContents` extracts the contents of the node described by the list, `getForest` extracts its forest, `getSubTree` returns the whole subtree, and `hasOffspring?` checks if the forest of the node is not empty.

```
  op getContents : Tree NatList ~> X$Elt .
  eq getContents(tree(C, F), N NL) = getContents(find(F, N), NL) .
  eq getContents(tree(C, F), nil) = C .

  op getForest : Tree NatList ~> Forest .
  eq getForest(tree(C, F), N NL) = getForest(find(F, N), NL) .
  eq getForest(tree(C, F), nil) = F .

  op getSubTree : Tree NatList ~> Tree .
  eq getSubTree(tree(C, F), N NL) = getSubTree(find(F, N), NL) .
  eq getSubTree(T, nil) = T .
```

```
 op hasOffspring? : Tree NatList ~> Bool .
 eq hasOffspring?(tree(C, F), N NL) = hasOffspring?(find(F, N), NL) .
 eq hasOffspring?(tree(C, F), nil) = F =/= mtForest .
endfm
```

We define now proof trees by instantiating the values contained in each node. We want to know the name of the equation (or the membership) associated with the node, and the lefthand and righthand sides of the reduction (or the term and sort of a membership). We declare a sort `Inference` that keeps these values, distinguishing between equations and memberships.

```
fmod PROOF-TREE-NODE is
 pr META-TERM .

 sort Inference .
 op _:_->_ : Qid Term Term -> Inference .
 op _:_:_ : Qid Term Type -> Inference .
```

We are also interested in the number of nodes in each subtree, so we define a constructor `node` that contains an inference and a natural number, with their corresponding look up functions.

```
 sort Node .
 op node : Inference Nat -> Node .

 var I : Inference .
 var N : Nat .

 op inf : Node -> Inference .
 op offspring : Node -> Nat .
 eq inf(node(I, N)) = I .
 eq offspring(node(I, N)) = N .
endfm
```

We use this module to create a view from the `TRIV` theory.

```
view PROOF-TREE-NODE from TRIV to PROOF-TREE-NODE is
 sort Elt to Node .
endv
```

We obtain our tree by instantiating the module `TREE` above.

```
fmod PROOF-TREE is
 pr TREE{PROOF-TREE-NODE} .
```

In addition, this module defines functions to obtain the different components from the root: `getLabel` extracts the statement identifier, `getFirstTerm` returns the first term from the inference, `getSecondTerm` extracts the second term in case the inference is related to an equation, and `getOffspring` returns the number of nodes in the tree.

```
 vars F F' : Forest .
 var  Q : Qid .
 vars T T' : Term .
 var  I : Inference .
```

```
var  N : Nat .
var  NL : NatList .
var  TR : Tree .
var  S : Type .

op getLabel : Tree -> Qid .
eq getLabel(tree(node(Q : T -> T', N), F)) = Q .
eq getLabel(tree(node(Q : T : S, N), F)) = Q .

op getFirstTerm : Tree -> Term .
eq getFirstTerm(tree(node(Q : T -> T', N), F)) = T .
eq getFirstTerm(tree(node(Q : T : S, N), F)) = T .

op getSecondTerm : Tree ~> Term .
eq getSecondTerm(tree(node(Q : T -> T', N), F)) = T' .

op getOffspring : Tree -> Nat .
eq getOffspring(tree(node(I, N), F)) = N .
```

We also have functions that extract the label and the number of nodes in the subtree of a concrete node.

```
op getLabel : Tree NatList ~> Qid .
eq getLabel(TR, NL) = getLabel(getSubTree(TR, NL)) .

op getOffspring : Tree NatList ~> Nat .
eq getOffspring(TR, NL) = getOffspring(getSubTree(TR, NL)) .
```

When a tree is modified by deleting some nodes, the information about the size of each subtree becomes obsolete. We use a function `calculateOffspring` that calculates the number of nodes in each subtree.

```
op calculateOffspring : Tree -> Tree .
op calculateOffspring* : Forest -> Forest .

ceq calculateOffspring(tree(node(I, N), F)) =
                       tree(node(I, getOffspring*(F') + 1), F')
 if F' := calculateOffspring*(F) .

eq calculateOffspring*(mtForest) = mtForest .
eq calculateOffspring*(TR F) = calculateOffspring(TR) calculateOffspring*(F) .

op getOffspring* : Forest -> Nat .
eq getOffspring*(mtForest) = 0 .
eq getOffspring*(TR F) = getOffspring(TR) + getOffspring*(F) .
```

The function `deleteSubTree` removes a subtree from the tree, updating the information about the number of nodes of each subtree, being the third parameter the size of the tree to be deleted.

```
op deleteSubTree : Tree NatList Nat ~> Tree .
op deleteSubTree* : Forest NatList Nat ~> Forest .
eq deleteSubTree(tree(node(I, N), F), N' NL, N'') =
   tree(node(I, sd(N, N'')), deleteSubTree*(F, N' NL, N'')) .
eq deleteSubTree(tree(node(I, N), F), nil, N') = mtForest .
```

```
 eq deleteSubTree*(TR F, s(N) NL, N') = TR deleteSubTree*(F, N NL, N') .
 eq deleteSubTree*(TR F, 0 NL, N) = deleteSubTree(TR, NL, N) F .
 eq deleteSubTree*(TR, nil, N) = mtForest .
endfm
```

5.2 Auxiliary modules

The `MAYBE` module adds a special value `maybe` to the sort used in its instantiation.

```
fmod MAYBE{X :: TRIV} is
 sort Maybe{X} .
 subsort X$Elt < Maybe{X} .
 op maybe : -> Maybe{X} [ctor] .
endfm
```

The module `MODULES` deals with the operations defined over modules. It defines the following functions:

```
fmod MODULES is
 pr META-LEVEL .
 pr MAYBE{Module} .
```

- `delete`, that deletes the equations and memberships specified by a label set. Since these labels will specify the suspicious statements, we only keep the trusted ones.

  ```
  var  Q : Qid .
  var  QS : QidSet .
  var  EqS : EquationSet .
  var  IL : ImportList .
  var  SS : SortSet .
  var  SSDS : SubsortDeclSet .
  var  ODS : OpDeclSet .
  var  MAS : MembAxSet .
  var  C : Condition .
  var  M : Module .
  var  AtS : AttrSet .
  vars T T' : Term .
  var  N : Nat .
  var  S : Type .

  op delete : EquationSet QidSet -> EquationSet .
  eq delete(eq T = T' [label(Q) AtS] . EqS, Q ; QS) = delete(EqS, QS) .
  eq delete(ceq T = T' if C [label(Q) AtS] . EqS, Q ; QS) = delete(EqS, QS) .
  eq delete(EqS, QS) = EqS [owise] .

  op delete : MembAxSet QidSet -> MembAxSet .
  eq delete(mb T : S [label(Q) AtS] . MAS, Q ; QS) = delete(MAS, QS) .
  eq delete(cmb T : S if C [label(Q) AtS] . MAS, Q ; QS) = delete(MAS, QS) .
  eq delete(MAS, QS) = MAS [owise] .
  ```

- `extractLabels` extracts all the labeled statements from a module. We use this function to know which statements must be treated if the user decides to debug without selecting specific lables.

```
op extractLabels : Module -> QidSet .
op extractLabels : EquationSet -> QidSet .
op extractLabels : MembAxSet -> QidSet .

eq extractLabels(M) = extractLabels(getEqs(M)) ; extractLabels(getMbs(M)) .

eq extractLabels(eq T = T' [label(Q) AtS] . EqS) = Q ; extractLabels(EqS) .
eq extractLabels(ceq T = T' if C [label(Q) AtS] . EqS) =
                                                  Q ; extractLabels(EqS) .
eq extractLabels(EqS) = none [owise] .

eq extractLabels(mb T : S [label(Q) AtS] . MAS) = Q ; extractLabels(MAS) .
eq extractLabels(cmb T : S if C [label(Q) AtS] . MAS) =
                                                  Q ; extractLabels(MAS) .
eq extractLabels(MAS) = none [owise] .
```

- `transform`, that given a module and a set of labels transforms the module by deleting the statements specified in the set.

```
op transform : Module QidSet -> Module .
eq transform(fmod Q is IL sorts SS . SSDS ODS MAS EqS endfm, QS) =
 fmod Q is
  IL
  sorts SS .
  SSDS
  ODS
  delete(MAS, QS)
  delete(EqS, QS)
 endfm .
```

- `generalEq` and `generalMb` return the given equation or membership as a conditional one.

```
op generalEq : Equation -> Equation .
eq generalEq(eq T = T' [AtS] .) = ceq T = T' if nil [AtS] . .
eq generalEq(Eq) = Eq [owise] .

op generalMb : MembAx -> MembAx .
eq generalMb(mb T : S [AtS] .) = cmb T : S if nil [AtS] . .
eq generalMb(MA) = MA [owise] .
```

- `in?`, that checks if an attribute set contains a label that is also included in a given set.

```
op in? : AttrSet QidSet -> Bool .
eq in?(label(Q) AtS, Q ; QS) = true .
eq in?(AtS, QS) = false [owise] .
```

- `owise?` checks if an attribute set contains the attribute `owise`.

```
op owise? : AttrSet -> Bool .
eq owise?(owise AtS) = true .
eq owise?(AtS) = false [owise] .
```

- `label`, that extracts the label from an attribute set.

```
op label : AttrSet ~> Qid .
eq label(label(Q) AtS) = Q .
```

- `reduce`, `normal`, and `type`, that abbreviate the composition of other functions.

```
op normal : Maybe{Module} Term ~> Term .
eq normal(M, T) = getTerm(metaNormalize(M, T)) .

op reduce : Maybe{Module} Term ~> Term .
eq reduce(M, T) = getTerm(metaReduce(M, T)) .

op type : Maybe{Module} Term ~> Term .
eq type(M, T) = getType(metaReduce(M, T)) .
endfm
```

Finally, we define pairs of term lists and forests, that will be used in the computation of the debugging tree.

```
fmod PAIR is
 pr PROOF-TREE .

 sort Pair .
 op <_,_> : TermList Forest -> Pair .
endfm
```

5.3 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 function `createTree` controls the construction of this tree. It receives the module where a suspicious inference took place, a correct module (or the constant `maybe` 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 reduction 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 (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.

```
fmod PROOF-TREE-CONSTRUCTION is
 pr MODULES .
 pr PAIR .
 pr MAYBE{Forest} * (op maybe to noProof) .

 vars OM TM M M' : Module .
 vars T T' T'' L R R' : Term .
 var  QS : QidSet .
```

```
var  CM : Maybe{Module} .
var  F : Forest .

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) .
```

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

```
op contract : Tree -> Tree .
eq contract(tree(node('root : T -> T', N),
        tree(node(Q : T -> T', N'), F))) = tree(node(Q : T -> T', N'), F) .
eq contract(TR) = TR [owise] .
```

It is also possible that the inference we are debugging is a membership. The `createTree` function is defined in a similar way to the one above, calculating the least sort possible for the term.

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

Where `createForest` computes the normal form of the term received as argument and then builds the proof tree for the sort inference.

We use the function `createForest` to create a forest of abbreviated trees. It receives as parameters the module where the reduction was done, the transformed module (that only allow trusted inferences), a correct module (possibly `maybe`) to check the inferences, two terms representing the inference whose proof tree we want to generate, and a set of suspicious equations and memberships. This function checks if the normalized forms of the terms are equal, if it can be reduced by using only trusted statements, and if the correct module can calculate this reduction; in such cases, there is no need to calculate the tree. 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 normal(OM, T) == normal(OM, T') .
ceq createForest(OM, TM, CM, T, T', QS) = mtForest
 if reduce(TM, T) == T' .
ceq createForest(OM, TM, M, T, T', QS) = mtForest
 if 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 reproduces a specific behavior of the *congruence* rule shown in Figure 1) and the forest of abbreviated trees generated by these reductions.

```
var  Q : Qid .
vars TL TL' : TermList .

op reduceSubterms : Module Module Maybe{Module} Term QidSet -> Pair .
op reduceSubterms : Module Module Maybe{Module} TermList QidSet Pair -> Pair .

ceq reduceSubterms(OM, TM, CM, Q[TL], QS) = < normal(OM, Q[TL']), F >
 if < TL', F > := reduceSubterms(OM, TM, CM, TL, QS, < empty, mtForest >) .

eq reduceSubterms(OM, TM, CM, empty, QS, < TL, F >) = < TL, F > .
ceq reduceSubterms(OM, TM, CM, (T, TL'), QS, < TL, F >) =
             reduceSubterms(OM, TM, CM, TL', QS, < (TL, T'),
                                    F createForest(OM, TM, CM, T, T', QS) >)
 if T' := reduce(OM, T) .
```

The function `applyEq` tries to apply (at the top) one equation,[2] by using the *replacement* rule from Figure 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.

```
vars C C' COND : Condition .

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.

```
vars SB SB' : Substitution .

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) .
```

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

If we cannot apply any equation without the `otherwise` attribute, we check the other equations (notice the use of the `owise` attribute in our own equation!).

```
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) /\
   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) [owise] .

eq applyEq(OM, TM, CM, T, T', QS, EqS) = mtForest [owise] .
```

We show now how the proof trees for the conditions in conditional equations (and memberships) are generated. Since `conditionForest` is used after having checked that the condition is fulfilled (by the function `metaMatch` above), we do not check it again.

We distinguish between the different types of conditions. If we have an equation, we add the trees of the reduction of the terms to their normal forms.

```
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. We generate the forest for the normal form of the righthand side.

```
eq conditionForest(T := T' /\ COND, OM, TM, CM, QS) =
              createForest(OM, TM, CM, T', reduce(OM, T), QS)
              conditionForest(COND, OM, TM, CM, QS) .
```

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.

```
var Ty S : Type .

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

Finally, the empty condition generates the empty forest.

```
eq conditionForest(nil, OM, TM, CM, QS) = mtForest .
```

To generate the forest for memberships we use 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 memberships.

```
var  ODS : OpDeclSet .
var  MAS : MembAxSet .

eq createForest(OM, TM, CM, T, S, QS, ODS, MAS) =
        if applyOp(OM, TM, CM, T, S, QS, ODS) =/= noProof then
                    applyOp(OM, TM, CM, T, S, QS, ODS)
        else
                    applyMb(OM, TM, CM, T, S, QS, MAS)
        fi .
```

To check the operators we examine that both the arity and co-arity of the term and the declaration fit. We recursively calculate the forest generated by the subterms. Notice that we never generate a new node for the application of an operator, because we always trust the signature.

```
var  TyL : TypeList .
var  OD : OpDecl .
var  AtS : AttrSet .
var  CONST : Constant .

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] .
```

The function `checkTypes` examines that all the subterms have the appropriate type in the operator declaration, while `createForest*` generates the corresponding forest.

```
op checkTypes : TermList TypeList Module -> Bool .
eq checkTypes(empty, nil, M) = true .
ceq checkTypes((T, TL), Ty TyL, M) = checkTypes(TL, TyL, M)
```

```
   if sortLeq(M, type(M, T), Ty) .
  eq checkTypes(TL, TyL, M) = false [owise] .

  op createForest* : Module Module Maybe{Module} TermList QidSet -> Forest .
  eq createForest*(OM, TM, CM, empty, QS) = mtForest .
  eq createForest*(OM, TM, CM, (T, TL), QS) =
       createForest(OM, TM, CM, T, type(OM, T), QS)
       createForest*(OM, TM, CM, TL, QS) .
```

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.

```
  var  MA : MembAx .

  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] .
 endfm
```

5.4 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 algorithm in charge of calculating the next node uses auxiliary functions that return pairs consisting of a list of natural numbers (identifying the best node found thus far) and a natural number (specifying the cost associated with this node). This pair, and the corresponding look up functions, are defined in the module `NAT-PAIR`.

```
 fmod NAT-PAIR is
  pr NAT-LIST .

  sort NPair .

  var NL : NatList .
  var N : Nat .

  op <_,_> : NatList Nat -> NPair .

  op first : NPair -> NatList .
  eq first(< NL, N >) = NL .
```

```
 op second : NPair -> Nat .
 eq second(< NL, N >) = N .
endfm
```

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. That is, a subtree whose size is the closest one to half the size of the tree.

```
fmod DIVIDE-QUERY-STRATEGY is
 pr PROOF-TREE .
 pr NAT-PAIR .

 var  I : Inference .
 vars N N' NODES LAST_DIFF BEST_DIFF NEW_DIFF : Nat .
 var  F : Forest .
 vars NL NL' BEST_NODE : NatList .
 var  ND : Node .
 var  T : Tree .

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

 op searchBestNode : Tree -> NatList .
```

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 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)) .
```

This function keeps the information about the last difference in order to stop searching when the current difference is bigger than the last one. 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, in this case the function returns the current best node and best difference.

```
 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 .
```

As said above, this function recursively traverses the forest and creates the new node identifiers with its accumulator parameter.

```
 op searchBestNode* : Forest Nat Nat Nat NatList NatList Nat -> NPair .

 eq searchBestNode*(mtForest, NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL, N) =
                                     < BEST_NODE, BEST_DIFF > .
 ceq searchBestNode*(T F, NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL, N) =
      if N' < BEST_DIFF then
        searchBestNode*(F, NODES, LAST_DIFF, N', NL', NL, s(N))
      else
        searchBestNode*(F, NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL, s(N))
      fi
  if < NL', N' > :=
     searchBestNode(T, NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL N) .
endfm
```

In the divide and query strategy, the user can decide to trust the statement associated to the current question. In this case, the tree will be pruned, deleting all the nodes associated with this statement.

```
fmod TREE-PRUNING is
 pr PROOF-TREE .

 op prune : Tree Qid -> Tree .
```

Notice that, since we cannot prune the root, we keep it and use an auxiliary function `pruneAux` for the rest of the tree. Once the tree has been pruned, the size of all the subtrees is recalculated.

```
 var  TR : Tree .
 vars T T' : Term .
 vars Q Q' : Qid .
 var  N : Nat .
 var  F : Forest .
 var  S : Type .
 var  PN : Node .

 eq prune(tree(node(Q : T -> T', N), F), Q') =
        calculateOffspring(tree(node(Q : T -> T', N), prune*(F, Q'))) .

 op pruneAux : Tree Qid -> Tree .
 eq pruneAux(tree(node(Q : T -> T', N), F), Q) = prune*(F, Q) .
 eq pruneAux(tree(node(Q : T : S, N), F), Q) = prune*(F, Q) .
 eq pruneAux(tree(PN, F), Q') = tree(PN, prune*(F, Q')) [owise] .
```

The function `prune*` recursively traverses the forest.

```
 op prune* : Forest Qid -> Forest .
 eq prune*(mtForest, Q) = mtForest .
 eq prune*(TR F, Q) = pruneAux(TR, Q) prune*(F, Q) .
endfm
```

5.5 Input/output

Now we implement the interaction between the user and the debugger by defining the behavior of the commands described in Section 4.2. This can be done in Maude itself by using the predefined module `LOOP-MODE` [6, Chapter 17], that handles the input/output and maintains the persistent state of the tool.

First we define the signature of these commands. The module `EXTENDED-SORTS` specifies how to introduce the name of (possibly parameterized) sorts.

```
fmod EXTENDED-SORTS is
 sorts SortToken ViewToken Sort ViewExp .

 subsorts SortToken < Sort .
 subsort ViewToken < ViewExp .

 op _{_} : Sort ViewExp -> Sort [prec 40] .
 op _,_ : ViewExp ViewExp -> ViewExp [assoc] .
endfm
```

With these definitions we create a module with all the possible commands.

```
fmod DD-COMMANDS is
 including EXTENDED-SORTS .
  sorts Token Bubble NeTokenList Command .

  op correct‘with_. : Token -> Command .
  op top-down‘strategy‘. : -> Command .
  op divide-query‘strategy‘. : -> Command .
  op debug_->_in_. : Bubble Bubble Token -> Command .
  op debug_->_in_with_. : Bubble Bubble Token NeTokenList -> Command .
  op debug_:_in_. : Bubble Sort Token -> Command .
  op debug_:_in_with_. : Bubble Sort Token NeTokenList -> Command .
  op node_. : Token -> Command .
  op all‘valid‘. : -> Command .
  op yes‘. : -> Command .
  op no‘. : -> Command .
  op trust‘. : -> Command .
endfm
```

The complete signature of the debugger is defined as follows:

```
fmod DD-SIGN is
 including DD-COMMANDS .

 sort Input .
 subsort Command < Input .
endfm
```

The `LOOP-MODE` module defines an operator `[_,_,_]` that receives an input stream (the first argument), an output stream (the third argument), and a state (given by its second argument) of sort `State`. Concrete applications must specify `State` to define the data that must be kept during the execution of the program. In our case, the state consists in several attributes enclosed by the operator `<_>`. These attributes are:

- the debugging `tree`, that initially is empty, and will be traversed during the debugging process.

- the `current` node of the tree that we are analyzing, represented by a list of natural numbers.
- the `strategy` to traverse the tree. The top-down strategy is represented by the number `1`, and divide and query is represented by the number `2`.
- the `module` where debugging takes place. It has sort `Maybe{Module}`, so its value initially is `maybe`.
- the correct module used to prune the debugging tree (`correction`). Since this value is optional, it also has sort `Maybe{Module}`, and its value is initially `maybe`.
- `input` and `output`, that are used to recover the user commands and to prompt the debugger information.

```
mod DD is
 inc LOOP-MODE .
 pr PRINT .
 pr TREE-PRUNING .
 pr DIVIDE-QUERY-STRATEGY .
 pr PROOF-TREE-CONSTRUCTION .
 pr CONFIGURATION .

 op <_> : AttributeSet -> State .

 op tree :_ : Forest -> Attribute .
 op current :_ : NatList -> Attribute .
 op strategy :_ : Nat -> Attribute .
 op module :_ : Maybe{Module} -> Attribute .
 op correction :_ : Maybe{Module} -> Attribute .
 op input :_ : TermList -> Attribute .
 op output :_ : QidList -> Attribute .
```

We define now a grammar that contains the signature previously defined.

```
 op GRAMMAR : -> Module .
 eq GRAMMAR =
 (fmod 'GRAMMAR is
  including 'DD-SIGN .
  ...) .
```

When an input stream is received, it is parsed by using this grammar, and if it is correct, inserted into the `input` attribute. Otherwise the input is discarded.

```
 vars M Q Q' : Qid .
 vars QIL QIL' QIL'' : QidList .
 var  ST : State .
 var  AtS : AttributeSet .
 vars N N' N'' : Nat .
 vars NL NL' : NatList .
 vars RP RP' : ResultPair .
 vars T T' T'' T''' : Term .
 vars MOD MOD' : Module .
 vars PT PT' : Tree .
 vars F F' F'' : Forest .
 vars MM MM' : Maybe{Module} .
```

```
var S : Sort .

crl [in] :
    [QIL, < input : empty, AtS >, QIL']
 => [nil,
    < input : getTerm(metaParse(GRAMMAR, QIL, 'Input)), AtS >,
    QIL']
 if QIL =/= nil /\ metaParse(GRAMMAR, QIL, 'Input) : ResultPair .
```

The `out` rule extracts a list of quoted identifiers from the `output` attribute and puts it in the output stream.

```
crl [out] :
    [QIL, < output : QIL', AtS >, QIL'']
 => [QIL, < output : nil, AtS >, (QIL'' QIL')]
 if QIL' =/= nil .
```

To start the debugging process we type the command `loop init-debug .`, that initializes the attributes explained above. Note that the default strategy is divide and query.

```
rl [loop] :
   init-debug
=> [nil, < tree : mtForest, current : nil, strategy : 2, module : maybe,
           correction : maybe, input : empty, output : nil >, nil] .
```

The two commands to select the strategy just change the value of this attribute.

```
eq [top-down-strategy] :
   < input : ('top-down`strategy`.. Command), output : nil, strategy : N, AtS >
 = < input : empty, output : ('Top-down 'strategy 'selected.),
     strategy : 1, AtS > .

eq [divide-query-strategy] :
   < input : ('divide-query`strategy`.. Command), output : nil,
     strategy : N, AtS >
 = < input : empty, output : ('Divide 'and 'Query 'strategy 'selected.),
     strategy : 2, AtS > .
```

The command to select a correct module inserts it in the `correction` attribute.

```
ceq [init-terms] :
    < input : ('correct`with_.['token[T]]), output : nil, correction : MM,
      tree : mtForest, AtS >
  = < input : empty, output : ('Module 'inserted.), correction : MOD,
      tree : mtForest, AtS >
 if Q := downQid(T)  /\
    MOD := upModule(Q, false) .
```

When the user introduces the terms of the invalid reduction, the module where the inference took place, and the labels of the suspicious equations and memberships to start the process, the debugging tree is computed and the first question, depending on the strategy selected, is prompted. We show below the case in which the divide and query strategy is selected.

```
ceq [init-terms-divide-query] :
    < input : ('debug_->_in_.['bubble[T], 'bubble[T'], 'token[T'']]),
      output : nil, tree : F, module : MM, current : NL, strategy : 2,
      correction : MM', AtS >
  = < input : empty, output : if getOffspring(PT) > 1 then
                                ('\n printNode(PT, NL', MOD))
                              else
                                (printCriticalNode(PT, nil, MOD)
                                   printCode(PT, nil, MOD))
                              fi,
      tree : PT, module : MOD, current : NL', strategy : 2, correction : MM', AtS >
 if Q := downQid(T'')  /\
    MOD := upModule(Q, true) /\
    RP := metaParse(MOD, downQidList(T), anyType) /\
    RP' := metaParse(MOD, downQidList(T'), anyType) /\
    PT := createTree(MOD, MM', getTerm(RP), getTerm(RP'), extractLabels(MOD)) /\
    NL' := searchBestNode(PT) .
```

A similar equation is applied if the user also introduces the set of suspicious labels. In this case, the top-down strategy is selected.

```
ceq [init-terms-top-down] :
    < input : ('debug_->_in_with_.['bubble[T], 'bubble[T'], 'token[T''],
                                   'neTokenList[T''']]),
      output : nil, tree : F, module : MM, current : NL, strategy : 1,
      correction : MM', AtS >
  = < input : empty, output : if getOffspring(PT) > 1 then
                                 (one-wrong? '\n printOffspring(PT, nil, MOD))
                              else
                                 (printCriticalNode(PT, nil, MOD)
                                   printCode(PT, nil, MOD))
                              fi,
      tree : PT, module : MOD, current : nil, strategy : 1, correction : MM', AtS >
 if Q := downQid(T'')  /\
    MOD := upModule(Q, true) /\
    QIL := downQidList(T''') /\
    RP := metaParse(MOD, downQidList(T), anyType) /\
    RP' := metaParse(MOD, downQidList(T'), anyType) /\
    PT := createTree(MOD, MM', getTerm(RP), getTerm(RP'), list2set(QIL)) .
```

If the transition associated with the current node is wrong, we check its subtree if it is not a leaf.

```
ceq [divide-query-traversal] :
    < input : ('no`.. Command), output : nil, strategy : 2,
      tree : PT, current : NL, module : MOD, AtS >
  = < input : empty, output : ('\n printNode(PT', NL', MOD)),
      strategy : 2, tree : PT', current : NL', module : MOD, AtS >
 if PT' := getSubTree(PT, NL) /\
    getOffspring(PT') > 1 /\
    NL' := searchBestNode(PT') .
```

If the current node's inference is trusted, we discard its subtree and prune the rest of the tree.

```
ceq [divide-query-traversal] :
    < input : ('trust‘.. Command), output : nil, strategy : 2,
      tree : PT, current : NL, module : MOD, AtS >
  = < input : empty, output : ('\n printNode(PT', NL', MOD)),
      strategy : 2, tree : PT', current : NL', module : MOD, AtS >
 if N := getOffspring(PT, NL) /\
    Q := getLabel(PT, NL) /\
    PT' := prune(deleteSubTree(PT, NL, N), Q) /\
    NL' := searchBestNode(PT') /\
    NL' =/= nil .
```

Once the tree has been updated after receiving an answer, if the current root has no children, the node is buggy and it is associated with the wrong rule, so we prompt its associated code.

```
ceq [divide-query-traversal] :
    < input : ('yes‘.. Command), output : nil, strategy : 2,
      tree : PT, current : NL, module : MOD, AtS >
  = < input : empty, output : (printCriticalNode(PT', nil, MOD)
                               printCode(PT', nil, MOD)),
      strategy : 2, tree : PT', current : nil, module : MOD, AtS >
 if N := getOffspring(PT, NL) /\
    PT' := deleteSubTree(PT, NL, N) /\
    getOffspring(PT') = 1 .
```

The equations for the top-down strategy are quite similar, if the user decides that a node is incorrect, we just select it for the next question by updating the `current` attribute.

```
ceq [top-down-traversal] :
    < input : ('node_.['token[T]]), output : nil, strategy : 1, tree : PT,
      current : NL, module : MOD, AtS >
  = < input : empty, output : (one-wrong? printOffspring(PT, NL N, MOD)),
      strategy : 1, tree : PT, current : (NL N), module : MOD, AtS >
 if N := downNat(T) /\
    N < size(getForest(PT, NL)) /\
    hasOffspring?(PT, NL N) .
```

If the selected node has no children, then we have found the buggy node and it is prompted

```
ceq [top-down-traversal] :
    < input : ('node_.['token[T]]), output : nil, strategy : 1, tree : PT,
      current : NL, module : MOD, AtS >
  = < input : empty, output : (printCriticalNode(PT, NL N, MOD)
                               printCode(PT, NL N, MOD)),
      strategy : 1, tree : PT, current : (NL N), module : MOD, AtS >
 if N := downNat(T) /\
    N < size(getForest(PT, NL)) /\
    not hasOffspring?(PT, NL N) .
```

In the same way, if all the inferences prompted are correct, the current node is the wrong one and it is prompted.

```
eq [top-down-traversal] :
    < input : ('all‘valid‘.. Command), output : nil, strategy : 1, tree : PT,
```

```
        current : NL, module : MOD, AtS >
    = < input : empty, output : (printCriticalNode(PT, NL, MOD)
                                 printCode(PT, NL, MOD)),
        strategy : 1, tree : PT, current : NL, module : MOD, AtS > .
endfm
```

Finally, we have also implemented equations that deal with errors by printing some messages.

6 Conclusions and future work

In this paper we have developed the foundations of declarative debugging of executable membership equational logic 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.

We want to improve the interaction with the user in several ways. One of these ways is to minimize the number of questions asked to the user, that can be achieved by trusting a whole module or keeping track of the questions already answered, in order to avoid asking the same question twice. Another way is to improve the user interface by providing 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 specications 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* 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 *LOPSTR*, pages 1–16, 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 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.

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

[6] 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.

[7] 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.

[8] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Wiley & Sons, 1990.

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

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

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

[12] 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.

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

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

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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, 2000. Springer-Verlag.

[21] P. Wadler. Why no one uses functional languages. *SIGPLAN Not.*, 33(8):23–27, August 1998.