Declarative Debugging of Maude Modules*

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

Technical Report SIC-6-08

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid

April, 2008

*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 modules: functional modules correspond to executable specifications in membership equational logic, while system modules correspond to rewrite theories. First we describe the construction of appropriate debugging trees for oriented equational and membership inferences and rewrite rules. These trees are obtained as the result of collapsing in proof trees all those nodes whose correction does not need any justification.

We include several extended examples to illustrate the use of the declarative debugger and its main features, such as two possible constructions of the debugging tree, two different strategies to traverse it, use of a correct module to reduce the number of questions asked to the user, selection of trusted vs. suspicious statements by means of labels, and trusting of statements "on the fly."

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.

Keywords: declarative debugging, rewriting logic, Maude, metalevel implementation

Contents

1 Introduction

In this paper we present a declarative debugger for *Maude specifications*, including equational functional specifications and concurrent systems specifications. Maude [9] 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 to specifications in *rewriting logic* [14], 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, 15], which, in addition to equations, allows the statement of *membership axioms* 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. Rewriting logic extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system. Maude *system modules* are used to define specifications in this logic.

Moreover, exploiting the fact that rewriting logic is *reflective* [8, 10], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [9, Chap. 14], a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications.

The Maude system supports several approaches for debugging Maude programs: tracing, term coloring, and using an internal debugger [9, Chap. 22]. The tracing facilities allow us to follow the execution of a specification, that is, the sequence of application of statements 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 Maude specifications.

Declarative debugging, also known as algorithmic debugging, was first introduced by E. Y. Shapiro [21]. It has been widely employed in the logic [12, 16, 23], functional [19, 18, 20], and multi-paradigm programming [7, 4, 13] 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 [17] 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 child nodes by some logical inference. Diagnosis proceeds by traversing the debugging tree, asking questions to an external oracle (generally the user) until a so-called *buggy node* is found. A buggy node is a node containing an erroneous result, but whose children 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. 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 [22].

The application of declarative debugging to Maude functional modules, that is, executable membership equational logic specifications, was already studied in our previous papers [6, 5]. The executability requirements of Maude functional modules mean that they are assumed to be confluent, terminating, and sort-decreasing[1] [9], so that, by orienting the equations from left to right, each term can be reduced to a unique canonical form, and semantic equality of two terms can be checked by reducing both of them to their respective canonical forms and checking that they coincide. These requirements are assumed in the form of the questions appearing in the debugging tree. In this paper, we considerably extend that work by also considering system modules. Now, since the specifications described in this kind of modules

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

can be non-terminating and non-confluent, their handling must be quite different.

In both cases, functional and system modules, the debugging process starts with an incorrect computation from the initial term to an unexpected one. The debugger then builds for that inference an appropriate debugging tree which is an abbreviation of the corresponding proof tree obtained by applying the inference rules of membership equational logic or rewriting logic. The abbreviation consists in collapsing all those nodes whose correction does not need any justification, such as those related with transitivity or congruence. Since the questions are located in the debugging tree, the answer allows the debugger to discard a subset of the questions, leading and shortening the debugging process. In the case of functional modules, the debugger builds a debugging tree whose nodes give rise to questions of the form "Is it correct that T fully reduces to T'?" which in general are easier to answer. However, in the absence of confluence and termination, this kind of questions do not make sense; thus, in the case of system modules, we have decided to develop two different trees whose nodes produce questions of the form "Is it correct that T is rewritten to T'?" where the difference consists in the number of steps involved in the rewrite. While one of the trees refers only to one-step rewrites, which are often easier to answer, the other one can also refer to many-steps rewrites that, although may be harder to answer, in general allow to discard a bigger subset of nodes. The user, depending on the debugged specification or his "ability" to answer questions involving several rewrite steps, can choose between these two kinds of trees.

The current version of the tool has the following characteristics:

- It supports all kinds of modules: for example, 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; modules can be parameterized; and operators' arguments can be `frozen` (see [9] for the meaning of all these concepts).

- In case of debugging a rewrite computation, two different debugging trees can be built: one whose questions are related to one-step rewrites and another whose questions are related to several steps. The latter tree is partially built so that any node corresponding to a one-step rewrite is expanded only when the navigation process reaches it.

- It provides two strategies to traverse the debugging tree: *top-down*, that traverses the tree from the root asking each time for the correctness of all the children of the current node, and then continues with one of the incorrect children; and *divide and query*, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise.

- Before starting the debugging process, the user can select a module containing only correct statements. By checking the correctness of the inferences with respect to this module (i.e., using this module as oracle) the debugger can reduce the number of questions asked to the user.

- It allows to debug specifications where some statements 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.

- The user can decide to use all the labeled statements as suspicious or can use only a subset by trusting labels and modules. 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.

- It provides an `undo` command, that allows the user to return to the previous state when a wrong answer has been provided.

As mentioned before, the Maude system includes the predefined `META-LEVEL` module supporting reflection in rewriting logic [9, 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` [9, Chapter 17], which can be used to specify input/output interactions with the user. However, instead of using this module directly, we extend Full Maude [9, Chapter 18], that includes features for parsing, evaluating, and pretty-printing terms, improving the input/output interaction. Moreover, Full Maude allows the specification of concurrent object-oriented systems, that can be also debugged. Thus, our declarative debugger, 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 and several examples 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 both membership equational logic and rewriting logic, and how their specifications are concretized in Maude functional and system modules, respectively. Section 3 describes the theoretical foundations of the debugging proof trees for inferences is both logics, namely, reductions, memberships, and rewrites. 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.

2 Rewriting logic and Maude

As mentioned in the introduction, Maude modules are executable rewriting logic especifications. Rewriting logic [14] is a logic of change very suitable for the specification of concurrent systems that is parameterized by an underlying equational logic, for which Maude uses membership equational logic [2, 15], which, in addition to equations, allows the statement of membership axioms characterizing the elements of a sort. In the following sections we present both logics and how their specifications are represented as Maude 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 axioms* 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 axiom, 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 Σ-*algebras* $\mathcal{A}$ consisting of a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1 \ldots k_n, k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. The meaning $[\![t]\!]_{\mathcal{A}}$ of a term t in an algebra $\mathcal{A}$ is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies an equation $t = t'$ (or the equation holds in the algebra), denoted $\mathcal{A} \models t = t'$, when both terms have the same meaning: $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. In the same way, satisfaction of a membership is defined as: $\mathcal{A} \models t : s$ when $[\![t]\!]_{\mathcal{A}} \in A_s$.

A membership equational logic specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of terms $[t]$. We refer to [2, 15] for a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules (that we adapt to our purposes in Figure 1 in Section 3.1), as well as the construction of initial and free algebras.

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, $t \to t'$. Such a statement holds in an algebra, denoted $\mathcal{A} \models t \to t'$, exactly when $\mathcal{A} \models t = t'$, i.e., when $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. Moreover, under those assumptions an equational condition $u = v$ in a conditional equation can be checked by finding a common term t such that $u \to t$ and $v \to t$. The notation we will use in the inference rules and debugging trees studied in Section 3 for this situation is $u \downarrow v$.

2.2 Maude functional modules

Maude functional modules [9, Chapter 4], 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 .
```

2.3 An example of functional module: sorted lists

Let us see a simple example of a functional module showing how to specify in Maude sorted lists. We use a module parameterized by the theory `TOSET` [9, 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 axiom. 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.

2.4 Rewriting logic

Rewriting logic extends equational logic by introducing the notion of *rewrites* corresponding to transitions between states; that is, while equations are interpreted as equalities and therefore they are symmetric, rewrites denote changes which can be irreversible.

A rewriting logic specification, or *rewrite theory*, has the form $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational specification and R is a set of *rules* as described below. From this definition, one can see that rewriting logic is built on top of equational logic, so that rewriting logic is parameterized with respect to the version of the underlying equational logic; in our case, Maude uses membership equational logic, as described in the previous sections. A rule in R has the general conditional form[2]

$$(\forall X)\ e \Rightarrow e' \Leftarrow \bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w_k'$$

where the head is a rewrite and the conditions can be equations, memberships, and rewrites; both sides of a rewrite must have the same kind. From these rewrite rules, one can deduce rewrites of the form $t \Rightarrow t'$ by means of general deduction rules introduced in [14] (see also [3]), that we adapt to our purposes in Figure 1 in Section 3.1.

[2]There is no need for the condition listing first equations, then memberships, and then rewrites: this is just a notational abbreviation, they can be listed in any order.

Models of rewrite theories are called $\mathcal{R}$-*systems* in [14]. Such systems are defined as categories that possess a (Σ, E)-algebra structure, together with a natural transformation for each rule in the set R. More intuitively, the idea is that we have a (Σ, E)-algebra, as described in Section 2.1, with transitions between the elements in each set A_k; moreover, these transitions must satisfy several additional requirements, including that there are identity transitions for each element, that transitions can be sequentially composed, that the operations in the signature Σ are also appropriately defined for the transitions, and that we have enough transitions corresponding to the rules in R. Then, if we keep in this context the notation $\mathcal{A}$ to denote an $\mathcal{R}$-system, a rewrite $t \Rightarrow t'$ is satisfied by $\mathcal{A}$, denoted $\mathcal{A} \models t \Rightarrow t'$, when there is a transition $[\![t]\!]_{\mathcal{A}} \rightarrow_{\mathcal{A}} [\![t']\!]_{\mathcal{A}}$ in the system between the corresponding meanings of both sides of the rewrite, where $\rightarrow_{\mathcal{A}}$ will be our notation for such transitions.

The rewriting logic deduction rules introduced in [14] are sound and complete with respect to this notion of model. Moreover, they can be used to build initial and free models; see [14] for details.

2.5 Maude system modules

Maude system modules [9, Chapter 6], introduced with syntax `mod ... endm`, are executable rewrite theories and their semantics is given by the initial system in the class of systems corresponding to the rewrite theory. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

The executability requirements for equations and memberships in a system module are the same as those of functional modules, namely, confluence, termination, and sort-decreasingness. With respect to rules, the satisfaction of all the conditions in a conditional rewrite rule is attempted sequentially from left to right, solving rewrite conditions by means of search; for this reason, we can have new variables in such conditions but they must become instantiated along this process of solving from left to right (see [9] for details). Furthermore, the strategy followed by Maude in rewriting with rules is to compute the normal form of a term with respect to the equations before applying a rule. This strategy is guaranteed not to miss any rewrites when the rules are *coherent* with respect to the equations [25, 9]. In a way quite analogous to confluence, this coherence requirement means that, given a term t, for each rewrite of it using a rule in R to some term t', if u is the normal form of t with respect to the equations and memberships in E, then there is a rewrite of u with some rule in R to a term u' such that $u' =_E t'$ (that is, the equation $t' = u'$ can be deduced from E).

The following section describes an example of a Maude system module with both equations and rules.

2.6 An example of system module: knight's tour problem

A knight's tour is a journey around the chessboard in such a way that the knight lands on each square exactly once. The legal move for a knight is two spaces in one direction, then one in a perpendicular direction. We want to solve the problem for a 3×4 chessboard with the knight initially located in one corner.

We represent positions in the chessboard as pairs of integers and journeys as lists of positions.

```
(mod KNIGHT is
  protecting INT .
  sorts Position Movement Journey Problem .
  subsort Position < Movement .
  subsorts Position < Journey < Problem .
  op [_,_] : Int Int -> Position .
  op nil : -> Journey .
  op __ : Journey Journey -> Journey [assoc id: nil] .

  vars N X Y : Int .
  vars P Q : Position .
  var  J : Journey .
```

The term `move P` represents a position reachable by the knight from position `P`. Since the reachable positions are not unique, this operation is defined by means of rewrite rules, instead of equations. The reachable positions can be outside the chessboard.

```
  op move_ : Position -> Movement .
  rl [mv1] : move [X, Y] => [X + 2, Y + 1] .
  rl [mv2] : move [X, Y] => [X + 2, Y - 1] .
```

```
  rl [mv3] : move [X, Y] => [X - 2, Y + 1] .
  rl [mv4] : move [X, Y] => [X - 2, Y - 1] .
  rl [mv5] : move [X, Y] => [X + 1, Y + 2] .
  rl [mv6] : move [X, Y] => [X + 1, Y - 2] .
  rl [mv7] : move [X, Y] => [X - 1, Y + 2] .
  rl [mv8] : move [X, Y] => [X - 1, Y - 2] .
```

The operation `legal` checks if a position is inside the 3×4 chessboard.

```
  op legal : Position -> Bool .
  eq [leg] : legal([X, Y]) = X >= 1 and Y >= 1 and X <= 3 and Y <= 4 .
```

The operation `contains(J, P)` checks if position `P` already occurs in the journey `J`.

```
  op contains : Journey Position -> Bool .
  eq [con1] : contains(P J, P) = true .
  eq [con2] : contains(J, P) = false [owise] .
```

`knight(N)` represents a journey where the knight has performed `N` hops. When no hops are taken, the knight remains at the first position `[1, 1]`. When `N > 0` the problem is recursively solved (using backtracking in an implicit way) as follows: first a legal journey of `N - 1` steps is found, then a new hop from the last position of that journey is performed, and finally it is checked that this last hop is legal and compatible with the other ones.

```
  op knight : Nat -> Problem .
  rl [k1] : knight(0) => [1, 1] .
  crl [k2] : knight(N) => J P Q
   if N > 0
   /\ knight(N - 1) => J P
   /\ move P => Q
   /\ legal(Q)
   /\ not(contains(J P, Q)) .
endm)
```

The solution to the 3×4 chessboard can be found by looking for a journey with 11 hops:

```
Maude> (rew knight(11) .)
```

But we obtain the following unexpected, wrong result, where the journey contains repeated positions.

```
result Journey : [1,1][2,3][3,1][2,3][3,1][2,3][3,1][2,3][3,1][2,3][3,1][2,3]
```

We will show how to debug this specification in Section 4.3.3.

3 Debugging trees for Maude specifications

Now we will describe debugging trees for both membership equational logic specifications and rewriting logic specifications. Since a membership equational logic specification coincides with a rewrite theory with an empty set of rules, our treatment will simply be at the level of rewrite theories. Our proof and debugging trees will include statements for reductions $e \rightarrow e'$, memberships $e : s$, and rewrites $e \Rightarrow e'$, and in the following sections we will describe how to build the debugging trees from the proof trees taking into account each kind of statement.

3.1 Proof trees

Before defining the debugging trees employed in our declarative debugging framework we introduce the semantic rules defining the semantics of a rewrite theory $\mathcal{R}$. The inference rules of the calculus can be found in Figure 1. The rules allow to deduce statements of the three kinds and are an adaptation of the rules presented in [2, 15] for membership equational logic and in [14, 3] for rewriting logic. Remember that the notation $\theta(u_i) \downarrow \theta(u_i')$ is an abbreviation of $\exists t_i . \theta(u_i) \rightarrow t_i \wedge \theta(u_i') \rightarrow t_i$. As usual, we represent deductions in the calculus as *proof trees*, where the premises are the child nodes of the conclusion at each inference step. We assume that the inference labels $(Rep_{\Rightarrow})$, $(Rep_{\rightarrow})$, and (Mb) decorating the

(**Reflexivity**)

$$\frac{}{e \Rightarrow e}\ (Rf_{\Rightarrow}) \qquad \frac{}{e \rightarrow e}\ (Rf_{\rightarrow})$$

(**Transitivity**)

$$\frac{e_1 \Rightarrow e' \qquad e' \Rightarrow e_2}{e_1 \Rightarrow e_2}\ (Tr_{\Rightarrow}) \qquad \frac{e_1 \rightarrow e' \qquad e' \rightarrow e_2}{e_1 \rightarrow e_2}\ (Tr_{\rightarrow})$$

(**Congruence**)

$$\frac{e_1 \Rightarrow e_1' \quad \ldots \quad e_n \Rightarrow e_n'}{f(e_1, \ldots, e_n) \Rightarrow f(e_1', \ldots, e_n')}\ (Cong_{\Rightarrow})$$

$$\frac{e_1 \rightarrow e_1' \quad \ldots \quad e_n \rightarrow e_n'}{f(e_1, \ldots, e_n) \rightarrow f(e_1', \ldots, e_n')}\ (Cong_{\rightarrow})$$

(**Replacement**)

$$\frac{\{\theta(u_i) \downarrow \theta(u_i')\}_{i=1}^{n}\ \{\theta(v_j) : s_j\}_{j=1}^{m}\ \{\theta(w_k) \Rightarrow \theta(w_k')\}_{k=1}^{l}}{\theta(e) \Rightarrow \theta(e')}\ (Rep_{\Rightarrow})$$

if $e \Rightarrow e' \Leftarrow \bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w_k'$

$$\frac{\{\theta(u_i) \downarrow \theta(u_i')\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(e) \rightarrow \theta(e')}\ (Rep_{\rightarrow})$$

if $e \rightarrow e' \Leftarrow \bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j$

(**Equivalence Class**)

$$\frac{e \rightarrow e' \quad e' \Rightarrow e'' \quad e'' \rightarrow e'''}{e \Rightarrow e'''}(EC)$$

(**Subject Reduction**)

$$\frac{e \rightarrow e' \quad e' : s}{e : s}(SRed)$$

(**Membership**)

$$\frac{\{\theta(u_i) \downarrow \theta(u_i')\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(e) : s}\ (Mb)$$

if $e : s \Leftarrow \bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j$

Figure 1: Semantic calculus for Maude modules

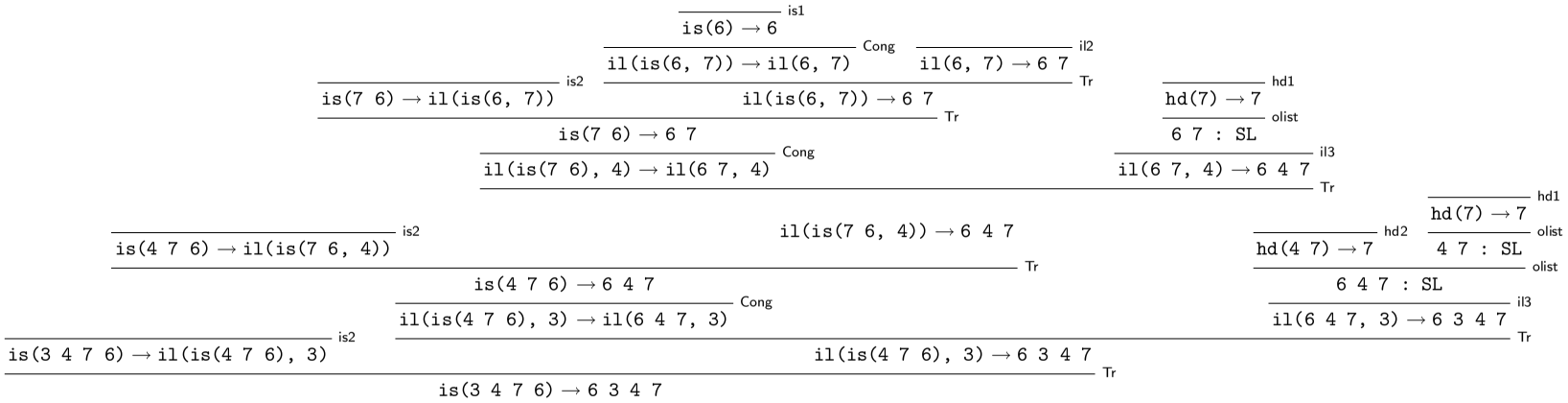

Figure 2: Proof tree for the sorted lists example

inference steps contain information about the particular rewrite rule, equation, and membership axiom, respectively, applied during the inference. This information will be used by the debugger in order to present to the user the incorrect fragment of code causing the error.

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 2.3. For obvious reasons, the operation names have been abbreviated in a self-explanatory way; furthermore, each node corresponding to an instance of the *replacement* or *membership* inference rules has been labeled with the label of the equation or membership statement which is being applied.

In our debugging framework we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. The intended interpretation must be an $\mathcal{R}$-system corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. Therefore the user expects that $\mathcal{I} \models e \Rightarrow e'$, $\mathcal{I} \models e \rightarrow e'$, and $\mathcal{I} \models e : s$ for each rewrite $e \Rightarrow e'$, reduction $e \rightarrow e'$, and membership $e : s$ computed w.r.t. the specification $\mathcal{R}$. As an $\mathcal{R}$-system, $\mathcal{I}$ must satisfy the following proposition:

Proposition 1 *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and let $\mathcal{A}$ be any $\mathcal{R}$-system. If a statement $e \Rightarrow e'$ (respectively $e \rightarrow e'$, $e : s$) can be deduced using the semantic calculus rules* reflexivity, transitivity, congruence, equivalence class *or* subject reduction *using premises that hold in $\mathcal{A}$, then $\mathcal{A} \models e \Rightarrow e'$ (respectively $\mathcal{A} \models e \rightarrow e'$, $\mathcal{A} \models e : s$).*

Proof. The result is a direct consequence of the definition of satisfaction of rewrite theories. For instance we check the result for the *transitivity* rules ($Tr_{\Rightarrow}$) and ($Tr_{\rightarrow}$), and for the *subject reduction* rule ($SRed$):

- ($Tr_{\Rightarrow}$). Suppose that $\mathcal{A} \models e_1 \Rightarrow e'$ and $\mathcal{A} \models e' \Rightarrow e_2$. Then $[\![e_1]\!]_{\mathcal{A}} \rightarrow_{\mathcal{A}} [\![e']\!]_{\mathcal{A}}$, $[\![e']\!]_{\mathcal{A}} \rightarrow_{\mathcal{A}} [\![e_2]\!]_{\mathcal{A}}$. Since $\rightarrow_{\mathcal{A}}$ is compositional, $[\![e_1]\!]_{\mathcal{A}} \rightarrow_{\mathcal{A}} [\![e_2]\!]_{\mathcal{A}}$, i.e., $\mathcal{A} \models e_1 \Rightarrow e_2$.
- ($Tr_{\rightarrow}$). if $\mathcal{A} \models e_1 \rightarrow e'$ and $\mathcal{A} \models e' \rightarrow e_2$ 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 $\mathcal{A} \models e_1 \rightarrow e_2$.
- ($SRed$). if $\mathcal{A} \models e \rightarrow e'$ 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*, *congruence* and *equivalence class* 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 a statement $e \Rightarrow e'$ (respectively $e \rightarrow e'$, $e : s$) 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 rules*, *wrong equations*, and *wrong membership axioms*, which are those specification pieces that can deduce something invalid from valid information.

Definition 1 *Let $r \equiv (af \Leftarrow \bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w_k')$ where af denotes an atomic formula, that is, r is either a rewrite rule, an oriented equation, or a membership axiom (in the last two cases $l = 0$) in some rewrite theory $\mathcal{R}$. Then:*

- *$\theta(r)$ is a* wrong rewrite rule instance *(respectively* wrong equation instance *and* wrong membership axiom instance*) w.r.t. an intended interpretation $\mathcal{I}$ when*
  1. *There exist terms $t_1, \ldots, t_n$ such that $\mathcal{I} \models \theta(u_i) \rightarrow t_i$, $\mathcal{I} \models \theta(u_i') \rightarrow t_i$ for $i = 1 \ldots n$.*
  2. *$\mathcal{I} \models \theta(v_j) : s_j$ for $j = 1 \ldots m$.*
  3. *$\mathcal{I} \models \theta(w_j) \Rightarrow \theta(w_j')$ for $k = 1 \ldots l$.*
  4. *$\theta(af)$ does not hold in $\mathcal{I}$.*
- *r is a* wrong rewrite rule *(respectively,* wrong equation *and* wrong membership axiom*) if it admits some wrong instance.*

The general schema of [17] presents declarative debugging as the search of *buggy nodes* (invalid nodes with all its children valid) in a debugging tree representing an erroneous computation. In our scheme instance, the proof trees constructed by the inferences of Figure 1 seem natural candidates for debugging trees. Although this is a possible option, we will use instead a suitable abbreviation of these trees. This is motivated by the following result:

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 statement associated to N is either a wrong rewrite rule, a wrong equation, or a wrong membership axiom.*

Proof. The first item is a straightforward consequence of Proposition 1: N buggy means N invalid with all its children valid, and the only possible inference rules at N are then *membership* and *replacement*.

For the second result we first observe that as a consequence of the previous part the label at the inference step with N as conclusion must be either $(Rep_{\Rightarrow})$, $(Rep_{\rightarrow})$, or (Mb). In the case of $(Rep_{\Rightarrow})$ the associated rewrite rule is wrong as a direct consequence of Definition 1: the condition 4 of the definition holds because N is invalid in $\mathcal{I}$, while the previous conditions, which state the validity of the statements in the rewrite rule condition instance, correspond to the premises of the $(Rep_{\Rightarrow})$ inference rule (see Figure 1), which are valid in $\mathcal{I}$ because N is buggy. Analogously it can be checked from inference rule (Mb) that if the conclusion is buggy the associated membership axiom is wrong, and the same in the case of $(Rep_{\rightarrow})$ but with an associated wrong equation. □

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:

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.

Thus, in order to minimize the number of questions asked to the user the debugger should consider the validity of $(Rep_{\Rightarrow})$, $(Rep_{\rightarrow})$, or (Mb). The rules for deriving an APT can be seen in Figure 3. The T_i represent proof trees corresponding to the premises in some inferences.

The abbreviation always starts by applying (APT_1). This rule simply duplicates the root of the tree and applies APT', which receives a proof tree and returns a forest. Hence without this duplication the result of the abbreviation could be a forest instead of a single tree. Moreover, it is easy to check that this duplication is safe in the sense that it can neither introduce nor remove buggy nodes (this will be a consequence of Theorem 1).

The rest of the APT rules correspond to function APT' and are assumed to be applied top-down: if several APT rules can be applied at the root of a proof tree, we must choose the first one, that is, the rule of least number. As a matter of fact, the figure includes rules for two different possible $APTs$, which we call *one-step* abbreviated proof tree (in short $APT^o(T)$), defined by all the rules in the figure excluding (APT_4^m), and *many-steps* abbreviated proof tree (in short $APT^m(T)$), defined by all the rules in the figure excluding (APT_4^o). Analogously, we will use the notation $APT'^o(T)$ (resp. $APT'^m(T)$) for the subset of rules of APT' excluding (APT_4^m) (resp. (APT_4^o)).

$(\mathbf{APT}_1)$	$APT\left(\dfrac{T_1 \dots T_n}{af}(R)\right)$	$= \dfrac{APT'\left(\dfrac{T_1 \dots T_n}{af}(R)\right)}{af}$
$(\mathbf{APT}_2)$	$APT'\left(\dfrac{}{e \Rrightarrow e}(Rf_{\Rrightarrow})\right)$	$= \emptyset$
$(\mathbf{APT}_3)$	$APT'\left(\dfrac{\dfrac{T_1 \dots T_n}{e_1 \to e'}(Rep_{\to}) \quad T'}{e_1 \to e_2}(Tr_{\to})\right)$	$= \left\{\dfrac{APT'(T_1) \dots APT'(T_n)\ APT'(T')}{e_1 \to e_2}(Rep_{\to})\right\}$
$(\mathbf{APT}_4^o)$	$APT'\left(\dfrac{T_1 \quad T_2}{e_1 \Rightarrow e_2}(Tr_{\Rightarrow})\right)$	$= APT'(T_1) \bigcup APT'(T_2)$
$(\mathbf{APT}_4^m)$	$APT'\left(\dfrac{T_1 \quad T_2}{e_1 \Rightarrow e_2}(Tr_{\Rightarrow})\right)$	$= \left\{\dfrac{APT'(T_1)\ APT'(T_2)}{e_1 \Rightarrow e_2}(Tr_{\Rightarrow})\right\}$
$(\mathbf{APT}_5)$	$APT'\left(\dfrac{T_1 \dots T_n}{e_1 \Rrightarrow e_2}(Cong_{\Rrightarrow})\right)$	$= APT'(T_1) \bigcup \dots \bigcup APT'(T_n)$
$(\mathbf{APT}_6)$	$APT'\left(\dfrac{T_1 \quad T_2}{e : s}(SRed)\right)$	$= APT'(T_1) \bigcup APT'(T_2)$
$(\mathbf{APT}_7)$	$APT'\left(\dfrac{T_1 \dots T_n}{e : s}(Mb)\right)$	$= \left\{\dfrac{APT'(T_1) \dots APT'(T_n)}{e : s}(Mb)\right\}$
$(\mathbf{APT}_8)$	$APT'\left(\dfrac{T_1 \dots T_n}{e_1 \Rrightarrow e_2}(Rep_{\Rrightarrow})\right)$	$= \left\{\dfrac{APT'(T_1) \dots APT'(T_n)}{e_1 \Rrightarrow e_2}(Rep_{\Rrightarrow})\right\}$
$(\mathbf{APT}_9)$	$APT'\left(\dfrac{T' \quad \dfrac{T_1 \dots T_n}{e \Rightarrow e'}(Rep_{\Rightarrow}) \quad T''}{e_1 \Rightarrow e_2}(EC)\right)$	$= \left\{\dfrac{APT'(T')\ APT'(T_1) \dots APT'(T_n)\ APT'(T'')}{e_1 \Rightarrow e_2}(Rep_{\Rightarrow})\right\}$
$(\mathbf{APT}_{10})$	$APT'\left(\dfrac{T_1 \dots T_n}{e_1 \Rightarrow e_2}(EC)\right)$	$= APT'(T_1) \bigcup \dots \bigcup APT'(T_n)$

(R) any inference rule $\quad \Rrightarrow$ either $\to$ or $\Rightarrow$ $\quad af$ either $e_1 \to e_2$, $e : s$ or $e_1 \Rightarrow e_2$

Figure 3: Transforming rules for obtaining abbreviated proof trees

The one-step debugging tree follows strictly the idea of keeping only nodes corresponding to the *replacement* and *membership* inference rules. However, the many-steps debugging tree also keeps nodes corresponding to the *transitivity* inference rule for rewrites. The user will choose which debugging tree (one-step or many-steps) will be used for the declarative debugging session, taking into account that the many-steps debugging tree usually leads to shorter debugging sessions (in terms of the number of questions) but with likely more complicated questions. The number of questions is usually reduced because keeping the transitivity nodes for rewrites gives to some parts of the debugging tree the shape of a balanced binary tree (each transitivity inference has two premises, i.e., two child subtrees), and this allows the debugger to use a very efficient divide and query navigation strategy similar to that presented in [21]. On the contrary, removing the transitivity inferences for rewrites (as rule (APT_4^o) does) produces flattened trees where this strategy is no longer efficient. On the other hand, in rewrites $e \Rightarrow e'$ appearing as conclusion of the transitivity inference rule, the term e' can contain the result of rewriting several subterms of e, and determining the validity of such nodes can be complicated, while in the one-step debugging tree each rewrite node $e \Rightarrow e'$ corresponds to a single rewrite applied at e and checking its validity is usually easier. The user must balance the pros and cons of each option, and choose the best one for each debugging session.

The rules (APT_3) and (APT_9) deserve a more detailed explanation. They keep the corresponding label ($Rep_{\Rightarrow}$) but changing the conclusion of the replacement inference in the lefthand side. For instance, (APT_3) replaces $e_1 \to e'$ by the conclusion of the next transitivity inference $e_1 \to e_2$. We do this as a pragmatic way of simplifying the structure of the APTs, since e_2 is obtained from e' and hence likely simpler (the root of the tree T' in (APT_3) must be necessarily of the form $e' \to e_2$ by the structure of the inference rule for transitivity in Figure 1). A similar reasoning explains the form of (APT_9). We will formally state below (Theorem 1) that these changes are safe from the point of view of the debugger.

Although $APT(T)$ is no longer a proof tree we keep the inference labels ($Rep_{\Rightarrow}$) and (Mb), assuming implicitly that they contain a reference to the rewrite rule, 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.

The following property of the abbreviated proof trees, in particular of the forests $APT'^o(T)$ and $APT'^m(T)$, will be useful when proving the correctness of the technique.

Lemma 1 *Let T be a finite proof tree representing an inference in the calculus of Figure 1 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root N of T is invalid in $\mathcal{I}$. Then:*

(a) If T contains only one node, then

$$APT'^o(T) = APT'^m(T) = \{T\}$$

(b) Both sets $APT'^o(T)$ and $APT'^m(T)$ contain a tree with an invalid root.

Proof. If T contains only one node N then N is an invalid node without children and therefore buggy. By Proposition 2 the inference step proving this node must be either a replacement or a membership. If it is a replacement the rule (APT_8) of Figure 3 must be applied and the result holds, since it returns a unary set with the same root. Analogously with a membership inference and rule (APT_7).

The second item can be proved by induction on the number of nodes of T, which we denote as $n(T)$. If $n(T) = 1$ the property is straightforward from the part (a) above because $T \in APT'^o(T), T \in APT'^m(T)$. If $n(T) > 1$ we distinguish cases depending on the rule for APT' that can be applied at the root of T:

- It cannot be (APT_2) by Proposition 2.
- If it is either (APT_3), (APT_4^m), (APT_7), (APT_8), or (APT_9), the result holds directly because the result is a unary set with the same (invalid) root.
- If it is either (APT_4^o), (APT_5), (APT_6), or (APT_{10}) then by Proposition 2 N has some invalid child, which corresponds to the root of some premise T_i. By the induction hypothesis, there is some $T' \in APT'(T_i)$ with invalid root. And by observing the rules of Figure 3 it can be checked that every subtree T_i of the root of T verifies $APT'(T_i) \subseteq APT'(T)$. Then $T' \in APT'(T)$.

□

Now we are ready to prove the correctness and completeness of the debugging technique based on APTs:

Theorem 1 *Let T be a finite proof tree representing an inference in the calculus of Figure 1 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root of T is invalid in $\mathcal{I}$. Then:*

- *Both $APT^o(T)$ and $APT^m(T)$ contain at least one buggy node (completeness).*
- *Any buggy node in $APT^o(T)$, $APT^m(T)$ has an associated wrong statement in $\mathcal{R}$ (correctness).*

Proof. For simplicity we will denote by $APT(T)$ both $APT^o(T)$ and $APT^m(T)$, and by $APT'(T)$ both $APT'^o(T)$ and $APT'^m(T)$, distinguishing the particular type of abbreviated proof tree when necessary.

- $APT(T)$ contains at least one invalid node, since its root is the root of T, and any debugging tree containing an invalid node contains a buggy node by Proposition 3.
- First we observe that the root of $APT(T)$ cannot be buggy, because if it is invalid then it has an invalid child (Lemma 1(b)) . Therefore any buggy node must be part of $APT'(T)$ (the premise in (APT_1)).

  Let N be a buggy node in $APT'(T)$. Then N is the root of some tree T_N, subtree of $APT'(T)$. By the structure of the APT' rules this means that there is a subtree T' of T s.t. $T_N \in APT'(T')$. We prove that N has an associated wrong statement in S by induction on the number of nodes of T', $n(T')$.

  If $n(T') = 1$ then T' contains only one node and $APT'(T') = \{T'\}$ by Lemma 1(a). Then the only possible buggy node is N, which means that N is also buggy in T and that the associated fragment of code is wrong by Proposition 2.

  If $n(T') > 1$ we examine the APT rule applied at the root of T':

  - (APT_2). This is not possible because then $APT'(T') = \emptyset$.
  - (APT_4^o), (APT_5), (APT_6) and (APT_{10}). Then $T_N \in APT'(T_i)$ for some child subtree T_i of the root of T' and the result holds by the induction hypothesis.
  - (APT_3). Then T' is of the form

$$\dfrac{\dfrac{T_1 \ldots T_n}{e_1 \to e'}(Rep_\to) \quad T''}{e_1 \to e_2}(Tr_\to)$$

    Hence $N \equiv (e_1 \to e_2)$ and T_N is

$$\dfrac{APT'(T_1) \ldots APT'(T_n) \;\; APT'(T'')}{e_1 \to e_2}(Rep_\to)$$

    Since N is buggy in T_N it is invalid w.r.t. $\mathcal{I}$. By Proposition 2, $e_1 \to e_2$ cannot be buggy in T', i.e. either T'' has an invalid root or $e_1 \to e'$ is invalid. But T'' cannot be invalid because $APT'(T'')$ is a child subtree of N and by Lemma 1(b) it would have an invalid root. Therefore $e_1 \to e'$ is invalid. Moreover, the roots of $T_1, \ldots, T_n$ are also valid by the same reason: $APT'(T_1), \ldots, APT'(T_n)$ are child subtrees of N in T_N and cannot have an invalid root. Therefore $e_1 \to e'$ is buggy in T', i.e., is buggy in T and by Proposition 2 the equation associated to label $(Rep_\to)$ is wrong. And this label is the same that can be found associated to N in the APT' T_N. Therefore the buggy node N of the APT' has an associated wrong equation.
  - (APT_4^m). We check that actually this rule cannot be applied to produce a buggy node and therefore must not be considered here. If (APT_4^m) is applied then T' must be of the form

$$\dfrac{T_1 \quad T_2}{e_1 \Rightarrow e_2}(Tr_\Rightarrow)$$

    N is $e_1 \Rightarrow e_2$ and T_N is

$$\dfrac{APT'(T_1) \;\; APT'(T_2)}{e_1 \Rightarrow e_2}(Tr_\Rightarrow)$$

    And N can be invalid but not buggy in T' (and hence in T) by Proposition 2, because it is the conclusion of a transitivity inference, and thus either T_1 or T_2 has an invalid root. Then by Lemma 1(b), either $APT'(T_1)$ or $APT'(T_2)$ have an invalid root and N is not buggy in T_N.

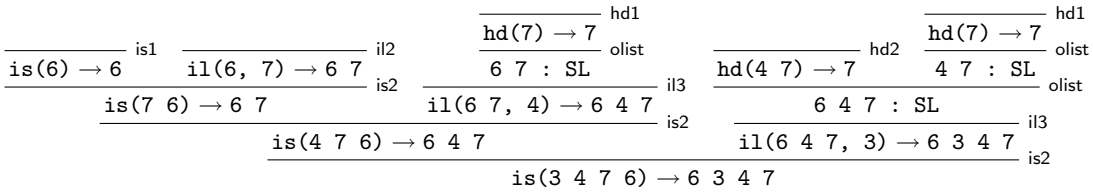

Figure 4: Abbreviated proof tree for the sorted lists example

– (APT_7). Then N is of the form $e : s$, T' is

$$\frac{T_1 \dots T_n}{e : s}(Mb)$$

and T_N

$$\frac{APT'(T_1) \dots APT'(T_n)}{e : s}(Mb)$$

Since N is buggy the roots of $APT'(T_1), \dots, APT'(T_n)$ are valid, which by Lemma 1 means that the roots of $T_1, \dots, T_n$ are also valid. Then N is buggy in T', and hence in T, and the membership axiom associated to the label (Mb) is wrong by Proposition 2.

– (APT_8). Analogous to (APT_7).

– (APT_9). Analogous to (APT_3).

□

The theorem states that we can safely employ the abbreviated proof tree as a basis for the declarative debugging of Maude system and 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

A rewrite theory has an underlying equational theory, containing equations and memberships, which is expected to satisfy the appropriate executability requirements, namely, it has to be terminating, confluent, and sort decreasing. Rules are assumed to be coherent with respect to the equations; for details, see [9].

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, and only their conditions will generate questions. 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 wrong statement must be labeled in order to be found.

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

As mentioned in the introduction, navigation of the debugging tree takes place by asking questions to an external oracle, which in our case is either the user or another module introduced by the user. In both cases the answers are assumed to be correct. If either the module is not really correct or the user provides an incorrect answer, the result is unpredictable. Notice that the information provided by the correct module need not be complete, in the sense that some functions can be only partially defined. In the same way, the signature of the correct module need not coincide with the signature of the module being debugged. If the correct module cannot help in answering a question, the user may have to answer it.

4.2 Commands

The debugger is initiated in Maude by loading the file `dd.maude` (available from `http://maude.sip.ucm.es/debugging`), which starts an input/output loop that allows the user to interact with the tool. Then, the user can enter Full Maude modules and commands, as well as commands for the debugger.

The user can choose between using all the labeled statements in the debugging process (by default) or selecting some of them by means of the command

```
(set debug select on .)
```

Once this mode is activated, the user can select and deselect statements by using

```
(debug select LABELS .)
(debug deselect LABELS .)
```

where `LABELS` is a list of labels separated by spaces.

Moreover, all the labels in statements of a flattened module can be selected or deselected with the following commands:

```
(debug include MODULES .)
(debug exclude MODULES .)
```

where `MODULES` is a list of module names separated by spaces.

The selection mode can be switched off by using the command

```
(set debug select off .)
```

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 .)
```

and can be deselected with the command

```
(delete correct module .)
```

Since rewriting is not assumed to terminate, a bound, which is 42 by default although iy can be `unbounded`, is used when searching in the correct module and can be set with the command

```
(set bound BOUND .)
```

When debugging rewrites, two different trees can be built: one whose questions are related to one-step rewrites and another whose questions are related to several steps. The user can switch between these trees with the commands

```
(one-step tree .)
(many-steps tree .)
```

being the first the default one.

The generated debugging 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 .)
```

The debugging process is started with the commands

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

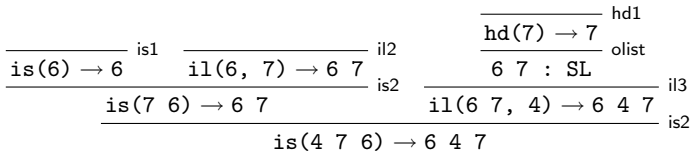

Figure 5: Abbreviated proof tree after the first answer

for wrong reductions, memberships, and rewrites. `MODULE-NAME` is the module where the inference took; if no module name is given, the current module is used by default.

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, the user 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, he 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 .)
```

Finally, we can return to the previous state in both strategies by using the command

```
(undo .)
```

4.3 Examples

We show how the debugger works by means of several examples: the sorted lists specification already introduced in Section 2.3, binary search trees, the knight's tour problem from Section 2.6, and the evaluation and computation semantics of a simple imperative language.

4.3.1 Sorted lists

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

```
Maude> (red insertion-sort(3 4 7 6) .)
result 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 with the default divide and query strategy by using the command

```
Maude> (debug insertion-sort(3 4 7 6) -> 6 3 4 7 .)
```

With this command the debugger computes the tree shown in Figure 4. The first question asked by the debugger is

```
Is this reduction (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 shown in Figure 5 is selected to continue the debugging. The next question is

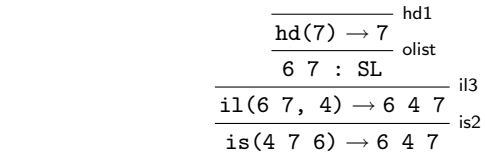

Figure 6: Abbreviated proof tree after the second answer

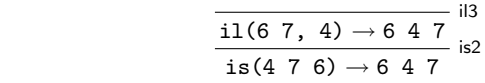

Figure 7: Abbreviated proof tree after the third answer

```
Is this reduction (associated with the equation is2) correct?

insertion-sort(7 6) -> 6 7

Maude> (yes .)
```

Since the list is sorted, we answer `yes`, so the subtree corresponding to this reduction 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 the subtree corresponding to this membership is also deleted (Figure 7) and the next question is asked.

```
Is this reduction (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 wrong 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.

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

We can check the fixed function by sorting again the list `3 4 7 6`.

```
Maude> (red insertion-sort(3 4 7 6) .)
result SortedList{NatAsToset} : 3 4 6 7
```

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 debug this inference by using the command

```
Maude> (debug 6 3 4 7 : SortedList{NatAsToset} .)
```

The first question the debugger asks 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 reduction (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. The right equation is

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

To debug this module we have used the default divide and query strategy. We illustrate now how to do it 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.

```
Maude> (top-down strategy .)

Top-down strategy selected.

Maude> (debug insertion-sort(3 4 7 6) -> 6 3 4 7 .)
```

The debugger asks now about the validity of the children of the root of the tree in Figure 4:

```
Please, choose a wrong node:

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 debugger selects this node as the current one and asks about the correction of its children:

```
Please, choose a wrong node:

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

```
Please, choose a wrong node:

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:

```
Please, choose a wrong node:

Node 0 : 6 4 7 : SortedList{NatAsToset}

Maude> (node 0 .)
```

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

```
Please, choose a wrong node:

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 consider a module defining the sorting function `sort` in a correct, although 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] .

 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 can use this module (instantiated with the view `NatAsToset`) to prune the debugging trees built by the `debug` commands if we previously introduce the command

```
Maude> (correct module CORRECT-SORTING{NatAsToset} .)

CORRECT-SORTING{NatAsToset} selected as correct module.
```

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 in SORTED-LIST-TEST : insertion-sort(3 4 7 6) -> 6 3 4 7 .)

Is this transition (associated with the equation il3) correct?

insert-list(6 4 7,3) -> 6 3 4 7

Maude> (no .)

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

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

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

As another example, we show how to specify binary search trees without repeated elements, whose nodes contain elements that satisfy the theory `STOSET` (defining a strict total order on them) [9, Sect. 8.3].

```
fmod SEARCH-TREE{X :: STOSET} is
 sorts NeSearchTree{X} SearchTree{X} Tree{X} .
 subsorts NeSearchTree{X} < SearchTree{X} < Tree{X} .

 op empty : -> SearchTree{X} [ctor] .
 op ___ : Tree{X} X$Elt Tree{X} -> Tree{X} [ctor] .
```

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

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

```
 vars E E' : X$Elt .
 vars L R : SearchTree{X} .
 vars L' R' : NeSearchTree{X} .

 mb [leaf]     : empty E empty : NeSearchTree{X} .
 cmb [1ch1] : L' E empty    : NeSearchTree{X} if max(L') < E .
 cmb [1ch2] : empty E R'    : NeSearchTree{X} if E < min(R') .
 cmb [2ch]  : L' E R'       : NeSearchTree{X}
  if max(L') < E /\ E < max(R') .

 ops min max : NeSearchTree{X} -> X$Elt .
 ceq [mn1] : min(empty E R) = E if empty E R : NeSearchTree{X} .
 ceq [mn2] : min(L' E R)    = min(L') if L' E R : NeSearchTree{X} .
 ceq [mx1] : max(L E empty) = E if L E empty : NeSearchTree{X} .
 ceq [mx2] : max(L E R')    = max(R') if L E R' : NeSearchTree{X} .
```

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

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

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

Now we can instantiate this module with the predefined module `INT` of integer numbers.

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

(fmod SEARCH-TREE-TEST is
 protecting SEARCH-TREE{IntAsStoset} .
 endfm)
```

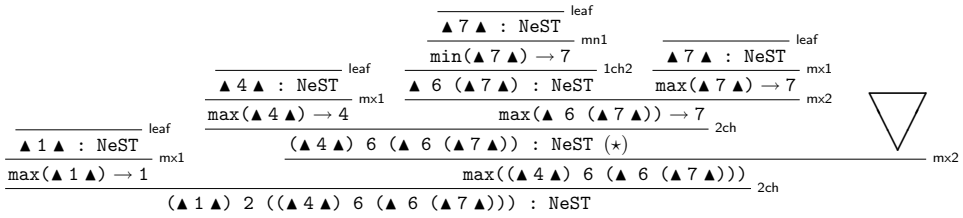

Figure 8: Abbreviated proof tree

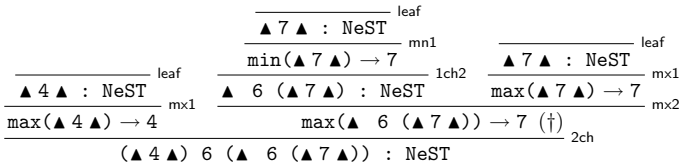

Figure 9: Abbreviated proof tree after the first question

Then, we reduce the following term

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

We obtain a tree with repetitions; moreover, Maude infers that it is a search tree!
We can debug the inference of this membership with the command

```
Maude> (debug
  (empty 1 empty) 2 ((empty 4 empty) 6 (empty 6 (empty 7 empty))) : NeSearchTree{Int} .)
```

that generates, by considering all the labeled statements as suspicious, the debugging tree shown in Figure 8, where ▲ denotes the `empty` search tree and ▽ represents the already depicted proof subtree with root `max(▲ 6 (▲ 7 ▲)) → 7`.

Since the default navigation strategy is divide and query, the debugger selects the node marked with ⋆ in the figure, and then asks the following question:

```
Is this membership (associated with the membership 2ch) correct?

(empty 4 empty) 6 (empty 6 (empty 7 empty)) : NeSearchTree{Int}

Maude> (no .)
```

Since the answer is `no`, the debugger discards the rest of the tree and focuses in the subtree with this node as root (see Figure 9). The next question corresponds to the node marked with † in the figure.

```
Is this transition (associated with the equation mx2) correct?

max(empty 6 (empty 7 empty)) -> 7

Maude> (yes .)
```

When the answer is `yes` the corresponding subtree is deleted, obtaining in this case the tree in Figure 10. The next question is

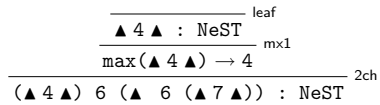

Figure 10: Abbreviated proof tree after the second question

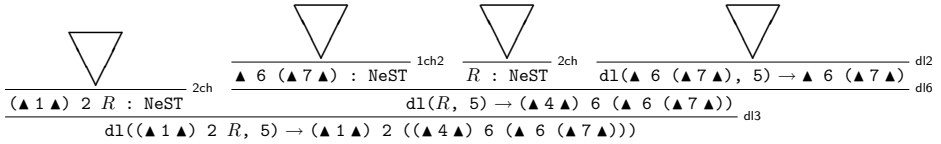

Figure 11: Abbreviated proof tree for the top down strategy

```
Is this transition (associated with the equation mx1) correct?

max(empty 4 empty) -> 4

Maude> (trust .)
```

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

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

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

```
Maude> (top-down strategy .)

Top-down strategy selected.

Maude> (set debug select on .)

Debug select is on.

Maude> (debug select leaf 1ch1 1ch2 2ch dl2 dl3 dl4 dl5 dl6 .)

Labels leaf 1ch1 1ch2 2ch dl2 dl3 dl4 dl5 dl6 are now suspicious.

Maude> (debug delete((empty 1 empty) 2 ((empty 4 empty) 5 (empty 6 (empty 7 empty))), 5)
       -> (empty 1 empty) 2 ((empty 4 empty) 6 (empty 6 (empty 7 empty))) .)
```

where we have decided to mark as suspicious the memberships and the non-trivial equations of `delete`. In this case, the debugger builds the proof tree (partially) shown in Figure 11 (where R denotes the search tree `(▲ 4 ▲) 5 (▲ 6 (▲ 7 ▲))`), so it asks the following questions:

```
Please, choose a wrong node:

Node 0 : (empty 1 empty) 2 ((empty 4 empty) 5 (empty 6 (empty 7 empty))) :
       NeSearchTree{Int}
Node 1 : delete((empty 4 empty) 5 (empty 6 (empty 7 empty)), 5) ->
       (empty 4 empty) 6 (empty 6 (empty 7 empty))

Maude> (node 1 .)

Please, choose a wrong node:

Node 0 : empty 6 (empty 7 empty) : NeSearchTree{Int}
Node 1 :(empty 4 empty) 5 (empty 6 (empty 7 empty)) : NeSearchTree{Int}
Node 2 : delete(empty 6 (empty 7 empty), 5) -> empty 6 (empty 7 empty)
```

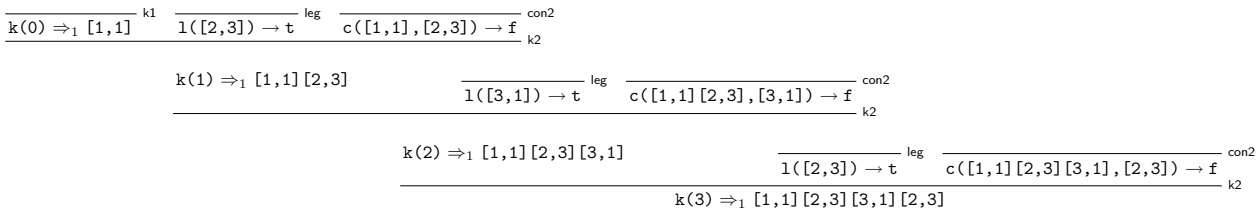

Figure 12: APT for the knight example

```
Maude> (all valid .)

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

The debugger concludes that the problem is within the equation `dl6`; to fix it, we must delete the element `E'` instead of `E` in the subtree.

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

4.3.3 Knight's tour problem

In Section 2.6 we described a system module that simulates a knight's tour around a chessboard. However, this system module contains a bug and the knight repeats some positions in its tour. This error is also obtained when looking for a 3 steps journey:

```
Maude> (rew knight(3) .)
result List : [1,1][2,3][3,1][2,3]
```

Thus, we debug this smaller computation. Moreover, after inspecting the rewrite rules describing the eight possible moves, we are sure that they are not responsible for the error; therefore, we trust them.

```
Maude> (set debug select on .)

Debug select is on.

Maude> (debug select con1 con2 leg k1 k2 .)

Labels con1 con2 k1 k2 leg are now suspicious.

Maude> (debug knight(3) =>* [1,1][2,3][3,1][2,3] .)
```

The default one-step tree construction strategy is used and the tree shown in Figure 12 is built, where every operation has been abbreviated with its first letter.

Since the tree is navigated by using the default divide and query strategy, the first two questions asked by the debugger are

```
Is this rewrite (associated with the rule k2) correct?

knight(1) =>1 [1,1][2,3]

Maude> (yes .)

Is this rewrite (associated with the rule k2) correct?

knight(2) =>1 [1,1][2,3][3,1]

Maude> (yes .)
```

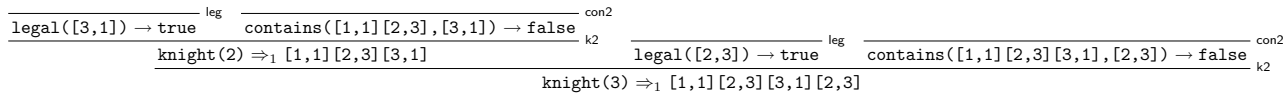

Figure 13: APT for the knight example after the first answer

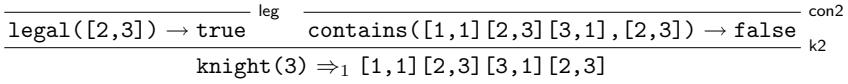

Figure 14: APT for the knight example after the second answer

Notice the form `=>1` of the arrow in the rewrites appearing in the questions, to emphasize that they are one-step rewrites.

In both cases the answer is `yes` because these paths are *possible*, legal behaviors of the knight when it can do one or two hops. The trees obtained after the first and second answers are shown in Figures 13 and 14, respectively. The next question is

```
Is this reduction (associated with the equation con2) correct?

contains([1,1][2,3][3,1],[2,3]) -> false

Maude> (no .)
```

Clearly, this is not a correct reduction, since position `[2,3]` is already in the path `[1,1][2,3][3,1]`. With this answer the debugger finds the error

```
The buggy node is:
contains([1,1][2,3][3,1],[2,3]) -> false
with the associated equation: con2
```

Looking at the definition of the `contains` operation, we realize that it defines the membership operation for *sets*, not for lists. Actually, it is checking if the position given as second argument is the head of the list. A correct definition of the `contains` operation is as follows:

```
 eq contains(nil, P) = false .
 eq contains(Q J, P) = P == Q or contains(J, P) .
```

With this function fixed, the problem can be solved:

```
Maude> (rew knight(11) .)
result Journey : [1,1][2,3][3,1][1,2][2,4][3,2][1,3][2,1][3,3][1,4][2,2][3,4]
```

4.3.4 WhileL evaluation semantics

We show in this section how to describe the evaluation semantics of a very simple programming language with arithmetic and Boolean expressions, assignments, sequential composition, conditionals, and loops [11, 24].

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 as quoted identifiers, although we also define ad hoc variables to facilitate its use. Variables and numbers are particular cases of expressions.

```
  subsorts Qid < Var < Exp .
  subsort Nat < Num < Exp .
```

```
  ops w x y z : -> Var .

  ops +. -. *. : -> Op .

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

In the same way, Boolean variables are defined:

```
  subsorts Qid < 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, conditional, and while loop. The fact that a program is just a command is reflected in our signature by a subsort declaration.

```
  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 by using the Maude predefined functions.

```
(fmod AP is
  pr WHILEL-SYNTAX .

  op Ap : Op Num Num -> Num .

  vars n n' : Num .

  eq Ap(+., n, n') = n + n' .
  eq Ap(*., n, n') = n * n' .
  eq Ap(-., n, n') = if n < n' then 0 else sd(n, n') fi .

  op Ap : BOp Boolean Boolean -> Boolean .

  var bv bv' : 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 been assigned more than one value, the leftmost assignment 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 .

  rl [CR] : < n, st > => n .

  rl [VarR] : < X, st > =>  st(X) .

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

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

```
  rl [BVarR] : < bx, st > =>  st(bx) .

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

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

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

We use this module to describe the semantics of statements.

```
(mod EVALUATION-WHILE is
  protecting EVALUATION-EXP-EVAL .

  subsort ENV < Statement .

  op <_,_> : Com ENV -> Statement .

  var X : Var .
  vars st st' st'' : ENV .
  var e : Exp .
  var v : Num .
  var be : BExp .
  vars C C' C'' : Com .
```

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

```
  crl [AsR] : < X := e, st > => < skip, st[v / X] >
           if < e, st > => v .
```

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

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

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

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

The while statement also distinguishes whether 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 then tries to evaluate the whole statement again.

```
  crl [WhileR1] : < While be Do C, st > => < skip, st >
               if < be, st > => F .
  crl [WhileR2] : < While be Do C, st > => < skip, st' >
               if < be, st > => T /\
                  < C, st > => < skip, st' > .
 endm)
```

If we execute now the program below to multiply x and y and keep the result in z

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

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

```
Maude> (top-down strategy .)

Top-down strategy selected.

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

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

```
Please, choose a wrong node:

Node 0 : < z := 0, x = 2 y = 3 z = 1 > =>1 < skip, x = 2 y = 3 z = 0 >
Node 1 : < While Not Equal(x,0) Do z := z +. y ; x := x -. 1, x = 2 y = 3 z = 0 >
         =>1 < skip, y = 3 z = 3 x = 1 >

Maude> (node 1 .)
```

The second node is erroneous, because x has not reached 0, so we select this node, and the following question is related to its children:

```
Please, choose a wrong node:

Node 0 : < Not Equal(x,0), x = 2 y = 3 z = 0 > =>1 T
Node 1 : < z := z +. y ; x := x -. 1, x = 2 y = 3 z = 0 >
         =>1 < skip, y = 3 z = 3 x = 1 >

Maude> (all valid .)
```

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

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

If we examine now the WhileR2 rule we realize that the body of the while loop is evaluated only once. We fix this rule as follows:

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

If we execute now the program in the fixed module, we obtain the right result.

```
Maude> (rew < z := 0 ; (While Not Equal(x, 0) Do
                        z := z +. y ; x := x -. 1), x = 2 y = 3 z = 1 > .)
result Statement : < skip, y = 3 z = 6 x = 0 >
```

4.3.5 WhileL computation semantics

In contrast to the evaluation semantics, the computation semantics describes the behavior of programs in terms of small steps [11, 24]. We define this behavior in the following module:

```
(mod COMPUTATION-WHILE is
  protecting EVALUATION-EXP .

  op <_,_> : Com ENV -> Statement .

  sort Statement2 .
  op (_,_) : Com ENV -> Statement2 .
  op Tick : -> Statement2 .

  var X : Var .
  vars st st' : ENV .
  var e : Exp .
  var v : Num .
  var be : BExp .
  vars C C' C'' : Com .

  eq skip ; C = C .
  eq C ; skip = C .

  crl [AsRc] : < X := e, st > => < skip, st[v / X] >
           if < e, st > => v .

  crl [IfRc1] : < If be Then C Else C', st > => < C'', st' >
            if < be, st > => T /\
               < C, st > => < C'', st' > /\ C =/= C'' .
  crl [IfRc2] : < If be Then C Else C', st > => < C'', st' >
            if < be, st > => F /\
               < C', st > => < C'', st' > /\ C' =/= C'' .

  crl [ComRc1] : < C ; C', st > => < C'' ; C', st >
             if < C, st > => < C'', st' > /\ C =/= C'' .
  crl [ComRc2] : < C ; C', st > => < C'', st' >
             if ( C, st ) => Tick /\
                < C', st > => < C'', st' > /\ C' =/= C'' .

  crl [WhileRc1] : < While be Do C, st > => < skip, st >
               if < be, st > => F .
  crl [WhileRc2] : < While be Do C, st > => < C ; (While be Do C), st >
               if < be, st > => T .
```

We also define the termination predicates for the language:

```
  rl [Skipt] : ( skip, st ) => Tick .

  crl [IfRt1] : ( If be Then C Else C', st ) => Tick
            if < be, st > => T /\
               ( C, st ) => Tick .
  crl [IfRt2] : ( If be Then C Else C', st ) => Tick
            if < be, st > => F /\
               ( C', st ) => Tick .

  crl [ComRt] : ( C ; C', st ) => Tick
            if ( C, st ) => Tick /\
               ( C', st ) => Tick .
endm)
```

If we rewrite now a program to swap the values of two variables, their values are not exchanged:

```
Maude> (rew < x := x -. y ;
              y := x +. y ;
```

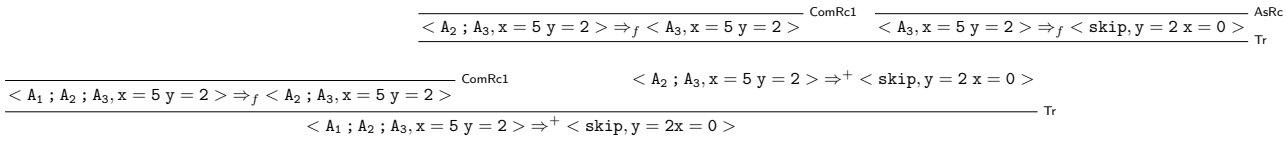

Figure 15: APT for the computation semantics example

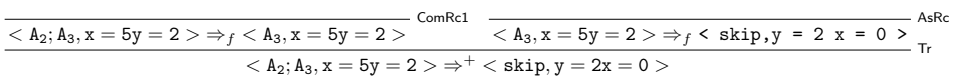

Figure 16: APT for the computation semantics example after the first answer

```
            x := y -. x, x = 5 y = 2 > .)
result Statement : < skip,y = 2 x = 0 >
```

We use the many-steps tree and the default divide and query strategy to debug this behavior:

```
Maude> (many-steps tree .)

Many-steps tree selected.

Maude> (debug < x := x -. y ; y := x +. y ; x := y -. x, x = 5 y = 2 >
          =>* < skip, y = 2 x = 0 > .)
```

The tool builds the debugging tree shown in Figure 15 where the assignments in the program have been abbreviated: $\texttt{A}_1$ = `x := x -. y`, $\texttt{A}_2$ = `y := x +. y`, $\texttt{A}_3$ = `x := y -. x`. The debugging trees above $\Rightarrow_f$ transitions have not been calculated yet; they are *frozen* until the navigation through the tree leads to them.

A node is selected and the following question is presented to the user:

```
Is this rewrite correct?

< y := x +. y ; x := y -. x, x = 5 y = 2 > =>+ < skip, y = 2 x = 0 >

Maude> (no .)
```

Notice the form `=>+` of the arrow in the rewrite appearing in the question, to emphasize that it is a many-steps rewrite.

The transition is wrong because the variables have not been properly updated. The debugger continues with the tree shown in Figure 16 and asks the following question:

```
Is this rewrite (associated with the rule AsRc) correct?

< x := y -. x,x = 5 y = 2 > =>1 < skip,y = 2 x = 0 >

Maude> (yes .)
```

Now the debugger computes the debugging tree associated with the left child in Figure 16. It is shown if Figure 17.

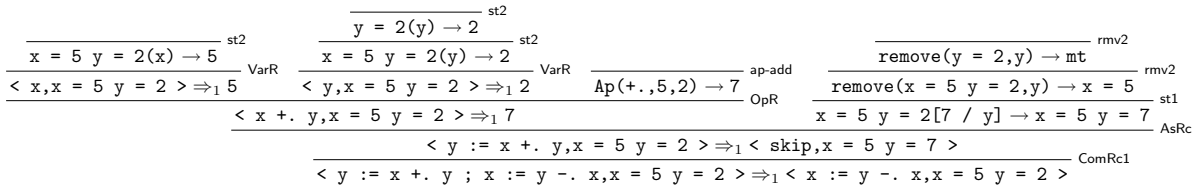

Figure 17: APT for the computation semantics example after the second answer

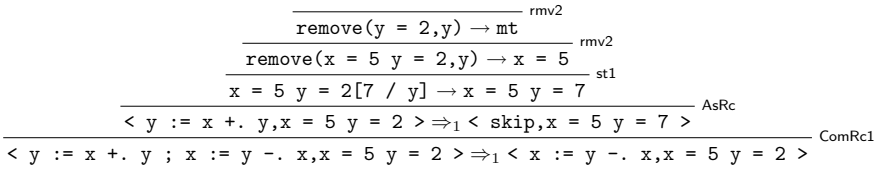

Figure 18: APT for the computation semantics example after the third answer

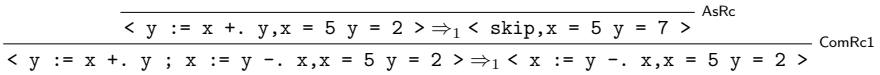

Figure 19: APT for the computation semantics example after the fourth answer

```
Is this rewrite (associated with the rule OpR) correct?

< x +. y, x = 5 y = 2 > =>1 7

Maude> (trust .)
```

We consider that the application of a primitive operation is simple enough to be trusted. The remaining tree is shown in Figure 18. The next question is related to the application of an equation to update the store

```
Is this reduction (associated with the equation st1) correct?

x = 5 y = 2[7 / y] -> x = 5 y = 7

Maude> (yes .)
```

This answers leads to the tree shown in Figure 19. Finally, a question about assignment is done:

```
Is this rewrite (associated with the rule AsRc) correct?

< y := x +. y, x = 5 y = 2 > =>1 < skip, x = 5 y = 7 >

Maude> (yes .)
```

With this information, the debugger is able to find the bug.

```
The buggy node is:
< y := x +. y ; x := y -. x, x = 5 y = 2 > =>1 < x := y -. x, x = 5 y = 2 >
with the associated rule: ComRc1
```

If we check the rule `ComRc1` we realize that the store is not properly updated. The righthand side of the rule should be `< C'' ; C', st' >`:

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

However, since we have the evaluation semantics of the language already specified and debugged, we can use that module as a *correct module* to reduce the number of questions to only one.

```
Maude> (correct module EVALUATION-WHILE .)

EVALUATION-WHILE  selected as correct module.

Maude> (debug in COMPUTATION-WHILE :
           < x := x -. y ; y := x +. y ; x := y -. x, x = 5 y = 2 >
        =>* < skip, y = 2 x = 0 > .)
```

After introducing these commands, the debugger builds the tree shown in Figure 20 and presents the following question:

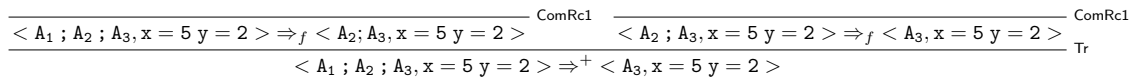

Figure 20: APT for the computation semantics example using a correct module

```
Is this rewrite (associated with the rule ComRc1) correct?

< y := x +. y ; x := y -. x, x = 5 y = 2 > =>1 < x := y -. x, x = 5 y = 2 >

Maude> (no .)
```

Having received a negative answer, the debugger tries to build the debugging tree associated to the left child in Figure 20 which had remained frozen until now. Such a tree is depicted in Figure 17; however, this time the information obtained from the correct module makes it possible to prune it so that the result consists only of the root. Having obtained a single node, it is pointed out as the buggy node.

```
The buggy node is:
< y := x +. y ; x := y -. x, x = 5 y = 2 > => < x := y -. x, x = 5 y = 2 >
with the associated rule: ComRc1
```

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 reflective capabilities, that allow us to use Maude terms and modules as data [9, Chapter 14]. Sections 5.1 to 5.3 describe the tree construction stage, where the abbreviated proof trees are constructed by means of functions that receive the initial symptom (a wrong computation), the module where it took place, a correct module (possibly empty), and a set of suspicious labels. The tree navigation is explained in Section 5.4, and the interaction with the user is explained in Section 5.5.

5.1 Proof tree definition

In this section we show how to represent in Maude the proof trees used in 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 [format (ngi o d d ++i n--i d)] .

 op mtForest : -> Forest [format (ni d)] .
 op __ : Forest Forest -> Forest [assoc id: mtForest] .

 *** Repeated computations
 eq F T F' T F'' = F T F' F'' .
```

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.

```
 op find : Forest Nat ~> Tree .
```

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

```
  op size : Forest -> Nat .
```

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 .
  op getForest : Tree NatList ~> Forest .
  op getSubTree : Tree NatList ~> Tree .
  op hasOffspring? : Tree NatList ~> Bool .

  ...
endfm
```

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

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

 op _:_->_ : Qid Term Term -> Inference [format (b o d b o d)] .
 op _:_:_ : Qid Term Type -> Inference [format (b o d b o d)] .
```

In the case of rewrites, we distinguish between nodes in the one-step tree,

```
  op _:_=>1_ : Qid Term Term -> Inference [format (b o d b o d)] .
```

and nodes in the many-steps tree:

```
  op _=>+_ : Term Term -> Inference [format (d b o d)] .
```

Since the many-steps tree is computed by demand, its leaves corresponding to one-step rewrites are kept as "frozen," and will be evaluated only if needed:

```
  op _:_=>f_ : Qid Term Term -> Inference .
```

In addition, we have an auxiliary node constructor

```
  op _:_=>_top_=>_ : Qid Term Term Term Term -> Inference .
```

that is only used while the debugging tree is being built out of the trace.

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 ProofTreeNode 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{ProofTreeNode} .
```

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 or a rule, and getOffspring returns the number of nodes in the tree.

```
  op getLabel : Tree ~> Qid .
  op getFirstTerm : Tree -> Term .
  op getSecondTerm : Tree ~> Term .
  op getOffspring : Tree -> Nat .
```

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

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

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 .

  ...
endfm
```

5.2 Auxiliary modules

In this section we describe the main auxiliary modules used by the tool.

The PAIR module defines pairs of elements, that satisfy the theory TRIV, and the corresponding projection functions:

```
fmod PAIR{X :: TRIV, Y :: TRIV} is
 sort Pair{X, Y} .
 op <_,_> : X$Elt Y$Elt -> Pair{X, Y} .

 var X : X$Elt .
 var Y : Y$Elt .

 op first : Pair{X, Y} -> X$Elt .
 op second : Pair{X, Y} -> Y$Elt .

 eq first(< X, Y >) = X .
 eq second(< X, Y >) = Y .
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. 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.

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

 op prune : Tree Qid -> Tree .
 op pruneAux : Tree Qid -> Tree .
 op prune* : Forest Qid -> Forest .
 ...
endfm
```

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

```
fmod MODULES is
 pr META-LEVEL .
 pr MAYBE{Module} * (op maybe to noModule) .
 pr CONVERSION .
```

- `functional2system`, that transforms a functional module into a system module by adding the empty set of rules.

```
  op functional2system : FModule -> SModule .
  eq functional2system(fmod Q is IL sorts SS . SSDS ODS MAS EqS endfm) =
     mod Q is IL sorts SS . SSDS ODS MAS EqS none endm .
```

- `deleteSuspicious`, that given a module and a set of suspicious labels, transforms the module by deleting these statements.

```
  op deleteSuspicious : Module QidSet -> Module .
  eq deleteSuspicious(FM, QS) = deleteSuspicious(functional2system(FM), QS) .
  eq deleteSuspicious(mod Q is IL sorts SS . SSDS ODS MAS EqS RlS endm, QS) =
   mod Q is
    IL
    sorts SS .
    SSDS
    ODS
    delete(MAS, QS)
    delete(EqS, QS)
    delete(RlS, QS)
   endm .

  op delete : MembAxSet QidSet -> MembAxSet .
  op delete : EquationSet QidSet -> EquationSet .
  op delete : RuleSet QidSet -> RuleSet .
```

- `extractLabels`, that extracts all the labels in a module. We use this function to know which statements must be treated if the user decides to debug without selecting specific labels.

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

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

```
  op generalEq : Equation -> Equation .
  op generalMb : MembAx -> MembAx .
```

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

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

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

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

  ...
endfm
```

The module SUBSTITUTION is in charge of applying substitutions to terms.

```
fmod SUBSTITUTION is
 pr MODULES .
```

The function substitute applies a substitution to a term, and then normalizes it:

```
 op substitute : Module Term Substitution -> Term .
 op substitute : Term Substitution -> Term .
 op substitute* : TermList Substitution -> TermList .
 ...
```

The function substituteHole substitutes the hole in a context by a term, and returns the normal form of the term.

```
 op substituteHole : Module Context Term -> Term .
 ...
endfm
```

5.3 Debugging tree construction

In this section we describe how the different abbreviated debugging trees are built. First, we describe the construction of debugging trees for reductions and memberships, and then we use them in the construction of trees for rewrites. Instead of creating the complete proof trees and then abbreviating them, we build the abbreviated proof trees directly.

5.3.1 Debugging trees for wrong reductions and memberships

The function createTree builds debugging trees for reductions or memberships. It exploits the fact that the equations and membership axioms are both *terminating* and *confluent*. It receives the module where a suspicious inference took place, a correct module (or the constant noModule when no such module is provided) to prune the tree, the term initially reduced, the (erroneous) result obtained, and the set of suspicious statement labels. It keeps the initial 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 deleteSuspicious), 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 statements, thus avoiding to build a tree that will be finally empty.

```
fmod FUNCTIONAL-TREE-CONSTRUCTION is
 pr SUBSTITUTION .
 pr PAIR{TermList, Forest} .
 pr MAYBE{Forest} * (op maybe to noProof) .

 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' := deleteSuspicious(M, QS) /\
     F := createForest(M, M', CM, T, 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(PN, TR)) = TR .
 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' := deleteSuspicious(M, QS) /\
     F := createForest(M, M', CM, 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 (as we will see below).

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 allows trusted inferences), a correct module (possibly `noModule`) to check the inferences, two terms representing the inference whose proof tree we want to generate, and a set of labels of suspicious equations and memberships. This function checks if the terms are equal, if the result can be reached by using only trusted statements, or if the correct module can calculate this inference; in such cases, there is no need to calculate the tree, so the empty forest is returned. Otherwise, it uses the function `createForest2`, which 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 .
eq createForest(OM, TM, CM, T, T', QS) =
 if T == T' or-else reduce(TM, T) == T' or-else reduce(CM, T) == T' then mtForest
 else createForest2(OM, TM, CM, T, T', QS)
 fi .
```

When the term being reduced is of the form `if T1 then T2 else T3 fi` Maude first evaluates `T1` and then, depending on the result, it evaluates either `T2` or `T3`. The first equation of `createForest2` handles this special behavior, so that only debugging trees that correspond to evaluated subterms are really built.

```
op createForest2 : Module Module Maybe{Module} Term Term QidSet ~> Forest .
eq createForest2(OM, TM, CM, 'if_then_else_fi[T1, T2, T3], T', QS)  =
   createForest(OM, TM, CM, T1, reduce(OM, T1), QS)
   if reduce(OM, T1) == 'true.Bool then
      createForest(OM, TM, CM, T2, T', QS)
   else
      if reduce(OM, T1) == 'false.Bool then
         createForest(OM, TM, CM, T3, T', QS)
      else
         createForest(OM, TM, CM, T2, reduce(OM, T2), QS)
         createForest(OM, TM, CM, T3, reduce(OM, T3), QS)
      fi
   fi .

ceq createForest2(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.

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

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, T, QS) = < T, mtForest > [owise] .

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,[3] by using the *replacement* rule from Figure 1, with the constraint that we cannot apply equations with the `otherwise` attribute if other

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

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.

```
op applyEq : Module Module Maybe{Module} Term Term QidSet -> Maybe{Forest} .
op applyEq : Module Module Maybe{Module} Term Term QidSet EquationSet -> Maybe{Forest} .

ceq applyEq(OM, TM, CM, T, T', QS) = mtForest
 if reduce(TM, T) == T' or-else reduce(CM, T) == 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.

```
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' := substitute(OM, R, SB) /\
    F := conditionForest(substitute(OM, C, SB), OM, TM, CM, QS)
         createForest(OM, TM, CM, R', T', QS) .
```

If we cannot apply any equation without the `otherwise` attribute, we check the other equations.

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

Finally, if no equation is applicable, the function returns `noProof`.

```
eq applyEq(OM, TM, CM, T, T', QS, EqS) = noProof [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 respective 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.

```
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 Figure 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 sortLeq(TM, type(TM, T), S) or-else sortLeq(CM, type(CM, T), 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 Figure 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.

```
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.

```
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, Ty, QS, op Q : nil -> Ty [AtS] . ODS) = mtForest
 if getName(CONST) = Q /\
    getType(CONST) = Ty .
```

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) .
```

Notice that the operator at top in the term can be different of the operator declaration if it is declared with the `iter` operator. We treat this case by eliminating the iteration from the top with the function `noIter`.

```
ceq applyOp(OM, TM, CM, Q'[T], Ty', QS, op Q : Ty -> Ty' [iter AtS] . ODS) =
    createForest(OM, TM, CM, Q''[T], type(OM, Q''[T]), QS)
 if Q[Q''[T]] := noIter(Q'[T]) /\
    sortLeq(OM, type(OM, Q''[T]), Ty) .

eq applyOp(OM, TM, CM, T, S, QS, ODS) = noProof [owise] .

op noIter : Term -> Term .
...
```

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

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

5.3.2 Debugging trees for wrong rewrites

We use a different methodology in the construction of the debugging tree for incorrect rewrites. Since these modules are not assumed to be confluent or terminating, we use the predefined breadth-first search function `metaSearchPath` to find from the initial term the wrong term introduced by the user, and then we use the returned trace to build the debugging tree. The trace returned by Maude when searching from `T` to `T'` is a list of steps of the form

```
{T1, Ty1, R1} ... {Tn, Tyn, Rn}
```

where `T1` is the normal form of `T`, `Ri` is the rule applied to (a subterm of) `Ti` to obtain `Ti+1` (which is already a normal form), and `T'` is the result of applying `Rn` to `Tn`.

The function `createRewTree`, given the module where the rewrite took place, a module with correct statements (possibly empty), the rewritten term, the result term, the set of suspicious labels, the type of tree selected (many-steps or one-step, identified by constants `ms` and `os` in the module `TREE-TYPE`), and the bound of the search in the correct modules, creates the corresponding debugging tree.

```
fmod SYSTEM-TREE-CONSTRUCTION is
 pr FUNCTIONAL-TREE-CONSTRUCTION .
 pr PAIR{Substitution, Context} .
 pr PAIR{TermList, NeCTermList} .
 pr PAIR{Forest, Forest} .
 pr MAYBE{Tree} * (op maybe to error) .
 pr TREE-TYPE .
 pr EXT-BOOL .

 op createRewTree : Module Maybe{Module} Term Term QidSet TreeType Bound -> Maybe{Tree} .

 eq createRewTree(OM, CM, T, T', QS, os, B) = oneStepTree(OM, CM, T, T', QS, B) .
 eq createRewTree(OM, CM, T, T', QS, ms, B) = manyStepsTree(OM, CM, T, T', QS, B) .
```

The function `oneStepTree` creates a complete debugging tree with only one-step rewrites in its nodes. It puts as root of the tree the complete inference, computes the tree for the reduction from the initial term to normal form with the function `createForest` from Section 5.3.1, and then computes the rest of the tree with the function `oneStepForest`. This correspond to a concrete application of the *equivalence class* inference rule from Figure 1.

```
op oneStepTree : Module Maybe{Module} Term Term QidSet Bound -> Maybe{Tree} .
ceq oneStepTree(OM, CM, T, T', QS, B) =
           contract(tree(node(T =>+ T', getOffspring*(F) + 1), F))
 if TM := deleteSuspicious(OM, QS) /\
    T1 := reduce(OM, T) /\
    F := createForest(OM, TM, CM, T, T1, QS)
         oneStepForest(OM, TM, CM, T1, T', QS, B) .

eq oneStepTree(OM, CM, T, T', QS, B) = error [owise] .
```

`oneStepForest` computes the trace of a rewrite with the predefined function `metaSearchPath` and uses it to generate a debugging tree by using `trace2forest`:

```
op oneStepForest : Module Module Maybe{Module} Term Term QidSet Bound -> Maybe{Forest} .
ceq oneStepForest(OM, TM, CM, T, T', QS, B) = F
 if TR := metaSearchPath(OM, T, T', nil, '*, unbounded, 0) /\
    F := trace2forest(OM, TM, CM, TR, T', QS, B) .

eq oneStepForest(OM, TM, CM, T, T', QS, B) = noProof [owise] .
```

The function `trace2forest` generates a forest of one-step rewrites by extracting each step of the trace and creating its corresponding tree with the function `stepForest`.

```
op trace2forest : Module Module Maybe{Module} Trace Term QidSet Bound -> Forest .
eq trace2forest(OM, TM, CM, nil, T, QS, B) = mtForest .
eq trace2forest(OM, TM, CM, {T, Ty, R}, T', QS, B) =
   stepForest(OM, TM, CM, T, T', R, QS, B, os) .
eq trace2forest(OM, TM, CM, {T, Ty, R} {T', Ty', R'} TR, T'', QS, B) =
   stepForest(OM, TM, CM, T, T', R, QS, B, os)
   trace2forest(OM, TM, CM, {T', Ty', R'} TR, T'', QS, B) .
```

The first equation of `stepForest` checks if the value can be obtained by using trusted statements or the correct module.

```
op stepForest : Module Module Maybe{Module} Term Term Rule QidSet Bound TreeType -> Forest .
ceq stepForest(OM, TM, CM, T, T', R, QS, B, TT) = mtForest
 if secure?(TM, CM, T, T', B) .
```

where `secure?` searches in the modules with correct statements if the rewrite is possible. Note the use of the bound, that can be selected by the user.

```
op secure? : Module Maybe{Module} Term Term Bound -> Bool .
eq secure?(TM, CM, T, T', B) = metaSearchPath(TM, T, T', nil, '+, B, 0) :: Trace
                        or-else metaSearchPath(CM, T, T', nil, '+, B, 0) :: Trace .
```

The second equation of `stepForest` deals with the other cases. It uses the function `getInfo` to obtain the substitution and the context of the application of the rule, and then applies this substitution in the lefthand and righthand sides to obtain the rewritten (sub)terms. The whole rewritten term is obtained by substituting the hole in the context by the subterm computed before (by using `substituteHole`). The corresponding normal forms are computed with the function `createForest` for reductions. This equation reproduces the abbreviation of the application of the *replacement*, *congruence*, and *equivalence class* inference rules.

```
ceq stepForest(OM, TM, CM, T, T', R, QS, B, TT) = if trusted?(R, QS) then F F'
        else tree(node(getName(R) : T1 =>1 T3, getOffspring*(F) + 1), F) F' fi
 if < SB, C > := getInfo(OM, T, T', getName(R), 0) /\
    T1 := substitute(OM, getLefthand(R), SB) /\
    T2 := substitute(OM, getRighthand(R), SB) /\
```

```
    T3 := reduce(OM, T2) /\
    F := conditionForest(substitute(OM, getCondition(R), SB), OM, TM, CM, QS, B, TT)
         createForest(OM, TM, CM, T2, T3, QS) /\
    F' := createForest(OM, TM, CM, substituteHole(OM, C, T3), T', QS) [owise] .

op getName : Rule ~> Qid .
op getCondition : Rule -> Condition .
op getRighthand : Rule -> Term .
op getLefthand : Rule -> Term .
...
```

where `trusted?` checks if the rule is not labeled or it is labeled but the label does not belong to the set of suspicious statements, and `getInfo` uses `metaXapply` to compute the substitution and the context needed to rewrite a term into another one.

```
op trusted? : Rule QidSet -> Bool .
eq trusted?(rl T => T' [label(Q) AtS] ., Q ; QS) = false .
eq trusted?(crl T => T' if COND [label(Q) AtS] ., Q ; QS) = false .
eq trusted?(R, QS) = true [owise] .

op getInfo : Module Term Term Qid Nat ~> Pair{Substitution, Context} .
ceq getInfo(M, T, T', Q, N) = < SB, C >
 if {T', Ty, SB, C} := metaXapply(M, T, Q, none, 0, unbounded, N) .
ceq getInfo(M, T, T', Q, N) = getInfo(M, T, T', Q, s(N))
 if {T'', Ty, SB, C} := metaXapply(M, T, Q, none, 0, unbounded, N) /\
    T' =/= T'' .
```

The new version of function `conditionForest` has as new parameters the bound of the search in the correct modules and a value indicating if the forest generated by the conditions must be either one-step or many-step.

```
op conditionForest : Condition Module Module Maybe{Module} QidSet Bound TreeType -> Forest .

eq conditionForest(nil, OM, TM, CM, QS, B, TT) = mtForest .
eq conditionForest(T = T' /\ COND, OM, TM, CM, QS, B, TT) =
                  createForest(OM, TM, CM, T, reduce(OM, T), QS)
                  createForest(OM, TM, CM, T', reduce(OM, T'), QS)
                  conditionForest(COND, OM, TM, CM, QS, B, TT) .
eq conditionForest(T := T' /\ COND, OM, TM, CM, QS, B, TT) =
                  createForest(OM, TM, CM, T', reduce(OM, T), QS)
                  conditionForest(COND, OM, TM, CM, QS, B, TT) .
eq conditionForest(T : S /\ COND, OM, TM, CM, QS, B, TT) =
                  createForest(OM, TM, CM, T, type(OM, T), QS)
                  conditionForest(COND, OM, TM, CM, QS, B, TT) .
```

Furthermore, it is also defined the case of rewrite conditions, that uses the corresponding tree construction function.

```
eq conditionForest(T => T' /\ COND, OM, TM, CM, QS, B, TT) =
    if TT == os then oneStepForest(OM, TM, CM, T, T', QS, B)
                else manyStepsTree2(OM, TM, CM, T, T', QS, B)
    fi
    conditionForest(COND, OM, TM, CM, QS, B, TT) .
```

The many-steps debugging tree is built with the function `manyStepsTree`. This tree is computed *by demand*, so that the debugging subtrees corresponding to one-step rewrites are only generated when they are pointed out as wrong. It uses an auxiliary function `manyStepsTree2`, that also receives as parameter the module cleaned of suspicious statements with `deleteSuspicious`.

```
op manyStepsTree : Module Maybe{Module} Term Term QidSet Bound -> Maybe{Tree} .
ceq manyStepsTree(OM, CM, T, T', QS, B) =
                 contract(tree(node(T =>+ T', getOffspring*(F) + 1), F))
 if F := manyStepsTree2(OM, deleteSuspicious(OM, QS), CM, T, T', QS, B) .
eq manyStepsTree(OM, CM, T, T', QS, B) = error [owise] .
```

This auxiliary function uses the function `metaSearchPath` to compute the trace. If it is not empty, the forest for the reduction of the initial term to normal form is built with the function `createForest` and the tree for the rewrites is appended to this forest. If the trace consists on only one step, it is expanded with the function `stepForest` shown above. Otherwise, the many-steps tree from the trace is build with the function `trace2tree`.

```
op manyStepsTree2 : Module Module Maybe{Module} Term Term QidSet Bound ~> Maybe{Forest} .
ceq manyStepsTree2(OM, TM, CM, T, T', QS, B) = F
 if {T'', Ty, R} TR := metaSearchPath(OM, T, T', nil, '*, unbounded, 0) /\
    F := createForest(OM, TM, CM, T, T'', QS)
         if TR =/= nil
         then trace2tree(OM, TM, CM, {T'', Ty, R} TR, T', QS, B, mtForest, 0)
         else stepForest(OM, TM, CM, T'', T', R, QS, B, ms)
         fi .
```

If the trace is empty, only the tree for the reduction is computed:

```
ceq manyStepsTree2(OM, TM, CM, T, T', QS, B) = createForest(OM, TM, CM, T, T', QS)
 if nil == metaSearchPath(OM, T, T', nil, '*, unbounded, 0) .
```

Finally, if the final term is not reachable from the initial term, an error is returned. Note that errors due to non-termination cannot be detected.

```
eq manyStepsTree2(OM, TM, CM, T, T', QS, B) = noProof [owise] .
```

The function `trace2tree` traverses the trace and for those steps which are not secure (that is, they use suspicious statements and cannot be inferred in the correct module) it builds a temporal node that will be used later to build the many-steps tree with the function `divide`.

```
op trace2tree : Module Module Maybe{Module} Trace Term QidSet Bound Forest Nat -> Tree .
eq trace2tree(OM, TM, CM, nil, T', QS, B, F, N) = mtForest .
ceq trace2tree(OM, TM, CM, {T, Ty, R}, T', QS, B, F, N) = divide(F, N)
 if secure?(TM, CM, T, T', B) .
eq trace2tree(OM, TM, CM, {T, Ty, R}, T', QS, B, F, N) =
              divide(F getTree(OM, T, T', R, QS), s(N)) [owise] .
ceq trace2tree(OM, TM, CM, {T, Ty, R} {T', Ty', R'} TR, T'', QS, B, F, N) =
          trace2tree(OM, TM, CM, {T', Ty', R'} TR, T'', QS, B, F, N)
 if secure?(TM, CM, T, T', B) .
eq trace2tree(OM, TM, CM, {T, Ty, R} {T', Ty', R'} TR, T'', QS, B, F, N) =
          trace2tree(OM, TM, CM, {T', Ty', R'} TR, T'', QS, B, F
                                    getTree(OM, T, T', R, QS), s(N)) [owise] .
```

where the function `getTree` uses the function `getInfo` to obtain the concrete subterms that have been rewritten, although it also keeps the information about the terms that have been rewritten at the top, which will be used later to create the inferences obtained by transitivity.

```
op getTree : Module Term Term Rule QidSet -> Tree .
ceq getTree(M, T, T', R, QS) = tree(node(Q : T1 => T3 top T => T', 1), mtForest)
 if Q := getName(R) /\
    < SB, C > := getInfo(M, T, T', Q, 0) /\
    T1 := substitute(M, getLefthand(R), SB) /\
    T2 := substitute(M, getRighthand(R), SB) /\
    T3 := reduce(M, T2) .
```

The function `divide` creates a balanced tree from the forest of leaves obtained from the trace. It divides the forest received as parameter into two forests of approximately the same size, recursively creates their trees, and uses them as children of a new binary tree that has as root the combination by *transitivity* of the rewrites in their roots.

```
op divide : Forest Nat ~> Tree .
```

We consider as basic cases the forests composed by one, two, or three trees.

```
  eq divide(A, 1) = freeze(A) .

  ceq divide(A A', 2) = tree(node(getFirstTerm(A) =>+ getSecondTerm(A'), 3), A2 A3)
   if A2 := freeze(A) /\
      A3 := freeze(A') .

  ceq divide(A F, 3) = tree(node(getFirstTerm(A) =>+ getSecondTerm(A'), 5), A2 A')
   if A' := divide(F, 2) /\
      A2 := freeze(A) .
```

We transform the leaves of the trees with the function `freeze` because the exta information is no longer useful:

```
  op freeze : Tree -> Tree .
  eq freeze(tree(node(Q : T => T' top T1 => T2, N), F)) = tree(node(Q : T =>f T', N), F) .
  eq freeze(A) = A [owise] .
```

The general case halves the forest and creates the corresponding trees, building a new tree with them:

```
  ceq divide(F, N) = tree(node(getFirstTerm(A) =>+ getSecondTerm(A'), s(N1)), A A')
   if N > 3 /\
      N' := N quo 2 /\
      < F', F'' > := takeDrop(F, N') /\
      A := divide(F', N') /\
      A' := divide(F'', sd(N, N')) /\
      N1 := getOffspring(A) + getOffspring(A') .
```

where the function `takeDrop` returns a pair composed by the first `N` elements of the forest and the rest:

```
  op takeDrop : Forest Nat -> Pair{Forest, Forest} .
  eq takeDrop(F, N) = takeDrop(F, N, mtForest) .

  op takeDrop : Forest Nat Forest -> Pair{Forest, Forest} .
  eq takeDrop(A F, s(N), F') = takeDrop(F, N, F' A) .
  eq takeDrop(F, N, F') = < F', F > [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 extra function to compute it (apart from those in module `TREE`). On the other hand, the divide and query strategy 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).

```
fmod DIVIDE-QUERY-STRATEGY is
  pr PROOF-TREE .
  pr PAIR{NatList, Nat} .

  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 .
```

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.

```
  op searchBestNode : Tree -> NatList .

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

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 differences 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. 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.

```
  op searchBestNode : Tree Nat Nat Nat NatList NatList -> Pair{NatList, Nat} .
  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 -> Pair{NatList, Nat} .

  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) .

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

5.5 The debugger environment

We implement our system on top of Full Maude, a language that extends Maude with support for object-oriented specification and advanced module operations [9, Part II]. The implementation of Full Maude includes code for parsing user input and pretty-printing; storing modules, theories, and views; and transforming object-oriented modules into system modules.

To parse some input using the built-in function `metaParse`, Full Maude needs the meta-representation of the signature in which the input has to be parsed. Thus, we define the signature of the debugger in a module that extends the Full Maude signature.

```
fmod DD-SIGNATURE is
 including FULL-MAUDE-SIGN .

 op debug_. : @Bubble@ -> @Command@ .
 op top-down‘strategy‘. : -> @Command@ .
 op divide-query‘strategy‘. : -> @Command@ .
 op one-step‘tree‘. : -> @Command@ .
 op many-steps‘tree‘. : -> @Command@ .
 op node_. : @Token@ -> @Command@ .
 op all‘valid‘. : -> @Command@ .
 op yes‘. : -> @Command@ .
 op no‘. : -> @Command@ .
 op trust‘. : -> @Command@ .
 op undo‘. : -> @Command@ .
 op correct‘module_. : @ModExp@ -> @Command@ .
 op delete‘correct‘module‘. : -> @Command@ .
 op set‘bound_. : @Token@ -> @Command@ .
 op set‘debug‘select‘on‘. : -> @Command@ .
 op set‘debug‘select‘off‘. : -> @Command@ .
 op debug-select_. : @NeTokenList@ -> @Command@ .
 op debug-deselect_. : @NeTokenList@ -> @Command@ .
 op debug-include_. : @Bubble@ -> @Command@ .
 op debug-exclude_. : @Bubble@ -> @Command@ .
endfm
```

This signature is included in the meta-module GRAMMAR to obtain the grammar DD-GRAMMAR, that allows to parse both Full Maude modules and commands and the debugger commands.

```
fmod META-DD-SIGN is
 inc META-FULL-MAUDE-SIGN .
 inc UNIT .

 op DD-GRAMMAR : -> FModule [memo] .
 eq DD-GRAMMAR = addImports((including 'DD-SIGNATURE .), GRAMMAR) .

 ...
endfm
```

The module DD-COMMAND-PROCESSING is in charge of processing the commands dealing with suspicious statements and the debugging command.

```
fmod DD-COMMAND-PROCESSING is
  pr COMMAND-PROCESSING .
  pr META-DD-SIGN .
  pr SYSTEM-TREE-CONSTRUCTION .
  pr PRINT .
```

The parsing of the debugging command returns a tuple containing the generated tree, the module where the computation took place, the set of suspicious statements, and a list of quoted identifiers indicating if an error has occurred.

```
  sort DebugTuple .
  op <_,_,_,_> : Tree Maybe{Module} QidSet QidList -> DebugTuple .
```

The parsing of the command is done in the GRAMMAR-DEB module, where the first bubble can contain either a module or just the initial term.

```
  op GRAMMAR-DEB : -> FModule [memo] .
  eq GRAMMAR-DEB = addOps(op '_->_. : '@Bubble@ '@Bubble@ -> '@Inference@ [none] .
                          op '_:_. : '@Bubble@ '@Bubble@ -> '@Inference@ [none] .
                          op '_=>*_. : '@Bubble@ '@Bubble@ -> '@Inference@ [none] .,
                          addSorts('@Inference@, GRAMMAR-RED)) .
```

The function `procDebug` processes a bubble and returns either a tree for the corresponding debug command or an error message. It receives the term to be parsed, a correct module (possibly `noModule`), a Boolean indicating if debug-select is on, the set of suspicious labels, the selected type of tree, the bound of the search in the correct module, the default module, and the Full Maude's database of modules.

After finding out the kind of the debugging command (reduction, membership, or rewrite) and if a module name has been selected by the command, the function `procDebug` builds the appropriate tree by using the functions `createTree` and `createRewTree` explained in Section 5.3.

```
  op procDebug : Term Maybe{Module} Bool QidSet TreeType Bound ModuleExpression
                 Database -> DebugTuple .
```

The processing of the commands for selecting or deselecting the labels of a list of modules is accomplished by the function `procInclude`. It receives a Boolean indicating if the selection option is enabled, the term to be parsed (with the list of modules) and the Full Maude's database, and it returns a pair with the set of labels from existing modules, and an error message for the problematic modules.

```
  sort IncludePair .
  op <_:_> : QidSet QidList -> IncludePair .

  op procInclude : Bool Term Database -> IncludePair .

  ...
endfm
```

The persistent state of Full Maude's system is given by a single object o f class `DatabaseClass`, which maintains the database of the system. We extend the Full Maude system by defining a subclass of `DatabaseClass` inheriting its behavior and adding new attributes to it.

```
 mod DD-DATABASE-HANDLING is
  inc DATABASE-HANDLING .
  pr DD-COMMAND-PROCESSING .
  pr TREE-PRUNING .
  pr DIVIDE-QUERY-STRATEGY .
  pr LIST{DDState} .
  pr LIST{Answer} .

  sort DDDatabaseClass .
  subsort DDDatabaseClass < DatabaseClass .

  op DDDatabase : -> DDDatabaseClass [ctor] .
```

The new attributes are:

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

  ```
  op tree :_ : Forest -> Attribute .
  ```

- the type of tree (`treeType`) that will be built when debugging rewrites. It takes the value `os` when the one-step tree is selected and `ms` when the many-steps tree is chosen.

  ```
  op treeType :_ : TreeType -> Attribute .
  ```

- the `current` node of the tree that we are analyzing, represented by a list of natural numbers.

  ```
  op current :_ : NatList -> Attribute .
  ```

- the `strategy` to traverse the tree. The top-down strategy is represented by the constant `td`, and divide and query is represented by `dq`.

  ```
  op strategy :_ : Strat -> Attribute .
  ```

- the `module` where debugging takes place. It has sort `Maybe{Module}`, so its value initially is `noModule`.

```
op module :_ : Maybe{Module} -> Attribute .
```

- 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 `noModule`.

```
op correction :_ : Maybe{Module} -> Attribute .
```

- the `bound` used in the correct module to search for answers when debugging rewrites.

```
op bound :_ : Bound -> Attribute .
```

- the value of the `select` option, that indicates if it is enabled.

```
op select :_ : Bool -> Attribute .
```

- the set of labels considered `suspicious`.

```
op suspicious :_ : QidSet -> Attribute [gather(&)] .
```

- the set of labels that has been considered suspicious for the construction of the current tree (`currentSuspicious`).

```
op currentSuspicious :_ : QidSet -> Attribute  [gather(&)] .
```

- the stack of previous states (`previousStates`), used to restore the state when the `undo` command is used.

```
op previousStates :_ : List{DDState} -> Attribute [gather(&)] .
```

- the list of `answers` already provided by the user, used to avoid making the same question twice.

```
op answers :_ : List{Answer} -> Attribute [gather(&)] .
```

- the `state` of the tool. It is `waiting` when the tool needs information from the user, `computing` when it is processing that information, and `finished` when the debugging process has concluded.

```
sort ToolState .
ops waiting computing finished : -> ToolState .

op state :_ : ToolState -> Attribute .
```

The initial values of these attributes are defined with the constant `init-state`:

```
op init-state : -> AttributeSet .

*** Initial values of the attributes (except input and output)
eq init-state = db : initialDatabase,
                default : 'CONVERSION, tree : mtForest, module : noModule,
                current : nil, strategy : dq, correction : noModule,
                previousStates : nil, answers : nil,
                bound : 42, state : waiting, treeType : os, suspicious : none,
                currentSuspicious : none, select : false .
```

The behavior of the debugger commands is described by means of rewrite rules that change the state of these attributes. Below we show some of the most interesting rules.

The rule `debug` starts the debugging process. It receives a term that will be processed with the function `procDebug` explained above. If there is no error (that is, the returned list of qids is `nil`), the tree, the module, and the set of suspicious labels are updated with the appropriate information, while the answers given by the user so far and the previous states are reset. However, if the command was incorrect, the error is shown and the state is set to `finished`.

```
crl [debug] :
    < O : DDDC | db : DB, input : ('debug_.[T]), output : nil,
                 default : ME, tree : F, module : MM, correction : MM',
                 previousStates : LS, answers : LA, state : TS,
                 treeType : TT, bound : BND, select : B, suspicious : QS,
                 currentSuspicious : QS', AtS >
 => if QIL == nil then
    < O : DDDC | db : DB, input : nilTermList, output : nil, default : ME,
                 tree : F', module : MM'', correction : MM',
                 previousStates : nil, answers : nil, state : computing,
                 treeType : TT, bound : BND, select : B, suspicious : QS,
                 currentSuspicious : QS'', AtS >
    else
    < O : DDDC | db : DB, input : nilTermList, output : QIL, default : ME,
                 tree : mtForest, module : MM, correction : MM',
                 previousStates : nil, answers : nil, state : finished,
                 treeType : TT, bound : BND, select : B, suspicious : QS,
                 currentSuspicious : QS', AtS >
    fi
 if < F', MM'', QS'', QIL > := procDebug(T, MM', B, QS, TT, BND, ME, DB) .
```

When the divide and query strategy is selected, the following rule handles the command:

```
rl [divide-query-strategy] :
   < O : DDDC | input : ('divide-query`strategy`..@Command@), output : nil,
                strategy : STRAT, state : TS, AtS >
=> < O : DDDC | input : nilTermList, output : ('\n 'Divide '& 'Query 'strategy
                                                'selected. '\n ),
                state : if TS == finished then finished else computing fi,
                strategy : dq, AtS > .
```

In the top down strategy, when the user introduces the identifier of a wrong node, the debugger updates the list of answers and the previous states, and changes the current tree by the appropriate child of the root.

```
crl [top-down-traversal] :
    < O : DDDC | input : ('node_.['token[T]]), strategy : td, tree : PT,
                 previousStates : LS, answers : LA, state : waiting, AtS >
 => < O : DDDC | input : nilTermList, strategy : td, tree : PT',
                 previousStates : LS < nil, PT, td >,
                 answers : LA getAnswer(PT', wrong), state : computing, AtS >
 if N := downNat*(T) /\
    N < size(getForest(PT, nil)) /\
    PT' := getSubTree(PT, N) .
```

where the function `getAnswer` constructs an answer given the current node and the answer given by the user.

When the divide and query strategy is selected and the user decides to trust a statement, the current subtree is deleted and the resulting tree pruned in order to delete the nodes associated with the trusted statement.

```
crl [divide-query-traversal] :
    < O : DDDC | input : ('trust`..@Command@), strategy : dq, tree : PT,
                 current : NL, previousStates : LS, answers : LA,
                 state : waiting, AtS >
```

```
   => < O : DDDC | input : nilTermList, strategy : dq, tree : PT', current : NL,
                   previousStates : LS < NL, PT, dq >,
                   answers : LA getAnswer(getSubTree(PT, NL), right),
                   state : computing, AtS >
  if N := getOffspring(PT, NL) /\
     Q := getLabel(PT, NL) /\
     PT' := prune(deleteSubTree(PT, NL, N), Q) .
```

When the user decides to switch the select mode on to use a subset of the labeled statements as suspicious, the `select` attribute is set to `true` and the set of currently suspicious statements is reset if the mode was disabled before.

```
  rl [select] :
     < O : DDDC | input : ('set'debug'select'on'..@Command@), select : B,
                  output : nil, suspicious : QS, AtS >
  => < O : DDDC | input : nilTermList, select : true,
                  output : ('\n 'Debug 'select 'is '\! 'on. '\o '\n),
                  suspicious : if B then QS else none fi, AtS > .

  ...

endm
```

The module `DD` manages the introduction of data by the user and the output of the debugger's answers. Full Maude uses the input/output facility provided by the `LOOP-MODE` module [9, Chapter 17], that consists in an operator `[_,_,_]` with an input stream (the first argument), an output stream (the third argument), and a state (given by its second argument).

```
mod DD is
  inc DD-DATABASE-HANDLING .
  inc LOOP-MODE .
  inc META-DD-SIGN .

  op o : -> Oid .

  --- State for LOOP mode:
  subsort Object < State .
  op init-debug : -> System .

  rl [init] :
     init-debug
  => [nil, < o : DDDatabase | input : nilTermList, output : nil, init-state >, dd-banner] .
```

The rule `in` below parses the data introduced by the user, that appears in the first argument of the loop, in the module `DD-GRAMMAR` and introduces it in the `input` attribute if it is correctly built.

```
  crl [in] :
      [QIL, < O : X@Database | input : nilTermList, Atts >, QIL']
   => [nil,
       < O : X@Database | input : getTerm(metaParse(DD-GRAMMAR, QIL, '@Input@)), Atts >,
       QIL']
   if QIL =/= nil /\
      metaParse(DD-GRAMMAR, QIL, '@Input@) : ResultPair .
```

The rule `out` is in charge of printing the messages from the debugger by moving the data in the `output` attribute to the third component of the loop.

```
  rl [out] :
     [QIL, < O : X@Database | output : (QI QIL'), Atts >, QIL'']
  => [QIL, < O : X@Database | output : nil, Atts >, (QIL'' QI QIL')] .
endm
```

The command `loop init-debug` initializes the state of the loop.

6 Conclusions and future work

In this paper we have developed the foundations of declarative debugging of executable rewriting logic specifications, and we have applied them to implement a debugger for Maude modules. The work encompasses and extends previous presentations [6, 5] on the declarative debugging of Maude functional modules, which constitute now a particular case of a more general setting.

We have described formally how debugging trees can be obtained from Maude proof trees, proving the correctness and completeness of the debugging technique. The tool based on these ideas allows the user to concentrate on the logic of the program disregarding the operational details. In order to deal with the possibly complex questions associated to rewrite statements, the tool offers the possibility of choosing between two different debugging trees: the one-step trees, with simpler questions and likely longer debugging sessions, and the many-steps trees which in general require fewer but more complex questions before finding the bug. The experience will show the user which one must be chosen in each case depending on the complexity of the specification.

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.

Future work will include the following:

- The current version of our debugger builds incrementally a debugging tree only when the user selects the many-steps strategy when debugging a rewrite computation. We plan to extend this idea to the debugging trees for reductions and memberships and for the one-step strategy for rewrites.
- We also want to improve the interaction with the user by providing a complementary graphical interface that allows the user to navigate the tree with more freedom.
- This interaction could be also improved by allowing the user to give the answer "don't know", that would postpone the answer to the question by asking alternative questions.
- We are investigating how to handle the `strat` operator attribute, that allows the specifier to define an evaluation strategy. This can be used to represent some kind of laziness.
- We also plan to study how to debug *missing answers* [16] in addition to the wrong answers we have treated thus far.

References

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

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

[3] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1):386–414, 2006.

[4] 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.

[5] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A declarative debugger for Maude functional modules. In *Proceedings Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008*. Elsevier, 2008. To appear.

[6] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. Declarative debugging of membership equational logic specifications. In *Ugo Montanari's Festschrift*, Lecture Notes in Computer Science. Springer, 2008. To appear.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

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

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

[15] 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.

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

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

[18] H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

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

[20] 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.

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

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

[23] 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. Springer, 2000.

[24] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67:226–293, 2006.

[25] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

[26] P. Wadler. Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, 1998.