Declarative Debugging of Missing Answers in Rewriting Logic*

Adrián Riesco, Alberto Verdejo, and Narciso Martí-Oliet

Technical Report SIC-6-09

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

October 2009

*Research supported by MEC Spanish project *DESAFIOS* (TIN2006-15660-C02-01) and Comunidad de Madrid program *PROMESAS* (S0505/TIC/0407).

Abstract

Rewriting logic is a logic of change, where rewrites correspond to transitions between states. One of the main characteristics of these transitions is that they can be nondeterministic, that is, given an initial state, there is a *set* of possible reachable states. Thus, an additional problem when debugging rewrite systems is that, although all the terms obtained could be correct, it is possible that not all the desired terms are computed, i.e., there are *missing answers*.

We propose a calculus that allows to infer, given an initial term, the complete set of reachable terms. We use an abbreviation of the proof trees computed with this calculus to build appropriate debugging trees for missing answers, whose adequacy for debugging is proved. We apply then this method to Maude specifications, a high-performance system based on rewriting logic, adding many options to build and navigate the tree. Several examples are shown to illustrate the use of the debugger and all of its features.

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 using operations in Maude itself. Even the user interface of the declarative debugger for Maude can be specified in Maude itself. We also describe in detail this metalevel implementation of our tool.

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

Contents

1 Introduction

In this paper we present a declarative debugger of missing answers for *Maude specifications*. Maude [8] 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* [13], 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* [1, 14], 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* [7, 9], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [8, 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 [8, Chap. 22]. The tracing facilities allow us to follow the execution of a specification, that is, the sequence of applications of statements (equations, memberships, or rules) that take place. We can select which operators or statements 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 "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 reached 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 computation that produced the wrong result. That is, when the user detects an unexpected statement application it is difficult to know where the incorrect computation 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 [25]. It has been widely employed in logic [11, 16, 27], functional [19, 18, 20], and multi-paradigm programming [6, 3, 12] 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 [26].

Declarative debugging of wrong answers in Maude specification was already studied in our previous papers [4, 5, 24, 23, 21]. In these works we presented how to debug wrong results due to errors in the statements of the specification. However, in a nondeterministic context such as that of Maude system modules other problems can arise. We show in this paper how to debug *missing answers*, that is, expected results that the specification is not able to compute. This kind of errors can be typically found in Maude by using its breadth-first search, that finds all the reachable terms from an initial one given a pattern, a condition, and a bound in the number of steps. To debug this kind of errors we have extended our calculus to deduce sets of reachable terms given an initial term, a bound in the number of rewrites, and a condition to be fulfilled. Finally, proof trees obtained with this calculus are abbreviated in order to shorten and ease the debugging process while preserving the correctness and completeness of the technique.

The current version of the tool has the following characteristics:

- It allows to debug *wrong answers*, that is, given an initial term we obtain either a wrong term (due to a reduction or a rewrite) or a wrong sort. The debugger can be used to detect errors caused by wrong equations, membership axioms, or rewrite rules. In this kind of debugging the tool indicates the specific rule responsible for the error.

- It also allows to debug *missing answers*. In a nondeterministic context such as Maude system modules, a missing answer is, given an initial term t, a bound in the number of steps n, and a condition c, a term reachable from t in at most n steps that fulfills c and that the system is not able to compute. In this kind of debugging, our tool is able to find errors due to wrong statements, missing rules, errors in the condition imposed to the reachable terms, and membership errors. In this kind of debugging, when there is a wrong statement the debugger points it out; while when there are missing rules the debugger detects the syntax operator whose terms do not have all their transitions defined.

- It supports all kinds of modules: for example, operators can be declared with any combination of axiom attributes; equations can be defined with the `otherwise` attribute; modules can be parameterized; and operators' arguments can be `frozen` (see [8] for the meaning of all these concepts).

- The tool allows to debug specifications where some statements are suspicious and have been labeled (each one with a different label). Thus, the judgments related to the unlabeled statements will be considered correct and will not generate nodes in the debugging tree. 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 judgment; 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 debugging tree.

- When debugging missing answers some operators and sorts can be pointed out as *final*, that is, the terms built with these operators at the top and *constructed* terms (i.e., terms built using only operators with the attribute `ctor`) having one of these sorts[1] cannot be further rewritten. Final sorts can also be identified "on the fly," removing the questions associated to the sort identified as final from the debugging tree.

- Before starting the debugging process, the user can select a module containing only correct statements. By checking the correctness of the judgments with respect to this module the debugger can reduce the number of questions asked to the user.

- It supports different kinds of searches when debugging missing answers: in *zero or more* steps, in *one or more* steps, and search of terms that cannot be further rewritten.

- In case of debugging a wrong rewrite computation, 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 latter tree is partially built so that any node corresponding to a one-step rewrite is expanded only when the navigation process reaches it.

- In the same way, when debugging missing answers the user can choose between two different trees: one whose questions are related to a set of terms obtained with just one rewrite step and another whose questions are associated with a set of terms obtained with several steps.

- 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*,[2] 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.

- The condition imposed to the reachable terms when debugging missing answers can be defined in a very complicated way, but the user generally has in mind the terms that must fulfill it. Thus,

[1] Note that if the least sort of a term is a subsort of one of these sorts, then the term will also be considered final.

[2] More concretely, we implement the Hirunkitti's divide and query algorithm, which is more efficient than the one originally introduced by Shapiro [26].

the debugger allows to prioritize questions related to the fulfillment of the condition from questions involving the statements defining it. Moreover, if this option is used the trees proving whether the condition holds or not are built on demand, so they will be computed only if needed.

- If the current question is too complicated the debugger allows to avoid it with a command `don't know`. However, this command can introduce incompleteness.
- The debugger provides an `undo` command, that allows the user to return to the previous state when an incorrect answer has been provided.

The rest of the paper is structured as follows. Section 2 provides a summary of the main concepts of rewriting logic, and how its specifications are written as Maude modules. Section 3 describes the calculus we have developed to debug missing answers, and how the proof trees obtained with this calculus have been adapted to obtain appropriate debugging trees. Section 4 presents the basic guidelines of the debugger, the possible questions asked to the user, and the complete list of commands, and illustrates how to use the tool by means of several examples, while Section 5 describes the Maude implementation of the tool by means of reflective techniques. 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 [13] 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 [1, 14], 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 *equations* $t = t'$, where t and t' are Σ-terms of the same kind, and *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 $s \in S_k$. Given a Σ-algebra $\mathcal{A}$ and a valuation $\sigma : X \longrightarrow \mathcal{A}$ mapping variables to values in the algebra, the meaning $[\![t]\!]^\sigma_{\mathcal{A}}$ of a term t is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies, under a valuation σ,

- an equation $t = t'$, denoted $\mathcal{A}, \sigma \models t = t'$, if and only if both terms have the same meaning: $[\![t]\!]^\sigma_{\mathcal{A}} = [\![t']\!]^\sigma_{\mathcal{A}}$; we also say that the equation holds in the algebra under the valuation.
- a membership $t : s$, denoted $\mathcal{A}, \sigma \models t : s$, if and only if $[\![t]\!]^\sigma_{\mathcal{A}} \in A_s$.

Satisfaction of Horn clauses is defined in the standard way. Finally, when terms are ground, valuations play no role and thus can be omitted. A membership equational logic specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of ground terms $[t]_E$, and where an equation or membership is satisfied if and only if it can be deduced from E by means of a sound and complete set of deduction rules [1, 14].

Since the membership equational logic specifications that we consider are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness [8], their equations $t = t'$ can be oriented from left to right, $t \rightarrow t'$. Such a statement holds in an algebra, denoted $\mathcal{A}, \sigma \models t \rightarrow t'$,

exactly when $\mathcal{A}, \sigma \models t = t'$, i.e., when $[\![t]\!]^\sigma_{\mathcal{A}} = [\![t']\!]^\sigma_{\mathcal{A}}$. 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 $u \downarrow v$. Also, the notation $t =_E t'$ means that the equation $t = t'$ can be deduced from E, equivalently, that $[t]_E = [t']_E$.

2.2 Maude functional modules

Maude functional modules [8, Chapter 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications and their semantics is given by the corresponding initial 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`). Conditions, in addition to memberships and equations, can also be *matching equations* $t := t'$, whose mathematical meaning is the same as that of an ordinary equation $t = t'$ but that operationally are solved by matching the righthand side t' against the pattern t in the lefthand side, thus instantiating possibly new variables in t.

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 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 q in R has the general conditional form[3]

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

where q is the rule label, 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 [13] (see also [2]).

Models of rewrite theories are called $\mathcal{R}$-*systems* in [13]. Such systems are defined as categories that possess a (Σ, E)-algebra structure, together with a natural transformation for each rule in the

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

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. The rewriting logic deduction rules introduced in [13] are sound and complete with respect to this notion of model. Moreover, they can be used to build initial models. Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, the initial model $\mathcal{T}_{\Sigma/E,R}$ for $\mathcal{R}$ has an underlying (Σ, E)-algebra $\mathcal{T}_{\Sigma/E}$ whose elements are equivalence classes $[t]_E$ of ground Σ-terms modulo E, and there is a transition from $[t]_E$ to $[t']_E$ when there exist terms t_1 and t_2 such that $t =_E t_1 \rightarrow_R^* t_2 =_E t'$, where $t_1 \rightarrow_R^* t_2$ means that the term t_1 can be rewritten into t_2 in zero or more rewrite steps applying rules in R, also denoted $[t]_E \rightarrow_{R/E}^* [t']_E$ when rewriting is considered on equivalence classes [13, 10].

For our purposes in this paper, we are interested in a subclass of models generalizing the initial models described above, that we will call *term models*, where the syntactic structure of terms is kept and associated notions such as variables, substitutions, and term rewriting make sense. Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, a Σ-term model has an underlying (Σ, E')-algebra whose elements are equivalence classes $[t]_{E'}$ of ground Σ-terms modulo some set of equations and memberships E' (which may be different from E), and there is a transition from $[t]_{E'}$ to $[t']_{E'}$ when $[t]_{E'} \rightarrow_{R'/E'}^* [t']_{E'}$, where the set of rules R' may also be different from R, that is, the term model is $\mathcal{T}_{\Sigma/E',R'}$ for some E' and R'. In such term models, the notion of valuation coincides with that of (ground) substitution. A term model $\mathcal{T}_{\Sigma/E',R'}$ satisfies, under a substitution θ,

- an equation $u = v$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u = v$, when $\theta(u) =_{E'} \theta(v)$, or equivalently, when $[\theta(u)]_{E'} = [\theta(v)]_{E'}$;
- a membership $u : s$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u : s$, when the Σ-term $\theta(u)$ has sort s according to the information in the signature Σ and the equations and memberships E' defining the term model $\mathcal{T}_{\Sigma/E',R'}$;
- a rewrite $u \Rightarrow v$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u \Rightarrow v$, when there is a transition in $\mathcal{T}_{\Sigma/E',R'}$ from $[\theta(u)]_{E'}$ to $[\theta(v)]_{E'}$, that is, when $[\theta(u)]_{E'} \rightarrow_{R'/E'}^* [\theta(v)]_{E'}$.

Satisfaction is extended to conditional sentences as usual. A Σ-term model $\mathcal{T}_{\Sigma/E',R'}$ satisfies a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ when $\mathcal{T}_{\Sigma/E',R'}$ satisfies the equations and memberships in E and the rewrite rules in R in this sense. For example, this is obviously the case when $E \subseteq E'$ and $R \subseteq R'$. As before, when the involved terms are ground, the substitution in the satisfaction notation can be omitted.

2.4 Maude system modules

Maude system modules [8, 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`), whose conditions can be equations, matching equations, memberships, and rewrites.

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 [8] 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 [29, 8]. 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 $u' =_E t'$.

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

2.5 An example of system module: maze

Given a maze, we want to obtain all the possible ways that allow us to reach the exit. First, we define the sorts `Pos`, `List`, and `State` that stand for positions in the labyrinth, lists of positions, and the path

traversed so far, respectively:

```
(mod MAZE is
  pr NAT .

  sorts Pos List State .
  subsort Pos < List .

  op [_,_] : Nat Nat -> Pos [ctor] .

  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
```

Terms of sort `State` are lists enclosed by curly brackets, that is, `{_}` is an "encapsulation operator" that ensures that the whole state is used:

```
  op {_} : List -> State [ctor] .
```

We assume an 8×8 labyrinth with the exit in the position `[8,8]`. The predicate `isSol` checks whether a list is a solution:

```
  vars X Y : Nat .
  var P Q : Pos .
  var L : List .

  op isSol : List -> Bool .
  eq [is1] : isSol(L [8,8]) = true .
  eq [is2] : isSol(L) = false [owise] .
```

The next position is computed with the (conditional) rule `expand`, that extends the solution with a new position by rewriting `next(L)` to obtain a new position and then checking whether the list with this new position is correct with `isOk`. Note that the choice of the next position, that could be initially erroneous, produces an implicit backtracking:

```
  crl [expand] : { L } => { L P } if next(L) => P /\ isOk(L P) .
```

The function `next` is defined in a nondeterministic way with the rules:

```
  op next : List -> Pos .

  rl [next1] : next(L [X,Y]) => [X, Y + 1] .
  rl [next2] : next(L [X,Y]) => [sd(X, 1), Y] .
  rl [next3] : next(L [X,Y]) => [X, sd(Y, 1)] .
```

`isOk(L P)` checks that the position `P` is within the limits of the labyrinth, not repeated in `L`, and not part of the wall by using an auxiliary function `contains`:

```
  op isOk : List -> Bool .
  eq isOk(L [X,Y]) = X >= 1 and Y >= 1 and X <= 8 and Y <= 8
                     and not(contains(L, [X,Y]))
                     and not(contains(wall, [X,Y])) .

  op contains : List Pos -> Bool .
  eq [c1] : contains(nil, P) = false .
  eq [c2] : contains(Q L, P) = if P == Q then true else contains(L, P) fi .
```

Finally, we define the `wall` of the labyrinth as a list of positions:

```
  op wall : -> List .
  eq wall =       [2,1]
                  [2,2]
                  [2,3]       [4,3] [5,3] [6,3] [7,3] [8,3]

            [1,5] [2,5] [3,5] [4,5]             [7,5]
                                                [7,6]
                                                [7,7]
                                                [7,8]       .
endm)
```

Now we can use the module to search the labyrinth's exit from the position `[1,1]` with the Maude command `search`:

```
Maude> (search {[1,1]} =>* {L:List} s.t. isSol(L:List) .)
search in MAZE :{[1,1]} =>* {L:List}.

No solution.
```

but it cannot find any path to escape. We will see in Section 4 how to debug this specification.

3 Debugging trees for missing answers

We describe in this section a calculus that allows to infer, given a term and some constraints, the *complete* set of reachable terms from this term that fulfill the requirements. The proof trees built with this calculus have nodes that justify why the terms are included in the corresponding sets (positive information) but also nodes that justify why there are no more terms (negative information). These latter nodes are then used in the debugging trees to localize as much as possible the reasons responsible of missing answers.

The calculus is introduced as an extension of the more standard calculus in [21] that allowed to deduce judgments corresponding to oriented equations $t \to t'$, memberships $t : s$, and rewrites $t \Rightarrow t'$. Once this extended calculus is presented, we show how to use it to define appropriate debugging trees for missing answers.

3.1 A calculus for missing answers

From now on, we assume a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ satisfying the Maude executability requirements; in particular E is confluent and terminating, maybe modulo some equational attributes such as associativity and commutativity, while R is coherent with respect to E. Then the equations corresponding to the equational attributes form the set A and the equations in $E - A$ can be oriented from left to right.

We introduce the inference rules used to obtain the set of reachable terms given an initial one, a pattern [8], a condition, and a bound in the number of rewrites. First, the pattern P and the condition $\mathcal{C}$ (that can use variables bound by the pattern) are put together by creating the condition $\mathcal{C}' \equiv P := \circledast \wedge \mathcal{C}$, where $\circledast$ is a "hole" that will be filled by concrete ground terms to check if they fulfill both the pattern and the condition. Throughout this paper we only consider a special kind of conditions and substitutions that operate over them, called *admissible*, that we define as follows:

Definition 1 *A condition $\mathcal{C} \equiv C_1 \wedge \cdots \wedge C_n$ is* admissible *if, for $1 \leq i \leq n$,*

- *C_i is an equation $u_i = u_i'$ or a membership $u_i : s$ and*

$$vars(C_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j), \text{ or}$$

- *C_i is a matching condition $u_i := u_i'$, u_i is a pattern and*

$$vars(u_i') \subseteq \bigcup_{j=1}^{i-1} vars(C_j), \text{ or}$$

- *C_i is a rewrite condition $u_i \Rightarrow u_i'$, u_i' is a pattern and*

$$vars(u_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j).$$

Definition 2 *A condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ is* admissible *if $P := t \wedge C_1 \wedge \cdots \wedge C_n$ is admissible for t any ground term.*

Definition 3 *Given an atomic condition C, we say that a substitution θ is* admissible for C *if*

- *C is an equation $u = u'$ or a membership $u : s$ and $vars(C) \subseteq dom(\theta)$, or*

- *C is a matching condition $u := u'$ and $vars(u') \subseteq vars(\theta)$, or*
- *C is a rewrite condition $u \Rightarrow u'$ and $vars(u) \subseteq vars(\theta)$.*

The calculus presented in this section (in Figures 1, 2, 3, and 8) will be used to deduce the following judgments, that we introduce together with their meaning for a Σ-term model $\mathcal{T}_{\Sigma/E',R'}$ defined by equations and memberships E' and by rules R':

- Given an admissible substitution θ for an atomic condition C, $\mathcal{T}_{\Sigma/E',R'} \models [C,\theta] \rightsquigarrow \Theta$ when
$$\Theta = \{\theta' \mid \mathcal{T}_{\Sigma/E',R'}, \theta' \models C \text{ and } \theta' \restriction_{dom(\theta)} = \theta\},$$
that is, Θ is the set of substitutions that fulfill the atomic condition C and extend θ.
- Given a set of admissible substitutions Θ for an atomic condition C, $\mathcal{T}_{\Sigma/E',R'} \models \langle C,\Theta\rangle \rightsquigarrow \Theta'$ when
$$\Theta' = \{\theta' \mid \mathcal{T}_{\Sigma/E',R'}, \theta' \models C \text{ and } \theta' \restriction_{dom(\theta)} = \theta \text{ for some } \theta \in \Theta\},$$
that is, Θ' is the set of substitutions that fulfill the condition C and extend any of the admissible substitutions in Θ.
- Given an admissible condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$, $\mathcal{T}_{\Sigma/E',R'} \models \mathit{fulfilled}(\mathcal{C},t)$ when there exists a substitution θ such that $\mathcal{T}_{\Sigma/E',R'}, \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ holds when $\circledast$ is substituted by t.
- Given an admissible condition $\mathcal{C}$ as before, $\mathcal{T}_{\Sigma/E',R'} \models \mathit{fails}(\mathcal{C},t)$ when there exists *no* substitution θ such that $\mathcal{T}_{\Sigma/E',R'}, \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ does not hold when $\circledast$ is substituted by t.
- $\mathcal{T}_{\Sigma/E',R'} \models t :_{ls} s$ when $\mathcal{T}_{\Sigma/E',R'} \models t : s$ and moreover s is the least sort with this property (with respect to the ordering on sorts obtained from the signature Σ and the equations and memberships E' defining the Σ-term model $\mathcal{T}_{\Sigma/E',R'}$).
- $\mathcal{T}_{\Sigma/E',R'} \models t \Rightarrow^{top} S$ when $S = \{t' \mid t \rightarrow_{R'}^{top} t'\}$, that is, the set S is formed by all the reachable terms from t by exactly one rewrite *at the top* with the rules R' defining $\mathcal{T}_{\Sigma/E',R'}$. Moreover, equality in S is modulo E', i.e., we are implicitly working with equivalence classes of ground terms modulo E'.
- $\mathcal{T}_{\Sigma/E',R'} \models t \Rightarrow^{q} S$ when $S = \{t' \mid t \rightarrow_{\{q\}}^{top} t'\}$, that is, the set S is the complete set of reachable terms (modulo E') obtained from t with one application of the rule $q \in R'$ at the top.
- $\mathcal{T}_{\Sigma/E',R'} \models t \Rightarrow_1 S$ when $S = \{t' \mid t \rightarrow_{R'}^{1} t'\}$, that is, the set S is constituted by all the reachable terms (modulo E') from t in exactly one step, where the rewrite step can take place anywhere in t.
- $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow_n^{\mathcal{C}} S$ when $S = \{t' \mid t \rightarrow_{R'}^{\leq n} t' \text{ and } \mathcal{T}_{\Sigma/E',R'} \models \mathit{fulfilled}(\mathcal{C},t')\}$, that is, S is the set of all the terms (modulo E') that satisfy the admissible condition $\mathcal{C}$ and are reachable from t in at most n steps.
- $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow +_n^{\mathcal{C}} S$ as before, but with reachability from t in at least one step and in at most n steps.
- $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow !_n^{\mathcal{C}} S$ when $S = \{t' \mid t \rightarrow_{R'}^{\leq n} t' \text{ and } \mathcal{T}_{\Sigma/E',R'} \models \mathit{fulfilled}(\mathcal{C},t') \text{ and } t' \not\rightarrow_{R'}\}$, that is, now the terms (modulo E') in S are *final*, meaning that they cannot be further rewritten.

We first introduce in Figure 1 the inference rules defining the relations $[C,\theta] \rightsquigarrow \Theta$ and $\langle C,\Theta\rangle \rightsquigarrow \Theta'$, and also $t :_{ls} s$, that will be used later to describe when a condition holds:

- Rule $\mathsf{EqC_1}$ indicates that if the two terms of an equality can be reduced to the same term then the current substitution is returned.
- In rule $\mathsf{EqC_2}$ we use the auxiliary function $nf(t)$, that stands for the normal form of t. If the normal forms of two terms are different (modulo axioms, indicated by $\not\equiv_A$), the equality cannot be fulfilled and the empty set of substitutions is returned.
- When we have a matching condition the substitution is extended with all the new substitutions θ' that satisfy the matching (modulo axioms), thus returning the empty set if the terms do not match, which is reflected by rule PatC.

$$\frac{\theta(t_1) \downarrow \theta(t_2)}{[t_1 = t_2, \theta] \rightsquigarrow \{\theta\}}\ \mathsf{EqC_1}$$

$$\frac{\theta(t_1) \to nf(\theta(t_1)) \quad \theta(t_2) \to nf(\theta(t_2))}{[t_1 = t_2, \theta] \rightsquigarrow \emptyset}\ \mathsf{EqC_2} \quad \text{if } nf(\theta(t_1)) \not\equiv_A nf(\theta(t_2))$$

$$\frac{\theta(t_2) \to t'}{[t_1 := t_2, \theta] \rightsquigarrow \{\theta'\theta \mid \theta'(\theta(t_1)) \equiv_A t'\}}\ \mathsf{PatC}$$

$$\frac{\theta(t) : s}{[t : s, \theta] \rightsquigarrow \{\theta\}}\ \mathsf{MbC_1}$$

$$\frac{\theta(t) \to t' \quad t' :_{ls} s'}{[t : s, \theta] \rightsquigarrow \emptyset}\ \mathsf{MbC_2} \quad \text{if } s' \not\leq s$$

$$\frac{t : s}{t :_{ls} s}\ \mathsf{Ls} \quad \text{if } s \leq ls(t)$$

$$\frac{\theta(t_1) \rightsquigarrow_{n+1}^{t_2 :=^{\circledast}} S}{[t_1 \Rightarrow t_2, \theta] \rightsquigarrow \{\theta'\theta \mid \theta'(\theta(t_2)) \in S\}}\ \mathsf{RlC} \text{ if } n = min(x \in \mathbb{N} : \forall i \geq 0\ (\theta(t_1) \rightsquigarrow_{x+i}^{t_2 :=^{\circledast}} S))$$

$$\frac{[C, \theta_1] \rightsquigarrow \Theta_1 \quad \cdots \quad [C, \theta_m] \rightsquigarrow \Theta_m}{\langle C, \{\theta_1, \ldots, \theta_m\}\rangle \rightsquigarrow \bigcup_{i=1}^{m} \Theta_i}\ \mathsf{SubsCond}$$

Figure 1: Calculus for substitutions

- Analogously to $\mathsf{EqC_1}$, rule $\mathsf{MbC_1}$ returns the current substitution if a membership condition holds.
- Rule $\mathsf{MbC_2}$ indicates that if the least sort of a certain term is not less or equal than the required sort, then the membership does not hold and the empty set of substitutions is returned.
- Rule Ls uses the inference rules for memberships already described in [21] to obtain the sort of a term. Then, the rule checks that this sort is less or equal than the least sort of the term, computed with $ls(t)$.
- Rewrite conditions are handled by rule RlC. This rule extends the set of substitutions by computing all the reachable terms that satisfy the pattern (using the relation $t \rightsquigarrow_n^{\mathcal{C}} S$ explained below) and then using these terms to obtain the new substitutions. Note that, as said above, the condition in the premise has a hole that will be filled with concrete terms to check if they match the pattern. Instead of using as bound of the search the smallest number that allows to find all the reachable terms from the current one, n, we use $n + 1$ in order to include in the tree the inferences proving that the terms reached in n steps are final.
- Finally, rule SubsCond computes the extensions of a set of admissible substitutions $\{\theta_1, \ldots, \theta_n\}$ by using the rules above with each of them.

Once these rules have been introduced, we can use them to define the rules defining the relation $t \rightsquigarrow_n^{\mathcal{C}} S$. First, we define in Figure 2 the rules related to $n = 0$ steps:

- Rule $\mathsf{Rf_1}$ indicates that when only zero steps are allowed and the current term fulfills the condition, the set of reachable terms consists only of this term.
- Rule Fulfill checks whether a term satisfies a condition. The premises of this rule check that all the atomic conditions hold, taking into account that it starts with a matching condition with a hole that must be filled with the current term and thus proved with the premise $\theta(P) \downarrow t$ (the rest of matching conditions are included in the equality conditions). Note that when the condition is satisfied we do not need to check *all* the substitutions, but only to assure that there exists *one* substitution that makes the condition true.
- Rule $\mathsf{Rf_2}$ complements $\mathsf{Rf_1}$ by defining the empty set as result when the condition does not hold.

$$\frac{\mathit{fulfilled}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \{t\}}\ \mathsf{Rf_1}$$

$$\frac{\theta(P) \downarrow t \ \ \{\theta(u_i) \downarrow \theta(u_i')\}_{i=1}^{n} \ \{\theta(v_j) : s_j\}_{j=1}^{m} \ \{\theta(w_k) \Rightarrow \theta(w_k')\}_{k=1}^{l}}{\mathit{fulfilled}(\mathcal{C}, t)}\ \mathsf{Fulfill}$$
$$\text{if } \mathcal{C} \equiv P := \circledast \wedge \textstyle\bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w_k'$$

$$\frac{\mathit{fails}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \emptyset}\ \mathsf{Rf_2}$$

$$\frac{[P := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0\rangle \rightsquigarrow \Theta_1 \ \cdots \ \langle C_k, \Theta_{k-1}\rangle \rightsquigarrow \emptyset}{\mathit{fails}(\mathcal{C}, t)}\ \mathsf{Fail} \ \text{ if } \mathcal{C} \equiv P := \circledast \wedge C_1 \ \wedge \ \ldots \ \wedge \ C_k$$

Figure 2: Calculus for solutions

$$\frac{\mathit{fulfilled}(\mathcal{C}, t) \ \ t \Rightarrow_1 \{t_1, \ldots, t_k\} \ \ t_1 \rightsquigarrow_n^{\mathcal{C}} S_1 \ \ldots \ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i \ \cup \ \{t\}}\ \mathsf{Tr_1}$$

$$\frac{\mathit{fails}(\mathcal{C}, t) \ \ t \Rightarrow_1 \{t_1, \ldots, t_k\} \ \ t_1 \rightsquigarrow_n^{\mathcal{C}} S_1 \ \ldots \ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i}\ \mathsf{Tr_2}$$

$$\frac{f(t_1, \ldots, t_m) \Rightarrow^{top} S_t \quad t_1 \Rightarrow_1 S_1 \quad \cdots \quad t_m \Rightarrow_1 S_m}{f(t_1, \ldots, t_m) \Rightarrow_1 S_t \ \cup \ \bigcup_{i=1}^{m} \{f(t_1, \ldots, u_i, \ldots, t_m) \mid u_i \in S_i\}}\ \mathsf{Rep}$$

$$\frac{t \Rightarrow^{q_1} S_{q_1} \quad \cdots \quad t \Rightarrow^{q_l} S_{q_l}}{t \Rightarrow^{top} \bigcup_{i=1}^{l} S_{q_i}}\ \mathsf{Top} \ \text{ if } \{q_1, \ldots, q_l\} = \{q \in R \mid q \ll_K^{top} t\}$$

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0\rangle \rightsquigarrow \Theta_1 \ \cdots \ \langle C_k, \Theta_{k-1}\rangle \rightsquigarrow \Theta_k}{t \Rightarrow^{q} \bigcup_{\forall\, \theta \in \Theta_k} \{\theta(r)\}}\ \mathsf{Rl} \ \text{ if } q : l \Rightarrow r \Leftarrow C_1 \ \wedge \ \ldots \ \wedge \ C_k \ \in R$$

$$\frac{t \rightarrow t_1 \quad t_1 \rightsquigarrow_n^{\mathcal{C}} \{t_2\} \cup S \quad t_2 \rightarrow t'}{t \rightsquigarrow_n^{\mathcal{C}} \{t'\} \cup S}\ \mathsf{Red_1}$$

Figure 3: Calculus for missing answers

- To check that a term does not satisfy a condition it is not enough to check that there exists a substitution that makes it to fail; we must make sure that there is no substitution that makes it true. We use the rules shown in Figure 1 to assure that the set of substitutions that satisfy the condition is empty. Note that the first set of substitutions is obtained from the first matching condition, filling the hole with the current term.

Now, we introduce in Figure 3 the rules defining the relation $t \rightsquigarrow_n^{\mathcal{C}} S$ when the bound n is greater than 0, which can be understood as searches in *zero or more* steps.

- Rules $\mathsf{Tr_1}$ and $\mathsf{Tr_2}$ show the behavior of the calculus when at least one step can be used. First, we check whether the condition holds (rule $\mathsf{Tr_1}$) or not (rule $\mathsf{Tr_2}$) for the current term, in order to introduce it in the result set. Then, we obtain all the terms reachable in one step with the relation $\Rightarrow_1$, and finally we compute the reachable solutions from these terms constrained by the same condition and the bound decreased in one step. The union of the sets obtained in this way and the initial term, if needed, corresponds to the final result set.

- Rule Rep shows how the set for one step is computed. The result set is the union of the terms obtained by applying each rule *at the top* (calculated with $t \Rightarrow^{top} S$) and the terms obtained by rewriting one step the arguments of the term. This rule can be straightforwardly adapted to

the more general case in which the operator f has some *frozen* arguments (i.e., that cannot be rewritten); the implementation of the debugger makes use of this more general rule.

- How to obtain the terms by rewriting at the top is explained by rule Top, that specifies that the result set is the union of the sets obtained with all the possible applications of each rule in the program. We have restricted these rules to those whose lefthand side, with the variables considered at the kind level, matches the term, represented with notation $q \ll_K^{top} t$, where q is the label of the rule and t the current term.

- Rule Rl uses the rules in Figure 1 to compute the set of terms obtained with the application of a single rule. First, the set of substitutions obtained from matching with the lefthand side of the rule is computed, and then it is used to find the set of substitutions that satisfy the condition. This final set is used to instantiate the righthand side of the rule to obtain the set of reachable terms.

- Finally, rule Red_1 allows to reduce the reachable terms in order to obtain their normal forms.

Now we prove that this calculus is correct in the sense that the derived judgments with respect to the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ coincide with the ones satisfied by the corresponding initial model $\mathcal{T}_{\Sigma/E,R}$, i.e., for any judgment φ, φ is derivable in the calculus if and only if $\mathcal{T}_{\Sigma/E,R} \models \varphi$. This is well known for the judgments corresponding to equations $t = t'$, memberships $t : s$, and rewrites $t \Rightarrow t'$ [14, 13].

Theorem 1 *The calculus of Figures 1, 2, and 3 is correct.*

Proof. By induction over proof trees; we distinguish cases over the different kinds of judgments:

- $[C, \theta] \rightsquigarrow \Theta$ is correct. We distinguish over the different kinds of conditions:
  - $C \equiv t_1 = t_2$. Since we work with admissible conditions, we know that $\theta(t_1)$ and $\theta(t_2)$ are ground, and thus the only possible substitution that can be included in Θ is θ. If the condition is fulfilled only rule EqC_1 can be used, and $\{\theta\}$ is returned, which is correct. Otherwise, only EqC_2 can be used, returning now the empty set which is again correct.
  - $C \equiv t_1 := t_2$. We assume that $\theta(t_2) \rightarrow t'$ is correct and thus we obtain by definition the complete set of substitutions that fulfill the matching.
  - $C \equiv t : s$. Like in equational conditions, $\theta(t)$ is ground and the resulting set can only contain θ. If the condition is fulfilled only MbC_1 can be applied and the set obtained is correct. Analogously, if the condition does not hold only MbC_2 can be used and the correct result is the empty set.
  - $C \equiv t_1 \rightarrow t_2$. We assume that the set of reachable terms from $\theta(t_1)$ that match $\theta(t_2)$ is correct, and thus by definition the set computed by rule RlC, the only one applicable here, is correct.

- $\langle C, \Theta \rangle \rightsquigarrow \Theta'$ is correct. The only rule that deals with this judgment is SubsCond. Assuming the premises correct, the conclusion is also correct.

- *fulfilled*$(\mathcal{C}, t)$. This judgment is correct when there exist a substitution that makes $\mathcal{C}$ with the hole filled by t hold. Rule Fulfill, the only one that can be used to prove this predicate, states this fact and thus the judgment is correct.

- *fails*$(\mathcal{C}, t)$. This judgment is correct when $\mathcal{C}$ with t filling its hole cannot be satisfied. Since the only rule that can be used for this predicate is Fail and the premise indicates that the set of substitutions that fulfill the condition is empty, the judgment is correct.

- $t \Rightarrow^q S$. This judgment is only computed with rule Rl. By hypothesis, all the substitutions that fulfill the conditions and make t match the lefthand side of the rule are in Θ_k, thus by definition the union of the application of all the substitutions in Θ_k to the lefthand side of the rule generate the set we are looking for and the judgment is correct.

- $t \Rightarrow^{top} S$. This judgment is only computed with rule Top. First, we notice that the rules in $\{q_1, \ldots, q_l\}$ are the only ones that can be applied to t (it does not match the lefthand side of the rest of rules) and thus the correctness is not affected by this selection. We know by hypothesis that each S_i, the set of reachable terms obtained from t with the rule q_i, is correct and hence the union of all these sets is by definition the set of reachable terms by rewriting at the top and the judgment is correct.

- $t \Rightarrow_1 S$. This judgment is only computed with rule Rep. By hypothesis, we know that S_t contains the set of reachable terms obtained by rewriting t at the top, while S_i contains the reachable terms in one step from t_i. Since the set of reachable terms in one step from t is the union of the terms obtained by one rewriting at the top and the set created by substituting each subterm by all the reachable terms in one step from it, the judgment is correct.

- $t \rightsquigarrow_n^{\mathcal{C}} S$. For this judgment, rule $\mathsf{Red_1}$ can always be applied. Since we work with a coherent theory, the set of reachable terms from both t and t_1 are the same, while t_2 and t' are in the same equivalence class and thus are equal modulo E.

  When $n = 0$, rules $\mathsf{Rf_1}$ or $\mathsf{Rf_2}$ are used and the result is straightforward.

  If $n > 1$ and the term fulfills the condition, rule $\mathsf{Tr_1}$ is applied. Since the condition holds, the result set must contain t that is added in the conclusion of the rule. Moreover, the terms $t_1, \ldots, t_k$ are the reachable terms from t in exactly one step, while S_i is the set of reachable terms from t_i in zero or more steps, that is, the union of the S_i is the set of reachable terms in at least one step and at most n, and thus the union of this set with the singleton set $\{t\}$ creates a correct set for this judgment. Analogously, when $n > 1$ and the condition does not hold, rule $\mathsf{Tr_2}$ is applied.

□

Once these rules are defined, we can build the tree corresponding to the search result shown in Section 2.5 for the maze example. We recall that we have defined a system to search a path out of a labyrinth but, given a concrete labyrinth with an exit, the program is unable to find it:

```
Maude> (search {[1,1]} =>* {L:List} s.t. isSol(L:List) .)
search in MAZE :{[1,1]} =>* {L:List}.

No solution.
```

First of all, we have to use a concrete bound to build the tree. It must allow to compute all the reachable terms, and in this case the least of these values is 4. We have (partially) depicted the tree in Figure 4, where we have abbreviated the equational condition `{L:List}` $:= \circledast \wedge$ `isSol(L:List) = true` by $\mathcal{C}$. The leftmost tree justifies that this condition does not hold for the initial term (this is the reason why $\mathsf{Tr_2}$ has been used instead of $\mathsf{Tr_1}$) and thus it is not a solution. Note that first the matching substitutions with the pattern are obtained (`L` $\mapsto$ `[1,1]` in this case), and then these substitutions are used to instantiate the rest of the condition, that for this term does not hold. The next tree shows the set of reachable terms in one step (the tree $*_1$, explained below, computes the terms obtained by rewrites at the top, while the tree on its right shows that the subterms cannot be further rewritten) and finally the rightmost tree, that has a similar structure to this one and will not be studied in depth, continues the search with the bound decreased in one step:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{\{[1,1]\}} \to \texttt{\{[1,1]\}}}\,\mathsf{Rf}}{[\texttt{\{L:List\}} := \texttt{\{[1,1]\}},\ \emptyset] \rightsquigarrow \texttt{L} \mapsto \texttt{[1,1]}\}}\,\mathsf{PatC} \quad \dfrac{\dfrac{}{\texttt{isSol([1,1])} \to \texttt{false}}\,\mathsf{Rep}_{\to}}{[\texttt{isSol(L)}, \texttt{L} \mapsto \texttt{[1,1]}] \rightsquigarrow \emptyset}\,\mathsf{EqC_2}}{\langle \texttt{isSol(L)}, \{\texttt{L} \mapsto \texttt{[1,1]}\}\rangle \rightsquigarrow \emptyset}\,\mathsf{SubsCond}}{\mathit{fails}(\mathcal{C}, \texttt{\{[1,1]\}})}\,\mathsf{Fail} \quad \dfrac{*_1 \quad \dfrac{\dfrac{}{\texttt{[1,1]} \Rightarrow^{top} \emptyset}\,\mathsf{Top} \quad \dfrac{\dfrac{}{1 \Rightarrow^{top} \emptyset}\,\mathsf{Top}}{1 \Rightarrow_1 \emptyset}\,\mathsf{Rep}}{\texttt{[1,1]} \Rightarrow_1 \emptyset}\,\mathsf{Rep}}{\texttt{\{[1,1]\}} \Rightarrow_1 \texttt{\{[1,1][1,2]\}}}\,\mathsf{Rep} \quad \dfrac{*_2}{\texttt{\{[1,1][1,2]\}} \rightsquigarrow_3^{\mathcal{C}} \emptyset}\,\mathsf{Tr_2}}{\texttt{\{[1,1]\}} \rightsquigarrow_4^{\mathcal{C}} \emptyset}\,\mathsf{Tr_2}$$

Figure 4: Tree for the maze example

The tree labeled with $*_1$ is sketched in Figure 5. In this tree the application of all the rules whose lefthand side matches the current term (`{[1,1]}`) are tried. In this case only the rule `expand` (abbreviated by `e`) can be used, and it generates a list with the new position `[1,2]`; the trees $*_3$ and $*_4$ are used to justify that the condition holds:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{\{[1,1]\}} \to \texttt{\{[1,1]\}}}\,\mathsf{Rf}}{[\texttt{\{L\}} := \texttt{\{[1,1]\}},\ \emptyset] \rightsquigarrow \{\texttt{L} \mapsto \texttt{[1,1]}\}}\,\mathsf{PatC} \quad *_3 \quad *_4}{\texttt{\{[1,1]\}} \Rightarrow^{\mathsf{e}} \texttt{\{[1,1][1,2]\}}}\,\mathsf{RI}}{\texttt{\{[1,1]\}} \Rightarrow^{top} \texttt{\{[1,1][1,2]\}}}\,\mathsf{Top}$$

Figure 5: Tree $*_1$ for the applications at the top

The tree $*_3$, shown in Figure 6, is in charge of inferring the set of substitutions obtained when checking the first condition of the rule `expand`, namely `next(L) => P`. The condition is instantiated with the substitution obtained from matching the term with the lefthand side of the rule (in this case $\mathtt{L} \mapsto \mathtt{[1,1]}$) and, since it is a rewrite condition, the set of reachable terms—computed with $*_5$, a list of premises similar to the ones shown for the root of the tree depicted in Figure 4—is used to extend this substitution, obtaining a set with three different substitutions:

$$\dfrac{\dfrac{\dfrac{*_5}{\mathtt{next([1,1])} \rightsquigarrow_1^{\mathtt{P}:=\circledast} \{\mathtt{[1,2]}, \mathtt{[1,0]}, \mathtt{[0,1]}\}}\,\mathsf{Tr_2}}{[\mathtt{next(L)} \Rightarrow \mathtt{P}, \mathtt{L} \mapsto \mathtt{[1,1]}] \rightsquigarrow \{\mathtt{L} \mapsto \mathtt{[1,1]P} \mapsto \mathtt{[1,2]}, \mathtt{L} \mapsto \mathtt{[1,1]P} \mapsto \mathtt{[1,0]}, \mathtt{L} \mapsto \mathtt{[1,1]P} \mapsto \mathtt{[0,1]}\}}\,\mathsf{RIC}}{\langle \mathtt{next(L)} \Rightarrow \mathtt{P}, \{\mathtt{L} \mapsto \mathtt{[1,1]}\}\rangle \rightsquigarrow \{\mathtt{L} \mapsto \mathtt{[1,1]P} \mapsto \mathtt{[1,2]}, \mathtt{L} \mapsto \mathtt{[1,1]P} \mapsto \mathtt{[1,0]}, \mathtt{L} \mapsto \mathtt{[1,1]P} \mapsto \mathtt{[0,1]}\}}\,\mathsf{SubsCond}$$

Figure 6: Tree $*_3$ for the first condition of `expand`

Now we use the substitutions obtained above to check the next condition of the rule `expand`, namely, `isOk(L P)`. The tree used to prove it, labeled with $*_4$, is presented in Figure 7, where the triangles represent proof trees in the calculus for wrong answers presented in [21] and the substitutions have been abbreviated as $\theta_1 \equiv \mathtt{L} \mapsto \mathtt{[1,1]P} \mapsto \mathtt{[1,2]}$, $\theta_2 \equiv \mathtt{L} \mapsto \mathtt{[1,1]P} \mapsto \mathtt{[1,0]}$, and $\theta_3 \equiv \mathtt{L} \mapsto \mathtt{[1,1]P} \mapsto \mathtt{[0,1]}$. The tree indicates that the first substitution is valid, but the other two substitutions are not, thus the set of substitutions is reduced and only the first one satisfies the two conditions. This substitution was used to create the reachable terms shown in the root of the tree in Figure 5:

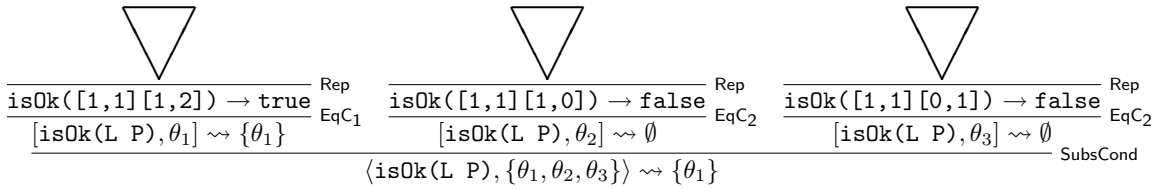

Figure 7: Tree $*_4$ for the second condition of `expand`

There are two additional kinds of search allowed in our framework: searches for final terms and searches in *one or more* steps. Figure 8 presents the inference rules for these cases:

- Rules $\mathsf{Rf_3}$ and $\mathsf{Rf_4}$ are applied when the set of reachable terms in one step is empty (that is, when the term is final). They check whether the term, in addition to being final, fulfills the condition in order to insert it in the result set when appropriate.
- Rule $\mathsf{Rf_5}$ specifies that, if the term is not final but no more steps are allowed, then the set of reachable final terms is empty.
- Rule $\mathsf{Tr_3}$ shows the transitivity for this kind of search. Since the term is not final, it is not necessary to check whether it fulfills the condition.
- Rule $\mathsf{Red_2}$ allows to reduce the reachable final terms in order to obtain their normal forms.
- If only zero steps are available in searches where at least one is required, the empty set is obtained, which is indicated in rule $\mathsf{Rf_6}$.
- When at least one step can be used we apply rule $\mathsf{Tr_4}$, that indicates that one step is used, and then the relation for zero or more steps is used with the results in order to obtain the final solutions.

The correctness of these inference rules with respect to the initial model $\mathcal{T}_{\Sigma/E,R}$ is proved in the following theorem:

Theorem 2 *The calculus of Figure 8 is correct.*

Proof.

$$\frac{\mathit{fulfilled}(\mathcal{C}, t) \quad t \Rightarrow_1 \emptyset}{t \rightsquigarrow !_n^{\mathcal{C}} \{t\}}\ \mathsf{Rf_3}$$

$$\frac{\mathit{fails}(\mathcal{C}, t) \quad t \Rightarrow_1 \emptyset}{t \rightsquigarrow !_n^{\mathcal{C}} \emptyset}\ \mathsf{Rf_4}$$

$$\frac{t \Rightarrow_1 S}{t \rightsquigarrow !_0^{\mathcal{C}} \emptyset}\ \mathsf{Rf_5} \quad S \neq \emptyset$$

$$\frac{t \Rightarrow_1 \{t_1, \ldots, t_k\} \quad t_1 \rightsquigarrow !_n^{\mathcal{C}} S_1 \quad \ldots \quad t_k \rightsquigarrow !_n^{\mathcal{C}} S_k}{t \rightsquigarrow !_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i}\ \mathsf{Tr_3} \ \text{ if } k > 0$$

$$\frac{t \to t_1 \quad t_1 \rightsquigarrow !_n^{\mathcal{C}} \{t_2\} \cup S \quad t_2 \to t'}{t \rightsquigarrow !_n^{\mathcal{C}} \{t'\} \cup S}\ \mathsf{Red_2}$$

$$\frac{}{t \rightsquigarrow +_0^{\mathcal{C}} \emptyset}\ \mathsf{Rf_6}$$

$$\frac{t \to t' \quad t' \Rightarrow_1 \{t_1, \ldots, t_k\} \quad t_1 \rightsquigarrow_n^{\mathcal{C}} S_1 \quad \ldots \quad t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow +_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i}\ \mathsf{Tr_4}$$

Figure 8: Calculus for final and one or more steps searches

- $t \rightsquigarrow !_n^{\mathcal{C}} S$. For this judgment, rule $\mathsf{Red_2}$ can always be applied. Since we work with a coherent theory, the set of reachable terms from both t and t_1 are the same, while t_2 and t' are equal modulo E.

  When $n = 0$, rules $\mathsf{Rf_3}$, $\mathsf{Rf_4}$, and $\mathsf{Rf_5}$ can be used. If t is not final only $\mathsf{Rf_5}$ can be used and, since no more steps are allowed, the empty set of results is returned, which is correct by definition. If t is final we have to check whether the term fulfills the condition; if the condition holds only $\mathsf{Rf_3}$ can be used and hence the singleton set consisting of the term is returned, while if the condition fails $\mathsf{Rf_4}$ is applied and the empty set is returned. In both cases the result is correct by definition.

  When $n > 0$ rules $\mathsf{Rf_3}$, $\mathsf{Rf_4}$, and $\mathsf{Tr_3}$ can be used. If the term is final, $\mathsf{Rf_3}$ and $\mathsf{Rf_4}$ are applied and the result holds as in the previous case. If the term is not final, then $\mathsf{Tr_3}$ is applied; the terms $t_1, \ldots, t_k$ are the reachable terms from t in exactly one step, while S_i is the set of reachable terms from t_i in zero or more steps, that is, the union of the S_i is the set of reachable terms in at least one step and at most n and, since the current term cannot be a solution because it is not final, the judgment is correct.

- $t \rightsquigarrow +_n^{\mathcal{C}} S$. We distinguish cases over n:

  When $n = 0$, only rule $\mathsf{Rf_6}$ can be applied; since the judgment requires at least one step, the set of reachable terms is empty by definition.

  When $n > 0$, rule $\mathsf{Tr_4}$ is applied. Since $t \to t'$ and the specification is coherent, we know that the set of reachable terms from both t and t' is the same; the terms $t_1, \ldots, t_k$ are the reachable terms from t in exactly one step, while S_i is the set of reachable terms from t_i in zero or more steps (note that the judgments in the premises are different from the one in the conclusion), that is, the union of the S_i is the set of reachable terms in at least one step and at most n and hence the judgment is correct.

□

Following the approach shown in [21], we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. This intended interpretation is a Σ-term model corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. Therefore the user *expects* that $\mathcal{I} \models \varphi$ for any judgment φ deduced with respect to the rewrite theory $\mathcal{R}$. As any Σ-term model, $\mathcal{I}$ must satisfy the following soundness propositions:

Proposition 1 *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, C an atomic condition, θ an admissible substitution, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If $[C, \theta] \rightsquigarrow \Theta$ or $\langle C, \Theta\rangle \rightsquigarrow \Theta'$ can be deduced using the rules from Figure 1 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models [C, \theta] \rightsquigarrow \Theta$ or $\mathcal{T}_{\Sigma/E',R'} \models \langle C, \Theta\rangle \rightsquigarrow \Theta'$, respectively.*

Proof. We apply the definition of satisfaction for each rule:

EqC$_1$ From the premises we deduce that $[\theta(t_1)]_{E'} = [\theta(t_2)]_{E'}$, that is, the condition is satisfied with the current substitution θ. Since θ already binds all the variables in the condition, it cannot be extended and θ itself is the result.

EqC$_2$ From the premises we deduce that $[\theta(t_1)]_{E'} \neq [\theta(t_2)]_{E'}$, thus the condition fails and there is no substitution that could satisfy it.

PatC We know that $[\theta(t_2)]_{E'} = [t']_{E'}$ and that matching conditions can have variables in its lefthand side that are not bound in θ. Thus, the substitution is extended with all the substitutions θ' that match t' and, since t' is equal (modulo E') to $\theta(t_2)$ by hypothesis, these are all the substitutions that satisfy the condition.

MbC$_1$ We know that the condition is fulfilled and θ binds all the variables, therefore it cannot be extended and the single substitution that verifies the condition is θ itself.

MbC$_2$ Similarly to EqC$_2$, we know by hypothesis that the condition does not hold, thus there is no substitution able to satisfy it and the empty set of substitutions is computed.

RlC In this case θ can be extended because rewrite conditions can contain new variables in their righthand side. We assume that S contains all the terms reachable from $\theta(t_1)$ that match the pattern t_2, and then use it to extend θ with all the substitutions θ' that bind the new variables in t_2 to match the terms in S, obtaining by definition all the substitutions that verify the condition.

SubsCond We assume that, for each θ_i, $1 \leq i \leq n$, we obtain the set of substitutions S_i that extend $[C, \theta_i]$. By definition, $\langle C, \{\theta_1, \ldots, \theta_n\}\rangle$ computes the set of substitutions that extend any $[C, \theta_i]$, i.e., the union of the S_i, thus the inference is sound.

□

Proposition 2 *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If $t \rightsquigarrow_0^{\mathcal{C}} S$ can be deduced using rules Rf$_1$ or Rf$_2$ from Figure 2 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow_0^{\mathcal{C}} S$.*

Proof. We apply the definition of satisfaction for each rule:

Rf$_1$ We know by hypothesis that the term t fulfills the condition thus, by definition, the set of reachable terms in zero steps is the singleton set with t as single element.

Rf$_2$ In a similar way to the case above, if the condition does not hold with the term t, then the set of reachable terms is empty.

□

Proposition 3 *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, n a natural number, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If $t \rightsquigarrow_n^{\mathcal{C}} S$ or $t \Rightarrow_1 S$ can be deduced using the rules from Figure 3 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow_n^{\mathcal{C}} S$ or $\mathcal{T}_{\Sigma/E',R'} \models t \Rightarrow_1 S$, respectively.*

Proof. We apply the definition of satisfaction for each rule:

Tr$_1$ We know that the condition is fulfilled by t, that t in exactly one step is rewritten to the set $\{t_1, \ldots, t_k\}$, and that each of these terms is rewritten in at most n steps to $S_1, \ldots, S_k$. Since $\{t_1, \ldots, t_k\}$ have been obtained in one step, the terms in $S_1, \ldots, S_k$ have been computed in at most $n + 1$ steps and in at least 1 step. Since we are looking for the solutions in zero or more steps, we have to compute the union of these sets with the set of reachable terms in zero steps, that in this case is the singleton set containing the term itself, because we are assuming it fulfills the condition. Thus, the inference is sound.

Tr$_2$ Analogous to the case above.

Rep We assume that all the possible rewrites in exactly one step at the top of $f(t_i)$, $0 \leq i \leq m$, lead to the set S_t and that all the reachable terms in exactly one step of each subterm t_i form the set S_i. By definition, all the reachable terms in exactly one step is the union of the set of all the terms obtained by rewrites at the top and the sets built by substituting each subterm by each reachable term from it (only one subterm is substituted at the same time), so the inference is sound.

Red$_1$ Since we know that $t \to t_1$, by coherence the same reachable terms are obtained from t and t_1. Moreover, since $t_2 =_{E'} t'$ we can substitute t_2 by t' and the set remains unchanged.

□

Proposition 4 *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, n a natural number, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If a statement $t \rightsquigarrow !_n^{\mathcal{C}} S$ or $t \rightsquigarrow +_n^{\mathcal{C}} S$ can be deduced using the rules from Figure 8 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow !_n^{\mathcal{C}} S$ or $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow +_n^{\mathcal{C}} S$, respectively.*

Proof.

Rf$_3$ In this case we know that the term fulfills the condition and that it is final, so by definition the set of final reachable terms consists exactly of the term itself.

Rf$_4$ If the term is final but it does not satisfy the condition, then the set of reachable states is empty by definition.

Rf$_5$ If no more steps can be used and the term is not final, the set of reachable terms is empty by definition.

Tr$_3$ We know that the term is not final, so we can split the search into two different searches, one in one step, that leads to $\{t_1, \ldots, t_k\}$ and another in n steps from these terms, that we know generate the sets $S_1, \ldots, S_k$. Thus, the result is the union of these sets.

Red$_2$ Analogous to Red$_1$ in Proposition 3.

Rf$_6$ By definition the relation requires at least one step, thus if only zero steps are available the result is the empty set.

Tr$_4$ First, we know that $t \to t'$, hence, by coherence, the same reachable terms are obtained from t and t'. Again, we distinguish the first step of the search, that leads to $\{t_1, \ldots, t_k\}$ and the next n steps. Since the terms in this second phase of the search have already evolved one step the single requirement is to fulfill the condition, and thus the union of the sets obtained with the relation for zero or more steps has to be the result.

□

Observe that these soundness propositions cannot be extended to the Ls, Fulfill, Fail, Top, and Rl inference rules, where the soundness of the conclusion depends not only on the calculus but also on the specification, which could be wrong.

Following the notation of [21], we will say that a judgment is *valid* when it holds in the intended interpretation $\mathcal{I}$, 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 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 debugging framework, we are able to locate wrong statements, wrong conditions, missing rules, and membership errors, which are defined as follows:

- Given a statement $A \Leftarrow C_1 \wedge \cdots \wedge C_n$ (where A is either an equation $l = r$, a membership $l : s$, or a rule $l \Rightarrow r$) and a substitution θ, the *statement instance* $\theta(A) \Leftarrow \theta(C_1) \wedge \cdots \wedge \theta(C_n)$ is *wrong* when all the atomic conditions $\theta(C_i)$ are valid in $\mathcal{I}$ but $\theta(A)$ is not.[4]
- Given a rule $l \Rightarrow r \Leftarrow C_1 \wedge \cdots \wedge C_n$ and a term t, the rule has a *wrong instance* if the judgments $[l := t, \emptyset] \rightsquigarrow \Theta_0$, $[C_1, \Theta_0] \rightsquigarrow \Theta_1$, $\cdots$, $[C_n, \Theta_{n-1}] \rightsquigarrow \Theta_n$ are valid in $\mathcal{I}$ but the application of Θ_n to the righthand side does not provide all the results expected for this rule.

[4] This is the kind of errors that can be detected with the calculus of wrong answers presented in [23].

- Given a condition $l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ and a term t, if $[l := t, \emptyset] \rightsquigarrow \Theta_0$, $[C_1, \Theta_0] \rightsquigarrow \Theta_1$, $\cdots$, $[C_n, \Theta_{n-1}] \rightsquigarrow \emptyset$ are valid in $\mathcal{I}$ (meaning that the condition does not hold for t) but the user expected the condition to hold, then we have a *wrong condition instance*.
- Given a condition $l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ and a term t, if there exists a substitution θ such that $\theta(l) = t$ and all the atomic conditions $\theta(C_i)$ are valid in $\mathcal{I}$, but the condition is not expected to hold, then we also have a *wrong condition instance*.
- A statement or condition is *wrong* when it admits a wrong instance.
- Given a term t, there is a *missing rule for* t if all the rules applied to t at the top lead to judgments $t \Rightarrow^{q_i} S_{q_i}$ valid in $\mathcal{I}$ but the union $\bigcup S_{q_i}$ does not contain all the reachable terms from t by using rewrites at the top.
- A specification has a *missing rule* if there exists a term t such that there is a missing rule for t.
- There is a *membership error* if the computed least sort of a term is strictly greater than the intended least sort but, for the time being, we are not able to detect the concrete reason for such an error.

Although we could use the proof trees shown thus far as debugging trees, we use an abbreviation that tries to keep only nodes related with the invalid nodes described above. In [21] we proved that a buggy node in the calculus of wrong answers was associated with a wrong statement; we extend this property with the following proposition:

Proposition 5 *Let N be a buggy node in some proof tree in the calculus of Figures 1, 2, 3, and 8 w.r.t. an intended interpretation $\mathcal{I}$. Then:*

1. *N is the consequence of a* Ls, Fulfill, Fail, Top, *or* RI *inference rule.*
2. *The error associated to N is a wrong statement, a missing rule, a wrong condition, or a membership error.*

Proof. The first item is a straightforward consequence of Propositions 1, 2, 3, and 4: N buggy means N invalid with all its children valid, and these are the only possible inference rules at N.

For the second property we study each inference rule separately:

Ls In this case the node is associated to a membership error, because the sort inferred for the term is correct by hypothesis but it is not the least one, thus it is a supersort. In this case we can only inform that the error is due to a membership that is not being applied, either because it does not exist or because its condition fails.

Fulfill If this node is buggy then there exists a substitution that satisfies the condition but the condition should not hold, thus we have a wrong condition. In this case the condition in the buggy node is pointed out as the error in the specification.

Fail In this case the set of substitutions that fulfill the condition is empty but the condition should hold, so the node is associated with a wrong condition. As in the case above, the error in the specification is related to the condition in the buggy node.

Top When this node is buggy all the possible rules have been applied at the top and their results are correct, but the union of these terms does not lead to all the intended reachable terms by rewriting the term at the top, so this node is related to a missing rule. In this case, we will point to the operator at the top of the term in the lefthand side of the buggy node as incompletely defined.

RI The nodes computing the set of substitutions that fulfill the condition of the rule are correct, but once the righthand side of the rule is instantiated with these substitutions there are reachable terms in the intended interpretation that are not in this set. Thus, in this case the buggy node is associated to a wrong rule and the rule applied in the node is pointed out as buggy.

□

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 6 *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, instead of using this proof tree T as debugging tree, we extend the notion of $APT(T)$ introduced in [21]. $APT(T)$ (from *Abbreviated Proof Tree*), or simply APT if the proof tree T is clear from the context, is a suitable abbreviation that shortens the proof tree and improves the questions posed to the user while keeping the completeness and correctness of the process. The rules to compute the APT, that must be applied in order, are shown in Figure 9:

- Rule ($\mathbf{APT}_1$) keeps the root of the tree and applies the general function APT', that returns a set of trees, to its premises.
- We distinguish between the one-step and the many-steps tree with rules ($\mathbf{APT}_2^o$) and ($\mathbf{APT}_2^m$). If the one-step option is selected, the rule ($\mathbf{APT}_2^o$) is applied and the conclusion of the rule is removed, while if the many-steps option is chosen the rule ($\mathbf{APT}_2^m$) is used and the conclusion is kept.
- Rule ($\mathbf{APT}_3$) associates the information kept about the rewrites at the top to the node with information about rewrites in one step.
- Rule ($\mathbf{APT}_4$) is in charge of keeping the more reduced set of terms when a Red node is found.
- Rule ($\mathbf{APT}_5$) indicates that the conclusion of the inference rules Fail, Fulfill, Ls, Top, and Rl are kept, since they contain relevant information for the debugging process.
- Finally, rule ($\mathbf{APT}_6$) indicates that in a case different from all the ones mentioned above the conclusion is deleted and APT' is recursively applied to the premises.

As said above, the reason for preferring the APT to the original proof tree is that it reduces the number of questions and moreover simplifies those that will be asked to the user while keeping the soundness and completeness of the technique. In particular, the following advantages are obtained:

- Nodes related to judgments about sets of substitutions are removed. Thus, the debugger does not prompt this kind of questions, that can be difficult to answer due to matching modulo.
- In the same way, the APT deletes questions about rewrites *at the top* of a given term (that can be difficult to answer due again to matching modulo) and associates the information of those nodes to questions related to the set of reachable terms in one step with rewrites in any position, that are in general easier to answer.
- It creates two different kinds of tree, one that contains judgments of rewrites with several steps and another that only contains rewrites in one step. The former present less questions, although they are harder to answer, while the latter asks more questions, that in general are easier to answer.
- Since the APT is calculated without computing the associated proof tree, it reduces the time and space needed to build the tree.

$$(\mathbf{APT}_1) \quad APT\left(\frac{T_1 \dots T_n}{af}\,\mathsf{R}_1\right) = \frac{APT'(T_1)\ \dots\ APT'(T_n)}{af}\,\mathsf{R}_1$$

$$(\mathbf{APT}_2^o) \quad APT'\left(\frac{T_1 \dots T_n}{af}\,\mathsf{Tr}_i\right) = APT'(T_1) \bigcup \dots \bigcup APT'(T_n)$$

$$(\mathbf{APT}_2^m) \quad APT'\left(\frac{T_1 \dots T_n}{af}\,\mathsf{Tr}_i\right) = \left\{\frac{APT'(T_1)\ \dots\ APT'(T_n)}{af}\,\mathsf{Tr}_i\right\}$$

$$(\mathbf{APT}_3) \quad APT'\left(\frac{\dfrac{T_1\ \dots\ T_n}{t \Rightarrow^{top} S'}\,\mathsf{Top}\ \ T_1' \dots T_m'}{t \Rightarrow_1 S}\,\mathsf{Rep}\right) = \left\{\frac{APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T_1')\ \dots\ APT'(T_m')}{t \Rightarrow_1 S}\,\mathsf{Top}\right\}$$

$$(\mathbf{APT}_4) \quad APT'\left(\frac{T\ \ \dfrac{T_1 \dots T_n}{af'}\,\mathsf{R}_1\ \ T'}{af}\,\mathsf{Red}_j\right) = \left\{\frac{APT'(T)\ APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T)}{af}\,\mathsf{R}_1\right\}$$

$$(\mathbf{APT}_5) \quad APT'\left(\frac{T_1 \dots T_n}{af}\,\mathsf{R}_2\right) = \left\{\frac{APT'(T_1)\ \dots\ APT'(T_n)}{af}\,\mathsf{R}_2\right\}$$

$$(\mathbf{APT}_6) \quad APT'\left(\frac{T_1 \dots T_n}{af}\,\mathsf{R}_1\right) = APT'(T_1) \bigcup \dots \bigcup APT'(T_n)$$

R_1 any inference rule R_2 Fulfill, Fail, Ls, Rl or Top $1 \le i \le 4$ $1 \le j \le 2$ af, af' any judgment

Figure 9: APT rules

For example, the application of the APT transformation to the proof tree for the maze example depicted in Figures 4 to 7 is shown in Figure 10. $APT'(*_5)$ and $APT'(*_2)$ represent the application of APT' to all the trees in $*_5$ and $*_2$, that were not shown; the triangles represent the application of the APT' function for the debugging trees for wrong answers; and † stands for a similar tree to the one on its left for the judgment `isOk([1,1][0,1]) → f`, where `f` abbreviates `false`. Note that in this tree we have made explicit the operator associated with rule Top to show that the transformation keeps track of the information associated to the potential buggy nodes.

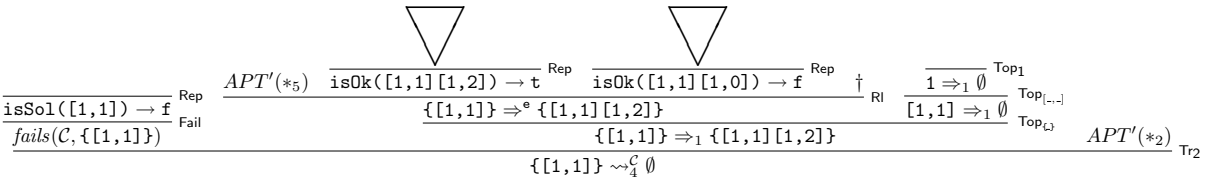

Figure 10: APT for the maze example

The following property of the abbreviated proof trees, in particular of the sets of trees computed with APT', will be used later when checking the correctness of our technique:

Lemma 1 *Let T be a finite proof tree representing an inference in the calculus of Figures 1, 2, 3, and 8 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$. Then the set $APT'(T)$ contains a tree with an invalid root iff the root N of T is invalid in $\mathcal{I}$.*

Proof. We prove it by induction on the number of nodes of T, which we denote as $n(T)$.

- If $n(T) = 1$, by Proposition 5 the inference step proving this node in T' must be Fulfill, Fail, Ls, Rl, or Top. In these cases the rule ($\mathbf{APT_5}$) is applied and the result holds, since it returns the same node.
- If $n(T) > 1$ we distinguish cases depending on the rule for APT' that can be applied at the root of T:
  - If it is ($\mathbf{APT}_2^m$), ($\mathbf{APT}_3$), ($\mathbf{APT}_4$), or ($\mathbf{APT}_5$), the result holds directly because the result is a singleton set with the same root.

– If it is ($\mathbf{APT}_2^o$) then by Propositions 3 and 4 N is invalid iff it 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 9 it can be checked that every subtree T_i of the root of T verifies $APT'(T_i) \subseteq APT'(T)$. Then $T' \in APT'(T)$.

– If it is ($\mathbf{APT}_6$) we remember that the rules are applied in order, thus the inference rule cannot be Fulfill, Fail, Ls, Rl, or Top, because in this case ($\mathbf{APT}_5$) would be applied. Thus, we can apply the same reasoning as in the case above and the result holds.

□

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

Theorem 3 *Let T be a finite proof tree representing an inference in the calculus of Figures 1, 2, 3, and 8 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:*

- *$APT(T)$ contains at least one buggy node (completeness).*
- *Any buggy node in $APT(T)$ has an associated wrong statement, missing rule, wrong condition, or membership error in $\mathcal{R}$ (correctness).*

Proof.

- $APT(T)$ 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 6.
- 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 such that $T_N \in APT(T')$. We prove that N has an associated wrong statement, missing rule, wrong condition, or membership error by induction on the number of nodes of T', $n(T')$:

  – If $n(T') = 1$, by Proposition 5 the inference step in the root of T' is Fulfill, Fail, Ls, Rl, or Top. Therefore, the rule ($\mathbf{APT_5}$) will be applied and the same node is returned, thus by Proposition 5 again the result holds.

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

  ($\mathbf{APT}_1$) and ($\mathbf{APT}_2^m$) We assume that ($\mathbf{APT}_1$) is applied to the original tree and thus the inference rule is Rf_1, Rf_2, Rf_3, Rf_4, Rf_5, Tr_1, Tr_2, Tr_3, Tr_4, Red_1, or Red_2, thus this case subsumes ($\mathbf{APT}_2^m$). We will check that these rules cannot be applied to generate a buggy node, and therefore it is not necessary to consider them. For example, if the root is labeled with Rf_1 then T' has the form

$$\frac{\mathit{fulfilled}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \{t'\}}\ \mathsf{Rf_1}$$

  $N \equiv t \rightsquigarrow_0^{\mathcal{C}} \{t'\}$ and T_N is

$$\frac{APT'(\mathit{fulfilled}(\mathcal{C}, t))}{t \rightsquigarrow_0^{\mathcal{C}} \{t'\}}\ \mathsf{Rf_1}$$

  N is invalid in T' but by Proposition 2 it cannot be buggy, hence $\mathit{fulfilled}(\mathcal{C}, t)$ is invalid. By Lemma 1 we know that $APT'(\mathit{fulfilled}(\mathcal{C}, t))$ contains a tree with an invalid root, and then N cannot be buggy.
  The proof for the rest of the rules is analogous.

  ($\mathbf{APT}_2^o$) and ($\mathbf{APT}_6$) $T_N \in APT'(T_i)$ for some child subtree T_i of the root of T' and the result holds by the induction hypothesis.

  ($\mathbf{APT}_3$) In this case T' has the form

$$\frac{\dfrac{T_1 \quad \ldots \quad T_n}{t \Rightarrow^{top} S'}\ \mathsf{Top} \qquad T_1' \quad \ldots \quad T_n'}{t \Rightarrow_1 S}\ \mathsf{Rep}$$

Thus $N \equiv t \Rightarrow_1 S$ and T_N is

$$\frac{APT'(T_1) \quad \ldots \quad APT'(T_n) \quad APT'(T'_1) \quad \ldots \quad APT'(T'_n)}{t \Rightarrow_1 S}\ \mathsf{Top}$$

By Proposition 3 we know that N cannot be buggy in T', thus either of $t \Rightarrow^{top} S'$ or the root of one of $T'_1 \ldots T'_n$ is invalid. However, if the root of one of the trees $T'_1 \ldots T'_n$ were invalid we know by Lemma 1 that the set obtained with APT' would contain a tree with an invalid root and then N cannot be buggy. Therefore, $t \Rightarrow^{top} S'$ is invalid but, for the same reason as before, $T_1 \ldots T_n$ cannot be invalid, so it is also buggy in T' and by Proposition 5 the rule label Top has associated a missing rule. Since this same label has been now assigned to N, the buggy node in the abbreviated proof tree has an associated missing rule.

($\mathbf{APT}_4$) We prove that rule Red_1 cannot be used to generate a buggy node, being the proof for Red_2 analogous. T' has the form

$$\frac{T \quad \dfrac{T_1 \ldots T_n}{t \rightsquigarrow_n^{\mathcal{C}} S'}\ \mathsf{R}_1 \quad T'}{t \rightsquigarrow_n^{\mathcal{C}} S}\ \mathsf{Red}_1$$

In this case $N \equiv t \rightsquigarrow_n^{\mathcal{C}} S$ and T_N is

$$\frac{APT'(T) \;\; APT'(T_1) \;\; \ldots \;\; APT'(T_n) \;\; APT'(T')}{t \rightsquigarrow_n^{\mathcal{C}} S}\ \mathsf{R}_1$$

By Proposition 3 we know that N cannot be buggy in T', thus one of its premises is invalid. T and T' cannot be invalid because in that case their respective set APT' would have a buggy node by Lemma 1 and N would not be buggy, thus $t \rightsquigarrow_n^{\mathcal{C}} S'$ is invalid. However, examining the inference rules we realize that the only possible rules for R_1 are Rf_1, Rf_2, Tr_1, Tr_2 or Red_1, thus we know by Propositions 2 and 3 that one of $T_1, \ldots, T_n$, let say T_{inv}, is invalid. But by Lemma 1 $APT'(T_{inv})$ contains a tree with an invalid root, and N cannot be buggy.

($\mathbf{APT}_5$) We present the proof for the inference rule $\mathsf{Fulfill}$, being the rest of cases analogous. T' has the form

$$\frac{T_1 \;\; \ldots \;\; T_n}{\mathit{fulfilled}(\mathcal{C}, t)}\ \mathsf{Fulfill}$$

Then $N \equiv \mathit{fulfilled}(\mathcal{C}, t)$ and T_N is

$$\frac{APT'(T_1) \;\; \ldots \;\; APT'(T_n)}{\mathit{fulfilled}(\mathcal{C}, t)}\ \mathsf{Fulfill}$$

Since N is buggy in T_N all the trees in $APT'(T_1) \;\; \ldots \;\; APT'(T_n)$ are valid and by Lemma 1 the roots of $T_1 \ldots T_n$ are also valid and N is buggy in T'. By Proposition 5 it is associated with a wrong statement in T' and, since we have the same label in T_N, the result holds.

□

4 Using the debugger

In this section we make explicit what is assumed about the modules introduced by the user. Then we present the different questions that can be presented to the user and the complete list of available commands. Finally, we illustrate how to use the debugger to debug several buggy examples.

4.1 Assumptions

Since we are debugging Maude modules, they are expected to satisfy the appropriate executability requirements. The specifications in functional modules have to be terminating, confluent, sort decreasing and, given an equation $t_1 = t_2 \Leftarrow C_1 \wedge \ldots \wedge C_n$, all the variables occurring in t_2 and $C_1 \ldots C_n$ must appear in t_1 or become instantiated by matching [8, Section 4.6]. While the equational part of system modules has to fulfill these requirements, rewrite rules must be coherent with respect to the equations

and, given a rule $t_1 \Rightarrow t_2 \Leftarrow C_1 \wedge \ldots \wedge C_n$, the variables occurring in t_2 and $C_1 \ldots C_n$ must appear in t_1 or become instantiated in matching or rewriting conditions [8, Section 6.3].

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. Likewise, when debugging missing answers some terms can be pointed out as final. Thus, this information has to be accurate in order to find the buggy node.

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.

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, it is not required to use the same signature in the correct and the debugged modules. If the correct module cannot help in answering a question, the user may have to answer it.

Finally, the patterns used in matching and rewrite conditions, as well as the signature, are supposed to be correct and they will not be considered during the debugging process.

4.2 Questions

We briefly describe in this section the different kinds of questions asked by the debugger, defining for each of them when they are considered correct in order to answer appropriately them during the debugging process. The possible questions are related to:

Reductions When a term t has been reduced by using equations to another term t', the debugger asks questions of the form "Is this reduction correct? $t \rightarrow t'$." These judgments are correct if the user expected t to be fully reduced to t' by using the equational part (equations and memberships) of the module.

Memberships When a sort s is inferred for a term t, the debugger prompts questions of the form "Is this membership correct? $t : s$." These judgments are correct if the expected least sort of t is a subsort of s or s itself.

Least sorts When the judgment refers to the least sort ls of a term t, the tool makes questions of the form "Did you expect t to have least sort ls?." In this case, the judgment is correct if the intended least sort of t is exactly ls.

Rewrites in one step When a term t is rewritten into another term t' in only one step, the debugger asks questions of the form "Is this rewrite correct? $t \Rightarrow_1 t'$," where t' has already been fully reduced by using equations. This judgment is correct if the user expected to obtain t' from t modulo equations with only one rewrite.

Rewrites in several steps When a term t is rewritten into another one t' after several rewrite steps, the debugger shows the question "Is this rewrite correct? $t \Rightarrow^+ t'$," where t' is fully reduced. This question is only prompted if the user selects the many-steps tree for wrong answers. This judgment is correct if t' is expected to be reachable from t.

Final terms When a term t cannot be further rewritten, the debugger asks "Did you expect t to be final?." This judgment is correct if the user expected that no rules can be applied to t.

Solutions When a term t fulfills the search condition, the debugger shows questions of the form "Did you expect t to be a solution?." This judgment is correct if t is one of the intended solutions. In the same way, if a term does not fulfill the search condition the debugger asks "Did you expect t not to be a solution?," that is correct if t is not one of the expected solutions.

Reachable terms in one step When all the possible applications of each rule in the current specification to a term t lead to a set of terms $\{t_1, \ldots, t_n\}$, with $n > 0$, the debugger prompts the question "Are the following terms all the reachable terms from t in one step? $t_1, \ldots, t_n$." This judgment is correct if all the expected terms from t in one step constitute the set $\{t_1, \ldots, t_n\}$.

Reachable terms with one rule Given a term t and a rule r, when all the possible applications of r to t produces a set of terms $\{t_1, \ldots, t_n\}$, the debugger presents questions of the form "Are the following terms all the reachable terms from t with one application of the rule r? $t_1, \ldots, t_n$." This judgment is correct if all the expected reachable terms from t with one application of r form the set $\{t_1, \ldots, t_n\}$. When $n = 0$ the debugger prompts questions of the form "Did you expect that no terms can be obtained from t by applying the rule r?," that is correct if the rule r is not expected to be applied to t.

Reachable terms in several steps Given an initial term t, a condition c, and a bound in the number of steps n, when all the terms reachable in at most n steps from t that fulfill c are $t_1, \ldots, t_m$, with $m > 0$, the debugger makes the following distinction:

- If the condition c defines the initial condition of the search, the tool poses questions of the form "Are the following terms all the possible solutions from t in n steps? $t_1, \ldots, t_m$," where the bound is omitted if it is unbounded. This judgment is correct if all the solutions that the user expected to obtain from t in at most n steps constitute the set $\{t_1, \ldots, t_m\}$. If $m = 0$ the debugger asks questions of the form "Did you expect that no solutions are reachable from t in n steps?," where the bound is again omitted if it is unbounded. In this case, the judgment is correct if no solutions were expected from t in at most n steps.
- If the condition c has been obtained from a rewrite condition $t' \Rightarrow p$, then c is just a matching condition with the pattern p, and n is unbounded. In this case, the questions have the form "Are the following terms all the reachable terms from t that match the pattern p? $t_1, \ldots, t_m$." This judgment is correct if all the terms that should be obtained from t and match the pattern p constitute the set $\{t_1, \ldots, t_m\}$. When $m = 0$ the questions have the form "Did you expect that no terms matching the pattern p can be obtained from t?," that is correct if t is expected to be final or all the terms reachable from t are not expected to match p.

These questions are only asked if the many-steps tree for missing answers is used.

We recommend to follow some tips to ease the questions asked during the debugging process:

- It is usually more complicated to answer questions related to many steps (both in wrong and missing answers) than questions related to one step. Thus, if a specification is complex it is better to debug it with a one-step tree.
- There are some sorts that are usually final, such as `Bool` and `Nat`, so identifying them as final can avoid several tedious questions.
- If an error is found using a complex initial term, this error can probably be reproduced with a simpler one. Using this simpler term leads to easier debugging sessions.
- When facing a problem with both wrong and missing answers, it is usually better to debug first the wrong answers, because questions related to them are usually easier to answer and fixing them can also solve the missing answers problem.
- When a question is related to a set of reachable terms that contains some wrong terms, it is recommended to point out one of these terms as erroneous instead of indicating the whole set as wrong.
- When using the top-down navigation strategy, several questions are prompted. To point out one as erroneous or all of them as valid will shorten the debugging process, while pointing one question as correct usually only eases the current set of questions. Thus, to indicate that a question is valid is only recommended for extremely complicated or large sets of questions.

4.3 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[5]

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

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

Moreover, all the labels in statements of a flattened module can be selected or deselected with the 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 .)
```

In a similar way, it is also possible to indicate that some terms are final, that is, that they cannot be further rewritten:

- By using the value `final` in the attribute `metadata` of an operator, that indicates that the terms built with this operator at the top are final.
- By selecting a set of final sorts. In this case, terms having one of these sorts (or having a subsort of these sorts) and built only with constructors (operators with the attribute `ctor`) are considered final.
- On the fly, as will be explained below.

In the first two cases, the user must activate the final sorts mode with the command

```
(set final select on .)
```

While the attribute `metadata` must be written in the Maude file, final sorts can be selected/deselected with the commands

```
(final select SORTS .)
(final deselect SORTS .)
```

where `SORTS` is a list of sort identifiers separated by spaces.

This option can be switched off with the command

```
(set final select off .)
```

A module with only correct definitions can be used to reduce the number of questions. In this case, 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, is used when searching in the correct module and can be set with the command

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

[5] Although these labels, as well as the set of labels from a module and the final sorts below, can be selected and deselected with the corresponding modes switched off, they will have effect only when these modes are activated.

where BOUND is either a natural number or the constant unbounded. Note that if it is 0 the correct module will not be used, while if it is unbounded the correct module is assumed to be terminating.

When debugging wrong 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, before starting the debugging process, with the commands

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

being the first the default one.

In the same way, when debugging missing answers we distinguish between trees whose nodes are related to sets of terms obtained with one (the default case) or many steps. The user can select them with the commands

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

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 in any moment by using the commands

```
(top-down strategy .)
(divide-query strategy .)
```

When debugging missing answers, the user can prioritize questions related to the fulfillment of the search condition from questions involving the statements defining it. This option, switched off by default, can be activated with the command

```
(solutions prioritized on .)
```

and can be switched off again with

```
(solutions prioritized off .)
```

The debugging process for wrong answers 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 .)
```

for wrong reductions, memberships, and rewrites respectively. MODULE-NAME is the module where the computation took place; if no module name is given, the current module is used by default. Similarly, we start the debugging of missing answers with the commands

```
(missing [[depth]] [in MODULE-NAME :] INITIAL-TERM =>* PATTERN [s.t. CONDITION] .)
(missing [[depth]] [in MODULE-NAME :] INITIAL-TERM =>+ PATTERN [s.t. CONDITION] .)
(missing [[depth]] [in MODULE-NAME :] INITIAL-TERM =>! PATTERN [s.t. CONDITION] .)
```

where the first command specifies a search in zero or more steps, the second one in one or more steps, and the last one only returns final terms. The depth argument indicates the bound in the number of steps allowed in the search, and it is considered unbounded when omitted, while MODULE-NAME has the same behavior as in the commands above.

How the process continues depends on the selected strategy. In the divide and query strategy, each question refers to one judgment that can be either correct or wrong. The different answers are transmitted to the debugger with the answers

```
(yes .)
(no .)
```

If the question asked is too difficult, the user can avoid to answer it with[6]

```
(don't know .)
```

[6]Notice that the question will not be asked again, thus this answer can lead to incompleteness.

In addition to these general answers, others can be introduced depending on the kind of question. If it corresponds to the application of a statement, instead of just answering `yes`, we can also *trust* some statements on the fly if we decide the bug is not there. To trust the current statement we answer

```
(trust .)
```

If a question refers to a set of reachable terms and one of these terms is not reachable, the user can point it out with the answer

```
(I is wrong .)
```

where `I` is the index of the wrong term in the set. With this answer the debugger focuses then on debugging this wrong judgment.

In case the question is related to a set of reachable *solutions*, if one of the solutions is reachable but it should not fulfill the search condition, the user can indicate it with

```
(I is not a solution .)
```

where `I` is the index of the term that should not be in the set. With this answer the user indicates that the definition of the search condition is erroneous and the debugger centers on it to continue the process.

If the question is about a final term, additional information can be given by answering

```
(its sort is final .)
```

that indicates to the debugger that all the constructed terms with the same sort as this term are final.

In case the top-down strategy is selected, several questions will be displayed in each step. The user can introduce then answers of the form `(N : answer .)`, where `N` is the index of the question and `answer` is the same answer that would be used in the divide and query strategy for this question. Thus, if there is an invalid question, the user can point it out with the answer

```
(N : no .)
```

while correct questions are answered with

```
(N : yes .)
```

As a shortcut to answer `(yes .)` to all the questions, the debugger provides the answer

```
(all : yes .)
```

When the user considers that a question is too complicated, it can be discarded with

```
(N : don't know .)
```

If one of the questions is associated with a program statement and the user decides that it can be trusted, it is indicated with

```
(N : trust .)
```

When a question presents a judgment relating a term to a set of terms, and the term in position `I` is not reachable from the initial one, then we can point it out with

```
(N : I is wrong .)
```

focusing then the debugging process on this wrong computation.

Furthermore, if the question refers to the set of reachable *solutions*, we can identify a reachable term that does not fulfill the search condition with the command

```
(N : I is not a solution .)
```

where `I` the index of the term in the set. With this answer the debugger concentrates on the definition of the search condition.

If one of the questions is related with a final term, on the fly information is given with

```
(N : its sort is final .)
```

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

```
(undo .)
```

4.4 Examples

We illustrate in this section how to use the debugger for missing answers by means of several examples.

4.4.1 An example of system module: Solving a maze

We show here how to debug the maze example presented in Section 2.5. We recall that we have specified a module to search a path out of a labyrinth but, given a concrete labyrinth with an exit, the program is unable to find it:

```
Maude> (search {[1,1]} =>* {L:List} s.t. isSol(L:List) .)
search in MAZE :{[1,1]} =>* {L:List}.

No solution.
```

We start the debugging process by introducing the command:

```
Maude> (missing {[1,1]} =>* {L:List} s.t. isSol(L:List) .)
```

With this command the debugger builds a debugging tree for missing answers in zero or more steps, with the default one-step strategy and the questions about solutions not prioritized. Once this tree is built, it is navigated with the divide and query strategy. In this case, the following question is prompted:

```
Did you expect {[1,1][1,2][1,3][1,4]} to be final?

Maude> (no .)
```

Since we expected to reach the position `[2,4]` from `[1,4]`, this state should be rewritten and thus it is not final. Following the strategy, the debugger takes the subtree with this node as root as current debugging tree and selects the next question:

```
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4]) in one step?

1 [1,5]
2 [1,3]
3 [0,4]

Maude> (no .)
```

Again, the answer is `no` because the set of terms is incomplete: we expected to find the movement to the right too. The debugging tree is updated again and the new question is:

```
Did you expect [1,4] to be final?

Maude> (yes .)
```

The answer is `yes` because we have not defined rules for positions, thus they cannot evolve. The strategy removes the tree with this node as root from the debugging tree and the following questions are:

```
Did you expect [1,3] to be final?

Maude> (yes .)

Did you expect [1,2] to be final?

Maude> (yes .)

Did you expect [1,1][1,2][1,3][1,4] to be final?

Maude> (yes .)
```

We use the same reasoning about final terms to answer these questions. The next questions are:

```
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4])
with one application of the rule next2 ?

1 [0,4]

Maude> (yes .)

Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4])
with one application of the rule next3 ?

1 [1,3]

Maude> (yes .)

Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4])
with one application of the rule next1 ?

1 [1,5]

Maude> (yes .)
```

All these questions are related to the appropriate application of certain rules; these rules move the last position of the list to the left, up, and down, and thus they are correct. With this information, the debugger is able to find the bug, prompting:

```
The buggy node is:
next([1,1][1,2][1,3][1,4]) =>1 {[1,5], [1,3], [0,4]}

The operator next is not completely defined.
```

In fact, if we check the code we realize that we forgot to define the rule that specifies movements to the right. We must add the rule:

```
  rl [next4] : next(L [X,Y]) => [X + 1, Y] .
```

However, we noticed that this session required to answer a lot of similar questions. We can enhance the behavior of the debugger by using features such as selection of final terms on the fly. For example, when the third question is prompted:

```
Did you expect [1,4] to be final?

Maude> (its sort is final .)

Terms of sort Pos are final.
```

we can indicate that not only this term, but all the terms with its sort are final. With this answer the debugging tree is pruned, and the next question is:

```
Did you expect [1,1][1,2][1,3][1,4] to be final?

Maude> (its sort is final .)

Terms of sort List are final.
```

We use again this answer, although in this case it does not reduce the number of questions. As before, the debugger finishes with the same three questions as above.

Although the number of questions has been reduced, we still face some questions that we would like to avoid about final terms. To do this, we can activate the final selection mode before starting the debugging:

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

Final select is on.
```

Once this mode is active, we can point out the sorts of the terms that will not be rewritten. Note that terms whose least sort is a subsort of the sorts selected will also be considered as final. For example, we consider in our specification the sorts `Nat` and `List` as final, which implicitly indicates that the sort `Pos`, subsort of `List`, is also final:

```
Maude> (final select Nat List .)

Sorts List Nat are now final.
```

Moreover, since we know that the rules `next1`, `next2`, and `next3` are correct, we can avoid questions about them by pointing the rest of statements as suspicious with the commands:

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

Debug select is on.

Maude> (debug select is1 is2 c1 c2 expand .)

Labels c1 c2 expand is1 is2 are now suspicious.
```

Once these options are introduced, we can start the debugging process with the same command as before:

```
Maude> (missing {[1,1]} =>* {L:List} s.t. isSol(L:List) .)

Are the following terms all the reachable terms from {[1,1][1,2][1,3]} in one step?

1 {[1,1][1,2][1,3][1,4]}

Maude> (yes .)

Are the following terms all the reachable terms from {[1,1][1,2]} in one step?

1 {[1,1][1,2][1,3]}

Maude> (yes .)
```

Given the labyrinth's limits and wall, we must go down in both cases to find the exit. The next question selected by the debugger is:

```
Did you expect that no terms can be obtained from {[1,1][1,2][1,3][1,4]} by applying the rule
expand ?

Maude> (no .)
```

As we know, the list of positions should evolve to find the exit. The debugger asks now:

```
Is this reduction (associated with the equation c2) correct?

contains([2,1][2,2][2,3][4,3][5,3][6,3][7,3][8,3][1,5][2,5][3,5]
         [4,5][7,5][7,6][7,7][7,8],[1,3]) -> false

Maude> (trust .)
```

We realize now that the equation `c2` is simple enough to be trusted, although we pointed it out as suspicious at the beginning of the session. We use the command `trust` to avoid more questions related to this equation, and the following question is prompted:

```
Is this reduction (associated with the equation c1) correct?

contains(nil,[1,5]) -> false

Maude> (trust .)
```

We consider that this equation can also be trusted. Finally, the answer to the next question allows to find the problem:

```
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4]) in one step?

1 [1,5]
2 [1,3]
3 [0,4]

Maude> (no .)

The buggy node is:
next([1,1][1,2][1,3]) =>1 {[1,4], [1,2], [0,3]}

The operator next is not completely defined.
```

Although in this example we have used the default divide and query navigation strategy, it is also possible to use the top-down one by using the command:

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

Top-down strategy selected.
```

We can also change the strategy during the debugging process. In this case we reduce the number of questions by considering that the sorts `Nat` and `List` are final and that the suspicious statements are the equations defining the solution, `is1` and `is2`:

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

Final select is on.

Maude> (final select Nat List .)

Sorts List Nat are now final.

Maude> (set debug select on .)

Debug select is on.

Maude> (debug select is1 is2 .)

Labels is1 is2 are now suspicious.
```

If we introduce now the debugging command, the first question is related to the validity of the children of the root:

```
Maude> (missing { [1,1] } =>* { L:List } s.t. isSol(L:List) .)

Question 1 :
Did you expect {[1,1]} not to be a solution?

Question 2 :
Are the following terms all the reachable terms from {[1,1]} in one step?

1 {[1,1][1,2]}

Question 3 :
Did you expect {[1,1][1,2]} not to be a solution?

Question 4 :
Are the following terms all the reachable terms from {[1,1][1,2]} in one step?

1 {[1,1][1,2][1,3]}
```

```
Question 5 :
Did you expect {[1,1][1,2][1,3]} not to be a solution?

Question 6 :
Are the following terms all the reachable terms from {[1,1][1,2][1,3]} in one step?

1 {[1,1][1,2][1,3][1,4]}

Question 7 :
Did you expect {[1,1][1,2][1,3][1,4]} not to be a solution?

Question 8 :
Did you expect {[1,1][1,2][1,3][1,4]} to be final?

Maude> (8 : no .)
```

We have to select a wrong question, or indicate otherwise that all are valid. In this case the eighth question is erroneous because the position `[2,4]` is reachable from `[1,4]` and it is free of wall, so we do not expect this term to be final. Once we indicate it the associated node is selected as current one and the next question refers to the validity of its children:

```
Question 1 :
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4]) in one step?

1 [1,5]
2 [1,3]
3 [0,4]

Maude> (1 : no .)
```

With these answers the debugger can detect the problem:

```
The buggy node is:
next([1,1][1,2][1,3][1,4]) =>1 {[1,5], [1,3], [0,4]}

The operator next is not completely defined.
```

4.4.2 Vending machine

We use in this section a slightly modified version of the vending machine [8, Section 6] to illustrate more features of our debugger:

```
(mod VENDING-MACHINE is
 pr NAT .
```

We define the sorts `Coin`, that represents the money introduced in the machine; `Item`, that designates the products sold; and `Marking`, that stands for multisets of money and products:

```
  sorts Coin Item Marking .
  subsorts Coin Item < Marking .
```

We declare now the elements of these sorts: `$` and `q` represent dollars and quarters, of sort `Coin`, while `a` and `c` designate apples and cakes, of sort `Item`:

```
  ops $ q : -> Coin [ctor] .
  ops a c : -> Item [ctor] .
  op null : -> Marking [ctor] .
  op __ : Marking Marking -> Marking [ctor assoc comm id: null] .
```

The behavior of the machine is defined with rules: dollars and quarters can be introduced at any time, we can either buy a cake with one dollar or buy an apple with one dollar and receive one quarter as change, and we obtain a dollar with four quarters:

```
  var M : Marking .

  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => c q .
  rl [change] : q q q q => $ .
```

Finally, we specify a function `num$` that counts the number of dollars in a marking:

```
  op num$ : Marking -> Nat .
  eq [n$1] : num$($ M) = s(num$(M)) .
  eq [n$2] : num$(M) = 1 [owise] .
 endm)
```

Once this module is introduced we can look for the reachable terms from one dollar, in at most four steps and at least one step, and containing exactly two dollars, with the command:

```
Maude> (search [, 4] $ =>+ M:Marking s.t. num$(M:Marking) = 2 .)
search [,4] in VENDING-MACHINE : $ =>+ M:Marking .
```

It returns 11 solutions but, since none of them contain two dollars, all the correct solutions are missing.[7] First, we select the many-steps tree for this debugging:

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

Many-steps tree selected when debugging missing answers.
```

Now we can introduce the command to start the debugging process:

```
Maude> (missing [4] $ =>+ M:Marking s.t. num$(M:Marking) = 2 .)
```

Once the tree is computed, the first question is selected with the default divide and query strategy:

```
Are the following terms all the reachable solutions from $ $ in at most 3 steps?

1  $ c c q q
2  $ c c q
3  $ c c
4  $ c q q q
5  $ c q q
6  $ c q
7  $ c

Maude> (1 is not a solution .)
```

Since these terms are reachable but are not valid solutions, we can point one of them out with the `no solution` command.[8] The next questions are related to the computation of the number of dollars in a marking:

```
Is this reduction (associated with the equation n$1) correct?

num$($ c c q q) -> 2

Maude> (no .)

Is this reduction (associated with the equation n$2) correct?

num$(c c q q) -> 1

Maude> (no .)
```

[7] Notice that, following the guidelines given in Section 4.2, we should debug first the reduction of the condition with any of these terms to `true`. In fact, that is what we are implicitly doing when using the answer `(1 is not a solution .)` in the first question.

[8] In these cases, as well as when the `wrong` answer is used, we could also answer `no`, but to indicate that a given term is not a solution usually eases and shortens the debugging process.

In these cases, the function fails to compute the correct number of dollars. With this information, the debugger deduces that a statement in the definition of the condition is buggy:

```
The buggy node is:
num$(c c q q) -> 1
with the associated equation: n$2
```

In fact, this equation returns 1 when there are no dollars in the marking, so we fix it as follows:

```
  eq [n$2] : num$(M) = 0 [owise] .
```

Of course, the top-down navigation strategy also works for trees whose nodes refer to many steps. Once we introduce the appropriate commands, the following question is prompted:

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

Top-down strategy selected.

Maude>  (many-steps missing tree .)

Many-steps tree selected when debugging missing answers.

Maude> (missing [4] $ =>+ M:Marking s.t. num$(M:Marking) = 2 .)

Question 1 :
Are the following terms all the reachable terms from $ in one step?

1  c
2  c q
3  $ $
4  $ q

Question 2 :
Are the following terms all the reachable solutions from c in at most 3 steps?

1  $ c c q
2  $ c c
3  $ c q q
4  $ c q
5  $ c

Question 3 :
Are the following terms all the reachable solutions from c q in at most 3 steps?

1  $ c c q q
2  $ c c q
3  $ c q q q
4  $ c q q
5  $ c q

Question 4 :
Are the following terms all the reachable solutions from $ $ in at most 3 steps?

1  $ c c q q
2  $ c c q
3  $ c c
4  $ c q q q
5  $ c q q
6  $ c q
7  $ c

Question 5 :
Are the following terms all the reachable solutions from $ q in at most 3 steps?
```

```
1  $ c q q q
2  $ c q q
3  $ c q
4  $ q q q q
5  $ q q q
6  $ q q
7  $ q

Maude> (5 : 1 is not a solution .)
```

We face different options here. If we check the last question, we realize that it shows the same question as the divide and query strategy above. Thus, we could select again the first term of this question as an erroneous solution and the following questions would be similar to the ones posed with the previous strategy and the same error would be discovered. However, we could try to follow another path, so we use the `undo` command to return to the question:

```
Maude> (undo .)

Question 1 :
Are the following terms all the reachable terms from $ in one step?

1  c
2  c q
3  $ $
4  $ q

Question 2 :
...

Maude> (1 : 2 is wrong .)
```

If we study the first question, we realize that the second term should not be reachable in one step, because the price of one cake is one dollar, thus we cannot exchange one dollar for a cake and a quarter. We introduce an answer that indicates that this term is not reachable and the debugger shows a new error:

```
The buggy node is:
$ =>1 c q
with the associated rule: buy-a
```

Actually, the rule that exchanges a dollar for an apple and a quarter is erroneous, because it returns a cake instead of an apple. We fix the rule as follows:

```
rl [buy-a] : $ => a q .
```

Notice that in this case the cause of the missing answer was an error in a program statement, while the previous bug was found in a statement defining the condition of the search.

Nevertheless, the appropriate debugging technique should be to fix each error the debugger finds and then test if the program works. In this example we fix the error in `n$2` and then we use the following command to check the program:

```
Maude> (search [, 4] $ =>+ M:Marking s.t. num$(M:Marking) = 2 .)
search [,4] in VENDING-MACHINE : $ =>+ M:Marking .

Solution 1
M:Marking --> $ $

Solution 2
M:Marking --> $ $ q

Solution 3
M:Marking --> $ $ c
```

```
Solution 4
M:Marking --> $ $ c q

Solution 5
M:Marking --> $ $ q q

Solution 6
M:Marking --> $ $ c q q

Solution 7
M:Marking --> $ $ q q q

No more solutions.
```

As we see, although these terms fulfill the condition, all the terms with apples are missing. We debug this error with the divide and query strategy and the many-steps tree:

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

Many-steps tree selected when debugging missing answers.

Maude> (missing [4] $ =>+ M:Marking s.t. num$(M:Marking) = 2 .)

Are the following terms all the reachable solutions from $ $ in at most 3 steps?

1  $ $ c q q
2  $ $ c q
3  $ $ c
4  $ $ q q q
5  $ $ q q
6  $ $ q
7  $ $

Maude> (1 is wrong .)
```

We cannot reach the first term in three steps because to obtain a cake we need a dollar and this rewrite does not return any change, so we answer `wrong 1` and the next question is prompted:

```
Is this rewrite (associated with the rule buy-a) correct?

$ =>1 c q

Maude> (no .)
```

As said above, it is not possible to obtain a cake and a quarter from a single dollar, thus this judgment is wrong. The debugger is able now to show the error:

```
The buggy node is:
$ =>1 c q
with the associated rule: buy-a
```

4.4.3 CCS semantics

We present now how to debug the semantics of CCS [15, 28]. First, we specify actions in the module `ACTION`:

```
fmod ACTION is
  protecting QID .

  sorts Label Act .
  subsorts Qid < Label < Act .

  op tau : -> Act .
  op ~_ : Label -> Label [prec 10] .
```

```
  eq ~ ~ L:Label = L:Label .
endfm)
```

The next module declares processes by defining the sort `ProcessId` of process identifiers and `Process` for processes:

```
(fmod PROCESS is
  protecting ACTION .

  sorts ProcessId Process .
  subsorts Qid < ProcessId < Process .
```

Then the concrete processes are defined. Note that we use the attribute `frozen`, that prevents the arguments from being rewritten:

- The idle process:

  ```
  op 0 : -> Process [ctor] .
  ```

- A process with an action as prefix:

  ```
  op _._ : Act Process -> Process [ctor frozen prec 25] .
  ```

- The summation process:

  ```
  op _+_ : Process Process -> Process [ctor frozen assoc comm prec 35] .
  ```

- The composition process:

  ```
  op _|_ : Process Process -> Process [ctor frozen assoc comm prec 30] .
  ```

- The process created by relabelling:

  ```
  op _[_/_] : Process Label Label -> Process [ctor frozen prec 20] .
  ```

- A restricted process:

  ```
    op _\_ : Process Label -> Process [ctor frozen prec 20] .
  endfm)
  ```

We can now define CCS contexts, associating identifiers to process definitions:

```
(fmod CCS-CONTEXT is
  protecting PROCESS .
```

We define the sort `Context` for contexts and the sort `Process?`, that we will use as supersort of `Process` to distinguish between defined and undefined processes:

```
  sorts Process? Context .
  subsort Process < Process? .

  op nil : -> Context [ctor] .
  op _&_ : Context Context -> Context [ctor assoc comm id: nil prec 42] .
  op _=def_ : ProcessId Process -> Context [ctor prec 40] .
```

The constant `context` will represent the context of each concrete application:

```
 op context : -> Context .
```

We also define a constant `not-defined`, that represents the undefined process:

```
  op not-defined : -> Process? [ctor] .
```

The function `_definedIn_` checks if a process identifier is defined in a given context:

```
  vars X X' : ProcessId .
  var P : Process .
  vars C C' : Context .

  op _definedIn_ : ProcessId Context -> Bool .
  eq [in1] : X definedIn nil = false .
  eq [in2] : X definedIn (X' =def P & C') = (X == X') or (X definedIn C') .
```

Finally, the function `def` extracts a process definition from a context, returning `not-defined` if the process is not defined:

```
  op def : ProcessId Context -> Process? .
  eq [df1] : def(X, nil) = not-defined .
  eq [df2] : def(X, (X' =def P) & C') = if X == X' then P
                                        else def(X, C') fi .
endfm)
```

With these modules specified, we are now able to define the semantics of CCS:

```
(mod CCS-SEMANTICS is
  protecting CCS-CONTEXT .

  sort ActProcess .
  subsort Process < ActProcess .

  op {_}_ : Act ActProcess -> ActProcess [ctor frozen] .
```

We define the semantics for each different kind of process:

- The process `A . P` can perform action `A` and it becomes `P`:

  ```
  vars A B : Act .
  vars P P' Q Q' R : Process .

  rl [prefix] : A . P => {A}P .
  ```

- When we have a summation of processes, we evolve one (modulo commutativity) and discard the other:

  ```
  crl [summ] : P + Q => P' if P => {A}P' .
  ```

- We evolve a composition of processes by performing an action in any of the inner processes:

  ```
  crl [cmp1] : P | Q => {A}(P' | Q) if P => {A}P' .
  ```

- In a composition of processes, we can also evolve them with `tau` if their first actions have complementary labels:

  ```
  vars L M : Label .

  crl [cmp2] : P | Q => {tau}(P' | Q') if P => {L}P' /\ Q => {~ L}Q' .
  ```

- In a restriction, we perform the action `A` if it is not forbidden:

  ```
  crl [res] : P \ L => {A}(P' \ L) if P => {A}P'
                                   /\ A =/= L /\ A =/= ~ L .
  ```

- If we have a relabeling we perform the next action of the process and then rename it if needed:

```
    crl [rlb1] : P[M / L] => {M}(P'[M / L]) if P =>{L}P' .
    crl [rlb2] : P[M / L] => {~ M}(P'[M / L]) if P =>{~ L}P' .
    crl [rlb3] : P[M / L] => {A}(P'[M / L]) if P =>{A}P'
                                            /\ A =/= L /\ A =/= ~ L .
```

- When we have a process identifier, we search the associated process in the context and evolve it:

```
    var X : ProcessId .

    crl [def] : X => {A}P if (X definedIn context) /\ def(X,context) => {A}P .
```

We define now the reflexive and transitive closure of this semantics. To do it, we declare a sort `TProcess` and a `frozen` operator `[_]`:

```
  sort TProcess .
  subsort TProcess < ActProcess .
  op [_] : Process -> TProcess [frozen] .
```

The rules for the closure are in charge of performing the actions of the process enclosed by the square brackets. The first rule defines the case when only one action is performed, while the second one specifies the performance of at least two actions:

```
  var AP : ActProcess .

  crl [rtc1] : [ P ] => {A}Q if P => {A}Q .
  crl [rtc2] : [ P ] => {A}AP if P => {A}Q /\ [ Q ] => AP /\ [ Q ] =/= AP .
endm)
```

In order to execute concrete examples, we have to define the value of the constant `context`. We present an example where this context describes a vending machine, where the user can introduce one penny (`'1p`) and obtain a little item (`'VenL`) or introduce two pence (`'2p`) and obtain a big item (`'VenB`). `'VenL` and `'VenB` are very similar: when buying a little item it is sold with `'little` and collected with `'collectL`, while big items are sold with `'big` and gathered with `'collectB`. Once these actions are performed, the processes finish: We cannot use a recursive call to `'Ven` instead of the idle process `0` here because the implicit search in rewrite conditions in the transitive closure could generate an infinite number of reachable terms in only one step.

```
(mod EXAMPLE is
  inc CCS-SEMANTICS .

  eq context =  ('Ven    =def  '1p . 'VenL + '2p . 'VenB)
                ('VenL   =def  'little . 'collectL . 0) &
                ('VenB   =def  'big . 'collectB . 0) .
```

We also define in this module the property that we will use in the search. In this case, we want to find the processes that have executed the action `'2p`:

```
  var AP : ActProcess .
  var A : Act .

  op 2pExec : Process -> Bool .
  eq [2p1] : 2pExec({A} AP) = A == '2p or 2pExec(AP) .
  eq [2p2] : 2pExec(AP) = false [owise] .
endm)
```

We perform a search for final terms that fulfill this condition with the command:

```
Maude> (search ['Ven] =>! T:TProcess s.t. 2pExec(T:TProcess) .)
search in EXAMPLE :['Ven] =>! T:TProcess .

No solution.
```

that does not find any solution. Since a path with this action is possible, we should find solutions with this search, so we debug the specification with:

```
Maude> (missing (['Ven]) =>! T:TProcess s.t. 2pExec(T:TProcess) .)

Are the following terms all the reachable terms from ['Ven] with
one application of the rule rtc2 ?

1 {'big}{'collectB}0
2 {'little}{'collectL}0

Maude> (1 is wrong .)
```

We cannot reach these terms from the initial one because the action associated with the insertion of coins does not appear. The next question is:

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

'VenB =>1 {'big}'collectB . 0

Maude> (trust .)
```

The process `'VenB` provides a big item that then must be collected, so this transition is correct. Moreover, this rule is simple enough and we can trust it. The debugger asks now:

```
Is this rewrite (associated with the rule rtc1) correct?

['collectB . 0] =>1 {'collectB}0

Maude> (yes .)
```

The process that has a simple action can only perform it and then the process finishes. The next question selected by the debugger is:

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

'1p . 'VenL + '2p . 'VenB =>1 'VenB

Maude> (no .)
```

In this case we expected to execute `'2p` before `'VenB`, but the first action has been skipped and thus this judgment is wrong. The following question is:

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

'2p . 'VenB =>1 {'2p}'VenB

Maude> (yes .)
```

This judgment is correct because the first action has been completed and now the second one has to be executed. With this information the debugger finds an error in the rule `summ`:

```
The buggy node is:
'1p . 'VenL + '2p . 'VenB =>1 'VenB
with the associated rule: summ
```

In fact, the rule executes one of the processes in the rewrite condition but it omits the first action in the result. The rule can be fixed as follows:

```
  crl [summ] : P + Q => {A}P' if P => {A}P' .
```

Now, we can use the search command shown above to check that the program is correct:

```
Maude> (search ['Ven] =>! T:TProcess s.t. 2pExec(T:TProcess) .)
search in EXAMPLE :['Ven] =>! T:TProcess .

No solution.
```

However, we obtain again that no solutions are reachable from the initial state. We start again the debugging process, in this case trusting the module with the behavior of the context, using the many-steps tree, and prioritizing the questions related to solutions, in order to delay questions about the equations defining the condition:

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

Many-steps tree selected when debugging missing answers.

Maude> (set debug select on .)

Debug select is on.

Maude> (debug include EXAMPLE .)

Labels 2p1 2p2 cmp1 cmp2 def df1 df2 in1 in2 prefix res rlb1
       rlb2 rlb3 rtc1 rtc2 summ are now suspicious.

Maude> (debug exclude CCS-CONTEXT .)

Labels df1 df2 in1 in2 are now trusted.

Maude> (solutions prioritized on .)

Solutions are prioritized.

Maude> (missing (['Ven]) =>! T:TProcess s.t. 2pExec(T:TProcess) .)
```

The first questions selected by the debugger are:

```
Are the following terms all the reachable terms from ['VenB]
that match the pattern AP:ActProcess ?

1 {'big}{'collectB}0
2 {'big}'collectB . 0
3 ['VenB]

Maude> (yes .)

Are the following terms all the reachable terms from ['VenL]
that match the pattern AP:ActProcess ?

1 {'little}{'collectL}0
2 {'little}'collectL . 0
3 ['VenL]

Maude> (yes .)
```

In both cases all the reachable solutions have been obtained, so the answer is `yes`. The next questions are:

```
Are the following terms all the reachable terms from ['Ven] with one application of the rule rtc1 ?

1 {'1p}'VenL
2 {'2p}'VenB

Maude> (yes .)

Are the following terms all the reachable terms from 'Ven in one step?

1 {'1p}'VenL
2 {'2p}'VenB

Maude> (yes .)
```

One step in the transitive closure is just one step in the process, and both judgments are correct. The next question is related to the reachable terms from the initial one:

```
Are the following terms all the reachable terms from ['Ven] in one step?

1 {'1p}{'little}'collectL . 0
2 {'1p}{'little}{'collectL}0
3 {'2p}{'big}'collectB . 0
4 {'2p}{'big}{'collectB}0
5 {'1p}'VenL
6 {'2p}'VenB

Maude> (yes .)
```

Note that in this case one application of a rule can execute several actions due to the rewrite conditions. These are all the terms reachable from the initial one, so we answer `yes` and the next question is shown:

```
Did you expect {'2p}'VenB to be final?

Maude> (yes .)
```

This process is final because the operator `{_}_` is `frozen` and no rules have been defined at the top. The following question is:

```
Did you expect {'2p}'VenB not to be a solution?

Maude> (no .)
```

This term is final and it has executed the action `'2p`, so it should be a solution. Once we point it out the debugger is able to find the error:

```
The buggy node is:
The term {'2p}'VenB is not a solution.

Either the condition or the pattern of the search is wrong.
Pattern: T:TProcess
Condition: 2pExec(T:TProcess) = true
```

If we check the pattern and the condition used in the search, we notice that we are using a variable of sort `TProcess` as pattern, but this is the sort used for the transitive closure, so we should use a variable of sort `ActProcess`. If we execute again the search with this modification:

```
Maude> (search ['Ven] =>! AP:ActProcess s.t. 2pExec(AP:ActProcess) .)
search in EXAMPLE :['Ven] =>! AP:ActProcess .

Solution 1
AP:ActProcess --> {'2p}{'big}{'collectB}0

Solution 2
AP:ActProcess --> {'2p}{'big}'collectB . 0

Solution 3
AP:ActProcess --> {'2p}'VenB

No more solutions.
```

all the possible solutions are found.

Note that, as we have seen, terms built with the frozen operator `{_}_` at the top are always final, so we can point it out by using the value `final` in the `metadata` attribute:

```
  op {_}_ : Act ActProcess -> ActProcess [ctor frozen metadata "final"] .
```

Now, we have to activate the `final` mode in order to use this assertion. For example, this last error can be found with only one question with the top-down navigation strategy:

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

Final select is on.

Maude> (top-down strategy .)

Top-down strategy selected.

Maude> (missing (['Ven]) =>! T:TProcess s.t. 2pExec(T:TProcess) .)

Question 1 :
Are the following terms all the reachable terms from ['Ven] in one step?

1 {'1p}{'little}'collectL . 0
2 {'1p}{'little}{'collectL}0
3 {'2p}{'big}'collectB . 0
4 {'2p}{'big}{'collectB}0
5 {'1p}'VenL
6 {'2p}'VenB

Question 2 :
Did you expect {'1p}{'little}'collectL . 0 not to be a solution?

Question 3 :
Did you expect {'1p}{'little}{'collectL}0 not to be a solution?

Question 4 :
Did you expect {'2p}{'big}'collectB . 0 not to be a solution?

Question 5 :
Did you expect {'2p}{'big}{'collectB}0 not to be a solution?

Question 6 :
Did you expect {'1p}'VenL not to be a solution?

Question 7 :
Did you expect {'2p}'VenB not to be a solution?

Maude> (4 : no .)

The buggy node is:
The term {'2p}{'big}'collectB . 0 is not a solution.

Either the condition or the pattern of the search is wrong.
Pattern: T:TProcess
Condition: 2pExec(T:TProcess) = true
```

where the `final` mode has reduced the number of questions shown in this question from 13 to 7.

5 Implementation

We show here how the ideas described in the previous sections are implemented. The current implementation extends the specification of the debugger for wrong answers presented in [21]: Section 5.1 describes how the debugging trees are modified to deal with missing answers, Section 5.2 presents the construction of the new debugging trees, and Section 5.3 explains some new commands used in the interaction with the user.

5.1 Proof tree extensions

In this section we present how to modify the debugging trees to allow missing answers. As explained in [21], the debugging tree is specified in a module parameterized with the theory `TRIV` where the main

operator is `tree`, that receives an element of sort `Elt` (required by the theory), a natural number with the size of the tree, and a `Forest` (defined as a list of trees):

```
op tree(_,_,_) : X$Elt Nat Forest -> Tree [format (ngi o d d d d ++i n--i d)] .
```

To instantiate the sort `Elt` we declare in another module the sort `Inference`, that defines the different inferences allowed in the nodes of a debugging tree. Thus, we have to specify new inferences in order to build trees for missing answers. We use the following notation to indicate the least sort of a given term:

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

All the nodes for inferences of reachable terms keep the initial term and the list of results. We distinguish between:

- The set of reachable terms in one step:

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

- The set of reachable terms by applying one rule, that also keeps the rule applied:

  ```
  op _=>q[_]{_} : Term Qid TermList -> Inference [format (d b o d d d d d d)] .
  ```

- The set of reachable terms when many steps are used. In this case we also keep the bound, the pattern, the condition, and a Boolean value indicating whether this search corresponds to the initial one, and thus these terms are the reachable *solutions* from the initial one, or corresponds to a search due to a rewrite condition:

  ```
  op _~>[_]{_}s.t._&_[_] : Term Bound TermList Term Condition Bool
                           -> Inference [format (d b o d d d d d d d d d d d d d)] .
  ```

We use the operator `sol` to indicate (in the fourth argument) if a term (the first argument) matches the pattern given as second argument and fulfills the condition given as third argument. When the questions about solutions are prioritized these nodes are frozen and are expanded on demand, so its fifth argument is a Boolean value indicating whether the node has been already expanded. Finally, the last Boolean value indicates whether this term is a solution of the initial search condition or it is a solution of a rewrite condition:

```
op sol : Term Term Condition Bool Bool Bool -> Inference [format (b o)] .
```

Once these new inferences are included in the appropriate module, we only have to update some look-up functions and we can create debugging trees for missing answers.

5.2 Debugging trees for missing answers

We show in this section how the debugging trees for missing answers are built. First, we specify a module `SEARCH-TYPE` that defines the kind of search, where the constant `zeroOrMore` designates searches in zero or more steps, `oneOrMore` searches in one or more steps, and `final` searches for final terms:

```
fmod SEARCH-TYPE is
 sort SearchType .
 ops zeroOrMore oneOrMore final : -> SearchType .
endfm
```

The module `MISSING-ANSWERS-TREE` is in charge of building the debugging tree for missing answers:

```
fmod MISSING-ANSWERS-TREE is
 pr SYSTEM-TREE-CONSTRUCTION .
 pr SEARCH-TYPE .
 pr PAIR{Forest, TermList} .
```

The debugging tree for missing answers is built with the function `createMissingTree`, that receives the module where the terms should be found, a correct module (possibly `undefMod`), the initial term, the pattern, the condition to be fulfilled, the bound in the number of rewrites for *wrong* rewrites, the number of steps that can be given in the search, the search type, the type of tree to be built (one-step or many-steps) for both wrong and missing answers, the set of suspicious labels, the set of sorts that are final, a Boolean value indicating if the search introduced by the user was unbounded, and a Boolean value pointing out if the questions about solutions are prioritized. The forest is generated with an auxiliary function `createMissingForest` that receives, in addition to the values above, a Boolean value indicating whether the forest currently built corresponds to the initial search or to a search due to a rewrite condition, being its value in this case `true`. Once the tree has been built, the questions associated with terms that the user has declared as final are pruned with `cleanTree*`:

```
op createMissingTree : Module Maybe{Module} Term Term Condition Bound Bound SearchType
                       TreeType TreeType QidSet Bool QidSet Bool Bool -> Tree .

ceq createMissingTree(M, CM, T, PAT, C, BW, BM, ST, TTW, TTM, QS, BFS, FS, UB?, SP) =
      contract(tree(T ~>[B'] {clean(extractTerms(F))} s.t. PAT & C [true],
                                        1 + getOffspring*(F), F))
 if TM := deleteSuspicious(M, QS) /\
    T' := getTerm(metaReduce(M, T)) /\
    F := cleanTree*(M, BFS, FS, createForest(M, TM, CM, T, T', QS, strat?(M))
                            createMissingForest(labeling(M), TM, CM, T', PAT,
                                    C, BW, BM, ST, TTW, TTM, QS, FS, UB?, SP, true)) /\
    B' := if UB? then unbounded else BM fi .
```

If the tree to be built cannot evolve (the bound is `0`) and zero or more steps can be used, then we use the function `solutionTree` to create a tree that proves whether the condition is satisfied or not:

```
op createMissingForest : Module Module Maybe{Module} Term Term Condition Bound Bound SearchType
                         TreeType TreeType QidSet QidSet Bool Bool Bool -> Forest .
eq createMissingForest(OM, TM, CM, T, PAT, C, BW, 0, zeroOrMore, TTW, TTM, QS,
                       FS, UB?, SP, FST) =
       solutionTree(OM, TM, CM, T, PAT, C, BW, zeroOrMore, TTW, TTM, QS, FS, SP, FST) .
```

`solutionTree` uses the function `solveCondition` to decide if the current term fulfills the condition, and then examines if the questions about solutions are prioritized. If they are prioritized the tree is built on demand and only the node referring to the fulfillment of the condition is generated. If the solutions are not prioritized, the function distinguishes whether the condition is fulfilled or not. In the first case, it uses the function `conditionForest`, already defined for wrong answers, to create a forest for the satisfaction of the condition; in the other case, the forest is computed with the function `buildConditionForestMissingAux`, that proves that the condition cannot be fulfilled:

```
op solutionTree : Module Module Maybe{Module} Term Term Condition Bound SearchType
                  TreeType TreeType QidSet QidSet Bool Bool -> Tree .
ceq solutionTree(OM, TM, CM, T, PAT, C, BW, ST, TTW, TTM, QS, FS, SP, FST) =
     tree(sol(T, PAT, C, SC, not SP, FST), getOffspring*(F) + 1, F)
 if SC := solveCondition(OM, T, PAT, C) /\
    F := if SP
         then mtForest
         else if SC
              then conditionForest(substitute(OM, C, metaMatch(OM, PAT, T, C, 0)),
                                              OM, TM, CM, QS, BW, TTW)
              else first(buildConditionForestMissingAux(OM, TM, CM, QS, FS, T, T, BW,
                         ST, TTW, TTM, SP, PAT := T /\ C, strat?(OM),
                         allSubs(OM, T, PAT := T, 0, mtSSB), 2, getNumConds(C) + 1))
              fi
         fi .
```

The function `solveCondition` receives the module where the search takes place, the term to be checked, the pattern, and the condition, and checks if the term is a solution by using the predefined function `metaMatch`, that examines if two terms match while fulfilling a given condition:

```
op solveCondition : Module Term Term Condition -> Bool .
eq solveCondition(M, T, PAT, C) = metaMatch(M, PAT, T, C, 0) =/= noMatch .
```

If the terms can still evolve (the bound is greater than `0`), we compute all the possible reachable terms in exactly one step with the function `oneStepMissingTree` and evolve each of them with `createMissingForest*`. The solutions obtained are gathered with `extractTerms`, while we check if the current term is a valid solution with the function `solveCondition`. Finally, if the tree selected by the user is for many-steps transitions we put a root to the generated forest specifying the number of steps, while if we want one-step transitions only the forest is returned:

```
ceq createMissingForest(OM, TM, CM, T, PAT, C, BW, s(N'), zeroOrMore, TTW, TTM,
                        QS, FS, UB?, SP, FST) =
            if TTM == os
            then RF
            else tree(T ~>[B'] {TL''} s.t. PAT & C [FST], 1 + getOffspring*(RF), RF)
            fi
 if tree(T =>1 {TL}, N, F) := oneStepMissingTree(OM, TM, CM, T, QS, FS, BW, zeroOrMore,
                                                 TTW, TTM, SP) /\
    F' := createMissingForest*(OM, TM, CM, TL, PAT, C, BW, N', zeroOrMore,
                                    TTW, TTM, QS, FS, UB?, SP, FST) /\
    TL' := if solveCondition(OM, T, PAT, C)
           then T
           else empty
           fi /\
    TL'' := clean((extractTerms(F'), TL')) /\
    CF := solutionTree(OM, TM, CM, T, PAT, C, BW, zeroOrMore, TTW, TTM, QS, FS, SP, FST) /\
    RF := CF tree(T =>1 {TL}, N, F) F' /\
    B' := if UB? then unbounded else s(N') fi .
```

If the terms can evolve at least one step and the search type is `oneOrMore`, we use the method `oneStepMissingTree` to evolve the terms, and then we compute the possible results from them for zero or more steps:

```
ceq createMissingForest(OM, TM, CM, T, PAT, C, BW, s(N'), oneOrMore, TTW, TTM,
                        QS, FS, UB?, SP, FST) =
             tree(T ~>[B'] {TL'} s.t. PAT & C [FST], 1 + getOffspring*(RF), RF)
 if tree(T =>1 {TL}, N, F) := oneStepMissingTree(OM, TM, CM, T, QS, FS, BW,
                                                 oneOrMore, TTW, TTM, SP) /\
    F' := createMissingForest*(OM, TM, CM, TL, PAT, C, BW, N', zeroOrMore,
                                    TTW, TTM, QS, FS, UB?, SP, FST) /\
    RF := tree(T =>1 {TL}, N, F) F' /\
    B' := if UB? then unbounded else s(N') fi /\
    TL' := clean(extractTerms(F')) .
```

If the bound in this kind of search is `0` no solutions can be obtained and the empty forest is returned:

```
eq createMissingForest(OM, TM, CM, T, PAT, C, BW, 0, oneOrMore, TTW, TTM,
                        QS, FS, UB?, SP, FST) = mtForest .
```

When a final result is reached (the application of `oneStepMissingTree` to the term computes the empty set of terms) in a `final` search, we use `solutionTree` to build the appropriate tree:

```
ceq createMissingForest(OM, TM, CM, T, PAT, C, BW, BM, final, TTW, TTM, QS,
                        FS, UB?, SP, FST) =
        solutionTree(OM, TM, CM, T, PAT, C, BW, final, TTW, TTM, QS, FS, SP, FST)
        tree(T =>1 {empty}, N, F)
 if tree(T =>1 {empty}, N, F) := oneStepMissingTree(OM, TM, CM, T, QS, FS, BW,
                                                    final, TTW,  TTM, SP) .
```

If the term can evolve (i.e., there are reachable terms in one step) but the number of available rewrites reaches `0` only the tree with the one-step inference is returned:

```
ceq createMissingForest(OM, TM, CM, T, PAT, C, BW, 0, final, TTW, TTM, QS, FS,
                        UB?, SP, FST) =
        tree(T =>1 {TL}, N, F)
 if tree(T =>1 {TL}, N, F) := oneStepMissingTree(OM, TM, CM, T, QS, FS, BW,
                                                 final, TTW, TTM, SP) /\
    TL =/= empty .
```

If the bound is greater than 0 and the set of reachable terms in one step is not empty, we continue the search with the terms in this set and the bound decreased in one step:

```
ceq createMissingForest(OM, TM, CM, T, PAT, C, BW, s(N'), final, TTW, TTM,
                        QS, FS, UB?, SP, FST) =
      if TTM == os
      then RF
      else tree(T ~>[B'] {TL'} s.t. PAT & C [FST], 1 + getOffspring*(RF), RF)
      fi
 if tree(T =>1 {TL}, N, F) := oneStepMissingTree(OM, TM, CM, T, QS, FS, BW,
                                                 final, TTW, TTM, SP) /\
    TL =/= empty /\
    F' := createMissingForest*(OM, TM, CM, TL, PAT, C, BW, N', final,
                               TTW, TTM, QS, FS, UB?, SP, FST) /\
    RF := tree(T =>1 {TL}, N, F) F' /\
    B' := if UB? then unbounded else s(N') fi /\
    TL' := clean(extractTerms(F')) .
```

Given a list of terms, the function `createMissingForest*` computes the forest obtained by evolving each of them:

```
op createMissingForest* : Module Module Maybe{Module} TermList Term Condition Bound Bound
                          SearchType TreeType TreeType QidSet QidSet Bool Bool Bool
                          -> Forest .
eq createMissingForest*(OM, TM, CM, empty, PAT, C, BW, BM, ST, TTW, TTM, QS,
                        FS, UB?, SP, FST) = mtForest .
eq createMissingForest*(OM, TM, CM, (T, TL), PAT, C, BW, BM, ST, TTW, TTM, QS, FS,
                        UB?, SP, FST) =
    createMissingForest(OM, TM, CM, T, PAT, C, BW, BM, ST, TTW, TTM, QS, FS, UB?, SP, FST)
    createMissingForest*(OM, TM, CM, TL, PAT, C, BW, BM, ST, TTW, TTM, QS, FS, UB?, SP, FST) .
```

We use the method `oneStepMissingTree` to obtain all the possible results obtained from a term with exactly one step. To do it, we first compute all the results reached by rewriting the term one step at top with `oneStepTop`, and then the subterms of the original term are rewritten one step with `oneStepCongruence`. The function `combineSubterms` is in charge of substituting the arguments in the initial term with these new subterms:

```
op oneStepMissingTree : Module Module Maybe{Module} Term QidSet QidSet Bound SearchType
                        TreeType TreeType Bool -> Tree .
ceq oneStepMissingTree(OM, TM, CM, T, QS, FS, BW, ST, TTW, TTM, SP) =
            tree(T =>1 {clean((TL, TL'))}, 1 + getOffspring*(F F'), F F')
 if tree(T =>1 {TL}, N, F) := oneStepTop(OM, TM, CM, T, QS, FS, BW, ST, TTW, TTM, SP) /\
    F' := oneStepCongruence(OM, TM, CM, T, QS, FS, BW, ST, TTW, TTM, SP) /\
    TL' := combineSubterms(OM, T, F') .
```

`oneStepTop` uses an auxiliary function `buildConditionForestMissing*` that traverses all the rules and returns all their possible applications:

```
op oneStepTop : Module Module Maybe{Module} Term QidSet QidSet Bound SearchType
                TreeType TreeType Bool -> Forest .
ceq oneStepTop(OM, TM, CM, T, QS, FS, BW, ST, TTW, TTM, SP) =
                           tree(T =>1 {TL}, 1 + getOffspring*(F), F)
 if < F, TL > := buildConditionForestMissing*(OM, TM, CM, QS, FS, T, BW, ST, TTW, TTM, SP,
                                  getRls(OM), strat?(OM)) .
```

`buildConditionForestMissing*` returns a pair with the generated forest and the computed terms. When the set of rules to be checked is `none`, it returns a pair with the empty forest and the empty term list:

```
op buildConditionForestMissing* : Module Module Maybe{Module} QidSet QidSet Term Bound
                                  SearchType TreeType TreeType Bool RuleSet Bool
                                  -> Pair{Forest, TermList} .

eq buildConditionForestMissing*(M, TM, CM, QS, FS, T, BW, ST, TTW, TTM, SP, none, ST?) =
                        < mtForest, empty > .
```

When at least one rule can be used, the function computes the reachable terms by using the method `buildConditionForestMissing`, while the rest of the rules are recursively traversed:

```
ceq buildConditionForestMissing*(M, TM, CM, QS, FS, T, BW, ST, TTW, TTM, SP, R RS, ST?) =
                        < F F', (TL, TL') >
 if < F, TL > := buildConditionForestMissing(M, TM, CM, QS, FS, T, BW, ST, TTW, TTM,
                                             SP, R, ST?) /\
    < F', TL' > := buildConditionForestMissing*(M, TM, CM, QS, FS, T, BW, ST,
                                                TTW, TTM, SP, RS, ST?) .
```

The function `buildConditionForestMissing` first checks if the lefthand side of the rule matches the current term. If this matching fails (checked with the function `match?`), the set of reachable terms is `empty` and only the forest to compute the normal form of the current term is generated:

```
op buildConditionForestMissing : Module Module Maybe{Module} QidSet QidSet Term Bound
                                 SearchType TreeType TreeType Bool Rule Bool
                                 -> Pair{Forest, TermList} .
ceq buildConditionForestMissing(M, TM, CM, QS, FS, T, BW, ST, TTW, TTM, SP,
                                R, ST?) = < F, empty >
 if not match?(M, T, R) /\
    F := createForest(M, TM, CM, T, getTerm(metaReduce(M, T)), QS, ST?) .
```

where `match?` uses the predefined function `metaMatch` to check if the terms match:

```
op match? : Module Term Rule -> Bool .
eq match?(M, T, rl T1 => T2 [AtS] .) = metaMatch(M, T1, T, nil, 0) :: Substitution .
eq match?(M, T, crl T1 => T2 if C [AtS] .) = metaMatch(M, T1, T, nil, 0) :: Substitution .
```

If the terms match in an unconditional rule, the function computes all the substitutions obtained from matching of the current term with the lefthand side of the rule with the function `allSubs`, and uses them to instantiate the righthand side with `applySubs`, that applies the substitutions and computes the normal form of the obtained terms. Note that if the rule currently applied has been trusted only the nodes corresponding to the reduction to normal form are kept, while if the rule is suspicious a node with all the reachable terms is created:

```
ceq buildConditionForestMissing(M, TM, CM, QS, FS, T, BW, ST, TTW, TTM, SP,
                                rl T1 => T2 [AtS] ., ST?) =
                    < F', TL >
 if SSB := allSubs(M, T, T1 := T, 0, mtSSB) /\
    < F, TL > := applySubs(M, TM, CM, QS, T2, SSB) /\
    F' := if trusted?(AtS, QS)
          then F
          else tree(T =>q[label(AtS)] {TL}, 1 + getOffspring*(F), F)
          fi [owise] .
```

where `trusted` checks if the rule is trusted, that is, it is not labeled or it has a label that is not included in the set of suspicious ones:

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

`allSubs` uses the predefined function `metaMatch` to obtain all the possible substitutions:

```
op allSubs : Module Term Condition Nat Set{Substitution} -> Set{Substitution} .
ceq allSubs(M, T, C, N, SSB) = allSubs(M, T, C, s(N), SB . SSB)
 if SB := metaMatch(M, T, T, C, N) .
eq allSubs(M, T, C, N, SSB) = SSB .
```

and applySubs uses the auxiliary function substitute to instantiate the variables with their corresponding values, and then it uses createForest to obtain the normal form of the instantiated terms:

```
op applySubs : Module Module Maybe{Module} QidSet Term Set{Substitution}
               -> Pair{Forest, TermList} .
eq applySubs(OM, TM, CM, QS, T, mtSSB) = < mtForest, empty > .
ceq applySubs(OM, TM, CM, QS, T, SB . SSB) = < F' F, (T'', TL) >
 if < F, TL > := applySubs(OM, TM, CM, QS, T, SSB) /\
    T' := substitute(OM, T, SB) /\
    T'' := getTerm(metaReduce(OM, T')) /\
    F' := createForest(OM, TM, CM, T', T'', QS, strat?(OM)) .
```

When a conditional rule is applied, we use the substitutions with its lefthand side in the function buildConditionForestMissingAux, that uses them to compute the substitutions that fulfill the condition and generate all the terms obtained with each of them:

```
ceq buildConditionForestMissing(M, TM, CM, QS, FS, T, BW, ST, TTW, TTM, SP,
                                crl T1 => T2 if C [AtS] ., ST?) =
    < F'', TL >
 if SSB := allSubs(M, T, T1 := T, 0, mtSSB) /\
    < F, TL > := buildConditionForestMissingAux(M, TM, CM, QS, FS, T, T2, BW, ST, TTW,
                                TTM, SP, T1 := T /\ C, ST?, SSB, 2, getNumConds(C) + 1) /\
    F' := createForest(M, TM, CM, T, getTerm(metaReduce(M, T)), QS, ST?) F /\
    F'' := if trusted?(AtS, QS)
           then F'
           else tree(T =>q[label(AtS)] {TL}, 1 + getOffspring*(F'), F')
           fi [owise] .
```

The function buildConditionForestMissingAux has as new parameters the condition extended with the matching with the lefthand side of the rule, the set of substitutions that satisfy the condition thus far, and two natural numbers: the index of the atomic condition currently evaluated and the size of the condition. Once the whole condition has been analyzed we use the set of substitutions to instantiate the righthand side of the rule:

```
op buildConditionForestMissingAux : Module Module Maybe{Module} QidSet QidSet Term Term
                                    Bound SearchType TreeType TreeType Bool Condition Bool
                                    Set{Substitution} Nat Nat -> Pair{Forest, TermList} .
eq buildConditionForestMissingAux(M, TM, CM, QS, FS, T, T2, BW, ST, TTW, TTM, SP, C,
                                  ST?, SSB, s(N), N) =
               applySubs(M, TM, CM, QS, T2, SSB) .
```

If there are no substitutions that fulfill the current fragment of the condition then it cannot be satisfied and the empty set of terms is obtained:

```
eq buildConditionForestMissingAux(M, TM, CM, QS, FS, T, T2, BW, ST, TTW, TTM, SP, C,
                                  ST?, mtSSB, N, N') = < mtForest, empty > .
```

Otherwise, we apply the substitutions obtained thus far to the current atomic condition and generate the corresponding forest with the function substituteAndCreateForest. Then, the new set of substitutions that satisfy the condition extended with the current atomic condition (obtained with getNFirst) is computed with allSubs:

```
ceq buildConditionForestMissingAux(M, TM, CM, QS, FS, T, T2, BW, ST, TTW, TTM, SP, C,
                                   ST?, SSB, N, N') = < F F', TL' >
 if N <= N' /\
    C' := getNFirst(C, N) /\
    F := substituteAndCreateForest(M, TM, CM, QS, FS, BW, ST, TTW, TTM, SP,
                                   ST?, getLast(C'), SSB) /\
    SSB' := allSubs(M, T, C', 0, mtSSB) /\
    < F', TL' > := buildConditionForestMissingAux(M, TM, CM, QS, FS, T, T2, BW, ST,
                                        TTW, TTM, SP, C, ST?, SSB', s(N), N') [owise] .
```

The function substituteAndCreateForest traverses a set of substitutions and generates the forest obtained by instantiating and solving the given condition with each of them. When the list of substitutions is empty, the empty forest is returned:

```
op substituteAndCreateForest : Module Module Maybe{Module} QidSet QidSet Bound SearchType
                               TreeType TreeType Bool Bool Condition Set{Substitution}
                               -> Forest .
eq substituteAndCreateForest(M, TM, CM, QS, FS, BW, ST, TTW, TTM, SP, ST?, C, mtSSB) = mtForest .
```

When at least one more substitution must be used the condition is instantiated with it and the function `createForestOnceSubstituted` is in charge of computing the forest:

```
ceq substituteAndCreateForest(M, TM, CM, QS, FS, BW, ST, TTW, TTM, SP, ST?, C, SB . SSB) =
          F F'
 if F := createForestOnceSubstituted(M, TM, CM, QS, FS, BW, ST, TTW, TTM, SP,
                                              ST?, substitute(M, C, SB)) /\
    F' := substituteAndCreateForest(M, TM, CM, QS, FS, BW, ST, TTW, TTM, SP, ST?, C, SSB) .
```

The auxiliary function `createForestOnceSubstituted` distinguishes between the different kinds of atomic condition:

- In matching conditions only the forest for the reduction of the term in the righthand side to its normal form is computed:

```
op createForestOnceSubstituted : Module Module Maybe{Module} QidSet QidSet Bound
                                 SearchType TreeType TreeType Bool Bool Condition
                                 -> Forest .
eq createForestOnceSubstituted(M, TM, CM, QS, FS, BW, ST, TTW, TTM, SP, ST?, T := T') =
      createForest(M, TM, CM, T', getTerm(metaReduce(M, T')), QS, ST?) .
```

- When the atomic condition is an equality the forest for the reduction to normal form of both terms is generated:

```
eq createForestOnceSubstituted(M, TM, CM, QS, FS, BW, ST, TTW, TTM, SP, ST?, T = T') =
        createForest(M, TM, CM, T, getTerm(metaReduce(M, T)), QS, ST?)
        createForest(M, TM, CM, T', getTerm(metaReduce(M, T')), QS, ST?) .
```

- In membership conditions the forest for the reduction of the term to normal form and for the inference of its least sort are always computed. If this least sort does not fulfill the condition a new node stating that it is the least sort is created; otherwise, only the forest is returned:

```
ceq createForestOnceSubstituted(M, TM, CM, QS, FS, BW, ST, TTW, TTM, SP, ST?, T : Ty) =
                  createForest(M, TM, CM, T, T', QS, ST?)
                  if sortLeq(M, Ty', Ty)
                  then F
                  else tree(T :s Ty', 1 + getOffspring*(F), F)
                  fi
 if RP := metaReduce(M, T) /\
    T' := getTerm(RP) /\
    Ty' := getType(RP) /\
    F := createForest(M, TM, CM, T', Ty', QS, ST?) .
```

- Finally, rewrite conditions generate a forest of missing answers with the pattern of the condition, `nil` as condition, `zeroOrMore` steps as type of search, and `false` as last argument, indicating that the current search corresponds to a rewrite condition:

```
ceq createForestOnceSubstituted(OM, TM, CM, QS, FS, BW, ST, TTW, TTM, SP, ST?, T => T') =
          createForest(OM, TM, CM, T, T'', QS, ST?)
          createMissingForest(OM, TM, CM, T'', T', nil, BW, getBound(OM, T, T', nil, 0),
                              zeroOrMore, TTW, TTM, QS, FS, true, SP, false)
 if T'' := getTerm(metaReduce(OM, T)) .
```

The function `oneStepCongruence` is in charge of rewriting the subterms of the current term. It uses an auxiliary function `oneStepCongruence*` that traverses the subterms taking into account the `frozen` attribute (obtained with the function `getFrozen`), that prevents some arguments from being rewritten:

```
op oneStepCongruence : Module Module Maybe{Module} Term QidSet QidSet Bound SearchType
                       TreeType TreeType Bool -> Forest .

ceq oneStepCongruence(OM, TM, CM, Q[TL], QS, FS, BW, ST, TTW, TTM, SP) = F
 if F := oneStepCongruence*(OM, TM, CM, TL, QS, FS, BW, ST, TTW, TTM, SP,
                            getFrozen(OM, getOps(OM), Q[TL]), 1) .
eq oneStepCongruence(OM, TM, CM, T, QS, FS, BW, ST, TTW, TTM, SP) = mtForest [owise] .
```

`oneStepCongruence*` iterates over a list of terms, applying `oneStepMissingTree` to those that are not frozen:

```
op oneStepCongruence* : Module Module Maybe{Module} TermList QidSet QidSet Bound SearchType
                        TreeType TreeType Bool NatList Nat -> Forest .
eq oneStepCongruence*(OM, TM, CM, (T, TL), QS, FS, BW, ST, TTW, TTM, SP, NL, N) =
              if in?(N, NL)
              then mtForest
              else oneStepMissingTree(OM, TM, CM, T, QS, FS, BW, ST, TTW, TTM, SP)
              fi
              oneStepCongruence*(OM, TM, CM, TL, QS, FS, BW, ST, TTW, TTM, SP, NL, s(N)) .
eq oneStepCongruence*(OM, TM, CM, empty, QS, FS, BW, ST, TTW, TTM, SP, NL, N) = mtForest .
```

The function `combineSubterms` replaces the subterms of a given term by the terms obtained by rewriting one step each of them. It uses an auxiliary function `combineSubtermsAux` with the list of arguments already used and the list of arguments to be rewritten as new parameters:

```
op combineSubterms : Module Term Forest -> TermList .
eq combineSubterms(M, Q[TL], F) = combineSubtermsAux(M, Q, F, empty, TL) .
eq combineSubterms(M, T, F) = empty [owise] .
```

The terms to be substituted are traversed with the function `combineSubtermsAux`, that extracts the terms rewritten one step from the forest with the function `getPossibleTerms` and then creates the new terms with `createNewTerms`.

```
op combineSubtermsAux : Module Qid Forest TermList TermList -> TermList .
eq combineSubtermsAux(M, Q, F, TL, empty) = empty .
ceq combineSubtermsAux(M, Q, F, TL, (T, TL')) = createNewTerms(M, Q, TL, TL', TL''),
                                                combineSubtermsAux(M, Q, F, (TL, T), TL')
 if TL'' := getPossibleTerms(F, T) .
```

where `createNewTerms` replaces the current subterm with each reachable term from it until no more terms are availabe:

```
op createNewTerms : Module Qid TermList TermList TermList -> TermList .
eq createNewTerms(M, Q, TL, TL', empty) = empty .
eq createNewTerms(M, Q, TL, TL', (T, TL'')) = createNewTerms(M, Q, TL, TL', TL''),
                                               getTerm(metaReduce(M, Q[TL, T, TL'])) .
```

and `getPossibleTerms` traverses a forest extracting the terms obtained from the given term in one step:

```
op getPossibleTerms : Forest Term -> TermList .
eq getPossibleTerms(mtForest, T) = empty .
eq getPossibleTerms(tree(T =>1 {TL}, N, F) F', T) = TL .
eq getPossibleTerms(A F, T) = getPossibleTerms(F, T) [owise] .
```

We explain now the auxiliary functions used in the module:

- The function `clean` deletes the repeated occurrences from a list of terms:

  ```
  op clean : TermList -> TermList .
  eq clean((TL, T, TL', T, TL'')) = clean((TL, T, TL', TL'')) .
  eq clean(TL) = TL [owise] .
  ```

- The function `extractTerms` obtains all the reachable terms from a given forest:

```
op extractTerms : Forest -> TermList .
eq extractTerms(mtForest) = empty .
eq extractTerms(tree(T ~>[B] {TL} s.t. PAT & C [FST], N, F) F') = TL, extractTerms(F') .
eq extractTerms(tree((sol(T, PAT, C, true, EXP, FST)), N, F) F') = T, extractTerms(F') .
eq extractTerms(A F) = extractTerms(F) [owise] .
```

- The function `getBound` computes the number of rewrites needed to obtain all the reachable terms from an initial term by using the predefined `metaSearch` function until no more terms are found:

```
op getBound : Module Term Term Trace Nat -> Nat .
ceq getBound(M, T, T', TR, N) = getBound(M, T, T', TR', s(N))
 if TR' := metaSearchPath(M, T, T', nil, '+, unbounded, N) .
eq getBound(M, T, T', TR, N) = length(TR) + 1 [owise] .
```

  where `length` computes the size of a trace:

```
op length : Trace -> Nat .
eq length(nil) = 0 .
eq length(TRS TR) = s(length(TR)) .
```

- Given a condition, the function `getNFirst` extracts its first `N` atomic conditions:

```
op getNFirst : Condition Nat -> Condition .
eq getNFirst(T = T' /\ C, s(N)) = T = T' /\ getNFirst(C, N) .
eq getNFirst(T := T' /\ C, s(N)) = T := T' /\ getNFirst(C, N) .
eq getNFirst(T => T' /\ C, s(N)) = T => T' /\ getNFirst(C, N) .
eq getNFirst(T : Ty  /\ C, s(N)) = T : Ty /\ getNFirst(C, N) .
eq getNFirst(C, N) = nil [owise] .
```

  `getLast` returns the last atomic condition:

```
op getLast : Condition -> Condition .
eq getLast(C /\ T = T') = T = T' .
eq getLast(C /\ T := T') = T := T' .
eq getLast(C /\ T => T') = T => T' .
eq getLast(C /\ T : Ty) = T : Ty .
eq getLast(nil) = nil .
```

  and `getNumConds` computes the number of atomic conditions:

```
op getNumConds : Condition -> Nat .
eq getNumConds(nil) = 0 .
eq getNumConds(T = T' /\ C) = s(getNumConds(C)) .
eq getNumConds(T := T' /\ C) = s(getNumConds(C)) .
eq getNumConds(T => T' /\ C) = s(getNumConds(C)) .
eq getNumConds(T : Ty  /\ C) = s(getNumConds(C)) .
```

- The function `cleanTree*` receives a Boolean value indicating whether the final mode is active, the set of final sorts and a forest, and deletes the repetitions in the forest and then traverses it applying `cleanTree` to each tree:

```
op cleanTree* : Module Bool QidSet Forest -> Forest .
eq cleanTree*(M, BFS, FS, mtForest) = mtForest .
eq cleanTree*(M, BFS, FS, F A F' A F'') = cleanTree*(M, BFS, FS, F A F' F'') .
eq cleanTree*(M, BFS, FS, A F) = cleanTree(M, BFS, FS, A)
                                 cleanTree*(M, BFS, FS, F) [owise] .
```

- `cleanTree` is in charge of pruning the tree. If the current node corresponds to a final inference (i.e., the set of reachable terms in one step is `empty`) it checks, with the functions `final?` and `finalSorts?` respectively, if it has been trusted with the attribute `metadata` or its sort has been introduced as final. If the inference has been trusted the whole subtree is removed; otherwise the function is applied to the forest:

```
op cleanTree : Module Bool QidSet Tree -> Forest .
ceq cleanTree(M, true, FS, tree(T =>1 {empty}, N, F)) =
     if final?(M, getOps(M), T) or-else finalSorts?(M, T, FS)
     then mtForest
     else tree(T =>1 {empty}, 1 + getOffspring*(F'), F')
     fi
 if F' := cleanTree*(M, true, FS, F) .
```

This function also deletes the nodes related to inferences of solutions of rewrite conditions, that only depend on the pattern matching:

```
eq cleanTree(M, BFS, FS, tree(sol(T, PAT, nil, SC, EXP, false), N, F)) =
                                                    cleanTree*(M, BFS, FS, F) .
```

Otherwise the inference is kept and the function `cleanTree*` is applied to the forest:

```
ceq cleanTree(M, BFS, FS, tree(I:Inference, N, F)) =
                               tree(I:Inference, 1 + getOffspring*(F'), F')
 if F' := cleanTree*(M, BFS, FS, F) [owise] .
```

- The function `final?` checks whether the operator at the top of the given term has the value `final` in the `metadata` attribute, using the auxiliary functions `noIter` to obtain the operator without the `iter` attribute and `createAssocTypeList` to create a list of parameters without flattening due to the `assoc` attribute:

```
op final? : Module OpDeclSet Term -> Bool .
ceq final?(M, op Q : nil -> Ty [AtS metadata("final")] . ODS, Ct) = true
 if Q == getName(Ct) .
ceq final?(M, op Q : Ty -> Ty' [AtS metadata("final")] . ODS, Q'[T]) = true
 if Q[Q''[T]] := noIter(Q'[T]) /\
    checkTypes(T, Ty, M) .
ceq final?(M, op Q : TyL -> Ty [AtS metadata("final")] . ODS, Q[TL]) = true
 if checkTypes(TL, createAssocTypeList(TyL, AtS, size(TL)), M) .
eq final?(M, ODS, T) = false [owise] .
```

- `finalSorts?` checks with the predefined function `wellFormed` and the module with only constructor operators, obtained with `onlyCtors`, that the term is a constructed term and then it traverses the list of sorts checking if any of them is the sort of the current term with the function `finalSort?`:

```
op finalSorts? : Module Term QidSet -> Bool .
ceq finalSorts?(M, T, QS) = false
 if not wellFormed(onlyCtors(M), T) .
eq finalSorts?(M, T, none) = false .
eq finalSorts?(M, T, Q ; QS) = finalSort?(M, T, Q) or-else
                               finalSorts?(M, T, QS) [owise] .
```

- `finalSort?` computes the least sort of the current term with the predefined function `leastSort` and then computes that this sort is less than or equal to the current one with `sortLeq`:

```
op finalSort? : Module Term Type -> Bool .
ceq finalSort?(M, T, Ty) = true
 if Ty' := leastSort(M, T) /\
    sortLeq(M, Ty', Ty) .
eq finalSort?(M, T, Ty) = false [owise] .
```

- We extract the list of frozen arguments with `getFrozen`, that takes into account the `iter` and `assoc` attributes:

```
op getFrozen : Module OpDeclSet Term -> NatList .
ceq getFrozen(M, op Q : Ty -> Ty' [AtS frozen(NL)] . ODS, Q'[T]) = NL
 if Q[Q''[T]] := noIter(Q'[T]) /\
    checkTypes(T, Ty, M) .
ceq getFrozen(M, op Q : TyL -> Ty [AtS frozen(NL)] . ODS, Q[TL]) = NL
 if checkTypes(TL, createAssocTypeList(TyL, AtS, size(TL)), M) .
eq getFrozen(M, ODS, T) = nil [owise] .
```

- We use the function `pruneFinalSort` to prune the debugging tree when a sort is considered final on the fly. It traverses the tree and uses `finalSort?` to find the nodes that have to be deleted:

```
op pruneFinalSort : Module Type Tree -> Tree .
ceq pruneFinalSort(M, Ty, tree(T =>1 {empty}, N, F)) =
     if finalSort?(M, T, Ty)
     then mtForest
     else tree(T =>1 {empty}, 1 + getOffspring*(F'), F')
     fi
 if F' := pruneFinalSort*(M, Ty, F) [owise] .
ceq pruneFinalSort(M, Ty, tree(I:Inference, N, F)) =
        tree(I:Inference, 1 + getOffspring*(F'), F')
 if F' := pruneFinalSort*(M, Ty, F) [owise] .
```

- The function `pruneFinalSort*` traverses a forest applying `pruneFinalSort` to each tree:

```
op pruneFinalSort* : Module Type Forest -> Forest .
eq pruneFinalSort*(M, Ty, mtForest) = mtForest .
eq pruneFinalSort*(M, Ty, A F) = pruneFinalSort(M, Ty, A)
                                 pruneFinalSort*(M, Ty, F) [owise] .
endfm
```

5.3 The debugger environment

We implement our system on top of Full Maude, that extends Maude with support for object-oriented specification and advanced module operations [8, 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 missing_. : @Bubble@ -> @Command@ .
 op top-down`strategy`. : -> @Command@ .
 op divide-query`strategy`. : -> @Command@ .
 op one-step`tree`. : -> @Command@ .
 op many-steps`tree`. : -> @Command@ .
 op one-step`missing`tree`. : -> @Command@ .
 op many-steps`missing`tree`. : -> @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-include_. : @Bubble@ -> @Command@ .
 op debug-exclude_. : @Bubble@ -> @Command@ .
 op debug-select_. : @NeTokenList@ -> @Command@ .
 op debug-deselect_. : @NeTokenList@ -> @Command@ .
 op solutions`prioritized`on`. : -> @Command@ .
 op solutions`prioritized`off`. : -> @Command@ .
```

```
  op set‘final‘select‘on‘. : -> @Command@ .
  op set‘final‘select‘off‘. : -> @Command@ .
  op final‘select_. : @Bubble@ -> @Command@ .
  op final‘deselect_. : @Bubble@ -> @Command@ .
  op yes‘. : -> @Command@ .
  op no‘. : -> @Command@ .
  op trust‘. : -> @Command@ .
  op its‘sort‘is‘final‘. : -> @Command@ .
  op _is‘wrong‘. : @Token@ -> @Command@ .
  op _is‘not‘a‘solution‘. : @Token@ -> @Command@ .
  op _:‘yes‘. : @Token@ -> @Command@ .
  op _:‘no‘. : @Token@ -> @Command@ .
  op _:‘don’t‘know‘. : @Token@ -> @Command@ .
  op _:‘trust‘. : @Token@ -> @Command@ .
  op _:‘its‘sort‘is‘final‘. : @Token@ -> @Command@ .
  op _:_is‘wrong‘. : @Token@ @Token@ -> @Command@ .
  op _:_is‘not‘a‘solution‘. : @Token@ @Token@ -> @Command@ .
  op all‘:‘yes‘. : -> @Command@ .
  op don’t‘know‘. : -> @Command@ .
  op undo‘. : -> @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, final sorts, and the debugging commands:

```
fmod DD-COMMAND-PROCESSING is
  pr COMMAND-PROCESSING .
  pr META-DD-SIGN .
  pr MISSING-ANSWERS-TREE .
  pr SEARCH-TYPE .
  pr PRINT .
```

The parsing of the debugging command for missing answers returns a term of sort `MissTuple`, that contains the generated tree, the module where the debugging takes place, the type of search, the pattern used in the search, the condition, the set of suspicious statements, and a list of quoted identifiers with an error message:

```
  sort MissTuple .
  op <_,_,_,_,_,_,_> : Forest Maybe{Module} SearchType Maybe{Term} Condition QidSet QidList
                       -> MissTuple .
```

The function `procMissing` receives a bubble with the term to be parsed, a correct module (possibly `undefMod`), a Boolean indicating whether selection mode for suspicious labels is activated, the set of suspicious labels, the selected type of tree (one-step or many-steps), the bound of the search in the correct module, the default module, and the Full Maude's database of modules, and returns a value of sort `MissTuple`, using to build the tree the function `createMissingTree` shown in the previous section:

```
  op procMissing : Term ModuleExpression Maybe{Module} TreeType TreeType Bool QidSet Bool
                   QidSet Bool Bound Database -> MissTuple .
  ...
endfm
```

The persistent state of Full Maude's system is given by a single object of 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] .
```

In addition to the attributes defined for this class in [21], we add the following ones to deal with missing answers:

- the type of tree when debugging missing answers; it takes the value `os` when the one-step tree is selected and `ms` when the many-steps tree is chosen:

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

- the type of tree currently used in the debugging of missing answers is kept in `currentTTM` in order to use it when the tree is expanded on demand:

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

- a Boolean value that indicates whether the selection of final sorts is enabled:

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

- the attribute `currentFS` keeps if the option to delete final sorts is activated in the current session, in order to use it again to build trees on demand:

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

- the set of final sorts:

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

- the set of final sorts specified for the current debugging session are kept in `currentFinal` to use them when a tree is expanded on demand:

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

- the `pattern` that must be matched by the terms to be valid solutions:

```
op pattern :_ : Maybe{Term} -> Attribute .
```

- the `condition` that must be fulfilled by the terms to be valid solutions:

```
op condition :_ : Condition -> Attribute .
```

- a Boolean value indicating whether the questions about solutions are prioritized:

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

- we keep in `currentSP` the value about the prioritization of the questions about solutions to use it when the tree is expanded on demand:

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

- the `searchType` used for the debugging of missing answers, that can take the values `zeroOrMore`, `oneOrMore`, and `final`:

```
op searchType :_ : SearchType -> Attribute .
```

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 dealing with missing answers.

The rule `missing` is in charge of parsing the initial command for the debugging of missing answers. If the parsing is correct, that is, if the error message obtained with the function `procMissing` is `nil`, the tuple provides the new debugging tree, the current module, the search type, the pattern, the search condition, and the suspicious labels. All these values are kept in the appropriate attributes in order to reuse them later if trees on demand are built. If the parsing fails, the attributes are not updated and the command is discarded:

```
crl [missing] :
    < O : DDDC | db : DB, input : ('missing_.[T]), output : nil,
                 default : ME, tree : F, module : MM, correction : MM',
                 previousStates : LS, answers : LA, state : TS, treeType : TT,
                 currentTTW : CTTW, bound : BND, treeTypeMissing : TT',
                 currentTTM : CTT, select : B, suspicious : QS, currentSuspicious : QS',
                 condition : C, pattern : MT, searchType : MST, solutionsPrioritized : SP,
                 currentSP : CSP, finalSorts : FS, currentFinal : CFS,
                 finalSelect : BFS, currentFS : BFS', 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, currentTTW : TT,
                 bound : BND, treeTypeMissing : TT', currentTTM : TT', select : B,
                 suspicious : QS, currentSuspicious : QS'', condition : C', pattern : PAT,
                 searchType : MST', solutionsPrioritized : SP, currentSP : SP,
                 finalSorts : FS, currentFinal : FS, finalSelect : BFS,
                 currentFS : BFS, AtS >
    else
    < O : DDDC | db : DB, input : nilTermList, output : QIL, default : ME, tree : F,
                 module : MM, correction : MM', previousStates : LS, answers : nil,
                 state : TS, treeType : TT, currentTTW : CTTW, bound : BND,
                 treeTypeMissing : TT', currentTTM : CTT, select : B, suspicious : QS,
                 currentSuspicious : QS', condition : C, pattern : MT, searchType : MST,
                 solutionsPrioritized : SP, currentSP : CSP, finalSorts : FS,
                 currentFinal : CFS, finalSelect : BFS, currentFS : BFS', AtS >
    fi
 if < F?, MM'', MST', PAT, C', QS'', QIL > :=
                      procMissing(T, ME, MM', TT, TT', B, QS, BFS, FS, SP, BND, DB) .
```

While using the top-down strategy, if the user introduces a command indicating that a term in one of the questions is reachable but it is not a solution, the rule `top-down-traversal-sol` processes it and updates the tree if the arguments are correct. First, it parses the introduced arguments with the function `downNat*` to check that they are natural numbers and then checks that the question referred by the first argument exists. In this case, it checks that the question is related to an inference of a set of solutions with `solutionsInference?` and that the index introduced as second argument corresponds with one of these solutions by using the auxiliary function `numTermsInRootSet`, that computes the number of terms in the inference in the root of the tree given as argument. If all the conditions hold, a new tree is computed with `conditionForest`:

```
crl [top-down-traversal-sol] :
    < O : DDDC | input : ('_:_is‘not‘a‘solution‘.['token[T'],'token[T]]),
```

```
                strategy : td, tree : PT, currentSuspicious : QS, treeType : TT,
                bound : BND, module : M, correction : MM, previousStates : LS,
                answers : LA, condition : C, pattern : PAT, state : waiting, AtS >
 => < O : DDDC | input : nilTermList, strategy : td,
                tree : tree(sol(T2, PAT, C, true, true, true), 1 + getOffspring*(F), F),
                currentSuspicious : QS, treeType : TT, bound : BND,
                module : M, correction : MM, previousStates : LS < nil, PT, td >,
                answers : LA getAnswer(PT, wrong),
                condition : C, pattern : PAT, state : computing, AtS >
 if *** We check the values introduced are numbers
    N := downNat*(T) /\
    N' := downNat*(T') /\
    *** We check the question selected exists
    N' <= size(getForest(PT, nil)) /\
    N' > 0 /\
    N'' := sd(N', 1) /\
    *** We check the question selected has a set of solutions as result of the inference
    solutionsInference?(getContents(PT, N'')) /\
    *** We check the term selected exists
    N <= numTermsInRootSet(getSubTree(PT, N'')) /\
    N > 0 /\
    *** We obtain the term that is not a solution
    T2 := getWrongTerm(getSubTree(PT, N''), N) /\
    *** We create the new tree
    F := conditionForest(substitute(M, C, metaMatch(M, PAT, T2, C, 0)),
                                      M, deleteSuspicious(M, QS), MM, QS, BND, TT) .
```

The rule `missing-wrong` is used when, using the divide and query strategy, the user points out that a certain term is not reachable. The rule checks that the current question is related to an inference of a set of terms with `setInference?` and that the selected question points to one of these terms, and then creates the debugging tree for wrong answers with `createRewTree`:

```
crl [missing-wrong] :
    < O : DDDC | input : ('_is`wrong`.['token[T]]), strategy : dq, tree : PT,
                current : NL, previousStates : LS, answers : LA, state : waiting,
                currentSuspicious : QS, bound : BND, module : M, correction : MM,
                currentTTW : TT, AtS >
 => < O : DDDC | input : nilTermList, strategy : dq, tree : PT',
                current : NL, previousStates : LS < NL, PT, dq >,
                answers : LA getAnswer(getSubTree(PT, NL), wrong),
                state : computing, currentSuspicious : QS, bound : BND,
                module : M, correction : MM, currentTTW : TT, AtS >
 if N := downNat*(T) /\
    setInference?(getContents(PT, NL)) /\
    N > 0 /\
    N <= numTermsInRootSet(getSubTree(PT, NL)) /\
    T1 := getFirstTerm(getSubTree(PT, NL)) /\
    T2 := getWrongTerm(getSubTree(PT, NL), N) /\
    PT' := createRewTree(labeling(M), MM, T1, T2, QS, TT, BND) .
```

When the debugging of missing answers reaches a node that is frozen (i.e., it is built with the operator `sol` and its fifth argument is `false`), the debugger builds the associated tree. If the condition holds for the current term (the third argument is `true`), a tree for wrong answers is built with `conditionForest`:

```
crl [sol-true] :
    < O : DDDC | tree : tree(sol(T, T', C, true, false, true), N, F), module : M,
                correction : MM, state : computing, currentSuspicious : QS, bound : BND,
                currentTTW : TT, currentFinal : FS, currentFS : BFS, AtS >
 => < O : DDDC | tree : tree(sol(T, T', C, true, true, true), 1 + getOffspring*(F'), F'),
                module : M, correction : MM, state : computing, currentSuspicious : QS,
                bound : BND, currentTTW : TT, currentFinal : FS, currentFS : BFS, AtS >
 if F' := conditionForest(substitute(M, C, metaMatch(M, T', T, C, 0)), M,
                          deleteSuspicious(M, QS), MM, QS, BND, TT) .
```

while if the condition does not hold a tree for missing answers is computed by using the function `buildConditionForestMissingAux`:

```
crl [sol-false] :
    < O : DDDC | tree : tree(sol(T, T', C, false, false, true), N, F), module : M,
                 correction : MM, state : computing, currentSuspicious : QS, bound : BND,
                 currentTTW : TT, currentTTM : TT', searchType : MST, currentSP : SP,
                 currentFinal : FS, currentFS : BFS, AtS >
 => < O : DDDC | tree : tree(sol(T, T', C, false, true, true), 1 + getOffspring*(F'), F'),
                 module : M, correction : MM, state : computing, currentSuspicious : QS,
                 bound : BND, currentTTW : TT, currentTTM : TT', searchType : MST,
                 currentSP : SP, currentFinal : FS, currentFS : BFS, AtS >
 if M' := deleteSuspicious(M, QS) /\
    F' := cleanTree*(M, BFS, FS, createForest(M, M', MM, T,
                                    getTerm(metaReduce(M, T)), QS, strat?(M))
          first(buildConditionForestMissingAux(M, M', MM, QS, FS, T, T, BND, MST, TT,
                               TT', SP, T' := T /\ C, strat?(M),
                               allSubs(M, T, T' := T, 0, mtSSB), 2, getNumConds(C) + 1))) .
```

In the divide and query strategy, when the user introduces that the sort of a certain term is final on the fly the rule `sort-final` is applied. It checks that the question is related to final terms with the function `finalQuestion?` and then prunes all the tree with the function `pruneFinalSort`:

```
crl [sort-final] :
    < O : DDDC | input : ('its'sort'is'final'..@Command@), output : nil,
                 tree : PT, current : NL, module : M, state : waiting, AtS >
 => < O : DDDC | input : nilTermList, output : ('\n '\b 'Terms 'of 'sort '\o Ty
                                                  '\b 'are 'final. '\o '\n),
                 tree : PT', current : NL, module : M, state : computing, AtS >
 if finalQuestion?(getContents(PT, NL)) /\
    T := getFirstTerm(getSubTree(PT, NL)) /\
    Ty := getType(metaReduce(M, T)) /\
    PT' := pruneFinalSort(M, Ty, PT) .
```

The final selection mode is switched on and off with the rules `final-on` and `final-off` respectively, that update the `finalSelect` attribute:

```
rl [final-on] :
   < O : DDDC | input : ('set'final'select'on'..@Command@), output : nil,
                finalSelect : B, AtS >
=> < O : DDDC | input : nilTermList, finalSelect : true,
                output : ('\n '\b 'Final 'select 'is 'on. '\o '\n), AtS > .

rl [final-off] :
   < O : DDDC | input : ('set'final'select'off'..@Command@), output : nil,
                finalSelect : B, AtS >
=> < O : DDDC | input : nilTermList, finalSelect : false,
                output : ('\n '\b 'Final 'select 'is 'off. '\o '\n), AtS > .
```

Final sorts are introduced with the rule `select-final`, that uses an auxiliary function `procFinals` to generate the list of new sorts and an error message, that must be `nil` when the sorts are correct. These sorts are added to the former final sorts in the attribute `finalSorts`, and an advisory message is generated with `advisory-final` if the final mode is switched off:

```
crl [select-final] :
    < O : DDDC | input : ('final'select_.[T]), output : nil,
                 finalSelect : B, finalSorts : QS, AtS >
 => < O : DDDC | input : nilTermList, finalSelect : B,
                 output : (advisory-final(B) QIL includeMsgFinal(QS')),
                 finalSorts : (QS ; QS'), AtS >
 if < QS' : QIL > := procFinals(T) .

 ...
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 [8, Chapter 17], that consists of 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:

```
loop init-debug .
```

6 Conclusions and future work

In this paper we have presented a calculus for rewriting logic specifications that allows to compute all the reachable terms from an initial one given some constraints. Based in this calculus, we have developed the foundations of declarative debugging of missing answers, and we have applied them to implement a debugger for Maude modules. This debugger extends our previous work [4, 5, 24, 23, 21] on declarative debugging of wrong answers of Maude modules.

We have described formally how debugging trees can be built from the trees obtained with the new calculus, 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. This version of the tool extends the features from former versions, like trusting of statements or the construction of different kinds of tree, to the debugging of missing answers and adds specific characteristics for this kind of debugging, like the prioritization of some questions or the trusting of final terms.

In our opinion, since the intended use of most of the debuggers (including the existing debugging techniques for Maude, such as tracing and term coloring, as well as the previous versions of our debugger) is to debug wrong answers, our tool fills the gap between them and nondeterministic specifications like the ones in Maude system modules.

Finally, we have also developed a graphical user interface that eases the use of our command-line tool and allows free navigation of the debugging tree, see [22] for more information.

Future work will include the following:

- The current version of the tool allows the user to introduce a correct but maybe incomplete module in order to shorten the debugging session. We plan to add a new command to introduce complete modules, which would greatly reduce the number of questions asked to the user.
- Although the current version of our debugger detects missing answers due to a wide spectrum of reasons (missing rules, wrong statements, errors in the search condition, and membership errors) we plan to extend the detected errors to those caused by missing equations or memberships.
- We plan to add new navigation strategies, like the ones presented in [26], that take into account not the size of the subtrees but the number of different potential errors in them.

References

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

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

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

[4] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. Declarative debugging of membership equational logic specifications. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models. Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 174–193. Springer, 2008.

[5] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A declarative debugger for Maude functional modules. In G. Roşu, editor, *Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008*, volume 238(3) of *Electronic Notes in Theoretical Computer Science*, pages 63–81. Elsevier, 2009.

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

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

[8] 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.

[9] 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.

[10] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. North-Holland, 1990.

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

[12] I. MacLarty. Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne, 2005.

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

[14] 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.

[15] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[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] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative debugging of Maude modules. Technical Report SIC-6-08, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2008. `http://maude.sip.ucm.es/debugging`.

[22] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. A declarative debugger for Maude specifications - User guide. Technical Report SIC-7-09, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2009. `http://maude.sip.ucm.es/debugging`.

[23] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative debugging of rewriting logic specifications. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques (WADT 2008)*, volume 5486 of *Lecture Notes in Computer Science*, pages 308–325. Springer, 2009.

[24] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. A declarative debugger for Maude. In J. Meseguer and G. Roşu, editors, *Algebraic Methodology and Software Technology — 12th International Conference, AMAST 2008 Urbana, IL, USA, July 28-31, 2008 Proceedings*, volume 5140 of *Lecture Notes in Computer Science*, pages 116–121. Springer, 2008.

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

[26] 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.

[27] A. Tessier and G. Ferrand. Declarative diagnosis in the CLP scheme. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2000.

[28] A. Verdejo and N. Martí-Oliet. Two case studies of semantics execution in Maude: CCS and LOTOS. *Formal Methods in System Design*, 27:113–172, 2005.

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