Declarative Debugging of Rewriting Logic Specifications[*]

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

Technical Report 02/10

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

March 2010

[*]Research supported by MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

Abstract

Declarative debugging is a semi-automatic technique that starts from an incorrect computation and locates a program fragment responsible for the error by building a tree representing this computation and guiding the user through it to find the error. Membership equational logic (*MEL*) is an equational logic that in addition to equations allows to state membership axioms characterizing the elements of a sort. Rewriting logic is a logic of change that extends *MEL* by adding rewrite rules, that correspond to transitions between states and can be nondeterministic. We propose here a calculus to infer reductions, sort inferences, normal forms and least sorts with the equational part, and rewrites and sets of reachable terms through rules. We use an abbreviation of the proof trees computed with this calculus to build appropriate debugging trees for both wrong (an incorrect result obtained from an initial result) and missing answers (results that are erroneous because they are incomplete), whose adequacy for debugging is proved. Using these trees we have implemented a declarative debugger for Maude, a high-performance system based on rewriting logic, whose use is illustrated with an example.

Keywords: declarative debugging, rewriting logic, Maude, wrong answers, missing answers

Contents

1 Introduction

Declarative debugging [26], also known as declarative diagnosis or algorithmic debugging, is a debugging technique that abstracts the execution details, which may be difficult to follow in declarative languages, and focus on the results. We can distinguish between two different kinds of declarative debugging: debugging of *wrong answers*, that is applied when a *wrong* result is obtained from an initial value and has been widely employed in the logic [16, 28], functional [18, 19], multi-paradigm [4, 13], and object-oriented [5] programming languages; and debugging of *missing answers* [7, 1], applied when a result is *incomplete*, which has been less studied because the calculus involved is more complex than in the case of wrong answers. Declarative debugging starts from an incorrect computation, the error symptom, and locates the code (or the absence of code) responsible for the error. To find this error the debugger represents the computation as a *debugging tree* [17], where each node stands for a computation step and must follow from the results of its child nodes by some logical inference. This tree is traversed by asking questions to an external oracle (generally the user) until a *buggy node*—a node containing an erroneous result, but whose children are all correct—is found. Hence, we distinguish two phases in this scheme: the debugging tree *generation* and its *navigation* following some suitable strategy [27].

We present here a declarative debugger for *Maude specifications.* Maude [9] is a high-level language and high-performance system supporting both equational and rewriting logic computation. Maude modules correspond to specifications in *rewriting logic* [14], a logic that allows the representation of many models of concurrent and distributed systems. This logic is an extension of *membership equational logic* [2], an equational logic that, in addition to equations, allows to state *membership axioms* characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system and can be nondeterministic. The Maude system supports several approaches for debugging Maude programs: tracing, term coloring, and using an internal debugger [9, Chap. 22]. The tracing facilities allow us to follow the execution of a specification, that is, the sequence of applications of statements that take place. The same ideas have been applied to the functional paradigm by the tracer *Hat* [8], where a graph constructed by graph rewriting is proposed as suitable trace structure. Term coloring consists in printing with different colors the operators used to build a term that does not fully reduce. Finally, the Maude internal debugger allows to define break points in the execution by selecting some operators or statements. When a break point is found the debugger is entered, where we can see the current term and execute the next rewrite with tracing turned on. However, these tools have the disadvantages that they are supposed to be used only when a wrong result *is found*; and both the trace and the Maude debugger (that is based on the trace) show the statements applied in the order in which they are executed and thus the user can lose the general view of the *proof* of the incorrect computation that produced the wrong result.

Declarative debugging of wrong answers of membership equation logic specifications was studied in [6], and was later extended to debugging of wrong answers in system modules in [22], while descriptions of the whole system can be found in [25, 20], where we present how to debug wrong results due to errors in the statements of the specification. In [24] we investigated how to apply declarative debugging of missing answers, traditionally attached to nondeterministic frameworks, to membership equation logic specifications. We achieve this it by broadening the concept of missing answers to deal with erroneous normal forms and least sorts. Finally, we extended the calculus developed thus far in [23] to debug missing answers in rewriting specifications, that is, expected results that the specification is not able to compute.

One of the main points of our approach is that, unlike other proposals like [7], it combines the treatment of wrong and missing answers and, moreover, allows to detect missing answers due to both missing rules and wrong statements. The state of the art can be found in [27], where different algorithmic debuggers are compared and that will include our debugger in its next version. Roughly speaking, our debugger has the pros of building different kinds of debugging trees (one-step and many-steps, a novelty in the declarative debugging world), applying the missing answers technique to debug normal forms and least sorts,[1] following a similar approach to [1], and only it and DDT [4] implement the Hirunkitti's divide and query navigation strategy, provide a graphical interface, and debugs missing answers; as cons, we do not allow answers like "maybe yes," "maybe not," and "inadmissible," and do not perform tree compression [27]. However, these features have recently been introduced in specific debuggers, and we expect to implement them in our debugger soon. Finally, some of the features shared by most of the debuggers are: the trees are abbreviated in order to shorten and ease the debugging process (in our case,

[1] Although the least sort error can be seen as a Maude-directed problem, normal forms are a common feature in several programming languages.

since we obtain the trees from a formal calculus, we are able to prove the correctness and completeness of the technique), which mitigates the main problem of declarative debugging, the complexity of the questions asked to the user; trusting of statements; `undo` and `don't know` commands; and different strategies to traverse the tree. We refer to [27, 21] for the meaning of these concepts. With respect to other approaches, such as the Maude sufficient completeness checker [9, Chap. 21] or the sets of descendants [12], our tool provides a wider approach, since we handle conditional statements and our equations are not required to be left-linear.

Exploiting the fact that rewriting logic is *reflective* [10], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [9, Chap. 14], a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. Therefore, we are able to generate and navigate the debugging tree of a Maude computation using operations in Maude itself. In addition, the Maude system provides another module, `LOOP-MODE` [9, Chap. 17], which can be used to specify input/output interactions with the user. However, instead of using this module directly, we extend Full Maude [9, Chap. 18], that includes features for parsing, evaluating, and pretty-printing terms, improving the input/output interaction. Moreover, Full Maude allows the specification of concurrent object-oriented systems, that can also be debugged. Thus, our declarative debugger, including its user interactions, is implemented in Maude itself.

The rest of the paper is structured as follows. Section 2 presents the preliminaries of our debugging approach. Section 3 describes our calculus while the next section explains how to transform the proof trees build with this calculus into appropriate debugging trees. Section 5 shows how to use the debugger, while Section 6 illustrates it with an example. Finally, Section 7 outlines the code of our tool and Section 8 concludes and presents some future work.

Additional examples, the source code of the tool, and other papers on the subject, including the user guide [21], where a graphical user interface for the debugger is presented, are all available from the webpage `http://maude.sip.ucm.es/debugging`.

2 Preliminaries

In the following sections we present both membership equational logic and rewriting logic, and how their specifications are represented as Maude modules. Then, we state the assumptions made on those specifications.

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 [2, 15].

Since the membership equational logic specifications that we consider are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness [9], their equations $t = t'$ can be oriented from left to right, $t \to t'$. Such a statement holds in an algebra, denoted $\mathcal{A}, \sigma \models t \to t'$, 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 [9, Chapter 4], introduced with syntax `fmod ... endfm`, are executable membership equational logic 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[2]

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

[2] 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.

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 [14] (see also [3]).

Models of rewrite theories are called *$\mathcal{R}$-systems* in [14]. Such systems are defined as categories that possess a (Σ, E)-algebra structure, together with a natural transformation for each rule in the set R. More intuitively, the idea is that we have a (Σ, E)-algebra, as described in Section 2.1, with transitions between the elements in each set A_k; moreover, these transitions must satisfy several additional requirements, including that there are identity transitions for each element, that transitions can be sequentially composed, that the operations in the signature Σ are also appropriately defined for the transitions, and that we have enough transitions corresponding to the rules in R. The rewriting logic deduction rules introduced in [14] are sound and complete with respect to this notion of model. Moreover, they can be used to build initial 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 [14, 11].

However, for our purposes in this paper, we are interested in a subclass of rewriting logic models [14] that we call *term models*, where the syntactic structure of terms is kept and associated notions such as variables, substitutions, and term rewriting make sense. These models will be used in the next section to represent the *intended interpretation* that the user had in mind while writing a specification. Since we want to find the discrepancies between the intended model and the initial model of the specification as written, we need to consider the relationship between a specification defined by a set of equations E and a set of rules R, and a model defined by possibly different sets of equations E' and of rules R'; in particular, when $E' = E$ and $R' = R$, the term model coincides with the initial model built in [14].

Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, with Σ a signature, E a set of equations, and R a set of rules, 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 rewriting is considered on equivalence classes [14, 11]. 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';
- 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 mentioned above, when $E' = E$ and $R' = R$ the term model coincides with the initial model for $\mathcal{R}$.

2.4 Maude system modules

Maude system modules [9, Chapter 6], introduced with syntax `mod ... endm`, are executable rewrite theories and their semantics is given by the initial system in the class of systems corresponding to the rewrite theory. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`), 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 [9] for details). Furthermore, the strategy followed by Maude in rewriting with rules is to compute the normal form of a term with respect to the equations before applying a rule. This strategy is guaranteed not to miss any rewrites when the rules are *coherent* with respect to the equations [29, 9]. In a way quite analogous to confluence, this coherence requirement means that, given a term t, for each rewrite of it using a rule in R to some term t', if u is the normal form of t with respect to the equations and memberships in E, then there is a rewrite of u with some rule in R to a term u' such that $u' =_E t'$.

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 paths to 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 .
```

Terms of sort Pos have the form [X,Y], where X and Y are natural numbers, and lists are built with nil and the juxtaposition operator __:

```
  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] .
```

The predicate isSol checks whether a list is a solution in a 5 × 5 labyrinth:

```
  vars X Y : Nat .
  var P Q : Pos .
  var L : List .
  op isSol : List -> Bool .
  eq [is1] : isSol(L [5,5]) = true .
  eq [is2] : isSol(L) = false [owise] .
```

The next position is computed with rule expand, that extends the solution with a new position by rewriting next(L) to obtain a new position and then checking whether this list is correct with isOk. Note that the choice of the next position, that could be initially wrong, 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 [n1] : next(L [X,Y]) => [X, Y + 1] .
  rl [n2] : next(L [X,Y]) => [sd(X, 1), Y] .
  rl [n3] : next(L [X,Y]) => [X, sd(Y, 1)] .
```

where sd denotes symmetric difference on natural numbers.

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 <= 5 and Y <= 5
           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] [3,2] [2,3] [4,3] [5,3] [1,5] [2,5] [3,5] [4,5] .
endm)
```

Now, we can use the module to search the labyrinth's exit from the position `[1,1]` with the Maude command `search`, but it cannot find any path to escape. We will see in Section 5 how to debug this specification.

```
Maude> (search {[1,1]} =>* {L:List} s.t. isSol(L:List) .)
No solution.
```

2.6 Assumptions

Since we are debugging Maude modules, they are expected to satisfy the appropriate executability requirements indicated in the previous sections. Namely, the specifications in functional modules have to be terminating, confluent, sort decreasing and, given an equation $t_1 = t_2$ *if* $C_1 \wedge \ldots \wedge C_n$, all the variables occurring in t_2 and $C_1 \ldots C_n$ must appear in t_1 or become instantiated by matching [9, 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 $t_1 \Rightarrow t_2$ *if* $C_1 \wedge \ldots \wedge C_n$, 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 [9, Section 6.3].

One interesting feature of our tool is that the user can 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 constructed terms (terms built only with constructors, indicated with the attribute `ctor`, and also known as data terms in other contexts) are considered to be in normal form, and some of these constructed terms can be pointed out as "final" (they cannot be further rewritten). 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 signature is supposed to be correct and will not be considered during the debugging process.

3 A calculus for debugging

Now we will describe debugging trees for both wrong and missing answers. First, Section 3.1 presents a calculus to deduce reductions, memberships and rewrites. We will extend this calculus in Section 3.2 to describe a calculus to compute normal forms, least sorts and sets of reachable terms. From now on, we assume a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ satisfying the assumptions stated in the previous section.

3.1 A calculus for wrong answers

We show here a calculus to deduce judgments for reductions $e \to e'$, memberships $e : s$, and rewrites $e \Rightarrow e'$. The inference rules for this calculus, shown in Figure 1, are an adaptation of the rules presented in [2, 15] for membership equational logic and in [14, 3] for rewriting logic. Remember that the notation $\theta(u_i) \downarrow \theta(u'_i)$ is an abbreviation of $\exists t_i . \theta(u_i) \to t_i \wedge \theta(u'_i) \to t_i$. As usual, we represent deductions in the calculus as *proof trees*, where the premises are the child nodes of the conclusion at each inference step. We assume that the inference labels ($Rep_{\Rightarrow}$), ($Rep_{\to}$), and (Mb) decorating the inference steps contain information about the particular rewrite rule, equation, and membership axiom, respectively, applied

(**Reflexivity**)

$$\frac{}{e \Rightarrow e}\ \mathsf{Rf}_{\Rightarrow} \qquad\qquad \frac{}{e \rightarrow e}\ \mathsf{Rf}_{\rightarrow}$$

(**Transitivity**)

$$\frac{e_1 \Rightarrow e' \qquad e' \Rightarrow e_2}{e_1 \Rightarrow e_2}\ \mathsf{Tr}_{\Rightarrow} \qquad\qquad \frac{e_1 \rightarrow e' \qquad e' \rightarrow e_2}{e_1 \rightarrow e_2}\ \mathsf{Tr}_{\rightarrow}$$

(**Congruence**)

$$\frac{e_1 \Rightarrow e'_1 \quad \ldots \quad e_n \Rightarrow e'_n}{f(e_1, \ldots, e_n) \Rightarrow f(e'_1, \ldots, e'_n)}\ \mathsf{Cong}_{\Rightarrow} \qquad \frac{e_1 \rightarrow e'_1 \quad \ldots \quad e_n \rightarrow e'_n}{f(e_1, \ldots, e_n) \rightarrow f(e'_1, \ldots, e'_n)}\ \mathsf{Cong}_{\rightarrow}$$

(**Replacement**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n}\ \{\theta(v_j) : s_j\}_{j=1}^{m}\ \{\theta(w_k) \Rightarrow \theta(w'_k)\}_{k=1}^{l}}{\theta(e) \Rightarrow \theta(e')}\ \mathsf{Rep}_{\Rightarrow}$$

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

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

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

(**Equivalence Class**) (**Subject Reduction**)

$$\frac{e \rightarrow e' \quad e' \Rightarrow e'' \quad e'' \rightarrow e'''}{e \Rightarrow e'''}\ \mathsf{EC} \qquad\qquad \frac{e \rightarrow e' \quad e' : s}{e : s}\ \mathsf{SRed}$$

(**Membership**)

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

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

Figure 1: Semantic calculus for Maude modules

during the inference. This information will be used by the debugger in order to present to the user the incorrect fragment of code causing the error.

For example, we can try to build the proof tree for the following reduction:

```
Maude> (red isOk([1,1][1,2]) .)
result Bool : true
```

Figure 2 depicts the proof tree associated associated to this reduction, where `c` stands for `contains`, `t` for `true`, `f` for `false`, *rhs* for `1 >= 1 and 2 >= 1 and 1 <= 5 and 2 <= 5 and not(c(L,[X,Y])) and not(c(wall,[X,Y]))`, t_1 for `if [1,1] == [1,2] then t else c(nil,[1,2])`, t_2 for `if t then t else c(nil,[1,2])`, and each $\bigtriangledown$ abbreviates a computation not shown here. In order to obtain the result we use the transitivity inference rule, whose left premise utilizes the replacement rule to apply the equation for `isOk`, obtaining the term *rhs*, that will be further reduced in the right premise to obtain `t` by means of another transitivity step. The left child of this last node reduces all the subterms in *rhs* to `t`, while the right one just applies the usual equations for conjunctions to obtain the final result. While the first reductions in the premises of (•) correspond to arithmetic computations and will not been shown here, the last two are more complex. Figure 2 describes the tree $\sharp_1$, that proves how one of the subterms using equations defined by the user is reduced to `t`, while $\sharp_2$ is very similar and will not be studied in depth. The tree $\sharp_1$ reduces in its left child the inner subterm to `f` by traversing the list of positions (in this case the only element in the list is `[1,1]`), reducing the `if_then_else_fi` term in t_1 and then applying the equation for the empty list `nil`. Then, the right child of the root applies the predefined equation for `not` to obtain the final result.

$$\dfrac{\dfrac{}{\texttt{isOk([1,1][1,2])} \to \mathit{rhs}}\,\mathsf{Rep}_{\to} \qquad \dfrac{\dfrac{\dfrac{\triangledown\ \ \triangledown}{\texttt{1 >= 1} \to \texttt{t}}\,\mathsf{Tr} \quad \dots \quad \dfrac{\triangledown\ \ \triangledown}{\texttt{2 <= 5} \to \texttt{t}}\,\mathsf{Tr} \quad \sharp_1 \quad \sharp_2}{(\bullet)\ \mathit{rhs} \to \texttt{t and ... and t}}\,\mathsf{Cong} \qquad \dfrac{\triangledown\ \ \triangledown}{\texttt{t and ... and t} \to \texttt{t}}\,\mathsf{Tr}}{\mathit{rhs} \to \texttt{t}}\,\mathsf{Tr}}{\texttt{isOk([1,1][1,2])} \to \texttt{t}}\,\mathsf{Tr}$$

Figure 2: Tree for the reduction of `isOk([1,1][1,2])`

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{c([1,1],[1,2])} \to t_1}\,\mathsf{Rep}_{\to} \qquad \dfrac{\dfrac{\dfrac{}{\texttt{[1,1] == [1,2]} \to \texttt{t}}\,\mathsf{Rep}_{\to}}{t_1 \to t_2}\,\mathsf{Cong} \qquad \dfrac{\dfrac{}{t_2 \to \texttt{c(nil,[1,2])}}\,\mathsf{Rep}_{\to} \quad \dfrac{}{\texttt{c(nil,[1,2])} \to \texttt{f}}\,\mathsf{Rep}_{\to}}{t_2 \to \texttt{f}}\,\mathsf{Tr}}{t_1 \to \texttt{t}}\,\mathsf{Tr}}{\texttt{c([1,1], [1,2])} \to \texttt{f}}\,\mathsf{Tr}}{\texttt{not(c([1,1],[1,2]))} \to \texttt{not(f)}}\,\mathsf{Cong} \qquad \dfrac{}{\texttt{not(f)} \to \texttt{t}}\,\mathsf{Rep}_{\to}}{\texttt{not(c([1,1],[1,2]))} \to \texttt{t}}\,\mathsf{Tr}$$

Figure 3: Tree $\sharp_1$ for `not(c([1,1],[1,2]))`

In our debugging framework 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 $\mathcal{I} \models e \Rightarrow e'$, $\mathcal{I} \models e \to e'$, and $\mathcal{I} \models e : s$ for each rewrite $e \Rightarrow e'$, reduction $e \to e'$, and membership $e : s$ computed w.r.t. the specification $\mathcal{R}$. As a term model, $\mathcal{I}$ must satisfy the following proposition:

Proposition 1 *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and let $\mathcal{T} = \mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If a statement $e \Rightarrow e'$ (respectively $e \to e'$, $e : s$) can be deduced using the semantic calculus rules* reflexivity*,* transitivity*,* congruence*,* equivalence class *or* subject reduction *using premises that hold in $\mathcal{T}$, then $\mathcal{T} \models e \Rightarrow e'$ (respectively $\mathcal{T} \models e \to e'$, $\mathcal{T} \models e : s$).*

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

- $\mathsf{Tr}_{\Rightarrow}$. Suppose that $\mathcal{T} \models e_1 \Rightarrow e'$ and $\mathcal{T} \models e' \Rightarrow e_2$. Then $[\![e_1]\!]_{\mathcal{A}} \to_{\mathcal{T}} [\![e']\!]_{\mathcal{A}}$, $[\![e']\!]_{\mathcal{A}} \to_{\mathcal{T}} [\![e_2]\!]_{\mathcal{A}}$. Since $\to_{\mathcal{T}}$ is compositional, $[\![e_1]\!]_{\mathcal{A}} \to_{\mathcal{T}} [\![e_2]\!]_{\mathcal{A}}$, i.e., $\mathcal{T} \models e_1 \Rightarrow e_2$.
- $\mathsf{Tr}_{\to}$. If $\mathcal{T} \models e_1 \to e'$ and $\mathcal{T} \models e' \to e_2$ then $[\![e_1]\!]_{\mathcal{A}} = [\![e']\!]_{\mathcal{A}}$ and $[\![e']\!]_{\mathcal{A}} = [\![e_2]\!]_{\mathcal{A}}$. Therefore $[\![e_1]\!]_{\mathcal{A}} = [\![e_2]\!]_{\mathcal{A}}$ and $\mathcal{T} \models e_1 \to e_2$.
- SRed. If $\mathcal{T} \models e \to e'$ and $\mathcal{T} \models e' : s$, then $[\![e]\!]_{\mathcal{A}} = [\![e']\!]_{\mathcal{A}}$ and $[\![e']\!]_{\mathcal{A}} \in A_s$, and hence $[\![e]\!]_{\mathcal{A}} \in A_s$.

The *reflexivity*, *congruence* and *equivalence class* rules are checked analogously. □

Observe that this proposition cannot be extended to the *membership* and *replacement* inference rules, where the correctness of the conclusion depends not only on the calculus but also on the associated specification statement, which could be wrong.

3.2 A calculus for missing answers

The calculus in this section, that extends the one shown in the previous section, will be used to infer the normal form of a term, the least sort of a term, and, given a term and some constraints, the *complete* set of reachable terms from this term that fulfill the requirements.[3] The proof trees built with this calculus have nodes that justify the positive information (why the normal form is reached, the least sort is obtained, and the terms are included in the corresponding sets) but also nodes that justify the negative information (why the normal form is no further reduced, why no a lesser sort can be obtained for the term, and why there are no more terms in the sets). These latter nodes are then used in the debugging trees to localize as much as possible the reasons responsible for missing answers. Throughout this paper we only consider a special kind of conditions and substitutions that operate over them, called *admissible*, that we define as follows:

[3] The requirements of this last inference mimic the ones used in the Maude's breadth-first search, which is usually used to detect the existence of missing answers.

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*

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

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

$$vars(u'_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j), \textit{ or}$$

- *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 *A* kind-substitution*, denoted by κ, is a mapping between variables and terms of the form $v_1 \mapsto t_1; \ldots ; v_n \mapsto t_n$ such that $\forall_{1 \leq i \leq n} . kind(v_i) = kind(t_i)$, that is, each variable has the same kind as the term it binds.*

Definition 4 *A* substitution*, denoted by θ, is a mapping between variables and terms of the form $v_1 \mapsto t_1; \ldots ; v_n \mapsto t_n$ such that $\forall_{1 \leq i \leq n} . sort(v_i) \geq ls(t_i)$, that is, the sort of each variable is greater than or equal to the least sort of the term it binds. Note that a substitution is a special type of kind-substitution where each term has the sort appropriate to its variable.*

Definition 5 *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 $vars(u') \subseteq dom(\theta)$, or*
- *C is a rewrite condition $u \Rightarrow u'$ and $vars(u) \subseteq dom(\theta)$.*

The calculus presented in this section (in Figures 4–7, and 12) will be used to deduce the following judgments, that we introduce together with their meaning for a Σ-term model $\mathcal{T}' = \mathcal{T}_{\Sigma/E',R'}$ defined by equations and memberships E' and by rules R':

- Given a term t and a kind-substitution κ, $\mathcal{T}' \models adequateSorts(\kappa) \rightsquigarrow \Theta$ when either $\Theta = \{\kappa\} \wedge \forall v \in dom(\kappa). \mathcal{T}' \models \kappa[v] : sort(v)$ or $\Theta = \emptyset \wedge \exists v \in dom(\kappa). \mathcal{T}' \not\models \kappa[v] : sort(v)$, where $\kappa[v]$ denotes the term bound by v in κ. That is, when all the terms bound in the kind-substitution κ have the appropriate sort, then κ is a substitution and it is returned; otherwise (at least one of the terms has an incorrect sort), the kind-substitution is not a substitution and the empty set is returned.

- Given an admissible substitution θ for an atomic condition C, $\mathcal{T}' \models [C, \theta] \rightsquigarrow \Theta$ when

$$\Theta = \{\theta' \mid \mathcal{T}', \theta' \models C \text{ and } \theta' \restriction_{dom(\theta)} = \theta\},$$

  that is, Θ is the set of substitutions that fulfill the atomic condition C and extend θ by binding the new variables appearing in C.

- Given a set of admissible substitutions Θ for an atomic condition C, $\mathcal{T}' \models \langle C, \Theta \rangle \rightsquigarrow \Theta'$ when

$$\Theta' = \{\theta' \mid \mathcal{T}', \theta' \models C \text{ and } \theta' \restriction_{dom(\theta)} = \theta \text{ for some } \theta \in \Theta\},$$

  that is, Θ' is the set of substitutions that fulfill the condition C and extend any of the admissible substitutions in Θ.

- $\mathcal{T}' \models disabled(a, t)$ when the equation or membership a cannot be applied to t at the top.

$$\frac{\theta(t_2) \to_{norm} t' \quad adequateSorts(\kappa_1) \leadsto \Theta_1 \quad \ldots \quad adequateSorts(\kappa_n) \leadsto \Theta_n}{[t_1 := t_2, \theta] \leadsto \bigcup_{i=1}^{n} \Theta_i}\ \mathsf{PatC}$$
if $\{\kappa_1, \ldots, \kappa_n\} = \{\kappa\theta \mid \kappa(\theta(t_1)) \equiv_A t'\}$

$$\frac{t_1 : sort(v_1) \quad \ldots \quad t_n : sort(v_n)}{adequateSorts(v_1 \mapsto t_1, \ldots, v_n \mapsto t_n) \leadsto v_1 \mapsto t_1, \ldots, v_n \mapsto t_n}\ \mathsf{AS_1}$$

$$\frac{t_i :_{ls} s_i}{adequateSorts(v_1 \mapsto t_1, \ldots, v_n \mapsto t_n) \leadsto \emptyset}\ \mathsf{AS_2}\ \text{ if } s_i \not\leq sort(v_i)$$

$$\frac{\theta(t) : s}{[t : s, \theta] \leadsto \{\theta\}}\ \mathsf{MbC_1} \qquad \frac{\theta(t) :_{ls} s'}{[t : s, \theta] \leadsto \emptyset}\ \mathsf{MbC_2}\ \text{ if } s' \not\leq s$$

$$\frac{\theta(t_1) \downarrow \theta(t_2)}{[t_1 = t_2, \theta] \leadsto \{\theta\}}\ \mathsf{EqC_1} \qquad \frac{\theta(t_1) \to_{norm} t_1' \quad \theta(t_2) \to_{norm} t_2'}{[t_1 = t_2, \theta] \leadsto \emptyset}\ \mathsf{EqC_2}\ \text{ if } t_1' \not\equiv_A t_2'$$

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

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

Figure 4: Calculus for substitutions

- $\mathcal{T}' \models t \to_{red} t'$ when either $\mathcal{T}' \models t \to_{E'}^{1} t'$ or $\mathcal{T}' \models t_i \to_{E'}^{!} t_i'$, with $t_i \neq t_i'$, for some subterm t_i of t such that $t' = t[t_i \mapsto t_i']$, that is, the term t is either reduced one step at the top or reduced by substituting a subterm by its normal form.
- $\mathcal{T}' \models t \to_{norm} t'$ when $\mathcal{T}' \models t \to_{E'}^{!} t'$, that is, t' is in normal form with respect to the equations E'.
- Given an admissible condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$, $\mathcal{T}' \models \mathit{fulfilled}(\mathcal{C}, t)$ when there exists a substitution θ such that $\mathcal{T}', \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ holds when $\circledast$ is substituted by t.
- Given an admissible condition $\mathcal{C}$ as before, $\mathcal{T}' \models \mathit{fails}(\mathcal{C}, t)$ when there exists *no* substitution θ such that $\mathcal{T}', \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ does not hold when $\circledast$ is substituted by t.
- $\mathcal{T}' \models t :_{ls} s$ when $\mathcal{T}' \models t : s$ 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 $\mathcal{T}'$).
- $\mathcal{T}' \models t \Rightarrow^{top} S$ when $S = \{t' \mid t \to_{R'}^{top} t'\}$, 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 $\mathcal{T}'$. Moreover, equality in S is modulo E', i.e., we are implicitly working with equivalence classes of ground terms modulo E'.
- $\mathcal{T}' \models t \Rightarrow^{q} S$ when $S = \{t' \mid t \to_{\{q\}}^{top} t'\}$, 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.
- $\mathcal{T}' \models t \Rightarrow_1 S$ when $S = \{t' \mid t \to_{R'}^{1} t'\}$, 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.
- $\mathcal{T}' \models t \leadsto_{n}^{\mathcal{C}} S$ when $S = \{t' \mid t \to_{R'}^{\leq n} t' \text{ and } \mathcal{T}' \models \mathit{fulfilled}(\mathcal{C}, t')\}$, 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.
- $\mathcal{T}' \models t \leadsto\!+_{n}^{\mathcal{C}} S$ as before, but with reachability from t in at least one step and in at most n steps.
- $\mathcal{T}' \models t \leadsto\!!_{n}^{\mathcal{C}} S$ when $S = \{t' \mid t \to_{R'}^{\leq n} t' \text{ and } \mathcal{T}' \models \mathit{fulfilled}(\mathcal{C}, t') \text{ and } t' \not\to_{R'}\}$, that is, now the terms (modulo E') in S are *final*, meaning that they cannot be further rewritten.

We first introduce in Figure 4 the inference rules defining the relations $[C, \theta] \leadsto \Theta$, $\langle C, \Theta\rangle \leadsto \Theta'$, and $adequateSorts(\kappa) \leadsto \Theta$. Intuitively, these judgments will provide positive information when they lead to nonempty sets (indicating that the condition holds in the first two judgments or that the kind-substitution is a substitution in the third one) and negative information when they lead to the empty set (indicating respectively that the condition fails or the kind-substitution is not a substitution):

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0\rangle \rightsquigarrow \Theta_1 \quad \ldots \quad \langle C_n, \Theta_{n-1}\rangle \rightsquigarrow \emptyset}{\mathit{disabled}(a, t)}\ \mathsf{Dsb}$$

if $a \equiv l \to r \Leftarrow C_1 \wedge \ldots \wedge C_n \in E$ or
$a \equiv l : s \Leftarrow C_1 \wedge \ldots \wedge C_n \in E$

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(l) \to_{red} \theta(r)}\ \mathsf{Rdc_1}\ \text{ if } l \to r \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \in E$$

$$\frac{t \to_n t'}{f(t_1, \ldots, t, \ldots, t_n) \to_{red} f(t_1, \ldots, t', \ldots, t_n)}\ \mathsf{Rdc_2}\ \text{ if } t \not\equiv_A t'$$

$$\frac{\mathit{disabled}(e_1, f(t_1, \ldots, t_n)) \quad \ldots \quad \mathit{disabled}(e_l, f(t_1, \ldots, t_n)) \quad t_1 \to_{norm} t_1 \quad \ldots \quad t_n \to_{norm} t_n}{f(t_1, \ldots, t_n) \to_n f(t_1, \ldots, t_n)}\ \mathsf{Norm}$$

if $\{e_1, \ldots, e_l\} = \{e \in E \mid e \ll_K^{top} f(t_1, \ldots, t_n)\}$

$$\frac{t \to_{red} t_1 \quad t_1 \to_{norm} t'}{t \to_{norm} t'}\ \mathsf{NTr}$$

$$\frac{t \to_{norm} t' \quad t' : s \quad \mathit{disabled}(m_1, t') \quad \ldots \quad \mathit{disabled}(m_l, t')}{t :_{ls} s}\ \mathsf{Ls}$$

if $\{m_1, \ldots, m_l\} = \{m \in E \mid m \ll_K^{top} t' \ \wedge\ \mathit{sort}(m) < s\}$

Figure 5: Calculus for normal forms and least sorts

- Rule PatC computes all the possible substitutions that extend θ and satisfy the matching of the term t_2 with the pattern t_1 by first computing the normal form t' of t_2, obtaining then all the possible kind-substitutions κ that make t' and $\theta(t_1)$ equal modulo axioms (indicated by $\equiv_A$), and finally checking that the terms assigned to each variable in the kind-substitutions have the appropriate sort with $\mathit{adequateSorts}(\kappa)$. The union of the set of substitutions thus obtained constitutes the set of substitutions that satisfy the matching.

- Rule $\mathsf{AS_1}$ checks whether the terms of the kind-substitution have the appropriate sort to match the variables. In this case the kind-substitution is a substitution and it is returned.

- Rule $\mathsf{AS_2}$ indicates that, if any of the terms in the kind-substitution has a sort bigger than the required one, then the it is not a substitution and thus the empty set of substitutions is returned.

- Rule $\mathsf{MbC_1}$ returns the current substitution if a membership condition holds.

- Rule $\mathsf{MbC_2}$ is used when the membership condition is not satisfied. It checks that the least sort of the term is not less than or equal to the required one, and thus the substitution does not satisfy the condition and the empty set is returned.

- Rule $\mathsf{EqC_1}$ returns the current substitution when an equality condition holds, that is, when the two terms can be joined.

- Rule $\mathsf{EqC_2}$ checks that an equality condition fails by obtaining the normal forms of both terms and then examining that they are different.

- 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.

- Finally, rule SubsCond computes the extensions of a set of admissible substitutions for C $\{\theta_1, \ldots, \theta_n\}$ by using the rules above with each of them.

We use these judgments to define the inference rules of Figure 5, that describe how the normal form and the least sort of a term are computed:

- Rule Dsb indicates when an equation or membership a cannot be applied to a term t. It checks that there are no substitutions that satisfy the matching of the term with the lefthand side of the statement and that fulfill its condition. Note that we check the conditions from left to right, following the same order as Maude and making all the substitutions admissible.

$$\frac{\mathit{fulfilled}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \{t\}}\;\mathsf{Rf}_1 \qquad\qquad \frac{\mathit{fails}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \emptyset}\;\mathsf{Rf}_2$$

$$\frac{\theta(P) \downarrow t \quad \{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m} \quad \{\theta(w_k) \Rightarrow \theta(w'_k)\}_{k=1}^{l}}{\mathit{fulfilled}(\mathcal{C}, t)}\;\mathsf{Fulfill}$$

$$\text{if } \mathcal{C} \equiv P := \circledast \wedge \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k$$

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

Figure 6: Calculus for solutions

- Rule Rdc_1 reduces a term by applying one equation when it checks that the conditions can be satisfied, where the matching conditions are included in the equality conditions. While in the previous rule we made explicit the evaluation from left to right of the condition to show that finally the set of substitutions fulfilling it was empty, in this case we only need one substitution to fulfill the condition and the order is unimportant.

- Rule Rdc_2 reduces a term by reducing a subterm to normal form (checking in the side condition that it is not already in normal form).

- Rule Norm states that the term is in normal form by checking that no equations can be applied at the top and that all its subterms are already in normal form.

- Rule NTr describes the transitivity for the reduction to normal form. It reduces the term with the relation $\rightarrow_{red}$ and the term thus obtained then is reduced to normal form by using again $\rightarrow_{norm}$.

- Rule Ls computes the least sort of the term t. It computes a sort for its normal form (that has the least sort of the terms in the equivalence class) and then checks that memberships deducing lesser sorts cannot be applied.

In these rules Dsb provides the negative information, proving why the statements (either equations or membership axioms) cannot be applied, while the remaining rules provide the positive information indicating why the normal form and the least sort are obtained:

Once these rules have been introduced, we can use them in the rules defining the relation $t \rightsquigarrow_n^{\mathcal{C}} S$. First, we present in Figure 6 the rules related to $n = 0$ steps:

- Rule Rf_1 indicates that when only zero steps can be used and the current term fulfills the condition, the set of reachable terms consists only of this term.

- Rule Rf_2 complements Rf_1 by defining the empty set as result when the condition does not hold.

- Rule $\mathsf{Fulfill}$ 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 verify that there exists *one* substitution that makes the condition true.

- 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 4 to prove that the set of substitutions that satisfy the condition (where the first set of substitutions is obtained from the first matching condition filling the hole with the current term) is empty. Note that, while rule $\mathsf{Fulfill}$ provides the positive information indicating that a condition is fulfilled, this one provides the negative information, proving that the condition does not hold.

Now we introduce in Figure 7 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:

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

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

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

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

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

$$\frac{t \rightarrow_{norm} t_1 \quad t_1 \rightsquigarrow^{\mathcal{C}}_{n} \{t_2\} \cup S \quad t_2 \rightarrow_{norm} t'}{t \rightsquigarrow^{\mathcal{C}}_{n} \{t'\} \cup S}\;\mathsf{Red_1}$$

Figure 7: Calculus for missing answers

- 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 Stp 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 $q \ll^{top}_{K} t$, where q is the label of the rule and t the current term.

- Rule Rl uses the rules in Figure 4 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. The kind of information provided by this rule corresponds to the information provided by the substitutions; if the empty set of substitutions is obtained (negative information) then the rule computes the empty set of terms, which also corresponds with negative information proving that no terms can be obtained with this program rule; analogously when the set of substitutions is nonempty (positive information). This information is propagated through the rest of inference rules justifying why some terms are reachable while others are not.

- Finally, rule $\mathsf{Red_1}$ reduces the reachable terms in order to obtain their normal forms. We use this rule to reproduce Maude behavior, first the normal form of the term is computed and then the rules are applied.

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 $\mathcal{T}_{\Sigma/E,R} \models \varphi$.

Theorem 1 *The calculus of Figures 4, 5, 6, and 7 is correct.*

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

- $\mathit{adequateSorts}(\kappa) \rightsquigarrow \Theta$ is correct. Given a kind-substitution κ, when it has the variables of the appropriate sorts only the rule $\mathsf{SubsCond}_1$ can be applied and the set containing κ is returned. If the matching fails AS_2 has to be applied and the empty substitution set is returned, being the judgment correct.

- $[C, \theta] \rightsquigarrow \Theta$ is correct. We distinguish subcases 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 $\theta(t_2) \rightarrow_{norm} t'$ so, given the complete set of kind-substitutions, we restrict them to those that are substitutions, thus returning the correct set.
  - $C \equiv t : s$. 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 $\mathsf{SubsCond}$. Assuming the premises correct, the conclusion is also correct.

- $\mathit{disabled}(e, t)$ is correct. The only rule that deals with this judgment is Dsb. Assuming the premises correct there are no substitutions satisfying the conditions and making the lefthand side of the equation or membership match the term, so it cannot be applied and the judgment is correct.

- $t \rightarrow_{red} t'$ is correct. In this case two rules can be used: Rdc_1 and Rdc_2. The first one covers reductions at the top, while the second one covers reductions on the subterms, thus dealing with all possibilities. Assuming the premises correct, in the first case we verify that one step is used because it corresponds to the application of one equation, while in the second one we check with the side condition that at least one step is used and thus the judgment is correct.

- $t \rightarrow_{norm} t'$ is correct. The rules that deal with this case are Norm and NTr, that distinguish whether the term is already in normal form or can be further reduced. In the first case if we assume the premises correct then the term is in normal form and then the same term has to be returned. In the second case, assuming the premises correct and a confluent specification, the conclusion is correct.

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

- $\mathit{fails}(\mathcal{C}, t)$. 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 Stp. 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.

- $t \rightsquigarrow^{\mathcal{C}}_n S$. For this judgment, rule 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 Rf_1 or Rf_2 are used and the result is straightforward.

  If $n > 0$ and the term fulfills the condition, rule 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 $n > 0$ and the condition does not hold, rule 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 suffice to compute all the reachable terms, and in this case the least of these values is 4. We have depicted the tree in Figure 8, where we have abbreviated the equational condition `{L:List}` $:= \circledast \wedge$ `isSol(L:List) = true` by $\mathcal{C}$ and `isSol(L) = true` by `isSol(L)`. The leftmost tree justifies that the search condition does not hold for the initial term (this is the reason why Tr_2 has been used instead of Tr_1) and thus it is not a solution. Note that first the substitutions from the matching 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, which is proved by $*_1$. The next tree shows the set of reachable terms in one step (the tree $*_2$, 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.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{1} \rightarrow_{norm} \texttt{1}}\,\mathsf{Norm}}{\texttt{[1,1]} \rightarrow_{norm} \texttt{[1,1]}}\,\mathsf{Norm}}{\texttt{\{[1,1]\}} \rightarrow_{norm} \texttt{\{[1,1]\}}}\,\mathsf{Norm}}{[\texttt{\{L:List\}} := \texttt{\{[1,1]\}},\ \emptyset] \rightsquigarrow \texttt{L} \mapsto \texttt{[1,1]}\,\}}\,\mathsf{PatC} \quad \dfrac{*_1}{\langle \texttt{isSol(L)}, \{\texttt{L} \mapsto \texttt{[1,1]}\}\rangle \rightsquigarrow \emptyset}\,\mathsf{SubsCond}}{\mathit{fails}(\mathcal{C}, \texttt{\{[1,1]\}})}\,\mathsf{Fail} \quad \dfrac{*_2 \quad \dfrac{\dfrac{}{\texttt{[1,1]} \Rightarrow^{top} \emptyset}\,\mathsf{Top} \quad \dfrac{\dfrac{}{\texttt{1} \Rightarrow^{top} \emptyset}\,\mathsf{Top}}{\texttt{1} \Rightarrow_1 \emptyset}\,\mathsf{Stp}}{\texttt{[1,1]} \Rightarrow_1 \emptyset}\,\mathsf{Stp}}{\texttt{\{[1,1]\}} \Rightarrow_1 \texttt{\{[1,1][1,2]\}}}\,\mathsf{Stp} \quad \dfrac{*_3}{\texttt{\{[1,1][1,2]\}} \rightsquigarrow^{\mathcal{C}}_3 \emptyset}\,\mathsf{Tr}_2}{\texttt{\{[1,1]\}} \rightsquigarrow^{\mathcal{C}}_4 \emptyset}\,\mathsf{Tr}_2$$

Figure 8: Tree for the maze example

The tree $*_1$ shows why the current list is not a solution (i.e., the tree provides the negative information proving that the this fragment of the condition does not hold). The reason is that the function `isSol` is reduced to `false`, when we needed it to be reduced to `true`.

$$\dfrac{\dfrac{\dfrac{}{\texttt{isSol([1,1])} \rightarrow_{red} \texttt{false}}\,\mathsf{Rdc}_1 \quad \dfrac{}{\texttt{false} \rightarrow_{norm} \texttt{false}}\,\mathsf{Norm}}{\texttt{isSol([1,1])} \rightarrow_{norm} \texttt{false}}\,\mathsf{NTr} \quad \dfrac{}{\texttt{true} \rightarrow_{norm} \texttt{true}}\,\mathsf{Norm}}{[\texttt{isSol(L) = true}, \texttt{L} \mapsto \texttt{[1,1]}] \rightsquigarrow \emptyset}\,\mathsf{EqC}_2$$

Figure 9: Tree $*_1$ for the search condition

The tree labeled with $*_2$ is sketched in Figure 10. In this tree the applications 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 tree $*_4$ is used to justify that the first condition of `expand` holds and extends the set of substitutions that fulfill the condition thus far to the set $\{\theta_1, \theta_2, \theta_3\}$, where $\theta_1 \equiv \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,2]}$, $\theta_2 \equiv \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,0]}$, and $\theta_3 \equiv \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[0,1]}$. The substitution θ_1 also fulfills the next condition, `isOk(L P)`, which is proved with the rule $\textsf{EqC}_1$ in ($\clubsuit$) (where the big triangle is the proof tree shown in Figure 2, proving that the condition holds), while the substitutions θ_2 and θ_3 fail; the trees $\bigtriangledown$ proving it are analogous to the one shown in Figure 9. This substitution θ_1 is thus the only one inferred in the root of the tree, where the node ($\clubsuit$) provides the positive information proving why the substitution is obtained and its siblings ($\bigtriangledown$) the negative information proving why the other substitutions are not in the set.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{1 \to_{norm} 1}\;\textsf{Norm}}{\texttt{\{[1,1]\}} \to_{norm} \texttt{\{[1,1]\}}}\;\textsf{Norm}}{[\texttt{\{L\}} := \texttt{\{[1,1]\}},\ \emptyset] \leadsto \{\texttt{L} \mapsto \texttt{[1,1]}\}}\;\textsf{PatC} \qquad *_4 \qquad \dfrac{\dfrac{\bigtriangledown \qquad \dfrac{}{\texttt{true} \to \texttt{true}}\;\textsf{Rf}_{\to}}{(\clubsuit)\ [\texttt{isOk(L P)}, \theta_1] \leadsto \{\theta_1\}}\;\textsf{EqC}_1 \qquad \bigtriangledown \quad \bigtriangledown}{\langle \texttt{isOk(L P)}, \{\theta_1, \theta_2, \theta_3\} \rangle \leadsto \{\theta_1\}}\;\textsf{SubsCond}}{\texttt{\{[1,1]\}} \Rightarrow^{\textsf{e}} \texttt{\{[1,1][1,2]\}}}\;\textsf{Rl}}{\texttt{\{[1,1]\}} \Rightarrow^{top} \texttt{\{[1,1][1,2]\}}}\;\textsf{Top}$$

Figure 10: Tree $*_2$ for the applications at the top

The tree $*_4$, shown in Figure 11, 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 $\texttt{L} \mapsto \texttt{[1,1]}$) and, since it is a rewrite condition, the set of reachable terms—computed with $*_5$, which will not be further discussed here—is used to extend this substitution, obtaining a set with three different substitutions (that we previously abbreviated as θ_1, θ_2, and θ_3).

There are two additional kinds of search allowed in our framework: searches for final terms and searches in *one or more* steps. Figure 12 presents the inference rules for these cases:

- Rules $\textsf{Rf}_3$ and $\textsf{Rf}_4$ 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 $\textsf{Rf}_5$ specifies that, if the term is not final but no more steps are allowed, then the set of reachable final terms is empty.
- Rule $\textsf{Tr}_3$ 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 $\textsf{Red}_2$ reduces 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 $\textsf{Rf}_6$.
- When at least one step can be used we apply rule $\textsf{Tr}_4$, 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 12 is correct.*

$$\dfrac{\dfrac{\dfrac{*_5}{\texttt{next([1,1])} \leadsto_2^{\texttt{P}:=\circledast} \{\texttt{[1,2]}, \texttt{[1,0]}, \texttt{[0,1]}\}}\;\textsf{Tr}_2}{[\texttt{next(L)} \Rightarrow \texttt{P}, \texttt{L} \mapsto \texttt{[1,1]}] \leadsto \{\texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,2]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,0]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[0,1]}\}}\;\textsf{RlC}}{\langle \texttt{next(L)} \Rightarrow \texttt{P}, \{\texttt{L} \mapsto \texttt{[1,1]}\} \rangle \leadsto \{\texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,2]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,0]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[0,1]}\}}\;\textsf{SubsCond}$$

Figure 11: Tree $*_4$ for the first condition of `expand`

$$\frac{\mathit{fulfilled}(\mathcal{C},t) \quad t \Rightarrow_1 \emptyset}{t \rightsquigarrow!^{\mathcal{C}}_{n} \{t\}}\ \mathsf{Rf}_3$$

$$\frac{\mathit{fails}(\mathcal{C},t) \quad t \Rightarrow_1 \emptyset}{t \rightsquigarrow!^{\mathcal{C}}_{n} \emptyset}\ \mathsf{Rf}_4$$

$$\frac{t \Rightarrow_1 S}{t \rightsquigarrow!^{\mathcal{C}}_{0} \emptyset}\ \mathsf{Rf}_5 \quad S \neq \emptyset$$

$$\frac{t \Rightarrow_1 \{t_1, \ldots, t_k\} \quad t_1 \rightsquigarrow!^{\mathcal{C}}_{n} S_1 \quad \ldots \quad t_k \rightsquigarrow!^{\mathcal{C}}_{n} S_k}{t \rightsquigarrow!^{\mathcal{C}}_{n+1} \bigcup_{i=1}^{k} S_i}\ \mathsf{Tr}_3 \quad \text{if } k > 0$$

$$\frac{t \to t_1 \quad t_1 \rightsquigarrow!^{\mathcal{C}}_{n} \{t_2\} \cup S \quad t_2 \to t'}{t \rightsquigarrow!^{\mathcal{C}}_{n} \{t'\} \cup S}\ \mathsf{Red}_2$$

$$\frac{}{t \rightsquigarrow+^{\mathcal{C}}_{0} \emptyset}\ \mathsf{Rf}_6$$

$$\frac{t \to t' \quad t' \Rightarrow_1 \{t_1, \ldots, t_k\} \quad t_1 \rightsquigarrow^{\mathcal{C}}_{n} S_1 \quad \ldots \quad t_k \rightsquigarrow^{\mathcal{C}}_{n} S_k}{t \rightsquigarrow+^{\mathcal{C}}_{n+1} \bigcup_{i=1}^{k} S_i}\ \mathsf{Tr}_4$$

Figure 12: Calculus for final and one or more steps searches

Proof.

- $t \rightsquigarrow!^{\mathcal{C}}_{n} S$. For this judgment, rule 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 Rf_3, Rf_4, and Rf_5 can be used. If t is not final only 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 Rf_3 can be used and hence the singleton set consisting of the term is returned, while if the condition fails Rf_4 is applied and the empty set is returned. In both cases the result is correct by definition.

  When $n > 0$ rules Rf_3, Rf_4, and Tr_3 can be used. If the term is final, Rf_3 and Rf_4 are applied and the result holds as in the previous case. If the term is not final, then 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.

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

  When $n = 0$, only rule 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 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 the previous section, we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. As any Σ-term model, $\mathcal{I}$ must satisfy the following soundness propositions:

Proposition 2 *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 adequateSorts$(\kappa) \rightsquigarrow \Theta$, $[C, \theta] \rightsquigarrow \Theta$, or $\langle C, \Theta\rangle \rightsquigarrow \Theta'$ can be deduced using the rules from Figure 4 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models$ adequateSorts$(\kappa) \rightsquigarrow \Theta$, $\mathcal{T}_{\Sigma/E',R'} \models [C, \theta] \rightsquigarrow \Theta$, and $\mathcal{T}_{\Sigma/E',R'} \models \langle C, \Theta\rangle \rightsquigarrow \Theta'$, 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.

AS$_1$ We know that the terms in the kind-substitution have the adequate sort, so it is a substitution.

AS$_2$ When one term in the kind-substitution has an incorrect sort the match fails.

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 3 *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and j a judgment deduced with the inference rules Dsb, Rdc$_2$, or NTr from Figure 5 from premises that hold in $\mathcal{T}_{\Sigma/E',R'}$. Then also $\mathcal{T}_{\Sigma/E',R'} \models j$.*

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

Dsb If the matching with the lefthand side and the conditions cannot be satisfied then it is straightforward to see that the statement cannot be applied.

Rdc$_2$ The substitution of a subterm by its normal form is correct if the normal form is correct.

NTr Since the specification is confluent, we can use any equations to evolve a term and then compute the normal form from this new term.

□

Proposition 4 *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 6 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 5 *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 7 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 t itself, because we are assuming it fulfills the condition. Thus, the inference is sound.

Tr$_2$ Analogous to the case above.

Stp 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 6 *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 12 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 5.

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.

4 Debugging Trees

We describe in this section how to obtain appropriate debugging trees from the proof trees introduced in the previous section. First, we describe the errors that can be found with these proof trees; then, we describe how they can be abbreviated in such a way that soundness and completeness are kept while easing the debugging sessions.

4.1 Debugging with proof trees

As explained in the previous sections, 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 $\mathcal{R}$. We will say that a judgment is *valid* when it holds in the intended interpretation $\mathcal{I}$, and *invalid* otherwise. Our goal is to find a buggy node (an invalid node with all its children correct) 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 7 *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.*

By using the proof trees computed with the calculus of the previous section as debugging trees we are able to locate wrong statements, missing statements, and wrong search conditions, 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.

- 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.

- 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 search 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 $\theta(l) \equiv_A t$ 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 search condition instance*.

- A statement or condition is *wrong* when it admits a wrong instance.

- Given a term t, there is a *missing equation for* t if the computed normal form of t does not correspond with the one expected in $\mathcal{I}$.

- A specification has a *missing equation* if there exists a term t such that there is a missing equation for t.

- Given a term t, there is a *missing membership for* t if the computed least sort for t does not correspond with the one expected in $\mathcal{I}$.

Rep$_{\to}$	Wrong equation
Rep$_{\Rightarrow}$	Wrong rule
Mb	Wrong membership
Rdc$_1$	Wrong equation
Norm	Missing equation
Ls	Missing membership
Fulfill	Wrong search condition
Fail	Wrong search condition
Top	Missing rule
Rl	Wrong rule

Table 1: Errors detected by the proof trees

- A specification has a *missing membership* if there exists a term t such that there is a missing membership for t.
- 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.

We relate these definitions with our calculus in the following proposition:

Proposition 8 *Let N be a buggy node in some proof tree in the calculus of Figures 1, 4, 5, 6, 7, and 12 w.r.t. an intended interpretation $\mathcal{I}$. Then:*

1. *N corresponds to the consequence of an inference rule in the first column of Table 1.*
2. *The error associated to N can be obtained from the inference rule as shown in the second column of Table 1.*

Proof. The first item is a straightforward consequence of Propositions 1, 2, 3, 4, 5, and 6: 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:

Rep$_{\to}$ In this case the associated rewrite rule is wrong as a direct consequence of wrong statement instance: N is invalid in $\mathcal{I}$, while the previous conditions, which state the validity of the statements in the rewrite rule condition instance, correspond to the premises of the (Rep$_{\to}$) inference rule (see Figure 1), which are valid in $\mathcal{I}$ because N is buggy.

Rep$_{\Rightarrow}$ and Mb Analogous to the case above.

Rdc$_1$ In this case it is possible to have an erroneous result when the conditions hold. The reason is that the equation can be wrong, and thus we would have a wrong equation instance.

Norm If the conclusion of this rule is erroneous but its premises hold means that the specification does not have all the required equations, that is, an error in this node is associated to a missing equation.

Ls Similarly to the case above, if the conclusion of this rule is wrong while its premises hold means that the specification lacks some membership, that is, an error in this node is associated to a missing membership.

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.

Rl 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.

□

We assume that the nodes inferred with these inference rules are decorated with some extra information to identify the error when they are pointed out as buggy. More specifically, nodes related to wrong statements keep the label of the statement, nodes related to missing statements keep the operator at the top that requires more statements to be defined, and nodes related to wrong conditions keep the condition. With this information when a wrong statement is found this specific statement is pointed out; when a missing statement is found, the debugger indicates the operator at the top of the term in the lefthand side of the statement that is missing; and when a wrong condition is found, the specific condition is shown. Actually, when a missing statement is found what the debugger reports is that a statement is missing *or* the conditions in the remaining statements are not the intended ones (thus they are not applied when expected and another one would be needed), but the error *is not located* in the statements used in the conditions, since they are also checked during the debugging process. Finally, it is important not to confuse missing answers with missing statements; the current calculus detects missing answers due to both wrong and missing statements and wrong search conditions.

4.2 Abbreviated Proof Trees

However, we will not use the proof trees T computed in the previous sections directly as debugging trees, but a suitable abbreviation which we denote by $APT(T)$ (from *Abbreviated Proof Tree*), or simply APT if the proof tree T is clear from the context. The reason for preferring the APT to the original proof tree is that it reduces and simplifies the questions that will be asked to the user while keeping the soundness and completeness of the technique. This transformation relies on the Proposition 8: only potential buggy nodes are kept.

The rules for deriving an APT can be seen in Figure 13. The abbreviation always starts by applying ($\mathbf{APT}_1$). This rule simply duplicates the root of the tree and applies APT', which receives a proof tree and returns a forest (i.e., a set of trees). Hence without this duplication the result of the abbreviation could be a forest instead of a single tree. The rest of the APT rules correspond to the function APT' and are assumed to be applied top-down: if several APT rules can be applied at the root of a proof tree, we must choose the first one, that is, the rule of least number. The following advantages are obtained with this transformation:

- Questions associated to nodes with reductions are improved (rules ($\mathbf{APT}_2$), ($\mathbf{APT}_3$), ($\mathbf{APT}_5$), ($\mathbf{APT}_6$), and ($\mathbf{APT}_7$)) by asking about normal forms instead of asking about intermediate states. For example, in rule ($\mathbf{APT}_2$) the error associated to $t_1 \to t_2$ is the one associated to $t_1 \to t'$, which is not included in the APT. We have chosen to introduce $t_1 \to t_2$ instead of simply $t_1 \to t'$ in the APT as a pragmatic way of simplifying the structure of the APTs, since t_2 is obtained from t' and hence likely simpler.

- The rule ($\mathbf{APT}_4$) deletes questions about rewrites *at the top* of a given term (that may 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, with the variants of the rules ($\mathbf{APT}_8$) and ($\mathbf{APT}_9$), two different kinds of tree, one that contains judgments of rewrites with several steps and another that only contains rewrites in one step. The one-step debugging tree follows strictly the idea of keeping only nodes corresponding to relevant information. However, the many-steps debugging tree also keeps nodes corresponding to the *transitivity* inference rules. The user will choose which debugging tree (one-step or many-steps) will be used for the debugging session, taking into account that the many-steps debugging tree usually leads to shorter debugging sessions (in terms of the number of questions) but with likely more complicated questions. The number of questions is usually reduced because keeping the transitivity nodes for rewrites gives to some parts of the debugging tree the shape of a balanced binary tree (each transitivity inference has two premises, i.e., two child subtrees), and this allows the debugger to use efficiently the divide and query navigation strategy. On the contrary, removing the transitivity inferences for rewrites (as rules ($\mathbf{APT}_8^o$) and ($\mathbf{APT}_9^o$) do) produces flattened trees where this strategy is no longer efficient. On the other hand, in rewrites $t \Rightarrow t'$ and searches $t \rightsquigarrow_n^{\mathcal{C}} S$ appearing as conclusion of a transitivity inference rule, the judgment can be more complicated because it combines several inferences. The user must balance the pros and cons of each option, and choose the best one for each debugging session.

- The rule ($\mathbf{APT}_{11}$) removes from the tree all the nodes do not associated with relevant information, since the rule ($\mathbf{APT}_{10}$) keeps the relevant information and the rules are applied in order. We remove, for example, nodes related to judgments about sets of substitutions, disabled statements, and rewrites with a concrete rule, that can be in general difficult to answer. Moreover, it removes from the tree trivial judgments like the ones related to reflexivity or congruence.

- Since the APT is built without computing the associated proof tree, it reduces the time and space needed to build the tree.

The following property of the abbreviated proof trees will be useful when proving the correctness of the technique.

Lemma 1 *Let T be a finite proof tree representing an inference in the calculus of Figure 1, 4, 5, 6, 7, and 12 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root N of T is invalid in $\mathcal{I}$. Then:*

(a) If T contains only one node, then
$$APT'(T) = \{T\}$$

(b) There is a $T' \in APT'(T)$ such that T' has an invalid root.

Proof. If T contains only one node N then N is an invalid node without children and therefore buggy. By Proposition 8 the inference step proving this node must be $\mathsf{Rep}_{\rightarrow}$, Mb, $\mathsf{Rep}_{\Rightarrow}$, Rdc_1, Norm, $\mathsf{Fulfill}$, Fail, Ls, Rl, or Top. In all these cases the rule ($\mathbf{APT}_{10}$) of Figure 13 must be applied and the result holds, since it returns a unary set with the same root.

The second item can be proved by induction on the number of nodes of T, which we denote as $n(T)$. If $n(T) = 1$ the property is straightforward from the part (a) above because $T \in APT'(T)$. 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 either ($\mathbf{APT}_2$), ($\mathbf{APT}_3$), ($\mathbf{APT}_4$), ($\mathbf{APT}_5$), ($\mathbf{APT}_6$), ($\mathbf{APT}_7$), ($\mathbf{APT}_8^m$), ($\mathbf{APT}_9^m$), or ($\mathbf{APT}_{10}$) the result holds directly because the result is a unary set with the same invalid root (in the case of ($\mathbf{APT}_7$) an equivalent root).

- If it is ($\mathbf{APT}_8^o$), ($\mathbf{APT}_9^o$), or ($\mathbf{APT}_{11}$) by Proposition 8 N has some invalid child, which corresponds to the root of some premise T_i. By the induction hypothesis, there is some $T' \in APT'(T_i)$ with invalid root. And by observing the rules of Figure 13 it can be checked that every subtree T_i of the root of T verifies $APT'(T_i) \subseteq APT'(T)$. Then $T' \in APT'(T)$.

□

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

Theorem 3 *Let T be a finite proof tree representing an inference in the calculus of Figure 1 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root of T is invalid in $\mathcal{I}$. Then:*

- *$APT(T)$ contains at least one buggy node (completeness).*

- *Any buggy node in $APT(T)$ has an associated wrong statement, missing statement, or wrong condition in $\mathcal{R}$ according to Table 1 (correctness).*

Proof. We prove each item separately:

$$(\mathbf{APT}_1)\quad APT\left(\frac{T_1\ldots T_n}{aj}R_1\right) = \frac{APT'\left(\frac{T_1\ldots T_n}{aj}R_1\right)}{aj}$$

$$(\mathbf{APT}_2)\quad APT'\left(\frac{\frac{T_1\ldots T_n}{t_1\to t'}\,\mathsf{Rep}_{\to}\quad T'}{t_1\to t_2}\,\mathsf{Tr}_{\to}\right) = \left\{\frac{APT'(T_1)\ldots APT'(T_n)\ APT'(T')}{t_1\to t_2}\,\mathsf{Rep}_{\to}\right\}$$

$$(\mathbf{APT}_3)\quad APT'\left(\frac{\frac{T_1\ \ldots\ T_n}{t\to t''}\,\mathsf{Rdc}_1\ \ T'}{t\to t'}\,\mathsf{NTr}\right) = \left\{\frac{APT'\left(T_1\right)\ \ldots\ APT'\left(T_n\right)\ APT'\left(T'\right)}{t\to t'}\,\mathsf{Rdc}_1\right\}$$

$$(\mathbf{APT}_4)\quad APT'\left(\frac{\frac{T_1\ \ldots\ T_n}{t\Rightarrow^{top}S'}\,\mathsf{Top}\ \ T'_1\ldots T'_m}{t\Rightarrow_1 S}\,\mathsf{Stp}\right) = \left\{\frac{APT'\left(T_1\right)\ \ldots\ APT'\left(T_n\right)\ APT'\left(T'_1\right)\ \ldots\ APT'\left(T'_m\right)}{t\Rightarrow_1 S}\,\mathsf{Top}\right\}$$

$$(\mathbf{APT}_5)\quad APT'\left(\frac{T'\quad \frac{T_1\ldots T_n}{t\Rightarrow t'}\,\mathsf{Rep}_{\Rightarrow}\quad T''}{t_1\Rightarrow t_2}\,\mathsf{EC}\right) = \left\{\frac{APT'(T')\ APT'(T_1)\ldots APT'(T_n)\ APT'(T'')}{t_1\Rightarrow t_2}\,\mathsf{Rep}_{\Rightarrow}\right\}$$

$$(\mathbf{APT}_6)\quad APT'\left(\frac{T\ \ \frac{T_1\ldots T_n}{aj'}R_1\ \ T'}{aj}\,\mathsf{Red}_i\right) = \left\{\frac{APT'\left(T\right)\ APT'\left(T_1\right)\ \ldots\ APT'\left(T_n\right)\ APT'\left(T\right)}{aj}R_1\right\}$$

$$(\mathbf{APT}_7)\quad APT'\left(\frac{T_{t\to_{norm}t'}\ T_1\ldots T_n}{t:_{ls}s}\,\mathsf{Ls}\right) = \left\{\frac{APT'\left(T_{t\to_{norm}t'}\right)\ APT'\left(T_1\right)\ \ldots\ APT'\left(T_n\right)}{t':_{ls}s}\,\mathsf{Ls}\right\}$$

$$(\mathbf{APT}_8^o)\quad APT'\left(\frac{T_1\quad T_2}{t_1\Rightarrow t_2}\,\mathsf{Tr}_{\Rightarrow}\right) = APT'(T_1)\bigcup APT'(T_2)$$

$$(\mathbf{APT}_8^m)\quad APT'\left(\frac{T_1\quad T_2}{t_1\Rightarrow t_2}\,\mathsf{Tr}_{\Rightarrow}\right) = \left\{\frac{APT'(T_1)\ APT'(T_2)}{t_1\Rightarrow t_2}\,\mathsf{Tr}_{\Rightarrow}\right\}$$

$$(\mathbf{APT}_9^o)\quad APT'\left(\frac{T_1\ldots T_n}{aj}\,\mathsf{Tr}_j\right) = APT'\left(T_1\right)\bigcup\ldots\bigcup APT'\left(T_n\right)$$

$$(\mathbf{APT}_9^m)\quad APT'\left(\frac{T_1\ldots T_n}{aj}\,\mathsf{Tr}_j\right) = \left\{\frac{APT'\left(T_1\right)\ \ldots\ APT'\left(T_n\right)}{aj}\,\mathsf{Tr}_j\right\}$$

$$(\mathbf{APT}_{10})\quad APT'\left(\frac{T_1\ldots T_n}{aj}R_2\right) = \left\{\frac{APT'(T_1)\ldots APT'(T_n)}{aj}R_2\right\}$$

$$(\mathbf{APT}_{11})\quad APT'\left(\frac{T_1\ldots T_n}{aj}R_1\right) = APT'(T_1)\bigcup\ldots\bigcup APT'(T_n)$$

R_1 any inference rule $\qquad$ R_2 either Mb, $\mathsf{Rep}_{\to}$, $\mathsf{Rep}_{\Rightarrow}$, Rdc_1, Norm, $\mathsf{Fulfill}$, Fail, Ls, Rl, or Top

$1\le i\le 2 \qquad 1\le j\le 4 \qquad aj, aj'$ any judgment

Figure 13: Transforming rules for obtaining abbreviated proof trees

- $APT(T)$ 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 7.
- First we observe that the root of $APT(T)$ cannot be buggy, because if it is invalid then it has an invalid child (Lemma 1(b)). Therefore any buggy node must be part of $APT'(T)$ (the premise in $(\mathbf{APT}_1)$).

  Let N be a buggy node occurring in $APT'(T)$. Then N is the root of some tree T_N, subtree of some $T' \in APT'(T)$. 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 in S by induction on the number of nodes of T', $n(T')$.

  If $n(T') = 1$ then T' contains only one node and $APT'(T') = \{T'\}$ by Lemma 1(a). Then the only possible buggy node is N, which means that N is also buggy in T and that the associated fragment of code is wrong by Proposition 8.

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

  $(\mathbf{APT}_2)$ Then T' is of the form

$$\dfrac{\dfrac{T_1 \dots T_n}{e_1 \to e'}\,{\scriptstyle (Rep_\to)} \qquad T''}{e_1 \to e_2}\,{\scriptstyle (\,Tr_\to)}$$

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

$$\dfrac{APT'(T_1) \dots APT'(T_n)\ APT'(T'')}{e_1 \to e_2}\,{\scriptstyle (Rep_\to)}$$

  Since N is buggy in T_N it is invalid w.r.t. $\mathcal{I}$. By Proposition 8, $e_1 \to e_2$ cannot be buggy in T', i.e. either T'' has an invalid root or $e_1 \to e'$ is invalid. But T'' cannot be invalid because $APT'(T'')$ is a child subtree of N and by Lemma 1(b) it would contain a tree T''' with invalid root, which is not possible because T''' is a child of the buggy node N in T_N. Therefore $e_1 \to e'$ is invalid. Moreover, the roots of $T_1, \dots, T_n$ are also valid by the same reason: $APT'(T_1), \dots, APT'(T_n)$ are child subtrees of N in T_N and cannot have an invalid root. Therefore $e_1 \to e'$ is buggy in T', i.e., is buggy in T and by Proposition 8 the equation associated to label $(Rep_\to)$ is wrong. And this label is the same that can be found associated to N in the APT' T_N. Therefore the buggy node N of the APT' has an associated wrong equation.

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

$$\dfrac{\dfrac{T_1 \quad \dots \quad T_n}{t \to_{red} t''}\,\mathsf{Rdc_1} \qquad T}{t \to_{norm} t'}\,\mathsf{NTr}$$

  Thus $t \to_{norm} t'$ and T_N is

$$\dfrac{APT'(T_1) \quad \dots \quad APT'(T_n) \quad APT'(T)}{t \to_{norm} t'}\,\mathsf{Rdc_1}$$

  By Proposition 8 we know that N cannot be buggy in T', thus either $t \to_{red} t''$ or the root of T is invalid. However, if the root of T 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, $t \to_{red} t''$ is invalid but, for the same reason as before, $T_1 \dots T_n$ cannot be invalid, so it is also buggy in T' and by Proposition 8 the rule label $\mathsf{Rdc_1}$ has associated a wrong equation. Since this same label has been now assigned to N, the buggy node in the abbreviated proof tree has an associated wrong equation.

  $(\mathbf{APT}_4)$ In this case T' has the form

$$\dfrac{\dfrac{T_1 \quad \dots \quad T_n}{t \Rightarrow^{top} S'}\,\mathsf{Top} \qquad T_1' \quad \dots \quad T_n'}{t \Rightarrow_1 S}\,\mathsf{Stp}$$

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

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

By Proposition 5 we know that N cannot be buggy in T', thus either of $t \Rightarrow^{top} S'$ or the root of one of $T'_1 \dots T'_n$ is invalid. However, if the root of one of the trees $T'_1 \dots T'_n$ 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, $T_1 \dots T_n$ cannot be invalid, so it is also buggy in T' and by Proposition 8 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.

($\mathbf{APT}_5$) **and** ($\mathbf{APT}_6$) Analogous to the previous cases.

($\mathbf{APT}_7$) T' has the form

$$\frac{T_{t \rightarrow_{norm} t'} \ T_1 \ \dots \ T_n}{t :_{ls} s}\ \mathsf{Ls}$$

Then $N \equiv t :_{ls} s$ and T_N is

$$\frac{APT'(T_{t \rightarrow_{norm} t'})\ APT'(T_1) \ \dots \ APT'(T_n)}{t' :_{ls} s}\ \mathsf{Ls}$$

Since N is buggy in T_N all the trees in $APT'(T_{t \rightarrow_{norm} t'})$ $APT'(T_1)$ $\dots$ $APT'(T_n)$ are valid and by Lemma 1 the roots of $T_{t \rightarrow_{norm} t'}$ T_1 $\dots$ T_n are also valid and N is buggy in T'. By Proposition 8 it is associated with a missing membership in T' and, since we have the same label in T_N, the result holds.

($\mathbf{APT}^o_8$), ($\mathbf{APT}^o_9$), ($\mathbf{APT}_{11}$) Then $T_N \in APT'(T_i)$ for some child subtree T_i of the root of T' and the result holds by the induction hypothesis.

($\mathbf{APT}^m_8$) We check that actually this rule cannot be applied to produce a buggy node and therefore must not be considered here. If (APT^m_8) is applied then T' must be of the form

$$\frac{T_1 \quad T_2}{e_1 \Rightarrow e_2}\,(\mathit{Tr}_{\Rightarrow})$$

N is $e_1 \Rightarrow e_2$ and T_N is

$$\frac{APT'(T_1)\ APT'(T_2)}{e_1 \Rightarrow e_2}\,(\mathit{Tr}_{\Rightarrow})$$

And N can be invalid but not buggy in T' (and hence in T) by Proposition 8, because it is the conclusion of a transitivity inference, and thus either T_1 or T_2 has an invalid root. Then by Lemma 1(b), either $APT'(T_1)$ or $APT'(T_2)$ have an invalid root and N is not buggy in T_N.

($\mathbf{APT}^m_9$) Analogous to the previous case

($\mathbf{APT}_{10}$) We present the proof for the inference rule Fulfill, being the rest of cases analogous. T' has the form

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

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

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

Since N is buggy in T_N all the trees in $APT'(T_1)$ $\dots$ $APT'(T_n)$ are valid and by Lemma 1 the roots of $T_1 \dots T_n$ are also valid and N is buggy in T'. By Proposition 8 it is associated with a wrong statement in T' and, since we have the same label in T_N, the result holds.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{(\spadesuit)\ \texttt{1} \rightarrow_{norm} \texttt{1}}\,\mathsf{Norm}_{\mathtt{s\_}}}{(\spadesuit)\ \texttt{[1,1]} \rightarrow_{norm} \texttt{[1,1]}}\,\mathsf{Norm}_{\mathtt{[\_,\_]}}}{(\spadesuit)\ \texttt{\{[1,1]\}} \rightarrow_{norm} \texttt{\{[1,1]\}}}\,\mathsf{Norm}_{\mathtt{\{\_\}}} \quad \cfrac{}{\texttt{isSol}(P_1) \rightarrow \texttt{f}}\,\mathsf{Rdc}_{\mathtt{is2}} \quad \star_1 \quad \triangledown \quad \cdots \quad \triangledown \quad \star_2}{\texttt{\{[1,1]\}} \rightsquigarrow^{\mathcal{C}}_4 \emptyset}\,\mathsf{Tr}_2$$

Figure 14: Abbreviated proof tree for the maze example

$$\cfrac{\cfrac{\cfrac{}{(\spadesuit)\ \texttt{1} \rightarrow_{norm} \texttt{1}}\,\mathsf{Norm}_{\mathtt{s\_}}}{(\spadesuit)\ \texttt{[1,1]} \rightarrow_{norm} \texttt{[1,1]}}\,\mathsf{Norm}_{\mathtt{[\_,\_]}} \quad \cfrac{*'_5 \quad \star_3 \quad \cfrac{}{(\Diamond)\ \texttt{isOk}(L_2) \rightarrow \texttt{f}}\,\mathsf{Rep}_{\perp} \quad \cfrac{}{(\Diamond)\ \texttt{isOk}(L_3) \rightarrow \texttt{f}}\,\mathsf{Rep}_{\perp}}{\texttt{\{[1,1]\}} \Rightarrow^{\mathsf{e}} \texttt{\{[1,1][1,2]\}}}\,\mathsf{Rl}_{\mathsf{e}} \quad \cfrac{\cfrac{}{(\heartsuit)\ \texttt{1} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathtt{s\_}}}{(\heartsuit)\ \texttt{[1,1]} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathtt{[\_,\_]}}}{\texttt{\{[1,1]\}} \Rightarrow_1 \texttt{\{[1,1][1,2]\}}}\,\mathsf{Top}_{\mathtt{\{\_\}}}$$

Figure 15: Abbreviated tree for $\star_1$

□

The theorem states that we can safely employ the abbreviated proof tree as a basis for the declarative debugging of Maude system and functional modules: the technique will find a buggy node starting from any initial symptom detected by the user. Of course, these results assume that the user answers correctly all the questions about the validity of the APT nodes asked by the debugger (see Section 2.6).

The trees in Figures 14–17 depict the (one-step) abbreviated proof tree for the maze example, where $\mathcal{C}$ stands for `{L:List}:=` $\circledast \wedge$ `isSol(L:List)`, P_1 for `[1,1]`, L_1 for `[1,1][1,2]`, L_2 for `[1,1][1,0]`, L_3 for `[1,1][0,1]`, `t` for `true`, `f` for false, `n` for `next`, `e` for `expand`, L for `[1,1][1,2][1,3][1,4]`, and $*'_5$ for the application of APT' to $*_5$ (from Figure 11). We have also extended the information in the labels with the operator or statement associated to the inference. More concretely, the tree in Figure 14 abbreviates the tree in Figure 8; the first two premises in the abbreviated tree stand for the first premise in the proof tree (which includes the tree in Figure 9), keeping only the nodes associated with relevant information according to Proposition 8: Norm, with the operator associated to the reduction, and Rdc_1, with the label of the associated equation. The tree $\star_1$, shown in Figure 15, abbreviates the second premise of the tree in Figure 8 as well as the trees in Figures 10 and 11; it only keeps the nodes referring to normal forms, searches in one step, that are now associated to the rule Top, each of them referring to a different operator (the operator `s_` is the successor constructor for natural numbers), and the applications of rules (Rl) and equations ($\mathsf{Rep}_{\rightarrow}$). Note that the equation describing the behavior of `isOk` has not got any label, which is indicated with the symbol $\perp$; we will show below how the debugger deals with these nodes. The tree $\star_2$, presented in Figure 16, shares these characteristics and only keeps nodes related to one-step searches and application of rules. The tree $\star_3$ abbreviates the proof tree for the reduction shown in Figure 2, where the important result of the abbreviation is that all replacement inferences are related now to reductions to normal form, thus easing the questions that will be asked to the user.

These abbreviation rules are combined with trusting mechanisms that further reduce the proof tree:

- Statements can be trusted in several ways: non labelled statements, which include the predefined functions, are always trusted (i.e., the nodes marked with ($\Diamond$) in Figures 15 and 17 will be discarded by the debugger); statements and modules can be trusted before starting the debugging process; and statements can also be trusted on the fly.

- A correct modules can be given before starting a debugging session. By checking the correctness of the judgments against this module, correct nodes can be deleted from the tree.

- Constructed terms (that is, terms built only with constructors, pointed out with the `ctor` attribute) of certain sorts or built with some operators can be considered *final*, which indicates that they cannot

$$\cfrac{\cfrac{\cfrac{\triangledown \quad \cdots \quad \triangledown}{\texttt{n}(L) \Rightarrow^{\mathtt{n1}} \texttt{[1,5]}}\,\mathsf{Rl}_{\mathtt{n1}} \quad \cfrac{\triangledown \quad \cdots \quad \triangledown}{\texttt{n}(L) \Rightarrow^{\mathtt{n2}} \texttt{[0,4]}}\,\mathsf{Rl}_{\mathtt{n2}} \quad \cfrac{\triangledown \quad \cdots \quad \triangledown}{\texttt{n}(L) \Rightarrow^{\mathtt{n3}} \texttt{[1,3]}}\,\mathsf{Rl}_{\mathtt{n3}}}{(\ddagger)\ \texttt{n}(L) \Rightarrow_1 \{\texttt{[1,5]}, \texttt{[0,4]}, \texttt{[1,3]}\}}\,\mathsf{Top}_{\mathtt{n}} \quad \triangledown \quad \cdots \quad \triangledown}{(\wr)\ \texttt{\{[1,1][1,2][1,3][1,4]\}} \Rightarrow^{\mathsf{e}} \emptyset}\,\mathsf{Rl}_{\mathsf{e}}$$

$$\cfrac{(\wr)\ \texttt{\{[1,1][1,2][1,3][1,4]\}} \Rightarrow^{\mathsf{e}} \emptyset}{(\dagger)\ \texttt{\{[1,1][1,2][1,3][1,4]\}} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathtt{\{\_\}}}$$

Figure 16: Abbreviated tree $\star_2$

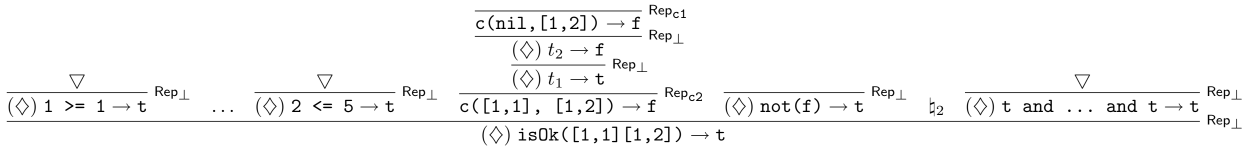

Figure 17: Abbreviated proof tree $\star_3$

be further rewritten. For example, we could consider terms of sorts `Nat` and `List` to be final and thus the nodes marked with (♡) in Figure 15 would be removed from the tree.

- Moreover, we consider that constructed terms are in normal form and thus they are automatically removed from the tree. For example, the nodes marked with (♠) in Figures 14 and 15 will be removed from the debugging tree.

5 Using the debugger

We introduce in this section the possible questions asked by the debugger and the commands used to interact with it.

5.1 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? $t \to t'$." 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.

Normal forms When a term cannot be further reduced and it is not a constructed term the debugger asks "Is t in normal form?," which is correct if the user expected t to be a normal form.

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 asks 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.

5.2 Commands

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

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

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

Once this mode is activated, the user can select and deselect statements by using[4]

```
(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, constructed terms having one of these sorts (or having a subsort of these sorts) 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 .)
```

[4] 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 the corresponding modes are activated.

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 .)
```

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 [in MODULE-NAME :] INITIAL-TERM -> NORMAL-FORM .)
(missing [in MODULE-NAME :] INITIAL-TERM : LEAST-SORT .)
(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 debugs erroneous normal forms, the second one erroneous least sorts, and the remaining ones refer to incomplete sets found when using search. More specifically, the third specifies a search in zero or more steps, the fourth one in one or more steps, and the last one only checks 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[5]

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

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* the statement 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 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 to a program statement and the user decides that it can be trusted, it is indicated with

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

When a question presents a judgment from 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 .)
```

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

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

```
(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 .)
```

6 Debugging session

We describe in this section how to debug the maze example shown 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. We start the debugging process with 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 questions about solutions not prioritized, and navigated with the default divide and query strategy. The first question is:

```
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. The next question is:

```
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 answer is no because the set of terms is incomplete: we expected to find the movement to the right too. The debugger asks now:

```
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 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` and the following question is prompted:

```
Is isOk([1,1][1,2][1,3][1,4]next([1,1][1,2][1,3][1,4])) in normal form?

Maude> (yes .)
```

This term is in normal form because we do not expect equations to be applied until the last term has been rewritten. The next question is:

```
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 debugger detects the problem with the next answer:

```
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:

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

Top-down strategy selected.
```

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.
```

We can follow how this strategy proceeds with the trees in Figures 14 and 16. Once we introduce the debugging command, the first question, which refers to the premises of the root in Figure 14 (although without some nodes, as the second one, deleted by the trusting mechanisms), is prompted:

```
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 .)
```

The eighth question (corresponding to the root of the tree in Figure 16, marked with (†)) is erroneous because position `[2,4]` is reachable from `[1,4]` and it is free of wall, so we do not expect this term to be final. The following questions are:[6]

```
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]

Question 2 :
Is isOk([1,1][1,2][1,3][1,4]next([1,1][1,2][1,3][1,4])) in normal form?

Maude> (1 : no .)
```

With this answer we have pointed out the node marked (‡) in Figure 16 as wrong. Since all its children correspond to applications of equations that were trusted (`n1`, `n2`, and `n3`, while the only suspicious statements were `is1` and `is2`), this node is now a leaf and thus it corresponds to a buggy node:

```
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.
```

7 Implementation

We show in this section how the ideas described in the previous sections are implemented. This implementation can be done in Maude itself by means of its reflective capabilities, that allow us to use Maude terms and modules as data [9, Chapter 14]. Sections 7.1 to 7.4 describe the tree construction stage, where the abbreviated proof trees are constructed by means of functions that receive the initial symptom, the module where it took place, a correct module (possibly empty), and a set of suspicious labels. The tree navigation is explained in Section 7.4, and the interaction with the user is explained in Section 7.5.

7.1 Proof tree definition

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

[6] Note that the child of this node, marked with ≀, is skipped because the corresponding equation has been trusted.

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

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

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

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

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

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

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

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

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

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

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

```
  op getContents : Tree NatList ~> X$Elt .
  op getForest : Tree NatList ~> Forest .
  op getSubTree : Tree NatList ~> Tree .
  op hasOffspring? : Tree NatList ~> Bool .

  ...
 endfm
```

We define now the sort `Judgment` to define the values kept in the debugging trees. When keeping reductions and memberships, we want to know the name of the statement associated with the node and the lefthand and righthand sides of the computation (or the term and sort of a membership).

```
 fmod DEBUGGING-TREE-NODE is
  pr META-LEVEL .
  sort Judgment .

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

If the type inferred is the least sort, we use the special notation below:

```
  op _:ls_ : Term Type -> Judgment [ctor format (d b o d)] .
```

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

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

and nodes in the many-steps tree:

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

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

```
op _=>f_ : Term Term -> Judgment [ctor] .
```

The nodes for debugging missing answers in system modules keep the initial term and the list of possible results. We distinguish between:

- The set of reachable terms in one step:

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

- The set of reachable terms by applying one rule:

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

- The set of reachable terms when many rewrite 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
                         -> Judgment [ctor format (d b o d d d d d d d d d d d d)] .
  ```

We use the operator `sol` to indicate (the Boolean value in the fourth argument) whether 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 it has a Boolean value (the fifth argument) 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 -> Judgment [ctor format (b o)] .
```

The operator `normal` indicates that a term is in normal form with respect to the equational theory:

```
op normal : Term -> Judgment [ctor format (r o)] .
```

Finally, we define a constant `unknown`, that will be used when the user answers `don't know` to any question:

```
 op unknown : -> Judgment [ctor] .
endfm
```

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

```
view DebuggingTreeNode from TRIV to DEBUGGING-TREE-NODE is
 sort Elt to Judgment .
endv
```

We obtain our tree by instantiating the module `TREE` above, mapping the sort `Elt` to `Judgment`:

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

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

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

Finally, the function `deleteSubTree` removes a subtree from the tree:

```
 op deleteSubTree : Tree NatList ~> Tree .
 ...
endfm
```

7.2 Auxiliary modules

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

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

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

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

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

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

When navigating the debugging tree, the user can decide to trust the statement associated to the current question. In this case, the tree will be pruned, deleting all the nodes associated with this statement:

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

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

The module STRAT defines the different navigation strategies available. The constant td stands for top-down, while dq refers to divide and query:

```
fmod STRAT is
 sort Strat .
 ops td dq : -> Strat .
endfm
```

The different types of tree that can be built are specified in the module TREE-TYPE below. The constant os identifies the one-step tree, while ms refers to the many-steps tree:

```
fmod TREE-TYPE is
 sort TreeType .
 ops os ms : -> TreeType .
endfm
```

The module DDState keeps the information needed to restore the state if the command undo is used: the current node, the debugging tree, and the navigation strategy:

```
fmod DDSTATE is
 pr NAT-LIST .
 pr PROOF-TREE .
 pr STRAT .

 sort DDState .

 op <_,_,_> : NatList Tree Strat -> DDState .
endfm
```

The view DDState is used to define a list of the previous states:

```
view DDState from TRIV to DDSTATE is
 sort Elt to DDState .
endv
```

The module SEARCH-TYPE defines the kinds of searches, 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 final zeroOrMore oneOrMore : -> SearchType .
endfm
```

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

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

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

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

- quitDefs, that deletes the memberships, equations, and rules definitions:

```
op quitDefs : Module -> Module [memo] .
eq quitDefs(fmod Q is IL sorts SS . SSDS ODS MAS EqS endfm) =
         fmod Q is IL sorts SS . SSDS ODS none none endfm .
eq quitDefs(mod Q is IL sorts SS . SSDS ODS MAS EqS RlS endm) =
         mod Q is IL sorts SS . SSDS ODS none none none endm .
```

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

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

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

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

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

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

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

- the functions `getLabel`, `getRighthand`, `getLefthand`, and `getCondition` extract the different parts of a rule:

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

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

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

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

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

- the function `onlyCtors` removes from the given module the operators that do not contain the `ctor` attribute. In order to obtain a valid module, it also removes all the equations, membership axioms, and rules:

  ```
  op onlyCtors : Module -> Module [memo] .
  eq onlyCtors(mod Q is IL sorts SS . SSDS ODS MAS EqS RlS endm) =
        mod Q is IL sorts SS . SSDS onlyCtors(ODS) none none none endm .
  eq onlyCtors(fmod Q is IL sorts SS . SSDS ODS MAS EqS endfm) =
        fmod Q is IL sorts SS . SSDS onlyCtors(ODS) none none endfm .

  op onlyCtors : OpDeclSet -> OpDeclSet [memo] .
  eq onlyCtors(none) = none .
  eq onlyCtors(op Q : TyL -> Ty [ctor AtS] . ODS) = op Q : TyL -> Ty [ctor AtS] .
                                                    onlyCtors(ODS) .
  eq onlyCtors(OD ODS) = onlyCtors(ODS) [owise] .
  ```

- the function `normalForm?` checks whether the term is in normal form with respect to the given module by using the predefined function `wellFormed` in the module that only contains constructors:

  ```
   op normalForm? : Module Term -> Bool .
   eq normalForm?(M, T) = wellFormed(onlyCtors(M), T) .

   ...
  endfm
  ```

The module `SUBSTITUTION` is in charge of applying substitutions to terms:

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

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

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

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

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

7.3 Debugging tree construction

In this section we describe how the different debugging trees are built. First, we describe the construction of debugging trees for wrong reductions, memberships, and rewrites and then we use them in the construction of the trees for erroneous normal forms, least sorts, and sets reachable terms. Instead of creating the complete proof trees and then abbreviating them, we build the abbreviated proof trees directly.

7.3.1 Debugging trees for wrong reductions and memberships

The function `createTree` builds debugging trees for wrong reductions and memberships. It exploits the fact that the equations and membership axioms are both *terminating* and *confluent*. It receives the module where a wrong inference took place, a correct module (or the constant `undefMod` when no such module is provided) to prune the tree, the initial term, the (erroneous) result obtained, and the set of suspicious statement labels. It keeps the initial reduction as the root of the tree and uses an auxiliary function `createForest` that, in addition to the arguments received by `createTree`, receives the module "cleaned" of suspicious statements (by using `deleteSuspicious`), and generates the forest of abbreviated trees corresponding to the reduction between the two terms given as arguments. The transformed module is used to improve the efficiency of the tree construction, because we can use it to check whether a term reaches its final form by using only trusted statements, preventing the debugger from building a tree that will be finally empty.

```
fmod FUNCTIONAL-TREE-CONSTRUCTION is

  op createTree : Module Maybe{Module} Term Term QidSet -> Tree .
  ceq createTree(M, CM, T, T', QS) =
              contract(tree('root@#$% : T -> T', getOffspring*(F) + 1, F))
   if ST? := strat?(M) /\
      M' := deleteSuspicious(M, QS) /\
      F := createForest(M, M', CM, T, T', QS) .
```

We use the function `createForest` to create a forest of abbreviated trees. It receives as parameters the module where the computation took place, the transformed module (that only contains trusted statements), a correct module (possibly `undefMod`) to check the inferences, two terms representing the inference whose proof tree we want to generate, and a set of labels of suspicious equations and memberships. The function checks whether the terms are equal, the result can be reached by using only trusted statements, or the correct module can calculate this inference; in such cases, there is no need to calculate the tree, so the empty forest is returned. Otherwise, it applies the function `createForest2`:

```
  op createForest : Module Module Maybe{Module} Term Term QidSet ~> Forest .
  eq createForest(OM, TM, CM, T, T', QS) =
   if T == T' or-else reduce(TM, T) == T' or-else reduce(CM, T) == T' then mtForest
   else createForest2(OM, TM, CM, T, T', QS)
   fi .
```

The function `createForest2` checks first whether is of the form `if T1 then T2 else T3 fi`. In this case, the debugger evaluates `T1` and then, depending on the result, it evaluates either `T2` or `T3` following the same evaluation strategy than Maude:

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

In other case, the debugger follows the Maude innermost strategy: it first tries to fully reduce the subterms (by means of the function `reduceSubterms`), and once all the subterms have been reduced, if the result is not the final one, it tries to reduce at the top (by using the function `applyEq`), to reach the final result by *transitivity*:

```
ceq createForest2(OM, TM, CM, T, T', QS) =
                                   if T'' == T' then F
                                   else F applyEq(OM, TM, CM, T'', T', QS)
                                   fi
 if < T'', F > := reduceSubterms(OM, TM, CM, T, QS) [owise] .
```

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

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

ceq reduceSubterms(OM, TM, CM, Q[TL], QS) = < normal(OM, Q[TL']), F >
 if < TL', F > := reduceSubterms(OM, TM, CM, TL, QS, < empty, mtForest >) .
eq reduceSubterms(OM, TM, CM, T, QS) = < T, mtForest > [owise] .

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

The function `applyEq` tries to apply (at the top) one equation,[7] by using the *replacement* rule from Figure 1, with the constraint that we cannot apply equations with the `otherwise` attribute if other equations can be applied. To apply an equation we check whether the term we are trying to reduce matches the lefthand side of the equation and its conditions are fulfilled. If this happens, we obtain a substitution (from both the matching with the lefthand side and the conditions) that we can apply to the righthand side of the equation. Note that, if we can obtain the transition in the correct module, the forest is not computed:

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

eq applyEq(OM, TM, CM, T, T', QS) =
       if reduce(TM, T) == T' or-else reduce(CM, T) == T' then mtForest
       else applyEq(OM, TM, CM, T, T', QS, getEqs(OM))
       fi .
```

First, we try to apply the equations without the `otherwise` attribute:

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

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

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

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

Finally, when no equation is applicable, the function returns noProof:

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

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

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

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

The case of the matching conditions is very similar. We generate the forest for the normal form of the righthand side:

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

In the membership case, we use the version of createForest that builds a forest for a membership inference where the sort is the least one assignable to the term in the condition:

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

Finally, the empty condition generates the empty forest:

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

 ...
endfm
```

7.3.2 Debugging trees for wrong rewrites

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

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

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

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

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

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

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

The function oneStepTree creates a complete debugging tree with only one-step rewrites in its nodes. It puts as root of the tree the complete inference, computes the tree for the reduction from the initial term to normal form with the function createForest from Section 7.3.1, and then computes the rest of the tree with the function oneStepForest. This correspond to a concrete application of the *equivalence class* inference rule from Figure 1:

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

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

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

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

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

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

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

The first equation of stepForest checks whether the value can be obtained by using trusted statements or the correct module:

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

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

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

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

```
ceq stepForest(OM, TM, CM, T, T', R, QS, B, TT) = if trusted?(R, QS) then F F'
        else tree(getLabel(R) : T1 =>1 T3, getOffspring*(F) + 1, F) F' fi
 if < SB, C > := getInfo(OM, T, T', getLabel(R), 0) /\
    T1 := substitute(OM, getLefthand(R), SB) /\
    T2 := substitute(OM, getRighthand(R), SB) /\
    T3 := reduce(OM, T2) /\
    F := conditionForest(substitute(OM, getCondition(R), SB), OM, TM, CM, QS, B, TT)
         createForest(OM, TM, CM, T2, T3, QS, strat?(OM)) /\
    F' := createForest(OM, TM, CM, substituteHole(OM, C, T3), T', QS, strat?(OM)) [owise] .
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
op trace2tree : Module Module Maybe{Module} Trace Term QidSet Bound Forest Nat -> Tree .
eq trace2tree(OM, TM, CM, nil, T', QS, B, F, N) = mtForest .
ceq trace2tree(OM, TM, CM, {T, Ty, R}, T', QS, B, F, N) = divide(F, N)
 if secure?(TM, CM, T, T', B) .
eq trace2tree(OM, TM, CM, {T, Ty, R}, T', QS, B, F, N) =
               divide(F tree(T =>f T', 1, mtForest), s(N)) [owise] .
ceq trace2tree(OM, TM, CM, {T, Ty, R} {T', Ty', R'} TR, T'', QS, B, F, N) =
          trace2tree(OM, TM, CM, {T', Ty', R'} TR, T'', QS, B, F, N)
 if secure?(TM, CM, T, T', B) .
eq trace2tree(OM, TM, CM, {T, Ty, R} {T', Ty', R'} TR, T'', QS, B, F, N) =
          trace2tree(OM, TM, CM, {T', Ty', R'} TR, T'', QS, B, F
                                           tree(T =>f T', 1, mtForest), s(N)) [owise] .
```

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

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

We consider as basic case the forest composed by only one tree:

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

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

```
ceq divide(F, N) = tree(getFirstTerm(A) =>+ getSecondTerm(A'), N1 + 1, A A')
 if N > 1 /\
    N' := N quo 2 /\
    < F', F'' > := takeDrop(F, N') /\
    A := divide(F', N') /\
    A' := divide(F'', sd(N, N')) /\
    N1 := getOffspring(A) + getOffspring(A') .
```

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

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

  op takeDrop : Forest Nat Forest -> Pair{Forest, Forest} .
  eq takeDrop(A F, s(N), F') = takeDrop(F, N, F' A) .
  eq takeDrop(F, N, F') = < F', F > [owise] .
 endfm
```

7.3.3 Debugging trees for missing answers

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} .
  pr MAYBE{Type} * (op maybe to typeError) .
```

The debugging tree for normal forms is built with the command createMissingTree. It receives the module where the reduction took place, a correct module, the initial term, the reached normal form, and a set of suspicious labels:

```
  op createMissingTree : Module Maybe{Module} Term Term QidSet -> Tree .
  ceq createMissingTree(M, CM, T, T', QS) = tree('root : T -> T'', getOffspring*(F) + 1, F)
   if TM := deleteSuspicious(M, QS) /\
      T'' := reduce(M, T') /\
      F := cleanTree*(M, false, none, createMissingForest(M, TM, CM, T, T'', QS)) .
```

The function createMissingForest checks whether the result can be obtained in the trusted or correct modules. When this happens, it only generates a forest proving the term is in normal form with proveNormal; in other case, it uses the auxiliary function createMissingForest2:

```
  op createMissingForest : Module Module Maybe{Module} Term Term QidSet -> Forest .
  ceq createMissingForest(OM, TM, CM, T, T', QS) = F
   if T == T' or-else reduce(TM, T) == T' or-else reduce(CM, T) == T' /\
      F := proveNormal(OM, TM, CM, T', QS) .
  eq createMissingForest(OM, TM, CM, T, T', QS) =
          createMissingForest2(OM, TM, CM, T, T', QS) [owise] .
```

createMissingForest2 generates the forest for the subterms with reduceSubtermsMissing and then distinguishes whether the final result has been reached, proving in that case the term is in normal form with proveNormal, or not, applying then the next equation with applyEqMissing:

```
  ceq createMissingForest2(OM, TM, CM, T, T', QS) =
                              if T'' == T' then F proveNormal(OM, TM, CM, T', QS)
                              else F applyEqMissing(OM, TM, CM, T'', T', QS)
                              fi
   if < T'', F > := reduceSubtermsMissing(OM, TM, CM, T, QS) [owise] .
```

The function reduceSubtermsMissing is in charge of reducing the subterms of the given term to normal form, returning a pair with the new term and the forest proving these reductions. It uses an auxiliary function reduceSubtermsMissing with an accumulator to keep the temporary results:

```
  op reduceSubtermsMissing : Module Maybe{Module} Module Term QidSet
                             -> Pair{TermList, Forest} .
  ceq reduceSubtermsMissing(OM, TM, CM, Q[TL], QS) = < normal(OM, Q[TL']), F >
   if < TL', F > := reduceSubtermsMissing(OM, TM, CM, TL, QS, < empty, mtForest >) .
  eq reduceSubtermsMissing(OM, TM, CM, T, QS) = < T, mtForest > [owise] .
```

This auxiliary function traverses the subterms, computing the appropriate tree with createMissingForest, and returning the accumulated pair once no more terms are available:

```
  op reduceSubtermsMissing : Module Maybe{Module} Module Term QidSet
                             Pair{TermList, Forest} -> Pair{TermList, Forest} .
  eq reduceSubtermsMissing(OM, TM, CM, empty, QS, < TL, F >) = < TL, F > .
  ceq reduceSubtermsMissing(OM, TM, CM, (T, TL'), QS, < TL, F >) =
           reduceSubtermsMissing(OM, TM, CM, TL', QS,
                      < (TL, T'), F createMissingForest(OM, TM, CM, T, T', QS) >)
   if T' := reduce(OM, T) .
```

The function applyEqMissing checks whether the current term is in normal form or the reduction is correct in the trusted or correct modules. In these cases, the empty forest is returned, while in other case applyEqMissing2 is used.

```
  op applyEqMissing : Module Module Maybe{Module} Term Term QidSet
                      -> Maybe{Forest} .
  eq applyEqMissing(OM, TM, CM, T, T', QS) =
         if reduce(TM, T) == T' or-else reduce(CM, T) == T' then mtForest
         else applyEqMissing2(OM, TM, CM, T, T', QS, getEqs(OM))
         fi .
```

applyEqMissing2 tries to apply one equation and then apply createMissingForest to the obtained term until the normal form is reached, following the approach for wrong answers. We first try to use equations defined without the owise attribute, and if they cannot be used then equations with this attribute are tried. This function returns noProof when the normal form have not been obtained but no equations can be applied to the current term:

```
  op applyEqMissing2 : Module Module Maybe{Module} Term Term QidSet
                      EquationSet -> Maybe{Forest} .
  ceq applyEqMissing2(OM, TM, CM, T, T', QS, Eq EqS) =
             if in?(AtS, QS) then
             tree(label(AtS) : T -> T', getOffspring*(F) + 1, F)
             else F
             fi
   if ceq LHS = RHS if C [AtS] . := generalEq(Eq) /\
      not owise?(AtS) /\
      sameKind(OM, type(OM, LHS), type(OM, T)) /\
      SB := metaMatch(OM, LHS, T, C, 0) /\
      RHS' := substitute(OM, RHS, SB) /\
      F := conditionForest(substitute(OM, C, SB), OM, TM, CM, QS)
           createMissingForest(OM, TM, CM, RHS', T', QS) .
  ceq applyEqMissing2(OM, TM, CM, T, T', QS, Eq EqS) =
            if in?(AtS, QS) then
            tree(label(AtS) : T -> T', getOffspring*(F) + 1, F)
            else F
            fi
   if ceq LHS = RHS if C [AtS] . := generalEq(Eq) /\
      owise?(AtS) /\
      sameKind(OM, type(OM, LHS), type(OM, T)) /\
      SB := metaMatch(OM, LHS, T, C, 0) /\
      RHS' := substitute(OM, RHS, SB) /\
      F := conditionForest(substitute(OM, C, SB), OM, TM, CM, QS)
           createMissingForest(OM, TM, CM, RHS', T', QS) .
  eq applyEqMissing2(OM, TM, CM, T, T', QS, EqS) = noProof [owise] .
```

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

```
  op match? : Module Term Equation -> Bool .
  ceq match?(M, T, eq T1 = T2 [AtS] .) = metaMatch(M, T', T, nil, 0) :: Substitution
   if T' := kindTerm(M, T1) /\
      sameKind(M, type(M, T), type(M, T')) .
  ceq match?(M, T, ceq T1 = T2 if C [AtS] .) = metaMatch(M, T', T, nil, 0) :: Substitution
   if T' := kindTerm(M, T1) /\
      sameKind(M, type(M, T), type(M, T')) .
  eq match?(M, T, Eq) = false [owise] .
```

The function buildConditionForestEqs proves that no equations can be applied to the given term. It uses an auxiliary function buildConditionForestEqs* that traverses all the equations in the module and check that each of them cannot be applied with buildConditionForestEqs2:

```
op buildConditionForestEqs : Module Module Maybe{Module} Term QidSet
                            -> Forest .
eq buildConditionForestEqs(OM, TM, CM, T, QS) =
             buildConditionForestEqs*(OM, TM, CM, T, QS, getEqs(OM)) .

op buildConditionForestEqs* : Module Module Maybe{Module} Term QidSet EquationSet
                             -> Forest .
eq buildConditionForestEqs*(OM, TM, CM, T, QS, none) = mtForest .
eq buildConditionForestEqs*(OM, TM, CM, T, QS, Eq EqS) =
        buildConditionForestEqs2(OM, TM, CM, T, QS, Eq)
        buildConditionForestEqs*(OM, TM, CM, T, QS, EqS) .
```

buildConditionForestEqs2 checks whether the term with the variables at the kind level matches the lefthand side of the equation. When it does not match the equation is discarded, while when the term matches the lefthand side we distinguish whether the equation is conditional or not. In the first case the function only proves that the term at the sort level does not match the lefthand side with sort-match (remember we are proving the equation cannot be applied), while in the second one we have also to compute why the condition does not hold, which is proved with buildConditionForestMissingAux:

```
op buildConditionForestEqs2 : Module Module Maybe{Module} Term QidSet Equation -> Forest .
ceq buildConditionForestEqs2(OM, TM, CM, T, QS, Eq) = F
 if not match?(OM, T, Eq) /\
    F := createForest(OM, TM, CM, T, getTerm(metaReduce(OM, T)), QS, strat?(OM)) .

ceq buildConditionForestEqs2(OM, TM, CM, T, QS, eq T1 = T2 [AtS] .) =
          F sort-match(OM, TM, CM, QS, T1, T)
 if SSB := allSubs(OM, T, T1 := T, 0, mtSSB) /\
    < F, TL > := applySubs(OM, TM, CM, QS, T2, SSB) [owise] .

ceq buildConditionForestEqs2(OM, TM, CM, T, QS, ceq T1 = T2 if C [AtS] .) =
          F sort-match(OM, TM, CM, QS, T1, T)
 if SSB := allSubs(OM, T, T1 := T, 0, mtSSB) /\
    < F, TL > := buildConditionForestMissingAux(OM, TM, CM, QS, T, T2,
                               T1 := T /\ C, SSB, 2, getNumConds(C) + 1) [owise] .
```

The function buildConditionForestMissingAux receives, in addition to the usual parameters, the condition to be checked, the set of substitutions that satisfy the condition thus far, the next atomic condition to be evaluated, and the total number of atomic conditions. When the index of the current atomic condition is greater than the number of conditions, it just computes the forest proving the reductions of the terms once the substitutions are applied with applySubs:

```
op buildConditionForestMissingAux : Module Module Maybe{Module} QidSet Term Term
                                    Condition Set{Substitution} Nat Nat
                                    -> Pair{Forest, TermList} .
eq buildConditionForestMissingAux(OM, TM, CM, QS, T, T2, C, SSB, s(N), N) =
              applySubs(OM, TM, CM, QS, T2, SSB) .
```

When the set of substitutions fulfilling the condition thus far is empty, the empty forest is returned:

```
eq buildConditionForestMissingAux(M, TM, CM, QS, T, T2, C, mtSSB, N, N') =
                        < mtForest, empty > .
```

While the function does not reach the last atomic condition, we extend the condition thus far with the next atomic condition with getNFirst, obtain the set of substitutions that fulfill this extended condition with allSubs, apply the previous set of substitutions to the atomic condition, and obtain the forest proving the extension of the set with substituteAndCreateForest:

```
ceq buildConditionForestMissingAux(M, TM, CM, QS, T, T2, C, SSB, N, N') =
                        < F F', TL' >
```

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

The function substituteAndCreateForest traverses the set of substitutions applying the auxiliary function createForestOnceSubstituted to each obtained condition:

```
op substituteAndCreateForest : Module Module Maybe{Module} QidSet Condition
                             Set{Substitution} -> Forest .
eq substituteAndCreateForest(OM, TM, CM, QS, C, mtSSB) = mtForest .

ceq substituteAndCreateForest(M, TM, CM, QS, C, SB . SSB) = F F'
 if F := createForestOnceSubstituted(M, TM, CM, QS, substitute(M, C, SB)) /\
    F' := substituteAndCreateForest(M, TM, CM, QS, C, SSB) .
```

createForestOnceSubstituted distinguishes between the different kinds of condition. For example, when a matching condition is found the forest for the reduction to normal form of the righthand side is computed with createMissingForest, while the forest for the matching is computed with the auxiliary function sort-match:

```
op createForestOnceSubstituted : Module Module Maybe{Module} QidSet Condition
                               -> Forest .
ceq createForestOnceSubstituted(OM, TM, CM, QS, T := T') =
      createMissingForest(OM, TM, CM, T', T'', QS)
      sort-match(OM, TM, CM, QS, T, T'')
 if T'' := reduce(OM, T') .
```

The debugging tree for incomplete sets of reachable terms 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 final sorts, a Boolean value indicating whether the search introduced by the user was unbounded, and a Boolean value pointing out whether 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 whether the current term fulfills the condition, and then examines whether 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 createMissingForest(OM, TM, CM, T, reduce(OM, T), QS)
                   sort-match(OM, TM, CM, QS, PAT, T)
                   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 whether 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 .
ceq solveCondition(M, T, PAT, C) = metaMatch(M, PAT, T, C, 0) =/= noMatch
 if sameKind(M, type(M, T), type(M, PAT)).
eq solveCondition(M, T, PAT, C) = false [owise] .
```

When 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 whether 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 whether 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 whether the terms match:

```
op match? : Module Term Rule -> Bool .
ceq match?(M, T, rl T1 => T2 [AtS] .) = metaMatch(M, T', T, nil, 0) :: Substitution
 if T' := kindTerm(M, T1) /\
    sameKind(M, type(M, T), type(M, T')) .
ceq match?(M, T, crl T1 => T2 if C [AtS] .) = metaMatch(M, T', T, nil, 0) :: Substitution
 if T' := kindTerm(M, T1) /\
    sameKind(M, type(M, T), type(M, T')) .
eq match?(M, T, R) = false [owise] .
```

If the terms match in an unconditional rule, the function computes all the substitutions obtained from the 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 whether 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 [owise] .
```

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?)
        sort-match(M, TM, CM, QS, T, T') .
  ```

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

```
ceq createForestOnceSubstituted(M, TM, CM, QS, FS, BW, ST, TTW, TTM, SP, ST?, T = T') =
        if T1 == T2 then
          createForest(M, TM, CM, T, getTerm(metaReduce(M, T)), QS, ST?)
          createForest(M, TM, CM, T', getTerm(metaReduce(M, T')), QS, ST?)
        else
          createMissingForest(M, TM, CM, T, T1, QS)
          createMissingForest(M, TM, CM, T', T2, QS)
        fi
 if T1 := getTerm(metaReduce(M, T)) /\
    T2 := getTerm(metaReduce(M, T')) .
```

- 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 initial forest is returned:

```
ceq createForestOnceSubstituted(M, TM, CM, QS, FS, BW, ST, TTW, TTM, SP, ST?, T : Ty) =
                   createMissingForest(M, TM, CM, T, T', QS) F
 if RP := metaReduce(M, T) /\
    T' := getTerm(RP) /\
    Ty' := getType(RP) /\
    sortLeq(M, Ty', Ty) /\
    F := createForest(M, TM, CM, T, T', QS, ST?)
         createForest(M, TM, CM, T', Ty', QS) .

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

- Finally, rewrite conditions generate a forest for 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 combined 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 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*, that receives a Boolean value indicating whether the final mode is active, the set of final sorts and a forest, and traverses this forest applying cleanTree to each tree:

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

- 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(J, N, F)) =
                              tree(J, 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 function `normalForm?` whether the term is a constructed term and then it traverses the list of sorts checking whether 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 normalForm?(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(J, N, F)) =
        tree(J, 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
```

7.4 Debugging tree navigation

Regarding the navigation of the debugging tree, we have implemented two strategies. In the top-down strategy the selection of the next node of the debugging tree is done by the user, thus we do not need any extra function to compute it (apart from those in module TREE). On the other hand, the divide and query strategy selects each time the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise.

The algorithm in charge of calculating the next node uses auxiliary functions that return pairs consisting of a list of natural numbers (identifying the best node found thus far) and a natural number (specifying the cost associated with this node):

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

  var  J : Judgment .
  vars N N' NODES LAST_DIFF BEST_DIFF NEW_DIFF : Nat .
  var  F : Forest .
  vars NL NL' BEST_NODE : NatList .
  var  T : Tree .
```

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

```
  op searchBestNode : Tree -> NatList .

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

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

```
  op searchBestNode : Tree Nat Nat Nat NatList NatList -> Pair{NatList, Nat} .
  ceq searchBestNode(T, NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL) =
                                                  < BEST_NODE, BEST_DIFF >
   if LAST_DIFF <= getDiff(NODES, getOffspring(T)) .
```

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

```
  ceq searchBestNode(tree(J, N, F), NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL) =
       if J == unknown then
         searchBestNode*(F, NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL, 0)
       else
         if NEW_DIFF <= BEST_DIFF then
           searchBestNode*(F, NODES, NEW_DIFF, NEW_DIFF, NL, NL, 0)
```

```
        else
          searchBestNode*(F, NODES, NEW_DIFF, BEST_DIFF, BEST_NODE, NL, 0)
        fi
      fi
   if NEW_DIFF := getDiff(NODES, N) /\
      LAST_DIFF > NEW_DIFF .
```

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

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

  eq searchBestNode*(mtForest, NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL, N) =
                                                   < BEST_NODE, BEST_DIFF > .

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

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

7.5 The debugger environment

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

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

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

  op debug_. : @Bubble@ -> @Command@ .
  op 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@ .
  op start`. : -> @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 wrong answers returns a tuple containing the generated tree, the module where the computation took place, the set of suspicious statements, and a list of quoted identifiers indicating the errors occurred during the parsing:

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

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

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

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

After finding out the kind of the debugging command (reduction, membership, or rewrite) and if a module name has been selected by the command, the function procDebug builds the appropriate tree by using the functions createTree and createRewTree explained in Section 7.3:

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

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

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

  op procInclude : Term Database -> IncludePair .

  ...
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] .
```

The new attributes are:

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

  ```
  op tree :_ : Forest -> Attribute [ctor] .
  ```

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

  ```
  op treeType :_ : TreeType -> Attribute [ctor] .
  ```

- the type of tree for wrong answers used in the current session is kept in order to use it when trees are built on demand:

  ```
  op currentTTW :_ : TreeType -> Attribute [ctor] .
  ```

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

  ```
  op treeTypeMissing :_ : TreeType -> Attribute [ctor] .
  ```

- 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 [ctor] .
  ```

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

```
op current :_ : NatList -> Attribute [ctor] .
```

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

```
op strategy :_ : Strat -> Attribute [ctor] .
```

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

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

- the correct module used to prune the debugging tree (correction). Since this value is optional, it has also sort Maybe{Module}, and its value is initially undefMod:

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

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

```
op bound :_ : Bound -> Attribute [ctor] .
```

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

```
op select :_ : Bool -> Attribute [ctor] .
```

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

```
op finalSelect :_ : Bool -> Attribute [ctor] .
```

- 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 [ctor] .
```

- the set of labels considered suspicious:

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

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

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

- the set of final sorts:

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

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

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

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

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

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

```
sort ToolState .
ops waiting computing finished missingEnd : -> ToolState [ctor] .

op state :_ : ToolState -> Attribute [ctor] .
```

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

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

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

```
op condition :_ : Condition -> Attribute [ctor] .
```

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

```
op solutionsPrioritized :_ : Bool -> Attribute [ctor] .
```

- 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 [ctor] .
```

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

```
op searchType :_ : SearchType -> Attribute [ctor] .
```

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

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

*** Initial values of the attributes (except input and output)
eq init-state = db : initialDatabase,
                default : 'CONVERSION, tree : mtForest, treeType : os, currentTTW : os,
                treeTypeMissing : os, currentTTM : os, current : nil, strategy : dq,
                module : undefMod, correction : undefMod, bound : 42, select : false,
                finalSelect : false, currentFS : false, suspicious : none,
                currentSuspicious : none, finalSorts : none, currentFinal : none,
                previousStates : nil, answers : nil, state : waiting, pattern : maybe,
                condition : nil, solutionsPrioritized : false, currentSP : false,
                searchType : final .
```

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

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

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

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—a function analogous to procDebug—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) .
```

The selection/deselection of the correct module is handled by the following rules:

- If a correct module expression is introduced, correct-module keeps the associated module if it exists, and shows an error message in other case.

```
crl [correct-module] :
    < O : DDDC | db : DB, input : ('correct`module_.[T]), output : nil, correction : MM, AtS >
 => if M? :: Module
    then < O : DDDC | db : DB, input : nilTermList, output : ('\n add-spaceR(printME(ME)) '\b
                                                             'selected 'as 'correct 'module. '\o '\n),
                      correction : M?, AtS >
    else < O : DDDC | db : DB, input : nilTermList, output : ('\n '\r 'Error: '\o getMsg(M?)),
                      correction : MM, AtS >
    fi
 if ME := parseModExp(T) /\
    M? := if compiledModule(ME, DB)
          then getFlatModule(ME, DB)
          else getFlatModule(modExp(evalModExp(ME, DB)), database(evalModExp(ME, DB)))
          fi .
```

- `correct-module-error` deals with the error when a wrong module expression is introduced:

```
crl [correct-module-error] :
    < O : DDDC | input : ('correct`module_.[T]), output : nil, AtS >
 => < O : DDDC | input : nilTermList, output : ('\n '\r 'Error: '\o 'Wrong
                                               'module 'expression. '\n), AtS >
 if ME? := parseModExp(T) /\
    not (ME? :: ModuleExpression) .
```

- The rule `delete-correct-module` sets the value of the correct module to `undefMod`:

```
rl [delete-correct-module] :
   < O : DDDC | input : ('delete`correct`module`..@Command@), output : nil,
                correction : MM, AtS >
=> < O : DDDC | input : nilTermList, output : ('\n '\b 'Correct 'module 'deleted. '\o '\n),
                correction : undefMod, AtS > .
```

The rule `top-down-strategy` fixes the value of the navigation strategy to `td`, and changes the state to `computing` if the debugging has not finished to show the appropriate question:

```
rl [top-down-strategy] :
   < O : DDDC | input : ('top-down`strategy`..@Command@), output : nil,
                strategy : STRAT, state : TS, AtS >
=> < O : DDDC | input : nilTermList, output : ('\n '\b 'Top-down 'strategy
                                              'selected. '\o '\n),
                strategy : td, state : if TS == finished then TS
                                                         else computing fi, AtS > .
```

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

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

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

```
crl [top-down-traversal-no] :
    < O : DDDC | input : ('_:`no`.['token[T]]), strategy : td, tree : PT,
                 previousStates : LS, answers : LA, state : waiting, AtS >
 => < O : DDDC | input : nilTermList, strategy : td, tree : PT',
                 previousStates : LS < nil, PT, td >,
                 answers : LA getAnswer(PT', wrong), state : computing, AtS >
```

```
  if UPT := removeUnknownChildren(PT) /\
     N := downNat*(T) /\
     N > 0 /\
     N <= size(getForest(UPT, nil)) /\
     PT' := getSubTree(UPT, sd(N, 1)) .
```

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

The rule missing-wrong is used when, debugging missing answers with 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 divide and query strategy is selected and the user decides to trust a statement, the current subtree is deleted and the resulting tree pruned in order to delete the nodes associated with the trusted statement:

```
crl [divide-query-traversal] :
    < O : DDDC | input : ('trust`..@Command@), strategy : dq, tree : PT,
                 current : NL, previousStates : LS, answers : LA,
                 state : waiting, AtS >
 => < O : DDDC | input : nilTermList, strategy : dq, tree : PT', current : NL,
                 previousStates : LS < NL, PT, dq >,
                 answers : LA getAnswer(getSubTree(PT, NL), right),
                 state : computing, AtS >
if Q := getLabel(PT, NL) /\
   PT' := prune(deleteSubTree(PT, NL), Q) .
```

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))) .
```

When a question related to a set of reachable solutions is presented to the user, he can point that one of these terms is not a solution, although it is reachable. In this case, the rule `no-result` checks that the question was really related to an inference of reachable solutions with `solutionsInference?` and that the term selected by the user exists, and then computes the tree of wrong answers for the fulfillment of the condition with `conditionForest`:

```
crl [no-result] :
    < O : DDDC | input : ('_is`not`a`solution`.['token[T]]), strategy : dq,
                 tree : PT, current : NL, module : M, correction : MM, treeType : TT,
                 bound : BND, currentSuspicious : QS, state : waiting,
                 condition : C, pattern : PAT, previousStates : LS,
                 answers : LA, AtS >
 => < O : DDDC | input : nilTermList, strategy : dq, current : NL,
                 tree : tree(sol(T2, PAT, C, true, true, true), 1 + getOffspring*(F), F),
                 module : M, correction : MM, treeType : TT, bound : BND,
                 currentSuspicious : QS, state : computing, condition : C,
                 pattern : PAT, previousStates : LS < NL, PT, dq >,
                 answers : LA getAnswer(PT, wrong), AtS >
 if N := downNat*(T) /\
    solutionsInference?(getContents(PT, NL)) /\
    N > 0 /\
    N <= numTermsInRootSet(getSubTree(PT, NL)) /\
    T2 := getWrongTerm(getSubTree(PT, NL), N) /\
    F := conditionForest(substitute(M, C, metaMatch(M, PAT, T2, C, 0)),
                                 M, deleteSuspicious(M, QS), MM, QS, BND, TT) .
```

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) .
```

When the user decides to switch the select mode on to use a subset of the labeled statements as suspicious, the `select` attribute is set to `true`:

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

The rules solution-expanded-on and solution-expanded-off are used to select and deselect if the solutions are expanded:

```
  rl [solution-expanded-on] :
     < O : DDDC | input : ('solutions`prioritized`on`..@Command@),
                  solutionsPrioritized : B, output : nil, AtS >
  => < O : DDDC | input : nilTermList, solutionsPrioritized : true,
                  output : ('\n '\b 'Solutions 'are 'prioritized. '\o '\n), AtS > .

  rl [solution-expanded-off] :
     < O : DDDC | input : ('solutions`prioritized`off`..@Command@),
                  solutionsPrioritized : B, output : nil, AtS >
  => < O : DDDC | input : nilTermList, solutionsPrioritized : false,
                  output : ('\n '\b 'Solutions 'are 'not 'prioritized. '\o '\n), AtS > .

  ...

 endm
```

The module DD manages the introduction of data by the user and the output of the debugger's answers. Full Maude uses the input/output facility provided by the LOOP-MODE module [9, Chapter 17], that consists 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 .
```

8 Conclusions and future work

We have presented in this paper a declarative debugger for Maude specifications. The debugging trees used in the debugging process are obtained from an abbreviation of a proper calculus whose adequacy for debugging has been proved. This work comprises our previous work on wrong [22, 6, 25] and missing answers [24, 23] and provides a powerful and complete debugger for Maude specifications. Moreover, we also provide a graphical user interface that eases the interaction with the debugger and allows to traverse the debugging tree with more freedom. The tree construction, its navigation, and the user interaction (excluding the GUI) have all been implemented in Maude itself. For more information, see `http://maude.sip.ucm.es/debugging`.

We plan to add new navigation strategies like the ones shown in [27] that take into account the number of different potential errors in the subtrees, instead of their size. Moreover, 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 intend to add a new command to introduce *complete* modules, which would greatly reduce the number of questions asked to the user. Finally, we also plan to create a test generator to test Maude specifications and debug the erroneous tests with the debugger.

References

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

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

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

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

[5] R. Caballero, C. Hermanns, and H. Kuchen. Algorithmic debugging of Java programs. In F. J. López-Fraguas, editor, *15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain*, volume 177 of *Electronic Notes in Theoretical Computer Science*, pages 75–89. Elsevier, 2007.

[6] 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.

[7] R. Caballero, M. Rodríguez-Artalejo, and R. del Vado Vírseda. Declarative diagnosis of missing answers in constraint functional-logic programming. In J. Garrigue and M. V. Hermenegildo, editors, *Proceedings of 9th International Symposium on Functional and Logic Programming, FLOPS 2008, Ise, Japan*, volume 4989 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2008.

[8] O. Chitil and Y. Luo. Structure and properties of traces for functional programs. In I. Mackie, editor, *Proceedings of the Third International Workshop on Term Graph Rewriting (TERMGRAPH 2006)*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 39–63, Amsterdam, The Netherlands, The Netherlands, 2007. Elsevier.

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

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

[11] 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.

[12] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In T. Nipkow, editor, *Proceedings of the 9th International Conference on Rewriting Techniques and Applications (RTA 98)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1998.

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

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

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

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

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

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

[19] B. Pope. *A Declarative Debugger for Haskell.* PhD thesis, The University of Melbourne, Australia, 2006.

[20] 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`.

[21] 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`.

[22] 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.

[23] A. Riesco, A. Verdejo, and N. Martí-Oliet. Declarative debugging of missing answers for Maude specifications. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*, Leibniz International Proceedings in Informatics, 2010. To appear.

[24] A. Riesco, A. Verdejo, and N. Martí-Oliet. Enhancing the debugging of Maude specifications. In *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications (WRLA 2010)*, Lecture Notes in Computer Science, 2010. To appear.

[25] 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.

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

[27] 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.

[28] 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.

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