International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 277-294
http://rewriting.loria.fr/rta/

DECLARATIVE DEBUGGING OF MISSING ANSWERS FOR MAUDE SPECIFICATIONS

ADRIÁN RIESCO [1] AND ALBERTO VERDEJO [1] AND NARCISO MARTÍ-OLIET [1]

[1] Facultad de Informática, Universidad Complutense de Madrid, Spain
E-mail address: ariesco@fdi.ucm.es,{alberto,narciso}@sip.ucm.es

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 of 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. In this paper we propose a calculus to infer normal forms and least sorts with the equational part, 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 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.

1. Introduction

Declarative debugging [20], 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 [12, 22], functional [14, 15], multi-paradigm [3, 9], and object-oriented [4] programming languages; and debugging of *missing answers* [5, 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* [13], 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)

1998 ACM Subject Classification: D.2.5 [Software Engineering]: Testing and Debugging – Debugging aids, D.3.2 [PROGRAMMING LANGUAGES]: Language Classifications – Nondeterministic languages.

Key words and phrases: Declarative debugging, Maude, Missing answers, Rewriting Logic.

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

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.277

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 [21].

In this paper we present a declarative debugger of missing answers for *Maude specifications*. Maude [6] is a high-level language and high-performance system supporting both equational and rewriting logic computation. The Maude system supports several approaches for debugging: tracing, term coloring, and using an internal debugger [6, Chap. 22]. 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 in Maude specifications was already studied in [17], where we presented how to debug wrong results due to errors in the statements of the specification. In [18] we extended the concept of missing answers, usually attached to incomplete sets of results, to deal with erroneous normal forms and least sorts in equational theories. However, in a nondeterministic context such as that of Maude modules other problems can arise. We show in this paper how to debug missing answers in rewriting specifications, that is, expected results that the specification is not able to compute. This kind of problems appears in Maude when using its breadth-first search, that finds all the reachable terms from an initial one given a pattern, a condition, and a bound in the number of steps. To debug this kind of errors we have extended our calculus to deduce sets of reachable terms given an initial term, a bound in the number of rewrites, and a condition to be fulfilled. Unlike other proposals like [5], our debugging framework combines the treatment of wrong and missing answers and, moreover, is able to detect missing answers due to both missing rules and wrong statements. The state of the art can be found in [21], 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) and applying the missing answers technique to debug normal forms and least sorts[1] (the different trees are a novelty in the declarative debugging world), and only it and DDT [3] implement the Hirunkitti's divide and query navigation strategy, provide a graphical interface, and debug missing answers; as cons, we do not provide answers like "maybe yes," "maybe not," and "inadmissible," and do not perform tree compression. 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, 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 [21, 16] for the meaning of these concepts. With respect to other approaches, such as the Maude sufficient completeness checker [6, Chap. 21] or the sets of descendants [8], our tool provides a wider approach, since we handle conditional statements and our equations are not required to be left-linear.

The rest of the paper is structured as follows. Section 2 provides a summary of the main concepts of rewriting logic and Maude specifications. Section 3 describes our calculus and Section 4 shows the debugging trees obtained from it. Finally, Section 5 concludes and mentions some future work.

Detailed proofs of the results can be found in [19], while additional examples, the source code of the tool, and other papers on the subject, including the user guide [16], where a graphical user interface for the debugger is presented, are all available from the webpage `http://maude.sip.ucm.es/debugging`.

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

2. Rewriting Logic and Maude

Maude modules are executable rewriting logic specifications. Rewriting logic [10] is a logic of change very suitable for the specification of concurrent systems that is parameterized by an underlying equational logic, for which Maude uses membership equational logic (*MEL*) [2], which, in addition to equations, allows to state of membership axioms characterizing the elements of a sort. Rewriting logic extends *MEL* by adding rewrite rules.

For our purposes in this paper, we are interested in a subclass of rewriting logic models [10] 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 Section 4 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 [10].

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 [10, 7]. 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 $\mathcal{R}$.

Maude functional modules [6, Chap. 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications that allow the definition of sorts (by means of keyword `sort(s)`); *subsort* relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`). Maude system modules [6, Chap. 6], introduced with syntax `mod ... endm`, are executable rewrite theories. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

We present how to use this syntax by means of an example. 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, where `sd` denotes the symmetric difference:

```
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)] .
```

`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 4.1 how to debug it.

3. A Calculus for Missing Answers

We describe in this section a calculus to infer, given a term and some constraints, the *complete* set of reachable terms from this term that fulfill the requirements. The proof trees built with this calculus have nodes that justify why the terms are included in the corresponding sets (positive information) but also nodes that justify why there are no more terms (negative information). These latter nodes are then used in the debugging trees to localize as much as possible the reasons responsible for missing answers. This calculus integrates the calculus to deduce substitutions, normal forms, and least sorts that was presented in [18], and that we reproduce here to give the reader an overall view of debugging of missing answers in Maude specifications. Moreover, these calculi extend the calculus in [17], used to deduce judgments corresponding to oriented equations $t \to t'$, memberships $t : s$, and rewrites $t \Rightarrow t'$, and to debug wrong answers. All the results in this paper refer to the

complete calculus comprising these three calculi, and thus we consider this work as the final step in the development of foundations for a complete declarative debugger for Maude.

From now on, we assume a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ satisfying the Maude executability requirements, i.e., E is confluent, terminating, maybe modulo some equational axioms such as associativity and commutativity, and sort-decreasing, while R is coherent with respect to E; see [6] for details. Equations corresponding to the axioms form the set A and the equations in $E - A$ can be oriented from left to right.

We introduce the inference rules used to obtain the set of reachable terms given an initial one, a pattern [6], a condition, and a bound in the number of rewrites. First, the pattern P and the condition $\mathcal{C}$ (that can use variables bound by the pattern) are put together by creating the condition $\mathcal{C}' \equiv P := \circledast \wedge \mathcal{C}$, where $\circledast$ is a "hole" that will be filled by the concrete terms to check if they fulfill both the pattern and the condition. Throughout this paper we only consider a special kind of conditions and substitutions that operate over them, called *admissible*. They correspond to the ones used in Maude modules and are defined as follows:

Definition 3.1. A condition $C_1 \wedge \cdots \wedge C_n$ 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 $vars(C_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$, or
- a matching condition $u_i := u_i'$, u_i is a pattern and $vars(u_i') \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$, or
- a rewrite condition $u_i \Rightarrow u_i'$, u_i' is a pattern and $vars(u_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$.

Note that the lefthand side of matching conditions and the righthand side of rewrite conditions can contain extra variables that will be instantiated once the condition is solved.

Definition 3.2. 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 3.3. 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 3.4. 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 3.5. 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 1–4) 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.
- $\mathcal{T}' \models \mathit{disabled}(a, t)$ when the equation or membership a cannot be applied to t at the top.
- $\mathcal{T}' \models t \rightarrow_{red} t'$ when either $\mathcal{T}' \models t \rightarrow^{1}_{E'} t'$ or $\mathcal{T}' \models t_i \rightarrow^{!}_{E'} t'_i$, 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.
- $\mathcal{T}' \models t \rightarrow_{norm} t'$ when $\mathcal{T}' \models t \rightarrow^{!}_{E'} t'$, 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 $S = \{t' \mid t \rightarrow^{top}_{R'} t'\}$, 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 $S = \{t' \mid t \rightarrow^{top}_{\{q\}} t'\}$, 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 $S = \{t' \mid t \rightarrow^{1}_{R'} t'\}$, 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.
- $\mathcal{T}' \models t \rightsquigarrow^{\mathcal{C}}_{n} S$ when $S = \{t' \mid t \rightarrow^{\leq n}_{R'} t' \text{ and } \mathcal{T}' \models \mathit{fulfilled}(\mathcal{C}, t')\}$, 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.

We first introduce in Figure 1 the inference rules defining the relations $[C, \theta] \rightsquigarrow \Theta$, $\langle C, \Theta\rangle \rightsquigarrow \Theta'$, and $\mathit{adequateSorts}(\kappa) \rightsquigarrow \Theta$. 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):

- 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 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 AS_2 indicates that, if any of the terms in the kind-substitution has a sort bigger than the required one, then it is not a substitution and thus the empty set of substitutions is returned.
- Rule MbC_1 returns the current substitution if a membership condition holds.

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

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

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

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

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

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

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

Figure 1: Calculus for substitutions

- 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 with equations, abbreviated as $t_1 \downarrow t_2$.
- 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 $t \rightsquigarrow_n^C S$ explained below) and then using these terms to obtain the new substitutions.
- Finally, rule $\mathsf{SubsCond}$ computes the extensions of a set of admissible substitutions $\{\theta_1, \ldots, \theta_n\}$ by using the rules above with each of them.

We use these judgments to define the inference rules of Figure 2, 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.
- Rule $\mathsf{Rdc_1}$ 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 $\mathsf{Rdc_2}$ reduces a term by reducing a subterm to normal form (checking in the side condition that it is not already in normal form).

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0\rangle \rightsquigarrow \Theta_1 \quad \dots \quad \langle C_n, \Theta_{n-1}\rangle \rightsquigarrow \emptyset}{\mathit{disabled}(a, t)}\ \mathsf{Dsb}$$
$$\text{if } a \equiv l \to r \Leftarrow C_1 \wedge \ldots \wedge C_n \in E \text{ or}$$
$$a \equiv l : s \Leftarrow C_1 \wedge \ldots \wedge C_n \in E$$

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

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

$$\frac{\mathit{disabled}(e_1, f(t_1, \dots, t_n)) \quad \dots \quad \mathit{disabled}(e_l, f(t_1, \dots, t_n)) \quad t_1 \to_{norm} t_1 \quad \dots \quad t_n \to_{norm} t_n}{f(t_1, \dots, t_n) \to_{norm} f(t_1, \dots, t_n)}\ \mathsf{Norm}$$
$$\text{if } \{e_1, \dots, e_l\} = \{e \in E \mid e \ll_K^{top} f(t_1, \dots, t_n)\}$$

$$\frac{t \to_{red} t_1 \quad t_1 \to_{norm} t'}{t \to_{norm} t'}\ \mathsf{NTr} \qquad \frac{t \to_{norm} t' \quad t' : s \quad \mathit{disabled}(m_1, t') \quad \dots \quad \mathit{disabled}(m_l, t')}{t :_{ls} s}\ \mathsf{Ls}$$
$$\text{if } \{m_1, \dots, m_l\} = \{m \in E \mid m \ll_K^{top} t' \ \wedge\ \mathit{sort}(m) < s\}$$

Figure 2: Calculus for normal forms and least sorts

$$\frac{\mathit{fulfilled}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \{t\}}\ \mathsf{Rf_1} \qquad \frac{\mathit{fails}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \emptyset}\ \mathsf{Rf_2}$$

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

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

Figure 3: Calculus for solutions

- Rule Norm states that the term is in normal form by checking that no equations can be applied at the top considering the variables at the kind level (which is indicated by $\ll_K^{top}$) 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 $\to_{red}$ and the term thus obtained then is reduced to normal form by using again $\to_{norm}$.
- 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, applicable at the top with the variables considered at the kind level, 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 3 the rules related to $n = 0$ steps:

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

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

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

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

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

$$\frac{t \rightarrow_{norm} t_1 \quad t_1 \rightsquigarrow_n^{\mathcal{C}} \{t_2\} \cup S \quad t_2 \rightarrow_{norm} t'}{t \rightsquigarrow_n^{\mathcal{C}} \{t'\} \cup S}\ \mathsf{Red}$$

Figure 4: Calculus for missing answers

- Rule $\mathsf{Rf_1}$ indicates that when only zero steps are used and the current term fulfills the condition, the set of reachable terms consists only of this term.
- Rule $\mathsf{Rf_2}$ complements $\mathsf{Rf_1}$ by defining the empty set as result when the condition does not hold.
- Rule Fulfill checks whether a term satisfies a condition. The premises of this rule check that all the atomic conditions hold, taking into account that it starts with a matching condition with a hole that must be filled with the current term and thus proved with the premise $\theta(P) \downarrow t$. 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 1 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 Fulfill provides the positive information indicating that a condition is fulfilled, this one provides negative information, proving that the condition fails.

Now we introduce in Figure 4 the rules defining the relation $t \rightsquigarrow_n^{\mathcal{C}} S$ when the bound n is greater than 0, which can be understood as searches in *zero or more* steps:

- Rules $\mathsf{Tr_1}$ and $\mathsf{Tr_2}$ show the behavior of the calculus when at least one step can be used. First, we check whether the condition holds (rule $\mathsf{Tr_1}$) or not (rule $\mathsf{Tr_2}$) for the current term, in order to introduce it in the result set. Then, we obtain all the terms reachable in one step with the relation $\Rightarrow_1$, and finally we compute the reachable solutions from these terms

constrained by the same condition and the bound decreased in one step. The union of the sets obtained in this way and the initial term, if needed, corresponds to the final result set.

- Rule 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 of the program. We restrict these rules to those whose lefthand side, with the variables considered at the kind level, matches the term.
- Rule Rl uses the rules in Figure 1 to compute the set of terms obtained with the application of a single rule. First, the set of substitutions obtained from matching with the lefthand side of the rule is computed, and then it is used to find the set of substitutions that satisfy the condition. This final set is used to instantiate the righthand side of the rule to obtain the set of reachable terms. 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 Red reduces the reachable terms in order to obtain their normal forms. We use this rule to reproduce Maude behavior, first the normal form is computed and then the rules are applied.

Now we state that this calculus is correct in the sense that the derived judgments with respect to the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ coincide with the ones satisfied by the corresponding initial model $\mathcal{T}_{\Sigma/E,R}$, i.e., for any judgment φ, φ is derivable in the calculus if and only if $\mathcal{T}_{\Sigma/E,R} \models \varphi$. This is well known for the judgments corresponding to equations $t = t'$, memberships $t : s$, and rewrites $t \Rightarrow t'$ [11, 10].

Theorem 3.6. *The calculus of Figures 1, 2, 3, and 4 is correct.*

Once these rules are defined, we can build the tree corresponding to the search result shown in Section 2 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. 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 5, 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 a decreased bound.

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

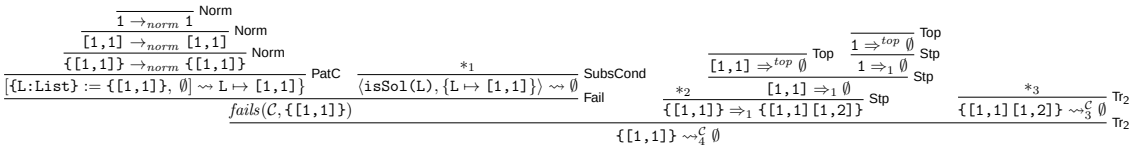

Figure 5: Tree for the maze example

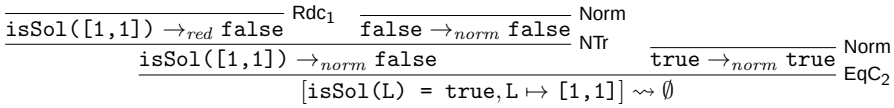

Figure 6: Tree $*_1$ for the search condition

The tree labeled with $*_2$ is sketched in Figure 7. 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 $\theta_1 \equiv$ `L` $\mapsto$ `[1,1]`; `P` $\mapsto$ `[1,2]`, $\theta_2 \equiv$ `L` $\mapsto$ `[1,1]`; `P` $\mapsto$ `[1,0]`, and $\theta_3 \equiv$ `L` $\mapsto$ `[1,1]`; `P` $\mapsto$ `[0,1]`. The substitution θ_1 also fulfills the next condition, `isOk(L P)`, which is proved with the rule EqC_1 in ♣ (where the big triangle is a computation in the calculus of [17] proving that the conditions of the equations hold), while the substitutions θ_2 and θ_3 fail; the trees $\bigtriangledown$ proving it are analogous to the one shown in Figure 6. This substitution θ_1 is thus the only one inferred in the root of the tree, where the node ♣ 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.

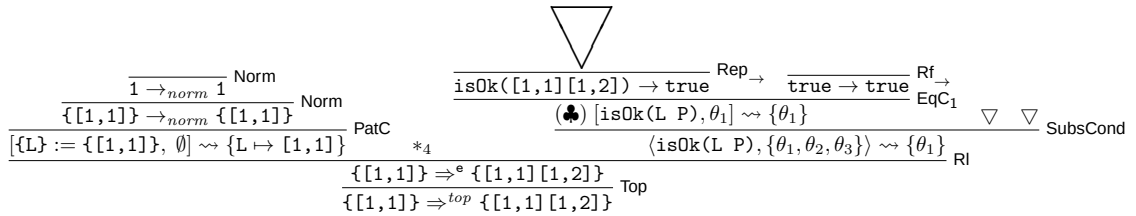

Figure 7: Tree $*_2$ for the applications at the top

The tree $*_4$, shown in Figure 8, 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 `L` $\mapsto$ `[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).

4. Debugging Trees

We describe in this section how to obtain appropriate debugging trees from the proof trees introduced in the previous section. Following the approach shown in [17], we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. This intended interpretation is a Σ-term model corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. Therefore the user *expects* that $\mathcal{I} \models \varphi$ for any judgment φ deduced with respect to the

$$\dfrac{\dfrac{\dfrac{*_5}{\texttt{next([1,1])} \rightsquigarrow_2^{\texttt{P}:=\circledast} \{\texttt{[1,2]}, \texttt{[1,0]}, \texttt{[0,1]}\}}\,\mathsf{Tr}_2}{[\texttt{next(L)} \Rightarrow \texttt{P}, \texttt{L} \mapsto \texttt{[1,1]}] \rightsquigarrow \{\texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,2]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,0]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[0,1]}\}}\,\mathsf{RIC}}{\langle \texttt{next(L)} \Rightarrow \texttt{P}, \{\texttt{L} \mapsto \texttt{[1,1]}\}\rangle \rightsquigarrow \{\texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,2]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,0]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[0,1]}\}}\,\mathsf{SubsCond}$$

Figure 8: Tree $*_4$ for the first condition of `expand`

rewrite theory $\mathcal{R}$. 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 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: 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. 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 $\mathcal{I}$. 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.[2]

In our debugging framework, 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. We will see in the next section that some extra information will be kept in the tree to provide this information. It is important not to confuse missing answers with missing

[2]Actually, 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.

statements; the current calculus detects missing answers due to both wrong and missing statements and wrong search conditions.

4.1. Abbreviated Proof Trees

We will not use proof trees T directly as debugging trees, but a suitable abbreviation which we denote by $APT(T)$ (from *Abbreviated Proof Tree*), or simply APT when T is clear from the context. The reason for preferring the APT 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 following proposition:

Proposition 4.1. *Let N be a buggy node in some proof tree in the calculus of Figures 1, 2, 3, and 4, w.r.t. an intended interpretation $\mathcal{I}$. Then:*

(1) *N is the consequence of a* $\mathsf{Rep}_{\rightarrow}$, $\mathsf{Rep}_{\Rightarrow}$, Mb, Rdc_1, Norm, Ls, Fulfill, Fail, Top, *or* RI *inference rule.*
(2) *The error associated to N is a wrong statement, a missing statement, or a wrong search condition.*

To indicate the error associated to the buggy node, 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.

The key idea in the APT, whose rules are shown in Figure 9, is to keep the nodes related to the inference rules indicated in Proposition 4.1, making use of the rest of rules to improve the questions asked to the user. 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:

- 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 $t \to t'$ is the one associated to $t \to t''$, which is not included in the APT. We have chosen to introduce $t \to t'$ instead of simply $t \to t''$ in the APT as a pragmatic way of simplifying the structure of the APTs, since t' is obtained from t'' and hence likely simpler.
- The rule ($\mathbf{APT}_4$) deletes questions about rewrites *at the top* (that can be difficult to answer due 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

$$
\begin{array}{lll}
(\mathbf{APT}_1) & APT\left(\dfrac{T_1 \dots T_n}{aj}\,{}_{R_1}\right) & = \dfrac{APT'\left(\dfrac{T_1 \dots T_n}{aj}\,{}_{R_1}\right)}{aj} \\[2ex]
(\mathbf{APT}_2) & APT'\left(\dfrac{\dfrac{T_1 \dots T_n}{t \to t''}\,\mathsf{Rep}_{\to} \quad T'}{t \to t'}\,\mathsf{Tr}_{\to}\right) & = \left\{\dfrac{APT'(T_1) \dots APT'(T_n)\ APT'(T')}{t \to t'}\,\mathsf{Rep}_{\to}\right\} \\[2ex]
(\mathbf{APT}_3) & APT'\left(\dfrac{\dfrac{T_1\ \dots\ T_n}{t \to t''}\,\mathsf{Rdc}_1\ \ T'}{t \to t'}\,\mathsf{NTr}\right) & = \left\{\dfrac{APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T')}{t \to t'}\,\mathsf{Rdc}_1\right\} \\[2ex]
(\mathbf{APT}_4) & APT'\left(\dfrac{\dfrac{T_1\ \dots\ T_n}{t \Rightarrow^{top} S'}\,\mathsf{Top}\ \ T'_1 \dots T'_m}{t \Rightarrow_1 S}\,\mathsf{Stp}\right) & = \left\{\dfrac{APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T'_1)\ \dots\ APT'(T'_m)}{t \Rightarrow_1 S}\,\mathsf{Top}\right\} \\[2ex]
(\mathbf{APT}_5) & APT'\left(\dfrac{T'\quad \dfrac{T_1 \dots T_n}{t \Rightarrow t'}\,\mathsf{Rep}_{\Rightarrow} \quad T''}{t_1 \Rightarrow t_2}\,\mathsf{EC}\right) & = \left\{\dfrac{APT'(T')\ APT'(T_1) \dots APT'(T_n)\ APT'(T'')}{t_1 \Rightarrow t_2}\,\mathsf{Rep}_{\Rightarrow}\right\} \\[2ex]
(\mathbf{APT}_6) & APT'\left(\dfrac{T\quad \dfrac{T_1 \dots T_n}{aj'}\,{}_{R_1}\ \ T'}{aj}\,\mathsf{Red}\right) & = \left\{\dfrac{APT'(T)\ APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T)}{aj}\,{}_{R_1}\right\} \\[2ex]
(\mathbf{APT}_7) & APT'\left(\dfrac{T_{t \to_{norm} t'}\ T_1 \dots T_n}{t :_{ls} s}\,\mathsf{Ls}\right) & = \left\{\dfrac{APT'(T_{t \to_{norm} t'})\ APT'(T_1)\ \dots\ APT'(T_n)}{t' :_{ls} s}\,\mathsf{Ls}\right\} \\[2ex]
(\mathbf{APT}_8^o) & APT'\left(\dfrac{T_1 \quad T_2}{t \Rightarrow t'}\,\mathsf{Tr}_{\Rightarrow}\right) & = APT'(T_1) \bigcup APT'(T_2) \\[2ex]
(\mathbf{APT}_8^m) & APT'\left(\dfrac{T_1 \quad T_2}{t \Rightarrow t'}\,\mathsf{Tr}_{\Rightarrow}\right) & = \left\{\dfrac{APT'(T_1)\ APT'(T_2)}{t \Rightarrow t'}\,\mathsf{Tr}_{\Rightarrow}\right\} \\[2ex]
(\mathbf{APT}_9^o) & APT'\left(\dfrac{T_1 \dots T_n}{aj}\,\mathsf{Tr}\right) & = APT'(T_1) \bigcup \dots \bigcup APT'(T_n) \\[2ex]
(\mathbf{APT}_9^m) & APT'\left(\dfrac{T_1 \dots T_n}{aj}\,\mathsf{Tr}_i\right) & = \left\{\dfrac{APT'(T_1)\ \dots\ APT'(T_n)}{aj}\,\mathsf{Tr}_i\right\} \\[2ex]
(\mathbf{APT}_{10}) & APT'\left(\dfrac{T_1 \dots T_n}{aj}\,{}_{R_2}\right) & = \left\{\dfrac{APT'(T_1) \dots APT'(T_n)}{aj}\,{}_{R_2}\right\} \\[2ex]
(\mathbf{APT}_{11}) & APT'\left(\dfrac{T_1 \dots T_n}{aj}\,{}_{R_1}\right) & = APT'(T_1) \bigcup \dots \bigcup APT'(T_n)
\end{array}
$$

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

$1 \leq i \leq 2 \qquad aj, aj'$ any judgment

Figure 9: Transforming rules for obtaining abbreviated proof trees

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.

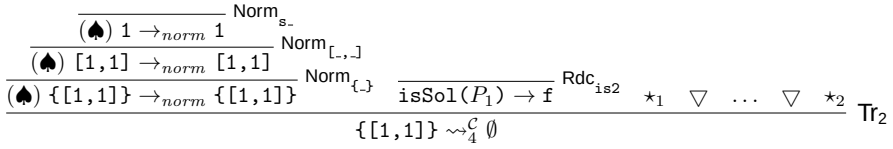

Figure 10: Abbreviated proof tree for the maze example

- The rule ($\mathbf{APT}_{11}$) removes from the tree all the nodes 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 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. We assume that the information introduced by the user during the session is correct.

Theorem 4.2. *Let T be a finite proof tree representing an inference in the calculus of Figures 1, 2, 3, and 4 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ s.t. 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 $\mathcal{R}$ (correctness).*

The trees in Figures 10–12 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 $*_5$. We have also extended the information in the labels with the operator or statement associated to the inference. More concretely, the tree in Figure 10 abbreviates the tree in Figure 5; the first two premises in the abbreviated tree abbreviate the first premise in the proof tree (which includes the tree in Figure 6), keeping only the nodes associated with relevant information according to Proposition 4.1: 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 11, abbreviates the second premise of the tree in Figure 5 as well as the trees in Figures 7 and 8; 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 12, shares these characteristics and only keeps nodes related to one-step searches and application of rules.

These APT rules are combined with trusting mechanisms that further reduce the proof tree (note that the correctness of these techniques relies on the decisions made by the user):

- Statements can be trusted in several ways: non labelled statements are always trusted (i.e., the nodes marked with ($\diamondsuit$) in Figure 11 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.

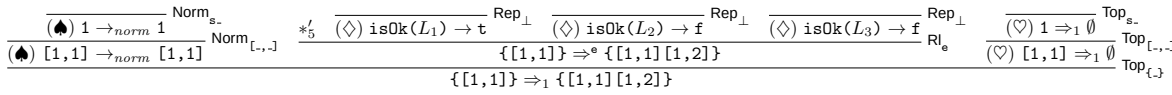

Figure 11: Abbreviated tree $\star_1$

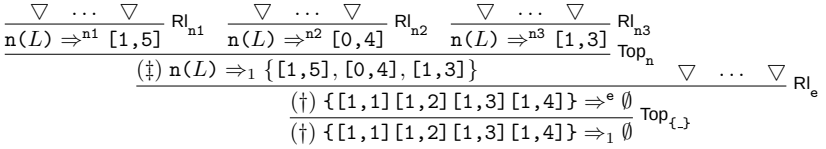

Figure 12: Abbreviated tree $\star_2$

- A correct module 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.
- We consider that constructed terms (terms built only with constructors, pointed out with the `ctor` attribute, and also known as data terms in other contexts) are in normal form and thus inferences of the form $t \to_{norm} t$ with t constructed are removed from the tree. This would remove from the tree the nodes marked with (♠) in Figures 10 and 11.
- Constructed terms of certain sorts or built with some operators can be considered *final*, which indicates that they cannot 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 11 would be removed from the tree.

Once this tree has been built, we can use it to debug the error shown in Section 2. Using the top-down navigation strategy our tool would show all the children of the root and ask the user to select an incorrect one. The last one (the root of $\star_2$) is incorrect and can be selected, and then the user has to answer about the validity of the child of this node. Since it is also incorrect the debugger selects it as current one (the path thus far has been marked with (†) in Figure 12) and the debugger shows its children. The first child (‡) is erroneous, but this time its children are all correct, so the tool points it out as buggy and it is associated to an erroneous fragment of code. More concretely, the rule used to infer this judgment was Top, and it is associated with the operator `next` (that was abbreviated as `n`), i.e., another rule for this operator is needed. Indeed, if we check the module we notice that the movement to the right has not been specified. We can fix it by adding:

```
rl [n4] : next(L [X,Y]) => [X + 1, Y] .
```

A detailed session of this example is available in the webpage `maude.sip.ucm.es/debugging`.

5. Conclusions and Future Work

We have presented in this paper a debugger of missing answers for Maude specifications. The trees for this kind of debugging are obtained from an abbreviation of a proper calculus whose adequacy for debugging has been proved. This work extends our previous work on wrong and missing answers [17, 18] 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 improves its traversal. The tree construction, its navigation, and the user interaction (excluding the GUI) have been all 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 [21] 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 test 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. Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05), Tallinn, Estonia*, pages 8–13. ACM Press, 2005.

[4] R. Caballero, 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.

[5] 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.

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

[7] 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.

[8] 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.

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

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

[11] 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.

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

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

[14] 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.

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

[16] 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`.

[17] 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.

[18] 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.

[19] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. Technical Report SIC 02/10, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2010. `http://maude.sip.ucm.es/debugging`.

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

[21] 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.

[22] 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.