Enhancing the Debugging of Maude Specifications*

Adrian Riesco, Alberto Verdejo, and Narciso Martí-Oliet

Facultad de Informática, Universidad Complutense de Madrid, Spain
`ariesco@fdi.ucm.es`, `{alberto,narciso}@sip.ucm.es`

Abstract. Declarative debugging is a semi-automatic technique that locates a program fragment responsible for the error by building a tree representing the computation and guiding the user through it to find the error. Two different kinds of errors are considered for debugging: *wrong answers*—a wrong result obtained from an initial value—and *missing answers*—a term that should be reachable but cannot be obtained from an initial value—, where the latter has only been considered in nondeterministic systems. However, we consider that missing answers can also appear in deterministic systems, when we obtain correct results that do not provide all the expected information, which corresponds, in the context of Maude modules, to terms whose normal form is not reached and to terms whose computed least sort is, although correct, bigger than the expected one. We present in this paper a calculus to deduce normal forms and least sorts, and a proper abbreviation of the trees obtained with it. These trees increase both the causes (missing equations and memberships) and the errors (erroneous normal forms and least sorts) detected in our debugging framework.

Keywords: declarative debugging, Maude, rewriting logic, membership equational logic, wrong answers, missing answers.

1 Introduction

Declarative debugging (also known as declarative diagnosis or algorithmic debugging) [17] is a debugging technique that abstracts the computation details to focus on results. It starts from an incorrect computation, the error symptom, and locates a program fragment responsible for the error. To find this error the debugger represents the computation as a *debugging tree* [10], 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. Traditional debugging techniques are devoted to fixing errors in specifications when an erroneous result, called a wrong answer, is found. Declarative debugging of this

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

kind of errors has been widely studied in the logic [9,19], functional [11,12], and multi-paradigm [3,7] programming languages. Another kind of errors, called *missing answers* [4,1], appears in nondeterministic systems when a term that should be reachable cannot be obtained from an initial one. This kind of errors has been less studied because it can only be applied to nondeterministic systems and because the associated calculus may be much more complicated than the one associated to wrong answers, making the debugging process unbearable.

Maude [5] is a high-level language and high-performance system supporting both equational and rewriting logic computation. Maude modules correspond to specifications in *rewriting logic* [8], 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. The Maude system supports several approaches for debugging: tracing, term coloring, and using an internal debugger [5, Chap. 22]. As part of an ongoing project to develop a declarative debugger for Maude specifications, we have already studied wrong answers in both functional and system modules [14] and missing answers in rewrites [15]. We now extend our framework by developing a calculus to deduce normal forms and least sorts seeing that the errors associated to these deductions correspond to missing answers in a deterministic framework. With this calculus we can detect errors due not only to wrong statements in a given specification but also to statements that the user *forgot* to specify,[1] indicating in this last case the operator at the top that the statement needs. These features improve our debugger in two ways: allowing to debug missing answers in the equational part of Maude modules and increasing the range of errors detected by the tool. For example, we can now debug missing answers when a rule cannot be applied because the term does not reach its normal form due to a missing equation or because the lefthand side does not match the term because it has a wrong least sort. We illustrate this improvement in Section 3 with a system module that, if debugged with the previous version of our tool, would print `Error: With the given information (labeling, correct module, and answers) it is impossible to debug.`, while in the current version the error is located.

The rest of the paper is organized as follows: after briefly introducing Maude modules with an example, Section 2 presents the calculus for missing answers and how the proof trees built with it are pruned in order to obtain appropriate debugging trees. Section 3 presents our tool by debugging some examples, while Section 4 concludes and outlines some future work.

The Maude source of the debugger, a user guide [13], additional examples, and other papers on this subject, including detailed proofs of the results [16], are all available from the webpage `http://maude.sip.ucm.es/debugging`.

[1] Note that the treatment of these missing statements is more powerful than the one currently applied in the Maude sufficient completeness checker [6], because it can be used with conditional and non left-linear statements.

1.1 An Example: Heaps

We show in this section how to specify in Maude binary heaps, that is, binary trees fulfilling that (1) all levels of the tree, except possibly the last one, are complete and, if the last level of the tree is not complete, the nodes of that level are filled from left to right; and (2) the value in each node is greater than the value in each of its children. The module `HEAP` defines binary trees (`BTree`) and `Heap`s and its nonempty variants (`NeBTree` and `NeHeap`), using a theory `TH` (not shown here) that defines the functions `min`, `max`, and a total order `_<_` over the elements of the sort `Elt`:

```
(fmod HEAP{X :: TH} is
  pr NAT .

  sorts BTree Heap NeBTree NeHeap .
  subsort NeHeap < NeBTree Heap < BTree .

  op mt : -> Heap [ctor] .
  op ___ : BTree X$Elt BTree -> NeBTree [ctor] .
```

We state by means of memberships when a binary tree is a heap:

```
  vars E E' : X$Elt .                    vars BT BT' : BTree .
  vars L L' R R' : Heap .                vars NL NR : NeHeap .

  cmb [h1] : NL E mt : NeHeap
   if max(NL) < E /\ depth(NL) == 1 .
  cmb [h2] : NL E NR : NeHeap
   if max(NL) < E /\ max(NR) < E /\
      (depth(NL) == depth(NR) and complete(NL)) or
      (depth(NL) == s(depth(NR)) and complete(NR)) .
```

where the auxiliary function `depth` computes the depth of a binary tree; `max` returns the value at the root of a nonempty heap (i.e., its maximum); and `complete` checks whether a binary tree is complete:

```
  op depth : BTree -> Nat .
  eq [dp1] : depth(mt) = 0 .
  eq [dp2] : depth(BT N BT') = max(depth(BT), depth(BT')) + 1 .

  op max : NeHeap -> X$Elt .
  ceq [max] : max(L E R) = E if L E R : NeHeap .

  op complete : BTree -> Bool .
  eq [cmp1] : complete(mt) = true .
  eq [cmp2] : complete(BT E BT') = complete(BT) and complete(BT') and
                                   depth(BT) == depth(BT') .
```

The function `insert` introduces a new element in a heap by sinking it to the appropriate position:

```
  op insert : X$Elt Heap ~> NeHeap .
  eq [ins1] : insert(E, mt) = mt E mt .
  ceq [ins2] : insert(E, L E' R) = L' max(E, E') R
   if L E' R : NeHeap /\
      not complete(L) or ((depth(L) > depth(R)) and complete(R)) /\
      L' := insert(min(E, E'), L) .
  ceq [ins3] : insert(E, L E' R) = L max(E, E') R'
   if L E' R : NeHeap /\
      not complete(R) or (depth(L) > depth(R)) and complete(L) /\
      R' := insert(min(E, E'), R) .
endfm)
```

We use a view HN (not shown here) to instantiate the values of the heap as natural numbers and we define a constant heap for testing:

```
(fmod NAT-HEAP is
  pr HEAP{HN} .
  op heap : -> NeHeap .
  eq heap = (mt 4 mt) 5 (mt 3 mt) .
endfm)
```

If we check in our specification the type of the constant heap:

```
Maude> (red heap .)
result NeBTree : (mt 4 mt) 5 (mt 3 mt)
```

we realize that although it has a correct sort (it is a NeBTree) its expected least sort, NeHeap, has not been obtained. We will show in Section 3 how to debug it.

2 Debugging Trees for Normal Forms and Least Sorts

We present in this section a calculus to compute the normal form and the least sort of a given term. The proof trees computed with this calculus contain the information proving why the term has been reduced to this normal form or this sort has been inferred (positive information) and also why the term has not been further reduced or a lesser sort has not been computed (negative information). The calculus is introduced as an extension of the calculus in [14] that allowed to deduce judgments corresponding to oriented equations $t \to t'$ and memberships $t : s$, and improves the calculus of missing answers of [15] by adding new causes to the errors debugged thus far. Once this extended calculus is presented, we show how to use it to define appropriate debugging trees.

2.1 A Calculus for Normal Forms and Least Sorts

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

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 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 2. *A* kind-substitution*, denoted by κ, is a mapping from variables to 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. *A* substitution*, denoted by θ, is a mapping from variables to 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 4. *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 (Figures 1 and 2) will be used to deduce the following judgments, that we introduce together with their meaning for a Σ-term model [8,16] $\mathcal{T}' = \mathcal{T}_{\Sigma/E',R'}$ 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.

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

if $\{\kappa_1, \ldots, \kappa_n\} = \{\kappa\theta \mid \kappa(\theta(t_1)) \equiv_A t'\}$

$$\frac{t_1 : sort(v_1) \quad \ldots \quad t_n : sort(v_n)}{adequateSorts(v_1 \mapsto t_1; \ldots; v_n \mapsto t_n) \rightsquigarrow \{v_1 \mapsto t_1; \ldots; v_n \mapsto t_n\}}\ \mathsf{AS}_1$$

$$\frac{t_i :_{ls} s_i}{adequateSorts(v_1 \mapsto t_1; \ldots; v_n \mapsto t_n) \rightsquigarrow \emptyset}\ \mathsf{AS}_2 \text{ if } s_i \not\leq sort(v_i)$$

$$\frac{\theta(t) : s}{[t : s, \theta] \rightsquigarrow \{\theta\}}\ \mathsf{MbC}_1 \qquad \frac{\theta(t) :_{ls} s'}{[t : s, \theta] \rightsquigarrow \emptyset}\ \mathsf{MbC}_2 \text{ if } s' \not\leq s$$

$$\frac{\theta(t_1) \downarrow \theta(t_2)}{[t_1 = t_2, \theta] \rightsquigarrow \{\theta\}}\ \mathsf{EqC}_1 \qquad \frac{\theta(t_1) \to_{norm} t'_1 \quad \theta(t_2) \to_{norm} t'_2}{[t_1 = t_2, \theta] \rightsquigarrow \emptyset}\ \mathsf{EqC}_2 \text{ if } t'_1 \not\equiv_A t'_2$$

$$\frac{\theta(t_1) \rightsquigarrow^{t_2 := \circledast}_{n+1} S}{[t_1 \Rightarrow t_2, \theta] \rightsquigarrow \{\theta'\theta \mid \theta'(\theta(t_2)) \in S\}}\ \mathsf{RIC} \qquad \frac{[C, \theta_1] \rightsquigarrow \Theta_1 \quad \cdots \quad [C, \theta_m] \rightsquigarrow \Theta_m}{\langle C, \{\theta_1, \ldots, \theta_m\}\rangle \rightsquigarrow \bigcup_{i=1}^{m} \Theta_i}\ \mathsf{SubsCond}$$

if $n = min(x \in \mathbb{N} : \forall i \geq 0\ (\theta(t_1) \rightsquigarrow^{t_2 := \circledast}_{x+i} S))$

Fig. 1. Calculus for substitutions

- Given a set of admissible substitutions Θ for an atomic condition C, $\mathcal{T}' \models \langle C, \Theta\rangle \rightsquigarrow \Theta'$ when $\Theta' = \{\theta' \mid \mathcal{T}', \theta' \models C$ and $\theta' \restriction_{dom(\theta)} = \theta$ for some $\theta \in \Theta\}$, that is, Θ' is the set of substitutions that fulfill the condition C and extend any of the admissible substitutions in Θ.
- Given an equation or membership a and a term t, $\mathcal{T}' \models disabled(a, t)$ when a cannot be applied to t at the top.
- Given two terms t and t', $\mathcal{T}' \models t \to_{red} t'$ when $\mathcal{T}' \models t \to^{1}_{E'} t'$ 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.
- Given two terms t and t', $\mathcal{T}' \models t \to_{norm} t'$ when $\mathcal{T}' \models t \to^{!}_{E'} t'$, that is, t' is in normal form with respect to the equations E'.
- Given a term t and a sort s, $\mathcal{T}' \models t :_{ls} s$ 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 $\mathcal{T}'$).

We introduce in Figure 1 the inference rules defining the relations $[C, \theta] \rightsquigarrow \Theta$, $\langle C, \Theta\rangle \rightsquigarrow \Theta'$, and $adequateSorts(\kappa) \rightsquigarrow \Theta$. 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 $adequateSorts(\kappa)$. 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 the least sort of any of the terms in the kind-substitution is 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.
- Rule 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 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 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 $t \leadsto_n^{\mathcal{C}} S$ explained in [16]) and then using these terms to obtain the new substitutions.
- Finally, rule SubsCond computes the extensions of a set of admissible substitutions $\{\theta_1, \ldots, \theta_n\}$ by using the rules above with each of them.

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 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).

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

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

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

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

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

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

Fig. 2. Calculus for normal forms and least sorts

- 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.

Theorem 1. *The calculus of Figures 1 and 2 is correct w.r.t. $\mathcal{R} = (\Sigma, E, R)$ in the sense that for any judgment φ, φ is derivable in the calculus if and only if $\mathcal{T}_{\Sigma/E,R} \models \varphi$, with $\mathcal{T}_{\Sigma/E,R}$ being the corresponding initial model.*

Once these rules have been presented, we can compute the proof tree associated to the erroneous computation shown in Section 1.1 for the heaps example. Remember that the least sort of the term `heap`, that should be `NeHeap`, was instead `NeBTree`. Figures 3 and 4 show the associated proof tree, where h stands for the term `(mt 4 mt) 5 (mt 3 mt)`, l for the lefthand side of the membership

$$\dfrac{\dfrac{\dfrac{}{\texttt{heap} \to_{red} h}\,\mathsf{Rdc_1} \quad \dfrac{}{h \to_{norm} h}\,\mathsf{Norm}}{\texttt{heap} \to_{norm} h}\,\mathsf{NTr} \qquad \dfrac{}{h : \texttt{NeBTree}}\,\mathsf{Mb} \quad T_1}{\texttt{heap} :_{ls} \texttt{NeBTree}}\,\mathsf{Ls}$$

Fig. 3. Proof tree for the heap example

$$\dfrac{\dfrac{\dfrac{}{h \to_{norm} h}\,\mathsf{Norm} \quad \dfrac{\dfrac{\nabla}{\texttt{mt 4 mt} :_{ls} \texttt{NeBTree}}\,\mathsf{Ls}}{adequateSorts(l,\theta)}\,\mathsf{AS_2}}{[l := h] \rightsquigarrow \emptyset}\,\mathsf{PatC} \quad \dfrac{}{\langle C_1, \emptyset\rangle \rightsquigarrow \emptyset}\,\mathsf{SubsCond} \quad \ldots \quad \dfrac{}{\langle C_n, \emptyset\rangle \rightsquigarrow \emptyset}\,\mathsf{SubsCond}}{disabled(\texttt{h2}, h)}\,\mathsf{Dsb}$$

Fig. 4. Proof tree T_1, proving the matching with `h2`

`h2`, namely `NL E NR` with `NL` and `NR` variables of sort `NeHeap` and `E` a natural number, C_1 and C_n are respectively the first condition and last condition of `h2`, θ is $\texttt{NL} \mapsto \texttt{mt 4 mt}; \texttt{E} \mapsto 5; \texttt{NR} \mapsto \texttt{mt 3 mt}$, and ∇ represents a tree similar to the one depicted in Figure 3.

The tree shown in Figure 3 illustrates that to compute the least sort of `heap` first it obtains its normal form and then it checks that no memberships can be applied to this term (and thus the sort is inferred by using the operator declarations). To check that no memberships are applied it only checks whether `h2` is used, because the other membership does not match the term with the variables at the kind level. The tree T_1, depicted in Figure 4, is in charge of this proof, that is, it provides the negative information proving that the membership cannot be applied. First, it checks that the lefthand side of the membership does not match the term because `mt 4 mt` has as least sort `NeBTree` and hence it does not match the variable `NL`, that has sort `NeHeap`. Since the empty set of substitutions is computed for this matching, the rest of conditions of the membership cannot be fulfilled, which is proved by the nodes associated with the rule SubsCond.

Following the approach shown in [14], 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 say that a judgment is *valid* when it holds in $\mathcal{I}$, and *invalid* otherwise. The basis of declarative debugging consists in searching *buggy nodes* (invalid nodes with all its children valid) [10] in a debugging tree standing for a problematic computation. In our debugging framework, we are able to locate wrong equations, wrong memberships, missing equations, and missing memberships,[2] which are defined as follows:

[2] It is important not to confuse wrong and missing answers with wrong and missing statements. The former are the initial symptoms that indicate the specifications fails, while the latter are the errors that generated this misbehavior.

- Given a statement $A \Leftarrow C_1 \wedge \cdots \wedge C_n$ (where A is either an equation $l = r$ or a membership $l : s$) 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 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.

Regarding missing statements, 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.

Proposition 1. *Let N be a buggy node in some proof tree in the calculus of Figures 1 and 2 w.r.t. an intended interpretation $\mathcal{I}$. Then the error associated to N is a wrong equation, a missing equation, or a missing membership.*

Although these are the errors detected by the calculus presented in this paper, since it is integrated with both the calculus of wrong answers [14] and the calculus for missing answers [15], the debugger as a whole can also detect wrong memberships and wrong and missing rules.

2.2 Abbreviated Proof Trees

We describe in this section how the proof trees shown in the previous section can be abbreviated in order to ease the questions posed to the user while keeping the completeness and correctness of the technique. To achieve this aim we extend the notion of $APT(T)$ introduced in [14]; $APT(T)$ (from *Abbreviated Proof Tree*) is obtained by a transformation based on deleting nodes whose correctness only depends on the correctness of their children. For example, nodes related to judgments about sets of substitutions, that can be complicated due to matching modulo, are removed.

The rules to compute the abbreviated proof tree, which are assumed to be applied in order (i.e., a rule cannot be applied if there is another one with a lower index that can be used), are described in Figure 5:

- Rule ($\mathbf{APT}_1$) keeps the root of the tree and applies the general function APT', that returns a set of trees, to the tree.
- Rule ($\mathbf{APT}_2$) improves the questions presented to the user when the inference rule NTr is used. This abbreviation associates the equation applied in the left branch (in the inference rule Rdc_1) to the judgment rooting the tree. In this way we ask about reductions to normal form instead of reductions in one step.

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

$$(\mathbf{APT}_2)\ APT'\left(\frac{\frac{T_1\ \dots\ T_n}{t\to t''}\mathsf{Rdc_1}\quad T'}{t\to t'}\mathsf{NTr}\right) = \left\{\frac{APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T')}{t\to t'}\mathsf{Rdc_1}\right\}$$

$$(\mathbf{APT}_3)\ APT'\left(\frac{T_{t\to_{norm}t'}\ T_1\dots T_n}{t :_{ls} s}\mathsf{Ls}\right) = \left\{\frac{APT'(T_{t\to_{norm}t'})\ APT'(T_1)\ \dots\ APT'(T_n)}{t' :_{ls} s}\mathsf{Ls}\right\}$$

$$(\mathbf{APT}_4)\ APT'\left(\frac{T_1\dots T_n}{aj}R_2\right) = \left\{\frac{APT'(T_1)\ \dots\ APT'(T_n)}{aj}R_2\right\}$$

$$(\mathbf{APT}_5)\ APT'\left(\frac{T_1\dots T_n}{aj}R_1\right) = APT'(T_1)\bigcup\dots\bigcup APT'(T_n)$$

R_1 any inference rule R_2 $\mathsf{Rdc_1}$, or Norm aj any judgment

Fig. 5. APT rules

- Rule $(\mathbf{APT}_3)$ improves the questions about least sorts by asking about the normal form of the term and thus the user is not in charge of computing it.
- Rule $(\mathbf{APT}_4)$ keeps the conclusion of the inference rules that contain debugging information.
- Rule $(\mathbf{APT}_5)$ discards the conclusion of the rules which do not contain debugging information.

Theorem 2. *Let T be a finite proof tree representing an inference in the calculus of Figures 1 and 2 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 equation, missing equation, or missing membership axiom in $\mathcal{R}$ (correctness).*

The abbreviated proof tree obtained by applying these rules to the proof tree depicted in Figures 3 and 4 is shown in Figure 6. This proof tree has been obtained by combining different features available in our tool:

- Judgments of the form $t \to_{norm} t$, that indicate that t is in normal form, are dropped from the proof tree if they are built only with constructors. In our example, the nodes corresponding to $h \to_{norm} h$ have been removed.
- Only labeled statements generate nodes in the abbreviated proof tree. For example, the equation to reduce the constant `heap` is not labeled and thus the node `heap` $\to_{red} h$ (or its corresponding abbreviation) does not appear in the abbreviated tree. Moreover, the debugger provides some other trusting mechanisms: statements and imported modules can be trusted before starting the debugging process; statements can also be trusted on the fly; and a correct module, introduced before starting the debugging process, can be used as oracle before asking the user.

$$\dfrac{\dfrac{\dfrac{}{(\ddagger)\ \texttt{mt} :_{ls} \texttt{Heap}}\ \mathsf{Ls}}{(\dagger)\ h :_{ls} \texttt{NeBTree}}\ \mathsf{Ls}}{\texttt{heap} :_{ls} \texttt{NeBTree}}\ \mathsf{Ls}$$

Fig. 6. Abbreviated proof tree for the heap example

- The signature is always considered correct, and hence judgments inferred by using it do not appear in the abbreviated tree. For example, the membership inference h : `BTree` only uses operator declarations and thus it does not appear in the final tree.
- The rest of nodes have been pruned by the *APT* rules. For example, they prevent all the judgments using substitutions from being asked.

Furthermore, the user can also follow some strategies to reduce the size of the debugging tree:

- 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 easier to answer and fixing them can also solve the missing answers problem.
- The Maude profiler [5, Chap. 22] indicates the most frequently used statements for a given computation. Trusting these statements will greatly reduce the size of the tree, although it requires the user to make sure that these statements are indeed correct.

Once the tree has been abbreviated we only have a subset of the original nodes and hence only the correctness of the judgments in these nodes concerns the debugging process. We present here the questions derived only from the calculus presented here, while the rest of the questions asked by the debugger can be found in [13]:

- When a term cannot be further reduced and it is not built only by constructors the debugger asks "Is t in normal form?," which is correct if the user expected t to be a normal form.
- 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 reduced to t'.
- When a sort s is inferred for a term t, the debugger prompts questions of the form "Is this membership correct? $t : s$." This judgment is correct if t has sort s.
- 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.

3 A Debugging Session

We describe in this section how to debug the specification shown in Section 1.1. To debug the error discovered in this specification (the least sort of the term `heap` is `NeBTree`) we use the command:

```
Maude> (missing heap : NeBTree .)
```

This command builds the tree depicted in Figure 6 and asks the following question, associated with the node marked with (†) in the figure:[3]

```
Is NeBTree the least sort of mt 4 mt ?
Maude> (no .)
```

Since we expected the term to have sort `NeHeap` the judgment is erroneous and the next question, that is associated to the node (‡) in Figure 6, is:

```
Is Heap the least sort of mt ?
Maude> (yes .)
```

With this answer the node (‡) disappears from the tree and the node (†) becomes buggy, because it is associated to an incorrect judgment and it has no children. The debugger presents the following message:

```
The buggy node is:
The least sort of mt 4 mt is NeBTree
Either the operator ___ needs more membership axioms or the conditions of
the current axioms are not written in the intended way.
```

Actually, if we check the specification we notice that the membership corresponding to the case when both heaps are empty was not stated. We should add to the specification the membership axiom:

```
  mb [h3] : mt E mt : NeHeap .
```

We can use now these heaps to implement another application. We present here a very simple specification of an auction. The module `AUCTION` defines the sort `People` as a multiset of `Person` (a pair of names and bids) and an `Auction` as some people and a heap, defined in `NS-HEAP`, containing elements of the form `[N,S]`, where `N` is a natural number standing for the bid and `S` a `String` with the name of the bidder. The winner of the auction will be the person on the top of the heap:

```
(mod AUCTION is
  pr NS-HEAP .

  sorts Person People Auction .
```

[3] Although the debugger provides two different navigation strategies, in this simple tree both of them choose the same node.

```
  subsort Person < People .

  op <_`,_> : String Nat -> Person [ctor] .
  op nobody : -> People [ctor] .
  op __ : People People -> People [ctor comm assoc id: nobody] .
  op _`[_`] : People Heap -> Auction [ctor] .
```

The rule bid inserts a bid into the heap:

```
  var N : Nat .                    var H : Heap .
  var P : People .                 var S : String .

  rl [bid] : (P < S, N >) [H] => P [insert([N,S], H)] .
endm)
```

If we search now for the possible winners of an auction, where `initial` stands for `< "aida", 5 > < "nacho", 4 > < "charlie", 3 > [mt]`:

```
Maude> (search in AUCTION : initial =>!
                            nobody [L:Heap [N:Nat, S:String] R:Heap] .)
No solution.
```

no solutions are found. Since one solution is expected, we debug the specification with the command:

```
Maude> (missing initial =>! nobody [ L:Heap [N:Nat, S:String] R:Heap ] .)
```

This command builds the corresponding debugging tree and traverses it with the default divide and query strategy, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise. The first question is:

```
Are the following terms all the reachable terms from
(< "aida", 5 > < "charlie", 3 > < "nacho", 4 >)[mt] in one step?
1 (< "aida", 5 > < "nacho", 4 >)[mt [3, "charlie"] mt]
2 (< "aida", 5 > < "charlie", 3 >)[mt [4, "nacho"] mt]
3 (< "charlie", 3 > < "nacho", 4 >)[mt [5, "aida"] mt]
Maude> (yes .)
```

The rule has inserted each person into the heap and thus the transition is correct. After some other questions related to rewrites in the style of [15], the debugger asks:

```
Is insert([4,"nacho"],mt[3,"charlie"]mt) in normal form?
Maude> (no .)
```

This term is not in normal form because we expected `insert` to be reduced. The next questions are also related to normal forms:[4]

[4] Note that, in these cases, the `String` values are not built with constructors and thus this question is not automatically removed by the debugger. If we defined our own constants for the names with the `ctor` attribute, these questions would not appear.

```
Is mt [3, "charlie"] mt in normal form?
Maude> (yes .)

Is [4,"nacho"] in normal form?
Maude> (yes .)
```

In these cases the judgment is correct because no equations should be applied to them. The next questions refer to reductions:

```
Is this reduction (associated with the equation dp1) correct?
depth(mt) -> 0
Maude> (trust .)

Is this reduction (associated with the equation cmp1) correct?
complete(mt) -> true
Maude> (trust .)
```

Since these reductions were associated to simple equations we have used the command `trust` to prevent the debugger from asking questions related to these equations again. The next question deals with memberships:

```
Is this membership (associated with the membership h3) correct?
mt [3, "charlie"] mt : NeHeap
Maude> (yes .)
```

The membership is correct because it only contains the value at the root. With this information the debugger finds the following bug:

```
The buggy node is:
insert([4,"nacho"], mt [3, "charlie"] mt) is in normal form.
Either the operator insert needs more equations or the conditions of
the current equations are not written in the intended way.
```

If we carefully inspect the equations for `insert` we notice that we have not treated the case where the tree is complete and a new level has to be started. We can add the appropriate equation or fix the equation `ins2`, that distinguishes a case that cannot occur in heaps. If we choose the latter, it should be fixed as follows:

```
  ceq [ins2] : insert(E, L E' R) = L' max(E, E') R
   if L E' R : NeHeap /\
      not complete(L) or ((depth(L) == depth(R)) and complete(R)) /\
      L' := insert(min(E, E'), L) .
```

4 Future Work

In this paper we have presented a calculus to debug erroneous normal forms and least sorts by abbreviating the proof trees obtained with it. This calculus, besides allowing to debug these new errors, improves the former versions of our debugger by allowing the debugging of new causes of missing answers in rewrites: missing equations and memberships. These debugging features have also been integrated with the graphical user interface [13].

Although the current version of the tool allows the user to introduce a correct but maybe incomplete module in order to shorten the debugging session [14], we also want to add a new command to introduce *complete* modules, which would greatly reduce the number of questions asked to the user. We also intend to add new navigation strategies like the ones shown in [18] that take into account the number of different potential errors in the subtrees, instead of their size.

Finally, we plan to use the new narrowing features of Maude to implement a test generator for Maude specifications. This generator would allow to check Maude specifications and then to invoke the debugger when one of the test cases fails.

Acknowledgements. We cordially thank Martin Wirsing for encouraging us to investigate the causes for missing answers in a deterministic context, and the referees for their useful comments.

References

1. Alpuente, M., Comini, M., Escobar, S., Falaschi, M., Lucas, S.: Abstract diagnosis of functional programs. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 1–16. Springer, Heidelberg (2003)
2. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236, 35–132 (2000)
3. Caballero, R.: 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 2005), Tallinn, Estonia, pp. 8–13. ACM Press, New York (2005)
4. Caballero, R., Rodríguez-Artalejo, M., del Vado Vírseda, R.: Declarative diagnosis of missing answers in constraint functional-logic programming. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 305–321. Springer, Heidelberg (2008)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Hendrix, J., Meseguer, J., Ohsaki, H.: A sufficient completeness checker for linear order-sorted specifications modulo axioms. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 151–155. Springer, Heidelberg (2006)
7. MacLarty, I.: Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne (2005)
8. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
9. Naish, L.: Declarative diagnosis of missing answers. New Generation Computing 10(3), 255–286 (1992)
10. Naish, L.: A declarative debugging scheme. Journal of Functional and Logic Programming (3) (1997)

11. Nilsson, H.: 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)
12. Pope, B.: A Declarative Debugger for Haskell. PhD thesis, The University of Melbourne, Australia (2006)
13. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: 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`
14. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: Declarative debugging of rewriting logic specifications. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 308–325. Springer, Heidelberg (2009)
15. Riesco, A., Verdejo, A., Martí-Oliet, N.: Declarative debugging of missing answers in rewriting logic. Technical Report SIC-6-09, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2009), `http://maude.sip.ucm.es/debugging`
16. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: 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`
17. Shapiro, E.Y.: Algorithmic Program Debugging. In: ACM Distinguished Dissertation. MIT Press, Cambridge (1983)
18. Silva, J.: A comparative study of algorithmic debugging strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
19. Tessier, A., Ferrand, G.: Declarative diagnosis in the CLP scheme. In: Deransart, P., Hermenegildo, M.V., Maluszynski, J. (eds.) DiSCiPl 1999. LNCS, vol. 1870, pp. 151–174. Springer, Heidelberg (2000)