Simplifying Questions in Maude Declarative Debugger by Transforming Proof Trees*

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

Facultad de Informática, Universidad Complutense de Madrid, Spain

Abstract. Declarative debugging is a debugging technique that abstracts the execution details that in general may be difficult to follow in declarative languages to focus on results. It relies on a data structure representing the wrong computation, the *debugging tree*, which is traversed by asking questions to the user about the correctness of the computation steps related to each node. Thus, the complexity of the questions is an important factor regarding the applicability of the technique. In this paper we present a transformation for debugging trees for Maude specifications that ensures that any subterm occurring in a question has been previously replaced by the most reduced form that it has taken during the computation, thus ensuring that questions become as simple as possible.

Keywords: declarative debugging, Maude, proof tree transformation.

1 Introduction

Declarative debugging [15], also called *algorithmic debugging*, is a debugging technique that abstracts the execution details, that may be difficult to follow in general in declarative languages, to focus on results. This approach, that has been used in logic [17], functional [11], and multi-paradigm [8] languages, is a two-phase process [10]: first, a data structure representing the computation, the so-called *debugging tree*, is built; in the second phase this tree is traversed following a *navigation strategy* and asking to an external oracle about the correctness of the computation associated to the current node until a *buggy node*, an incorrect node with all its children correct, is found. The structure of the debugging tree must ensure that buggy nodes are associated to incorrect fragments of code, that is, finding a buggy node is equivalent to finding a bug in the program. Note that, since the oracle used to navigate the tree is usually the user, the number and complexity of the questions are the main issues when discussing the applicability of the technique.

Maude [4] is a high-level language and high-performance system supporting both equational and rewriting logic computation. Maude modules correspond to specifications in rewriting logic [9], a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. This logic is an extension of equational logic, that in the Maude case corresponds to membership equational logic (MEL) [1], which, in addition to equations, allows the statement of membership axioms

* Research supported by MEC Spanish projects *DESAFIOS10* (TIN2009-14599-C03-01) and *STAMP* (TIN2008-06622-C03-01), Comunidad de Madrid programs *PROMETIDOS* (S2009/TIC1465) and *PROMESAS* (S-0505/TIC/0407), and UCM-BSCH-GR58/08-910502.

G. Vidal (Ed.): LOPSTR 2011, LNCS 7225, pp. 73–89, 2012.

characterizing the elements of a sort. Rewriting logic extends MEL by adding rewrite rules, that represent transitions in a concurrent system.

In previous papers we have faced the problem of declarative debugging of Maude specifications both for *wrong answers* (incorrect results obtained from a valid input), and for missing answers (incomplete results obtained from a valid input); a complete description of the system can be found in [14]. Conceptually debugging trees in Maude are obtained in two steps. First a *proof tree* for the erroneous result (either a wrong or missing answer) in a suitable semantic calculus is considered. Then this tree is pruned by removing those nodes that correspond to logic inference steps that does not depend on the program and are consequently valid. The result is an *abbreviated proof tree* (APT) which has the property of requiring less questions to find the error in the program. Moreover, the terms in the APT nodes appear in their most reduced forms (for instance function calls have been replaced by their results). Although unnecessary from the theoretical point of view, this property of containing terms in their most reduced form has been required since the earlier works in declarative debugging (see Section 2) since otherwise the debugging process becomes unfeasible in practice due to the complexity of the questions performed to the user.

However, the situation changes when debugging Maude specifications with the `strat` attribute [4], that directs the evaluation order and can prevent some arguments from being reduced, that is, this attribute introduces a particular notion of laziness, making some subterms to be evaluated later than they would be in a "standard" Maude computation. For this reason we will use in this paper a slightly different notion of normal form that takes into account `strat`: a term is in normal form if neither the term nor its subterms has been further reduced in the current computation. When dealing with specifications with this attribute the APT no longer contains the terms in their most reduced forms, and thus the questions performed by the tool become too involved.

The purpose of this work is to define a program transformation that converts an arbitrary proof tree T built for a specification with the `strat` attribute into a proof tree T' whose APT contains all the subterms in their most reduced form. Since T' is also a proof tree for the same computation the soundness and completeness of the technique obtained in previous papers remain valid. Note that this improvement, described for the equational subset of Maude (where `strat` is applied) improves the questions asked in the debugging of both wrong and missing answers, including system modules, because reductions are used by all the involved calculi. Note that, although we present here a transformation for arbitrary proof trees, our tool builds the debugging trees in such a way that some of this transformations are not needed (more specifically, we do not need the "canonical" transformation we will see later). We prefer to build the proof tree and then transform it to make the approach conservative: the user can decide whether he wants to use the transformation or not.

The rest of the paper is organized as follows: the following section introduces some related work and shows the contributions of our approach with respect to related proposals. Section 3 introduces Maude functional modules, the debugging trees used to debug this kind of modules, and the trees we want to obtain to improve the debugging process. Section 4 presents the transformations applied to obtain these trees and the theoretical results that ensure that the transformation is safe. Finally, we present the conclusions and discuss some related ongoing work.

The source code of the debugger, examples, and much more information is available at `http://maude.sip.ucm.es/debugging/`. Detailed proofs of the results shown in this paper and extended information about the transformations can be found in [2].

2 Related Work

Since the introduction of declarative debugging [15] the main concerns with respect to this technique were the complexity of the questions performed to the user, and also that the process can become very tedious, and thus error-prone. The second point is related to the number of questions and has been addressed in different ways [14,16]: nodes whose correction only depends on the correction of their children are removed; statements and modules can be trusted, and thus the corresponding nodes can be removed from the debugging tree; a database can be used to prevent debuggers from asking the same question twice; trees can be compressed [5], which consists in removing from the debugging tree the children of nodes that are related to the same error as the father, in such a way that the father will provide all the debugging information; a different approach consists in *adding* nodes to the debugging tree to balance it and thus traverse it more efficiently [7]; finally, other techniques reduce the number of questions by allowing complex answers, that direct the debugging process in a more specific direction, e.g. [8] provides an answer to point out a specific subterm as erroneous.

This paper faces the first concern, the complexity of the questions, considering the case of Maude specifications including the `strat` attribute. This attribute can be used to alter the execution order, and thus the same subterm can be found in different forms in the tree. The unpredictability of the execution order was already considered in the first declarative debuggers proposed for lazy functional programming. In [12] the authors proposed two ways of constructing the debugging trees. The first one was based on source code transformations and the introduction of an impure primitive employed for ensuring that all the subterms take the most reduced form (or a special symbol denoting unevaluated calls). This idea was implemented in Buddha [13], a declarative debugger for Haskell, and in the declarative debugger of the functional-logic language Toy [3]. The second proposal was to change the underlying language implementation, which offers better performance. This technique was exploited in [11], where an implementation based on graph reduction was proposed for the language Haskell.

In this paper we address a similar problem from a different point of view. We are interested in proving formally the adequacy of the proposal and thus we propose a transformation at the level of the proof trees, independent of the implementation. The transformation takes an arbitrary proof tree and generates a new proof tree. We prove that the transformed tree is a valid proof tree with respect to rewriting logic calculus underlying Maude and that the subterms in questions are in their most reduced form.

3 Declarative Debugging in Maude

We present here Maude and the debugging trees used to debug Maude specifications.

3.1 Maude

For our purposes in this paper we are interested in the equational subset of Maude, which corresponds to specifications in MEL [1]. Maude functional modules [4], introduced with syntax `fmod ... endfm`, are executable MEL specifications and their semantics is given by the corresponding initial membership algebra in the class of algebras satisfying the specification. In a functional module we can declare sorts (by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`). The executability requirements for equations and memberships are confluence, termination, and sort-decreasingness [4].

We illustrate the features described before with an example. The `LAZY-LISTS` module below specifies lists with a lazy behavior. At the beginning of the module we define the sort `NatList` for lists of natural numbers, which has `Nat` as a subsort, indicating that a natural number constitutes the singleton list:

```
(fmod LAZY-LISTS is
  pr NAT .
  sort NatList .
  subsort Nat < NatList .
```

Lists are built with the operator `nil` for empty lists and with the operator _ _ for bigger lists, which is associative and has `nil` as identity. It also has the attribute `strat(0)` indicating that only reductions at the top (the position `0`) are allowed:

```
  op nil : -> NatList [ctor] .
  op _ _ : NatList NatList -> NatList [ctor assoc id: nil strat(0)] .
```

Next, we define a function `from` that generates a potentially infinite list starting from the number given as argument. Note that the attribute `strat(0)` in _ _, used in the right-hand side of the equation, does not permit reductions in the subterms of `N from(s(N))`, thus preventing an infinite computation because no equations can be applied to `from(s(N))`:

```
  op from : Nat -> NatList .
  eq [f] : from(N) = N from(s(N)) .
```

where `f` is a label identifying the equation. The module also contains a function `take` that extracts the number of elements indicated by the first argument from the list given as the second argument. Since the `strat(0)` attribute in _ _ prevents the list from evolving, we take the first element of the list and apply the function to the rest of the list in a matching condition, thus separating the terms built with _ _ into two different terms and allowing the lazy lists to develop all the needed elements:

```
  op take : Nat NatList -> NatList .
  ceq [t1] : take(s(N), N' NL) = N' NL' if NL' := take(N, NL) .
  eq [t2] : take(N, NL) = 0 [owise] .
```

where `owise` stands for otherwise, indicating that the equation is used when no other equation can be applied. Finally, the function `head` extracts the first element of a list, where `~>` indicates that the function is partial:

(**Reflexivity**)

$$\frac{}{t \to t}\ \mathsf{Rf}$$

(**Congruence**)

$$\frac{t_1 \to t'_1 \quad \ldots \quad t_n \to t'_n}{f(t_1,\ldots,t_n) \to f(t'_1,\ldots,t'_n)}\ \mathsf{Cong}$$

(**Transitivity**)

$$\frac{t_1 \to t' \quad t' \to t_2}{t_1 \to t_2}\ \mathsf{Tr}$$

(**Replacement**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(t) \to \theta(t')}\ \mathsf{Rep}$$
$$\text{if } t \to t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j$$

Fig. 1. Semantic calculus for reductions

```
  op head : NatList ~> Nat .
  eq [h] : head(N NL) = N .
 endfm)
```

We can now introduce the module in Maude and reduce the following term:

```
Maude> (red take(2 * head(from(1)), from(1)) .)
result NatList : 1 2 0
```

However, instead of returning the first two elements of the list, it appends `0` to the result. The unexpected result of this computation indicates that there is some error in the program. The following sections show how to build a debugging tree for this reduction, how to improve it, and how to use this improved tree to debug the specification.

3.2 Debugging Trees

Debugging trees for Maude specifications [14] are conceptually built in two steps:[1] first, a proof tree is built with the proof calculus in Figure 1, which is a modification of the calculus in [1], where we use the notation $t \downarrow t'$ to indicate that t and t' are reduced to the same term (which is used for both equality conditions of the form $t = t'$ and matching conditions $t := t'$, where t may contain new variables) and we assume that the equations are terminating and confluent and hence they can be oriented from left to right, and that replacement inferences keep the label of the applied statement in order to point it out as wrong when a buggy node is found. In the second step a pruning function, called *APT*, is applied to the proof tree in order to remove those nodes whose correctness only depends on the correctness of their children (and thus they are useless for the debugging process) and to improve the questions asked to the user. This transformation can be found in [14].

Figure 1 describes the part of MEL we will use throughout this paper, the extension to full MEL is straightforward and can be found in [2]. The figure shows that the proof trees can infer *judgments* of the form $t \to t'$, indicating that t is reduced to t' by using equations. The inference rules in this calculus are reflexivity, that proves that a term can be reduced to itself; congruence, that allows to reduce the subterms; transitivity, used to compose reductions; and replacement, that applies a equation to a term if a substitution

[1] The implementation applies these two steps at once.

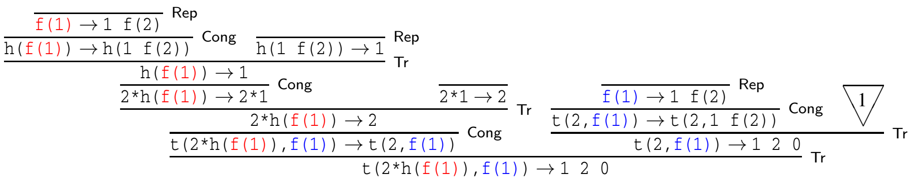

Fig. 2. Proof tree for the reduction on Section 3.1

θ making the term match the lefthand side of the equation and fulfilling the conditions is found. It is easy to see that the only inference rule whose correctness depends on the specification is replacement; intuitively, nodes inferred with this rule will be the only ones kept by *APT*. Thus, *APT* removes some nodes from the tree and can attach the debugging information to some others in order to ease the questions asked to the user, but cannot modify the judgments in the nodes, since it would require to modify the whole structure of the tree, as we will see later.

We show in Figures 2 and 3 the proof tree associated to the reduction presented in the previous section, obtained following Maude execution strategies,[2] where `t` stands for `take`, `h` for `head`, and `f` for `from`. The left child of the root of the tree in Figure 2 obtains the number of elements that must be extracted from the list, while the right child unfolds the list one step further and takes the element thus obtained, repeating this operation until all the required elements have been taken. Note that Maude cannot reduce the second argument of `t`, `f(1)`, to its normal form (with respect to the tree) `1 2 3 f(4)` with three consecutive replacement steps because the attribute `strat(0)` prevents it.

From the point of view of declarative debugging, this tree is not very satisfactory, because it contains nodes like `t(2,1 f(2))` $\to$ `1 2 0`, the root of the tree in Figure 3, where the subterm `f(2)` is not fully reduced, which forces the user to obtain its expected result and then (mentally) substitute it in the node in order to answer the question about the correction of the node. We show the APT corresponding to this tree in Figure 4; note that a transformation like *APT* cannot improve this kind of questions because there is no node with the information we want to use, and thus the node (†) described above is kept and will be used in the debugging process. Intuitively, we would like to gather all the replacements related to the same term so we can always ask about terms with the subterms in normal form, like `t(2,1 2` $\perp$`)`, where $\perp$ is a special symbol indicating that a term could be further reduced but its value is not necessary. The next section explains how to transform proof trees in order to obtain questions with this form.

When examining a proof tree we are interested in distinguishing whether two syntactically identical terms are copies of the same term or not. The reason is that it is more natural for the user to have each copy in its more reduced form, without considering the reductions of other copies of the same term (as happens with the term `f(1)` in the

[2] Actually, the value `3` in Figure 3 has been computed to mimic Maude's behavior. Once it has obtained `take(0, f(3))` it tries to reduce its subterms, obtaining `3` although it will be never used. All the transformations in this paper also work if this term is not computed.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\spadesuit)\ \texttt{f(2)} \to \texttt{2 f(3)}}\ \mathsf{Rep}}{\texttt{t(1,f(2))} \to \texttt{t(1,2 f(3))}}\ \mathsf{Cong} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{f(3)} \to \texttt{3 f(4)}}\ \mathsf{Rep}}{\texttt{t(0,f(3))} \to \texttt{t(0,3 f(4))}}\ \mathsf{Cong} \qquad \dfrac{}{\texttt{t(0,3 f(4))} \to \texttt{0}}\ \mathsf{Rep}}{\texttt{t(0,f(3))} \to \texttt{0}}\ \mathsf{Tr}}{\texttt{t(1,2 f(3))} \to \texttt{2 0}}\ \mathsf{Rep}}{(\diamondsuit)\ \texttt{t(1,f(2))} \to \texttt{2 0}}\ \mathsf{Tr}}{\texttt{t(2,1 f(2))} \to \texttt{1 2 0}}\ \mathsf{Rep}$$

Fig. 3. Proof tree for the subtree $\overset{1}{\bigtriangledown}$ on Figure 2

$$\dfrac{\dfrac{}{\texttt{f(1)} \to \texttt{1 f(2)}}\ \mathsf{Rep} \quad \dfrac{}{\texttt{h(1 f(2))} \to \texttt{1}}\ \mathsf{Rep} \quad \dfrac{}{\texttt{f(1)} \to \texttt{1 f(2)}}\ \mathsf{Rep} \quad \dfrac{\dfrac{}{\texttt{f(2)} \to \texttt{2 f(3)}}\ \mathsf{Rep} \quad \dfrac{\dfrac{}{\texttt{f(3)} \to \texttt{3 f(4)}}\ \mathsf{Rep} \quad \dfrac{}{\texttt{t(0,3 f(4))} \to \texttt{0}}\ \mathsf{Rep}}{\texttt{t(1,2 f(3))} \to \texttt{2 0}}\ \mathsf{Rep}}{(\dagger)\ \texttt{t(2,1 f(2))} \to \texttt{1 2 0}}\ \mathsf{Rep}}{\texttt{t(2*h(f(1)),f(1))} \to \texttt{1 2 0}}\ \mathsf{Tr}$$

Fig. 4. Abbreviated proof tree for the proof tree in Figure 2

example above; one of these terms is reduced to `1 f(2)` while the second one is reduced to `1 2 3 f(4)`). We achieve this goal by "painting" related terms in a proof tree with the same color. Hence the same term can be repeated in several places in a proof tree, but only those copies coming from the same original term will have the same color. We refer to colored terms as *c-terms* and to trees with colored terms in their nodes as *c-trees*. When talking about colored trees, $t_1 = t_2$ means that t_1 and t_2 are equally colored. Therefore talking about two occurrences of a c-term t implicitly means that there are two copies of the same term equally colored. Intuitively, all the terms in the lefthand side of the root have different colors; the replacement inference rule introduces new colors, while the reflexivity, transitivity, and congruence rules propagate them. More details can be found in [2].

It is worth observing that computation trees represent a particular computation that has already taken place in Maude. This means that we can be sure that the debugged program satisfies the constraints required by Maude functional modules: equations must be terminating, confluent, and sort-decreasing. Other requirements such as left-linearity or constructor-based rules are not required in these modules. The details of how to carry out a computation correspond to Maude and not to the debugger, which only represents computations. During the tree construction process, the debugger already knows the appropriate substitutions used in the associated computation as well as the places where they must be applied for each computation step; our algorithms just modify the tree taking into account this information. A subtle detail that allows us to move the computations forward is that the lefthand side of Maude equations must be a pattern, and thus "frozen" terms (i.e., terms that are not built with constructors and cannot be reduced because of the `strat` attribute) such as `N . from(N')` cannot be used. Notice that in Maude, term sharing is introduced incrementally by equational simplification, because it analyzes righthand sides of equations to identify its shared subterms [6]. More details can be found in [2].

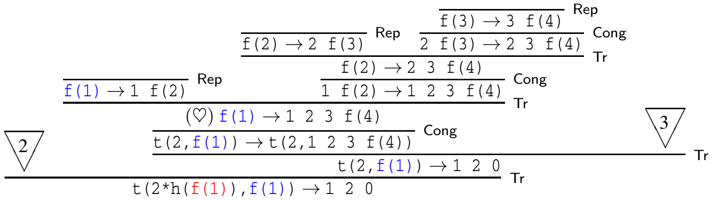

Fig. 5. Proof tree for the reduction on Section 3.1

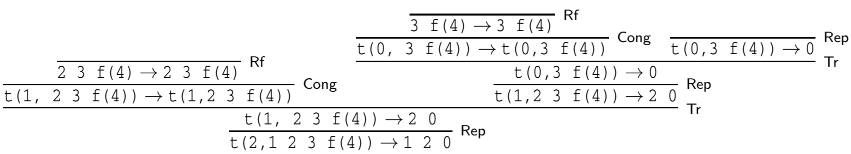

Fig. 6. Proof tree for the subtree $\overset{3}{\nabla}$ on Figure 5

3.3 The Lists Example Revisited

As explained in the previous section, the debugging tree in Figure 4 presents the drawback of containing nodes of the form `t(2,1 f(2))` → `1 2 0`, whose correction is difficult to state because the subterms must be mentally reduced by the user in order to compute the final result. We give in this section the intuitions motivating the transformations in the next section, transforming the trees and in Figures 2 and 3, that give rise to the proof trees in Figures 5 and 6. The tree $\overset{2}{\nabla}$ in Figure 5 has the same left premise as the one in Figure 2, which shows the importance of coloring the terms in the proof trees, because the algorithm distinguishes between the two `f(1)` thanks to their different colors. The part of the tree depicted in Figure 5 shows how the reduction of the subterms is "anticipated" by the algorithm in the previous section and thus the node (♡) performs the reduction of the second argument of `t`, `f(1)`, to its normal form (with respect to the tree), and all the replacement steps that were needed to reach it are contained in this subtree. The tree $\overset{3}{\nabla}$ in Figure 6 shows the other part of the transformations: we get rid of the relocated replacement inferences by using reflexivity steps.

The APT of our transformed proof tree is depicted in Figure 7. It has removed all the useless information like reflexivity and congruence inferences, and has associated the replacement inferences, that contain debugging information, to the transitivity inferences below them, returning a debugging tree where the lefthand side of all the reductions have their subterms in normal form, as expected because the transformation works driven by the *APT* transformation.

Since this transformation has been implemented in our declarative debugger, we can start a debugging session to find the error in the specification described in Section 3.1. The debugging process starts with the command:

```
Maude> (debug take(2 * head(from(1)), from(1)) -> 1 2 0 .)
```

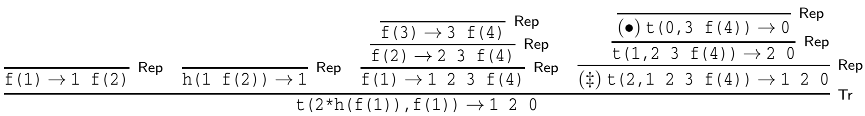

Fig. 7. Abbreviated proof tree for the transformed tree

This command builds the tree shown in Figure 7,[3] which is traversed following the navigation strategy divide and query [16], that selects in each case a node rooting a subtree with approximately half the size of the whole tree, and the first question, associated with the node (‡) in Figure 7, is:

```
Is this reduction (associated with the equation t1) correct?
take(2,1 2 3 ?:NatList) -> 1 2 0
Maude> (no .)
```

where we must interpret `?:NatList` as a term that has not reached a normal form (in the sense it is not built with operators with the `ctor` attribute) but whose value is irrelevant to compute the final result. The answer is `(no .)` because we expected to take only `1 2`. Note that this node is the transformed version of the node (†) in Figure 4, that would perform the question:

```
Is this reduction (associated with the equation t1) correct?
take(2,1 from(2)) -> 1 2 0
```

which is more difficult to answer because we have to think first about the reduction of `from(2)` and then use the result to reason about the complete reduction. With the answer given above the subtree rooted by (‡) in Figure 7 is considered as the current one and the next questions are:

```
Is this reduction (associated with the equation t1) correct?
take(1,2 3 ?:NatList) -> 2 0
Maude> (no .)
Is this reduction (associated with the equation t2) correct?
take(0,3 ?:NatList) -> 0
Maude> (no .)
```

We answer `(no .)` in both cases for the same reason as in the previous case. With these answers we have discovered that the node (•) is wrong. Hence, since it has no children, it is the buggy node and is associated with a wrong statement:

```
The buggy node is: take(0,3 ?:NatList) -> 0
with the associated equation: t2
```

In fact, the equation `t2` returns `0` but it should return `nil`.

4 Transforming Debugging Trees

In this section we present the transformation that ensures that the abbreviated proof tree contains every term reduced as much as possible. This transformation is a two-step process. First, a sequence of three tree transformations prepares the proof tree for the

[3] Actually, it builds the tree in Figure 4 and then transforms it into the tree in Figure 7.

second phase. We call the trees obtained by this transformation *canonical trees*. The second phase takes a canonical tree as input and applies the algorithm that replaces terms by its most reduced forms. The result is a proof tree for the same computation whose APT verifies that every term is reduced as much as possible.

4.1 Reductions

We need to formally define the concepts of reduction and number of steps, which will be necessary to ensure that a tree is in its most reduced form.

Definition 1. *Let T be an APT, and t,t' two c-terms. We say that $t \to t'$ is a reduction w.r.t. T if there is a node $N \in T$ of the form $t_1 \to t_2$ verifying:*

- *$pos(t,t_1) \neq \emptyset$, where $pos(t,t_1)$ is the set of positions of t containing t_1.*
- *$t' = t[t_1 \mapsto t_2]$, where $t[t_1 \mapsto t_2]$ represents the replacement of every occurrence of the c-term t_1 by t_2 in t.*

In this case we also say that t is reducible (w.r.t. T). A reduction chain *for t will be a sequence of reductions $t_0 = t \to t_1 \to t_2 \to \cdots \to t_n$ s.t. each $t_i \to t_{i+1}$ is a reduction and that t_n cannot be further reduced w.r.t. T.*

Definition 2. *Let T be an APT. Then:*

- *The number of reductions of a term t w.r.t. T, denoted as $reduc(t,T)$ is the sum of the length of all the possible different reduction chains of t w.r.t. T.*
- *The number of reductions of a node of the form $N = f(t_1,\ldots,t_n) \to t$ w.r.t. T, denoted as $reduc(N,T)$ is defined as $(\sum_{i=1}^{n} reduc(t_i,T)) + reduc(t,T)$.*

In this definition the length of a reduction chain $t_0 \to \cdots \to t_n$ is defined as n. Remember that the aim of this paper is to present a technique to put together these reductions chains, transforming appropriately the proof tree, and using colors to distinguish terms; when dealing with commutative or associativity, we will assume flatten terms with the subterms ordered in an alphabetical fashion. Moreover, our technique assumes that there is only one normal form for each c-term in the tree.

Definition 3. *We say that an occurrence of a c-term t occurring in an APT T is in* normal form w.r.t. *T if there is no reduction for any c-subterm of t in T.*

Definition 4. *Let T be an APT. We say that T is* confluent *if every c-term t occurring in T has a unique normal form with respect to T.*

Note that this notion of confluence is different from the usual notion of confluence required in Maude functional modules: it requires all the copies of a (colored) term, that can be influenced by the `strat` attribute, to be reduced to the same term. In the rest of the paper we assume that, unless stated otherwise, all the APTs are colored and confluent. With these definitions we are ready to define the concept of *norm*:

(**InsCong₁**)

$$InsCong\left(\dfrac{T_1 \ldots T_m}{f(t_1,\ldots,t_n) \to t}\mathsf{Rep}\right) =$$

$$\dfrac{\dfrac{\dfrac{}{t_1 \to t_1}\mathsf{Rf} \;\ldots\; \dfrac{}{t_n \to t_n}\mathsf{Rf}}{f(t_1,\ldots,t_n) \to f(t_1,\ldots,t_n)}\mathsf{Cong} \quad \dfrac{InsCong(T_1)\ldots InsCong(T_m)}{f(t_1,\ldots,t_n) \to t}\mathsf{Rep}}{f(t_1,\ldots,t_n) \to t}\mathsf{Tr}$$

(**InsCong₂**)

$$InsCong\left(\dfrac{T_1 \ldots T_m}{aj}\mathsf{R}\right) = \dfrac{InsCong(T_1)\ldots InsCong(T_m)}{aj}\mathsf{R}$$

aj any judgment, R any inference rule, $n > 0$

Fig. 8. Insert Congruences (*InsCong*)

Definition 5. *Let T be a proof tree, and $T' = APT(T)$. The* norm *of T, represented by $\parallel T \parallel$, is the sum of the lengths of all the reduction chains that can be applied to terms in T'. More formally, given the reduc function in Definition 2:*

$$\parallel T \parallel = \sum_{\substack{N \in T' \\ N \neq root(T')}} reduc(N, T')$$

Thus, the norm is the number of reductions that can be performed in the corresponding APT. Our goal is to obtain proof trees with associated norm 0, ensuring that the questions performed to the user contain terms as reduced as possible. This is the purpose of the proof tree transformations in the following section, which start with some initial proof tree and produces an equivalent proof tree with norm 0.

4.2 Canonical Trees

Canonical trees are obtained from proof trees as explained in the following definition.

Definition 6. *We define the canonical form of a proof tree T, which will be denoted from now on as $Can(T)$, as*

$$Can(T) = RemInf(NTr(InsCong(T)))$$

where InsCong (insert congruences), NTr (normalize transitivities), and RemInf (remove superfluous inferences) are defined in Figures 8, 9, and 10, respectively.

It is assumed that the rules of each transformation are applied top-down. The first transformation, *InsCong*, prepares the proof tree for allowing reductions on the arguments t_i of judgments of the form $f(t_1,\ldots,t_n) \to t$ by introducing congruence inferences before these judgments take place. Initially no reduction is applied, and each argument is simply reduced to itself using a reflexivity inference. Replacing these reflexivities by non-trivial reductions for the arguments is the role of the algorithm introduced in the

$(\mathbf{NTr_1})$

$$NTr\left(\cfrac{\cfrac{T_{t_1\to t_2} \quad T_{t_2\to t_3}}{t_1 \to t_3}\mathsf{Tr} \quad T_{t_3\to t_4}}{t_1 \to t_4}\mathsf{Tr}\right) = NTr\left(\cfrac{NTr(T_{t_1\to t_2}) \quad NTr\left(\cfrac{T_{t_2\to t_3} \quad T_{t_3\to t_4}}{t_2 \to t_4}\mathsf{Tr}\right)}{t_1 \to t_4}\mathsf{Tr}\right)$$

$(\mathbf{NTr_2})$

$$NTr\left(\frac{T_1 \ldots T_n}{aj}\mathsf{R}\right) = \frac{NTr(T_1)\ldots NTr(T_n)}{aj}\mathsf{R} \quad aj \text{ any judgment, } \mathsf{R} \text{ any inference rule}$$

Fig. 9. Normalize Transitivities (*NTr*)

next section. The next transformation, *NTr*, takes care of righthand sides. The idea is that transitivity inferences occurring as left premises of other transitivity are associated to intermediate, not fully-reduced computations. Thus, *NTr* ensures that righthand sides can be completely reduced by the algorithm in the next section. Finally, *RemInf* eliminates some superfluous steps involving reflexivities, and combines consecutive congruences in a "bigger step" single congruence, which avoids the production of unnecessary intermediate results in the proof tree. This last process is done with the help of an auxiliary transformation *merge* (Figure 11), that combines two trees by using a transitivity.

A proof tree in canonical form is also a proof tree proving the same judgment.

Proposition 1. *Let T be a proof tree. Then $Can(T)$ is a proof tree with the same root.*

Moreover, applying these transformations cannot produce an increase of the norm:

Proposition 2. *Let T be a proof tree and $T' = Can(T)$. Then $\| T \| \geq \| T' \|$.*

4.3 Reducing the Norm of Canonical Trees

We describe in this section the main transformation applied to the proof trees. This transformation relies on the following proposition, that declares that in any proof tree in canonical form there exist (1) a node with a reduction $t_1 \to t_1'$ such that t_1' is in normal form, that will be used to further reduce the terms, (2) a node that contains a reduction $t_2 \to t_2'$, with $t_1 \in t_2'$ (t_1 is a subterm of t_2'), which means that t_2' can be further reduced by the previous reduction, and (3) a node such that it is not affected by the transformations in the previous nodes. We will use node (1) to improve the reductions in node (2); this transformation will only affect the nodes in the subtree that has (3) as root:

Proposition 3. *Let T be a confluent c-proof tree in canonical form such that $\| T \| > 0$. Then T contains:*

1. *A node related to a judgment $t_1 \to t_1'$ such that:*
   - *It is either the consequence of a transitivity inference with a replacement as left premise, or the consequence of a replacement inference which is not the left premise of a transitivity.*
   - *t_1' is in normal form w.r.t. T.*

$$(\textbf{RemInf}_1)\quad \mathit{RemInf}\left(\dfrac{\dfrac{T_{t_1\to t_1'}\dots T_{t_n\to t_n'}}{f(t_1,\dots,t_n)\to f(t_1',\dots,t_n')}\,\mathsf{Cong}\qquad \dfrac{T_{t_1'\to t_1''}\dots T_{t_n'\to t_n''}}{f(t_1',\dots,t_n')\to f(t_1'',\dots,t_n'')}\,\mathsf{Cong}}{f(t_1,\dots,t_n)\to f(t_1'',\dots,t_n'')}\,\mathsf{Tr}\right) =$$

$$\dfrac{\mathit{RemInf}\left(\mathit{merge}\left(T_{t_1\to t_1'},\ T_{t_1'\to t_1''}\right)\right)\dots \mathit{RemInf}\left(\mathit{merge}\left(T_{t_n\to t_n'},\ T_{t_n'\to t_n''}\right)\right)}{f(t_1,\dots,t_n)\to f(t_1'',\dots,t_n'')}\,\mathsf{Cong}$$

$$(\textbf{RemInf}_2)\quad \mathit{RemInf}\left(\dfrac{\dfrac{T_{t_1\to t_1'}\dots T_{t_n\to t_n'}}{f(t_1,\dots,t_n)\to f(t_1',\dots,t_n')}\,\mathsf{Cong}\qquad \dfrac{\dfrac{T_{t_1'\to t_1''}\dots T_{t_n'\to t_n''}}{f(t_1',\dots,t_n')\to f(t_1'',\dots,t_n'')}\,\mathsf{Cong}\quad T_{f(t_1'',\dots,t_n'')\to e}}{f(t_1',\dots,t_n')\to e}\,\mathsf{Tr}}{f(t_1,\dots,t_n)\to e}\,\mathsf{Tr}\right) =$$

$$\mathit{RemInf}\left(\dfrac{\dfrac{\mathit{merge}\left(T_{t_1\to t_1'},\ T_{t_1'\to t_1''}\right)\dots \mathit{merge}\left(T_{t_n\to t_n'},\ T_{t_n'\to t_n''}\right)}{f(t_1,\dots,t_n)\to f(t_1'',\dots,t_n'')}\,\mathsf{Cong}\qquad \mathit{RemInf}(T_{f(t_1'',\dots,t_n'')\to e})}{f(t_1,\dots,t_n)\to e}\,\mathsf{Tr}\right)$$

$$(\textbf{RemInf}_3)\quad \mathit{RemInf}\left(\dfrac{\dfrac{F}{t\to t_1}\,\mathsf{Cong}\qquad \dfrac{\dfrac{}{t_1\to t_1}\,\mathsf{Rf}\quad T_{t_1\to t'}}{t_1\to t'}\,\mathsf{Tr}}{t\to t'}\,\mathsf{Tr}\right) = \mathit{RemInf}\left(\dfrac{\dfrac{F}{t\to t_1}\,\mathsf{Cong}\qquad T_{t_1\to t'}}{t\to t'}\,\mathsf{Tr}\right)$$

$$(\textbf{RemInf}_4)\quad \mathit{RemInf}\left(\dfrac{T^{\mathsf{Rf}}\quad T'}{aj}\,\mathsf{Tr}\right) = \mathit{RemInf}\left(\dfrac{T'\quad T^{\mathsf{Rf}}}{aj}\,\mathsf{Tr}\right) = \mathit{RemInf}(T'),\ aj \text{ any judgment}$$

$$(\textbf{RemInf}_5)\quad \mathit{RemInf}\left(\dfrac{T_1\dots T_n}{aj}\,\mathsf{R}\right) = \dfrac{\mathit{RemInf}(T_1)\dots \mathit{RemInf}(T_n)}{aj}\,\mathsf{R},\ aj \text{ any judgment, } \mathsf{R} \text{ any inference rule}$$

Fig. 10. Remove superfluous inferences (*RemInf*)

$$(\textbf{Merge}_1)$$
$$merge\left(\frac{T_{t\to t_1}\ \ T_{t_1\to t'}}{t\to t'}\mathsf{Tr}, T_{t'\to t''}\right) = \frac{T_{t\to t_1}\ \ merge\,(T_{t_1\to t'}, T_{t'\to t''})}{t\to t''}\mathsf{Tr}$$
$$(\textbf{Merge}_2)$$
$$merge\,(T_{t\to t'}, T_{t'\to t''}) = \frac{T_{t\to t'}\ \ T_{t'\to t''}}{t\to t''}\mathsf{Tr}$$

Fig. 11. Merge Trees

2. *A node related to a judgment* $t_2 \to t_2'$ *with* $t_1 \in t_2'$.
3. *A node related to a judgment* $t_3 \to t_3'$ *consequence of a transitivity step, with* $t_1 \notin t_3'$.

Algorithm 1 presents the transformation in charge of reducing the norm of the proof trees until it reaches 0. It first selects a node N_{ible} (from *reducible node*), that contains a term that has been further reduced during the computation,[4] a node N_{er} (from *reducer node*) that contains the reduction needed by the terms in N_{ible}, and a node p_0 limiting the range of the transformation. Note that we can distinguish two parts in the subtree rooted by the node in p_0, the left premise, where N_{ible} is located, and the right premise, where N_{er} is located. Then, we create some copies of these nodes in order to use them after the transformations. For example, the first step of the loop for the proof tree in Figures 2 and 3 would set N_{ible} to `f(2)` $\to$ `2 f(3)` and N_{er} to `f(3)` $\to$ `3 f(4)`; they are located in the subtree rooted by p_0, the node ($\diamondsuit$).

Step 6 replaces the proof of the reduction $t_1 \to t_1'$ by reflexivity steps $t_1' \to t_1'$. Since the algorithm is trying to use this reduction before its current position, a natural consequence will be to transform all the appearances of t_1 in the path between the old and the new position by t_1', what means that in this particular place we would obtain the reduction $t_1' \to t_1'$ inferred, by Proposition 3, by either a transitivity or a replacement rule, and with the appropriate proof trees as children. Since this would be clearly incorrect, the whole tree is replaced by a reflexivity. In our example, the replacement `f(3)` $\to$ `3 f(4)` (N_{er}) would be transformed into the reflexivity step `3 f(4)` $\to$ `3 f(4)`.

Step 7 replaces all the occurrences of t_1 by t_1' in the right premise of p_0, as explained in the previous step. In this way, the right premise of p_0 is a new subtree where t_1 has been replaced by t_1' and all the proofs related to $t_1 \to t_1'$ have been replaced by reflexivity steps $t_1' \to t_1'$. Note that intuitively these steps are correct because t_1' is required to be in normal form, the tree is confluent, and the norm of this tree is 0, that is, all the possible reductions of terms with the same color have been previously modified by the algorithm to create a $t_1 \to t_1'$ proof. In our example, the appearances of `f(3)` in the right premise of the node ($\diamondsuit$) are replaced by `3 f(4)`; this subtree is already a proof tree.

Step 8 replaces the occurrences of t_1 by t_1' in the left premise of p_0. We apply this transformation only in the righthand sides because they are in charge of transmitting the information, and in this way we prevent the algorithm from changing correct values (inherited perhaps from the root). This substitution can be used thanks to the position p_0, which ensures that only the righthand sides are affected. In our example, we substitute the term `f(3)` by `3 f(4)` in the left child of ($\diamondsuit$).

[4] We select the first one in post-order to ensure that this node is the one that generated the term.

Step 9 combines the reduction in N_{ible} with the reduction in N_{er} (actually, it merges their copies, since the previous transformations have modified them). If the term t_1 we are further reducing corresponds to the term t'_2 in the lefthand side of the judgment in N_{ible}, then it is enough to use a transitivity to "join" the two subtrees. In other case, the term we are reducing is a subterm of t'_2 and thus we must use a congruence inference rule to reduce it, using again a transitivity step to infer the new judgment. This last step would generate, in our example, a node combining the replacement (♠) and the one in N_{er} in a transitivity step, giving rise to the node `f(2)` $\to$ `2 3 f(4)`; in this way the left child of (♢), and consequently the tree, becomes a proof tree again.

Finally, these transformations make the trees to lose their canonical form, and hence the canonical form of the tree is computed again in step 10.

Algorithm 1. *Let T be a proof tree in canonical form.*

1. *Let $T_r = T$*
2. *Loop while $\parallel T_r \parallel > 0$*
3. *Let $N_{er} = t_1 \to t'_1$ be a node satisfying the conditions of item 1 in Proposition 3, $N_{ible} = t_2 \to t'_2$ the first node in T's post-order verifying the conditions of item 2 in Proposition 3, and p_0 the position of the subtree of T rooted by the first (furthest from the root) ancestor of N_{ible} satisfying item 3 in Proposition 3, such that the right premise of the node in p_0, T_{rp}, has $\parallel T_{rp} \parallel = 0$.*
4. *Let C_{er} be a copy of the tree rooted by N_{er}.*
5. *Let C_{ible} be a copy of the tree rooted by N_{ible} and p_{ible} the position of N_{ible}.*
6. *Let T_1 be the result of replacing in T all the subtrees rooted by N_{er} by a reflexivity inference step with conclusion $t'_1 \to t'_1$.*
7. *Let T_2 be the result of substituting all the occurrences of the c-term t_1 by t'_1 in the right premise of the subtree at position p_0 in T_1.*
8. *Let T_3 be the result of substituting all the occurrences of the c-term t_1 with t'_1 in the righthand sides of the left premise of the subtree at position p_0 in T_2.*
9. *Let T_4 be the result of replacing the subtree at position p_{ible} in T_3 by the following subtree:*
   (a) if $t'_2 = t_1$.

$$\frac{C_{ible} \quad C_{er}}{t_2 \to t'_1}\ \mathsf{Tr}$$

   (b) if $t'_2 \neq t_1$.

$$\frac{C_{ible} \quad \dfrac{C_{er}}{t'_2 \to t'_2[t_1 \mapsto t'_1]}\ \mathsf{Cong}}{t_2 \to t'_2[t_1 \mapsto t'_1]}\ \mathsf{Tr}$$

10. *Let T_r be the result of normalizing T_4.*
11. *End Loop*

The next theorem is the main result of this paper. It states that after applying the algorithm we obtain a proof tree for the same computation whose nodes are as reduced as possible. Thus, the declarative debugging tool that uses this tree as debugging tree will ask questions in its most simplified form.

Theorem 1. *Let T be a proof tree in canonical form. Then the result of applying Algorithm 1 to T is a proof tree T_r such that $root(T_r) = root(T)$ and $\| T_r \| = 0$.*

Observe that we have improved the "quality" of the information in the nodes without increasing the number of questions, since the transformations do not introduce new replacement inferences in the APT.

5 Concluding Remarks and Ongoing Work

One of the main criticisms to declarative debugging is the high complexity of the questions performed to the user. Thus, if the same computation can be represented by different debugging trees, we must choose the tree containing the simplest questions. In Maude, an improvement in this direction is to ensure that the judgments involving reductions are presented to the user with the terms reduced as much as possible. We have presented a transformation that allows us to produce debugging trees fulfilling this property starting with any valid proof tree for a wrong computation. The result is a debugging tree with questions as simple as possible without increasing the number of questions, which is specially useful when dealing with the `strat` attribute. Moreover, the theoretical results supporting the debugging technique presented in previous papers remain valid since we have proved that our transformation transforms proof trees into proof trees for the same computation.

Although for the sake of simplicity we have focused in this paper on the equational part of Maude, this transformation has been applied to all the judgments $t \to t'$ appearing in the debugging of both wrong (including system modules) and missing answers. However, our calculus for missing answers also considers judgments $t \to_{norm} t'$, indicating that t' is the normal form of t; when facing the `strat` attribute, the inferences for these judgments have the same problem shown here; we are currently working to define a transformation for this kind of judgment.

References

1. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236, 35–132 (2000)
2. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: Improving the debugging of membership equational logic specifications. Technical Report SIC-02-11, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (March 2011), `http://maude.sip.ucm.es/debugging/`
3. Caballero, R., Rodríguez-Artalejo, M.: DDT: A Declarative Debugging Tool for Functional-Logic Languages. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 70–84. Springer, Heidelberg (2004)
4. 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)
5. Davie, T., Chitil, O.: Hat-Delta: One right does make a wrong. In: 7th Symposium on Trends in Functional Programming, TFP 2006 (2006)

6. Eker, S.: Term rewriting with operator evaluation strategies. In: Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, WRLA 1998. Electronic Notes in Theoretical Computer Science, vol. 15, pp. 311–330 (1998)
7. Insa, D., Silva, J., Riesco, A.: Balancing execution trees. In: Gulías, V.M., Silva, J., Villanueva, A. (eds.) Proceedings of the 10th Spanish Workshop on Programming Languages, PROLE 2010, pp. 129–142. Ibergarceta Publicaciones (2010)
8. MacLarty, I.: Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne (2005)
9. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
10. Naish, L.: A declarative debugging scheme. Journal of Functional and Logic Programming 1997(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. Nilsson, H., Sparud, J.: The evaluation dependence tree as a basis for lazy functional debugging. Automated Software Engineering 4, 121–150 (1997)
13. Pope, B.: Declarative Debugging with Buddha. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 273–308. Springer, Heidelberg (2005)
14. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: Declarative debugging of rewriting logic specifications. Journal of Logic and Algebraic Programming (2011) (to appear)
15. Shapiro, E.Y.: Algorithmic Program Debugging. ACM Distinguished Dissertation. MIT Press (1983)
16. Silva, J.: A Comparative Study of Algorithmic Debugging Strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
17. 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)