Improving the Debugging of Membership Equational Logic Specifications*

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

Technical Report SIC-02-11

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

March 2011

*Research supported by MEC Spanish project *DESAFIOS* (TIN2006-15660-C02-01), Comunidad de Madrid program *PROMESAS* (S0505/TIC/0407), TIN2008-06622-C03-01, S-0505/TIC/0407, and UCM-BSCH-GR58/08-910502.

Abstract

Declarative debugging is a debugging technique that abstracts the execution details, that can be difficult to follow in general in declarative languages, to focus on results. It relies on a data structure called *debugging tree*, that represents the computation and is traversed by asking questions to the user about the correction 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

Contents

1 Introduction

Declarative debugging [8], also called *algorithmic debugging*, is a debugging technique that abstracts the execution details, that can be difficult to follow in general in declarative languages, to focus on results. It is a two-phase process [5]: 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 correction of the computations associated to the current node until a *buggy node*, that is an incorrect node with correct children, is found. The structure of the debugging tree must ensure that buggy nodes are associated to incorrect fragments of the program. Thus, finding a buggy node equals to finding a bug in the program. Since the oracle is usually the user, the number and complexity of the questions are the main issues when discussing the applicability of the technique.

In the case of the declarative language Maude [3], we have addressed the problem of reducing the number of questions in previous papers [6, 7]. In these works a debugging tree corresponds to a *proof tree* for the wrong result in a suitable semantic calculus. In order to reduce the number of questions, all the nodes in the tree that correspond to valid logic inferences are removed, keeping only those nodes whose validity rely on the program statements. This simplified tree is called the *abbreviated proof tree*, or APT. It has been proven that applying declarative debugging to APTs results in a correct and complete debugging technique.

This paper faces the second issue: the complexity of the questions performed to the user. In particular our goal is to display every term contained in a question in the most reduced form that it has reached during the computation. For instance, if the APT contains a node $f(a) \to b$ and it also contains $a \to c$, then a question about $f(c) \to b$ is preferable to a question about $f(a) \to b$ in terms of simplicity. This principle must be applied recursively, considering for instance if c has been reduced to another value. However, the first naïve approach that consists of replacing each term by its reduced form is not safe, because in general it produces APTs that do not correspond to a valid proof tree, and the technique can become incorrect or incomplete.

Thus, our goal is to transform a proof tree T into another proof tree T' with the same root but whose corresponding APT presents terms in their most reduced form, and then use this APT for debugging. Since T' is also a proof tree for the same computation the soundness of the technique is not compromised, and the theoretical results presented in previous papers remain valid.

The rest of the paper is organized as follows: Section 2 introduces Maude functional modules and the debugging trees used to debug this kind of modules. Section 3 add *colors* to proof trees. Section 4 presents the concept of reduction in relation to proof tree. Next, Section 5 presents the initial transformations applied to the trees. This initial transformation is required by the algorithm of Section 6, which also include the main theoretical result of this report. Finally, the work ends presenting some conclusions.
The theoretical principles introduced in this paper have been implemented in the declarative debugger for Maude that can be found at `http://maude.sip.ucm.es/debugging/`.

2 Debugging Proof Trees

We present in this section Maude and the debugging trees used to debug Maude specifications.

2.1 Maude

For our purposes in this paper we are interested in the equational subset of Maude, which corresponds with specifications in Membership Equational Logic (MEL) [2, 4]

Maude functional modules [3, Chap. 4], introduced with syntax `fmod ... endfm`, are executable membership equational 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 [3].

(**Reflexivity**)

$$\frac{}{e \to e}\ \mathsf{Rf}_{\to}$$

(**Congruence**)

$$\frac{e_1 \to e_1' \quad \ldots \quad e_n \to e_n'}{f(e_1, \ldots, e_n) \to f(e_1', \ldots, e_n')}\ \mathsf{Cong}_{\to}$$

(**Transitivity**)

$$\frac{e_1 \to e' \quad e' \to e_2}{e_1 \to e_2}\ \mathsf{Tr}_{\to}$$

(**Replacement**)

$$\frac{\{\theta(u_i) \downarrow \theta(u_i')\}_{i=1}^n \quad \{\theta(v_j) : s_j\}_{j=1}^m}{\theta(e) \to \theta(e')}\ \mathsf{Rep}_{\to}$$
if $e \to e' \Leftarrow \bigwedge_{i=1}^n u_i = u_i' \wedge \bigwedge_{j=1}^m v_j : s_j$

(**Subject Reduction**)

$$\frac{e \to e' \quad e' : s}{e : s}\ \mathsf{SRed}$$

(**Membership**)

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

Figure 1: Semantic calculus for Maude functional modules

2.2 Debugging trees

The debugging trees for Maude specifications [7] 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 [2, 4], where we assume that the equations are terminating and confluent and hence they can be oriented from left to right, and that replacement and membership inference rules keep the applied statement as part of the rule in order to point this statement as wrong when a buggy node is found. In the second step a pruning function, called APT (see Figure 2), 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. More specifically, this transformation tries to present questions where all the subterms of the term being reduced are in normal form.

Proof trees and abbreviated proof tree nodes contain *judgments*. A judgment is either of the form $f(t_1, \ldots, t_n) \to t$ or $t : s$. If N is a node in a tree T we will use the notation T_N to represent the subtree of T rooted by N, $root(T)$ to indicate the judgment at the root node of T, and $children(T)$ for referring to the forest of children nodes of the root of T.

3 Coloring Proof Trees

When examining a proof tree we are interested in distinguishing whether two syntactically identical terms are copies of the same term or not. The idea is to achieve this goal by "painting" with the same color related terms in a proof tree. 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 will refer to colored terms as *c-terms* and to trees with colored terms in their nodes as *c-trees*. Next we introduced some basic definitions about term positions and colors. The notation for term positions is standard in term rewriting texts, see for instance [1].

Definition 1.

1. *A position of a term is a sequence of natural numbers separated by the symbol . that determines one of its subterms. Given a term t by $pos(t)$ we denote the set of positions in t, which is defined as $pos(X) = \epsilon$; $pos(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{i.p \mid i \in \{1, \ldots, n\} \wedge p \in pos(t_i)\}$, where ϵ denotes the empty or top position. By $t|_p$ we denote the subterm of t at position $p \in pos(t)$, defined as $t|_\epsilon = t$; $f(t_1, \ldots, t_n)|_{i.p} = t_i|_p$. We extend positions to judgments in proof trees by considering the symbols $\to$ and $:$ as infix function symbols. For instance $(f(t_1, \ldots, t_n) \to t)|_1 = f(t_1, \ldots, t_n)$, $(f(t_1, \ldots, t_n) : s)|_2 = s$.*

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

$$(\mathbf{APT}_1) \quad APT\left(\frac{T_1 \dots T_n}{af}(R)\right) = \frac{APT'\left(\frac{T_1 \dots T_n}{af}(R)\right)}{af} \ ((R) \text{ any inference rule})$$

$$(\mathbf{APT}_2) \quad APT'\left(\frac{}{e \to e}(Rf)\right) = \emptyset$$

$$(\mathbf{APT}_3) \quad APT'\left(\frac{\frac{T_1 \dots T_n}{e_1 \to e'}(Rep) \quad T'}{e_1 \to e_2}(Tr)\right) = \left\{\frac{APT'(T_1) \dots APT'(T_n)\ APT'(T')}{e_1 \to e_2}(Rep)\right\}$$

$$(\mathbf{APT}_4) \quad APT'\left(\frac{T_1 \quad T_2}{e_1 \to e_2}(Tr)\right) = APT'(T_1) \ \cup \ APT'(T_2)$$

$$(\mathbf{APT}_5) \quad APT'\left(\frac{T_1 \dots T_n}{e_1 \to e_2}(Cong)\right) = APT'(T_1) \ \cup \dots \ \cup \ APT'(T_n)$$

$$(\mathbf{APT}_6) \quad APT'\left(\frac{T_1 \quad T_2}{e : s}(SRed)\right) = APT'(T_1) \ \cup \ APT'(T_2)$$

$$(\mathbf{APT}_7) \quad APT'\left(\frac{T_1 \dots T_n}{e : s}(Mb)\right) = \left\{\frac{APT'(T_1) \dots APT'(T_n)}{e : s}(Mb)\right\}$$

$$(\mathbf{APT}_8) \quad APT'\left(\frac{T_1 \dots T_n}{e_1 \to e_2}(Rep)\right) = \left\{\frac{APT'(T_1) \dots APT'(T_n)}{e_1 \to e_2}(Rep)\right\}$$

Figure 2: Transforming rules for obtaining abbreviated proof trees

2. *We say that a term t is colored if there is a mapping color $:: t \times pos(t) \to \mathbb{N}$. We say that the natural number $color(t', p)$ is the color of t' for any $t' = t|_p$.*

3. *We say two terms t_1, t_2 are* equally colored *if $pos(t_1) = pos(t_2)$ and $color(t_1, p) = color(t_2, p)$ for every $p \in pos(t_1)$.*

4. *We say that a proof tree T is* colored *if every judgment contained in a node of T is colored, and if:*

   (a) *Every subterm at the root node N of the tree have a different, new color, i.e. $color(N, p_1) \neq color(N, p_2)$ for every $p_1 \neq p_2$, $p_1, p_2 \in pos(N)$.*

   (b) *Every use of a program equation or membership axiom r in a* replacement *or* membership *inference step must be colored with new colors and must verify $color(r, p_1) \neq color(r, p_2)$ for every $p_1 \neq p_2$, $p_1, p_2 \in pos(r)$, except for the occurrences of the same variable, which must have the same color.*

   (c) *In the inference rules at Figure 1 the same symbols represent equally colored occurrences of the same term. For instance in a reflexivity step the lefthand and righthand side must correspond to equally colored terms.*

When talking about colored trees, the notation $t_1 = t_2$ indicates that t_1 and t_2 are equally colored. Hence, talking about two occurrences of a c-term t means implicitly two copies of the same term equally colored.

The following lemma will be useful when relating occurrences of the same term in a colored tree:

Lemma 1. *Let T be a colored proof tree. Then:*

1. *Every term t occurring in a node N of T different from the root*

   (a) *Occurs also in the parent of N, or*

   (b) *Occurs also in a sibling of N, or*

   (c) *Neither of the two previous possibilities hold and t is a new term introduced by a membership axiom or equation that does not occur in T out of the subtree rooted by N.*

2. *If a node N contains an occurrence of some term t but its parent N' does not include t, then t cannot occur out of the subtree rooted by N'.*

3. *Let N_1, N_2 be two nodes in T containing occurrences of the same term t, and let N be the deepest common ancestor to N_1, N_2. Then all the nodes in the path from N to N_1 and from N to N_2, except possibly N, contain t.*

4. *Let T_1, T_2 be two sibling proof trees in T and t a term in $root(T_1)$ not occurring in $root(T_2)$. Then t cannot occur in T_2.*

5. *If a node N is of the form $t \to t'$ then the subtree T_N:*

   (a) *Cannot contain any node either of the form $a \to b$ or $a : s$ with $a \neq t$ and t occurring in a.*

   (b) *Cannot contain any node of the form $a \to b$ with $b \neq t$ and t occurring in b.*

Proof.

1. From the definition of colored proof tree and the structure of the semantic rules of Figure 1 applied to the parent of N:

   - Reflexivity. This is not possible since this rule contains no premises and N must be one of the premises.
   - Transitivity. If N is the first premise, every subterm of t_1 occurs at its parent and every subterm of t' as its sibling. Analogous for the other premise.
   - Congruence. Then N is a premise of the form $t_i \to t'_i$ and all its subterm occur in its parent.
   - Subject Reduction. If N is the premise $t \to t'$, every subterm of t occur also in its parent, and every subterm of t' in its sibling. Analogous for t' in the second premise.
   - Membership. All the subterm coming from the instantiation by θ of a variable x occurring in the lefthand side t of the membership axiom occur also at is parent in $\theta(t)$. The rest of the subterms occurring in the premises correspond to new terms which by the proof tree construction will have a new different color and that therefore cannot occur out of its own subtree.
   - Replacement. Analogous to the membership inference.

2. As we have seen in the previous result the repeated occurrences are shared either by the siblings or by the parent. But if the parent of N does not contain t then the occurrences of the term can reach the siblings of N, its immediate descendants and recursively to other nodes in the subtree rooted by N', but t cannot occur out of this subtree where new occurrences of the same term will have different colors and will therefore considered different.

3. We can prove that t is in every node between N and N_1 using induction on the number n of nodes in the path from N to N_1 (analogous for the nodes in the path from N to N_2): if $n = 0$ or $n = 1$ (basis) the result holds trivially: if $n = 0$ then $N = N_1$, and if $n = 1$ then N if the parent of N_1. If $n > 1$ (inductive step) then we consider the parent N' of N_1, $N' \neq N$. N' must contain t because otherwise applying the previous result t cannot occur out of the subtree rooted by N' and N_2 is out of this subtree since N' is not an ancestor of N_2. Then N' contains t, and the result follows applying the inductive hypothesis to the path from N to N'.

4. By contradiction: assume that N is a node of T_2 containing t. The deepest common ancestor to $root(T_1)$ and N must be obviously some ancestor N' of $root(T_1)$. Since T_1 and T_2 are siblings then N' is ancestor of $root(T_2)$, and hence $root(T_2)$ is in the path from N' to N. Then by the previous point $root(T_2)$ should contain t, which is a contradiction.

□

4 Reductions

Definition 2. *Let T be an APT, and t, t'. We say that $t \to t'$ is a reduction w.r.t. T if there is a node $N \in T$ of the form $a \to b$ verifying:*

- $pos(t, a) \neq \emptyset$

- $t' = t[a \mapsto b]$.

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 $t_0 = t \to t_1 \to t_2 \dots$ s.t. each $t_i \to t_{i+1}$ is a reduction w.r.t. T.*

We are interested in the number of steps necessary to reduce all the nodes in an APT:

Definition 3. *Let T be an APT. Then:*

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

In this definition the length of a reduction chain $t_0 \to \dots \to t_n$ is defined as n.

Definition 4. *We say that an occurrence of a c-term t occurring in an APT T is in* normal form w.r.t. *T is there is no reduction for any c-subterm of t in T.*

The following result ensures that every reduction chain ends with a normal form w.r.t. T.

Lemma 2. *Let T be an APT and t a c-term in T. Then every reduction chain $t \to t_1 \to t_2 \to \dots$ starting at t is finite.*

Proof. If t is already a normal form w.r.t. T the result holds trivially since there is no reduction chain starting by t. Otherwise we define an order $\sqsubseteq$ between terms in a tree T based on the following auxiliary definitions:

- $SubTerm(t) = \{t' \mid t' = t\mid_p \text{ for some } p \in pos(t)\}$.
- $|T|$ = number of nodes in T.
- $maxTree(T, t) = max\{\, |T^{\mathsf{Rep}}_{t \to t}| \mid T^{\mathsf{Rep}}_{t \to t} \in T\}$.
- $t \preceq t' = maxTree(t) \leq maxTree(t')$.
- $multi(t) = \{\!|\, maxTree(T, t') \mid t' \in SubTerm(t) \,|\!\}$, with $\{\!|\ \dots\ |\!\}$ the notation for representing multisets. Observe that every subterm of t contributes to $multi(t)$ with a single value even if it occurs several times in t.
- $\sqsubseteq$ = multiset order induced by $\preceq$ on *multi*.

Given any reduction chain $t_0 = t \to t_1 \to t_2 \to \dots$ we show that $t_{i+1} \sqsubseteq t_i$ for every $i \geq 0$ which means that the reduction chain cannot be infinite, thus ending in a normal form. The reduction $t_i \to t_{i+1}$ means that there are positions $p_1 \dots, p_k \in pos(t_i)$, $k > 0$, such that $t_i\mid_{p_1} = t_i\mid_{p_k} = f(t'_1, \dots, t'_n)$, with f some function symbol, and a node N in T of the form $f(t'_1, \dots, t'_n) \to b$ conclusion of a replacement inference step with $t_{i+1} = t_i[f(t'_1, \dots, t'_n) \mapsto b]$.

Then proving $multi(b) \sqsubseteq multi(f(t'_1, \dots, t'_n))$ proves $multi(t_{i+1}) \sqsubseteq multi(t_i)$. The multiset $multi(f(t'_1, \dots, t'_n))$ must be of the form

$$multi(f(t'_1, \dots, t'_n)) = \{\!|\, v_1, \dots, v_r, m |\!\}$$

where $m = maxTree(f(t'_1, \dots, t'_n)) \geq |\ T_N\ |$, with T_N the subtree of T rooted by N, and for every $j = 1 \dots r$, $v_j = maxTree(a')$ for some $a' \in SubTerm(t'_i)$, $1 \leq n$. Now we check the form of $multi(t_{i+1})$, which depends on the subexpressions of b. b corresponds to an instance of the righthand side of the replacement rule applied at N. Therefore every subexpression t' of b is either:

- An expression which was already in the lefthand side $f(t'_1, \dots, t'_n)$. This can only happen if it corresponds to the instance of some variable x in the replacement rule such that x occurs in both the left and the righthand side. In this case t' is a subexpression of some t'_i with $1 \leq i \leq n$, and therefore $maxTree(t') = v_j$ for some $1 \leq j \leq r$.

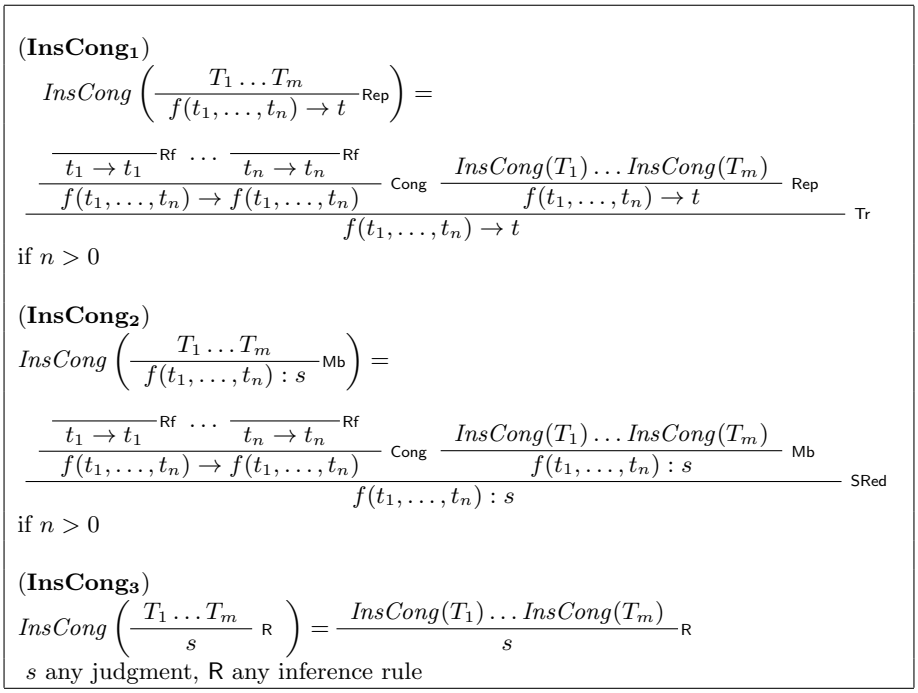

Figure 3: Insert Congruences (*InsCong*)

- A value of the form $c(t'_1, \ldots, t'_n)$ with c occurring only in the righthand side of the original replacement rule. Then by the construction of the colored proof trees $c(t'_1, \ldots, t'_n)$ has a different new color that cannot occur out of T_N. Therefore any replacement for $c(t'_1, \ldots, t'_n)$ must be inside T_N and $maxTree(c(t'_1, \ldots, t'_n)) <| T_N |$.

Then $multi(b)$ can be obtained from $multi(f(t'_1, \ldots, t'_n))$ by possible removing some of the values v_j, $1 \leq j \leq r$ and by replacing m by some values $m' < m$. Therefore $multi(b) \sqsubseteq multi(f(t'_1, \ldots, t'_n))$.

□

Our technique assumes that there is only one normal form for each expression in the tree.

Definition 5. *Let T be a colored proof tree. We say that T is* confluent *if every c-expression e occurring in T has a unique normal form w.r.t. T.*

Lemma 3. *If t is in normal form w.r.t. T then every subexpression of t is in normal form w.r.t. T.*

Proof. Straightforward from Definition 4. □

5 Canonical Trees

A given statement can allow different proof trees. In order to simplify the transformation algorithm it will be useful to assume that the proof trees are in a *canonical form* defined as follows.

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

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

where the transformations InsCong *(insert congruences),* NTr *(normalize transitivities), and* RemInf *(remove superfluous inferences) are defined in Figures 3, 4, and 5 respectively.*

It is assumed that the rules of each transformation are applied top-down, for example if both (**NTr1**) and (**NTr2**) can be applied only the first one will be chosen.

Next we define and prove the possible forms of the trees obtained after each transformation. We start by proving that the proof trees obtained by the *InsCong* have the following form, which we call $form_I$:

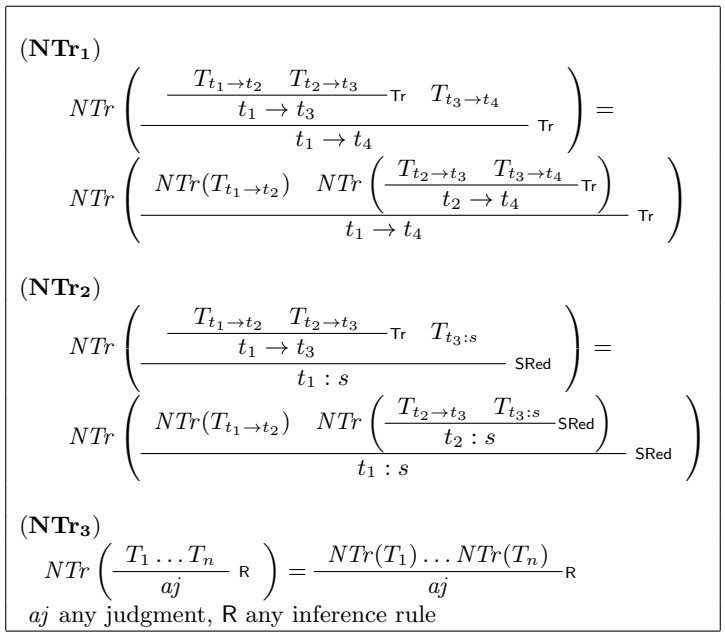

Figure 4: Normalize Transitivities (NTr)

Definition 7. *We define the class $form_I$ as the set of trees consisting of:*

1. *A single node corresponding to a reflexivity inference step.*
2. *A tree with a congruence step at the root whose premises are in $form_I$.*
3. *A replacement or membership applied to a term t with $arity(t) = 0$ whose premises are in $form_I$.*
4. *A transitivity or a subject reduction whose premises are trees in $form_I$.*
5. *A transitivity (respectively a subject reduction) whose left premise is a tree in $form_I$ rooted by a congruence, and whose right premise is rooted by a replacement (respectively a membership) with premises in $form_I$.*

Lemma 4. *Let T a proof tree. Then $InsCong(T)$ is in $form_I$.*

Proof. We apply induction in the number of nodes n of T, distinguishing cases depending on the inference step applied at the root of T.

Basis, $n = 1$

- Reflexivity. In this case the tree does not change and we obtain the same reflexivity, that corresponds to a $form_I$ tree of type 1.
- Unconditional replacement. Assuming that the inference in the root is $t \to t'$, we distinguish whether $arity(t) = 0$ or not. In the first case (**InsCong$_3$**) is applied, the tree does not change, and it corresponds to a $form_I$ tree of type 3. In other case, (**InsCong$_1$**) is applied and a $form_I$ tree of type 4 is obtained.
- Unconditional membership. Analogous to the case above.

Inductive step, $n > 1$

- Conditional replacement. Assuming that the inference in the root is $t \to t'$, we distinguish whether $arity(t) = 0$ or not:
  - If $arity(t) = 0$ then (**InsCong$_3$**) is applied. The result trees have the form $form_I$ by hypothesis, obtaining a tree of type 3.

– If $arity(t) > 0$ then (**InsCong$_1$**) is applied. By hypothesis, the application of the function to the premises returns trees of the form $form_I$, the left premise has this form by construction and the right premise is rooted by a replacement and its premises have the form $form_I$, then this tree has type 5.

- Conditional membership. Analogous to the case above.
- Congruence. By hypothesis the premises have form $form_I$, and then the transformation has type 2.
- Transitivity. By hypothesis the premises have form $form_I$, and then the transformation has type 4.

□

Next we define a new form, called $form_N$, proving that $NTr(InsCong(T))$ is of this form.

Definition 8. *We define the class $form_N$ as the set of trees consisting of:*

1. *A single node corresponding to a reflexivity inference step.*
2. *A tree with a congruence step at the root whose premises are in $form_N$.*
3. *A replacement or membership with root corresponding to a term t with $arity(t) = 0$ whose premises are in $form_N$.*
4. *A transitivity or a subject reduction whose premises are trees in $form_N$, and with the left premise not a transitivity.*
5. *A transitivity (respectively a subject reduction) whose left premise is a tree in $form_N$ rooted by a congruence, and whose right premise is rooted either by a replacement (respectively a membership) with premises in $form_N$, or by a transitivity (respectively a subject reduction) with a replacement of arity greater than 0 and with premises in $form_N$ as left premise, and a tree in $form_N$ as right premise.*

Lemma 5. *Let T a proof tree of the form $form_I$. Then $NTr(T)$ is in $form_N$.*

Proof. We apply complete induction, distinguishing cases on the possible forms of T according to Definition 7.

1. Reflexivity. In this case (**NTr$_3$**) is applied, the tree does not change and it has type 1.
2. Congruence. By (**NTr$_3$**) the result is a congruence and by the induction hypothesis its premises are in form $form_N$. Then the tree is in $form_N$, type 2.
3. Replacement or membership for a term with arity 0. Analogous to the previous case and producing a $form_N$ tree of type 3.
4. Transitivity with premises in form $form_I$. If the left premise is not a transitivity, then (**NTr$_3$**) is applied and by using the induction hypothesis we obtain a $form_N$ tree of type 4.

   If the left premise is a transitivity then the original tree has the form:

   $\dfrac{A_1 \quad A_2}{t \to t'}\ \mathsf{Tr}$ with $A_1 \equiv \dfrac{B_1 \quad B_2}{t \to t_1}\ \mathsf{Tr}$, A_1', A_2' and A_2 in $form_I$. Then the transformation rule (**NTr$_1$**) yields:

   $$NTr\left(\frac{\dfrac{A_1' \quad A_2'}{t \to t'}\ \mathsf{Tr}}{}\right)$$

   and

   $$A_1' \equiv \frac{\dfrac{B_1' \quad B_2'}{t \to t_1}\ \mathsf{Tr}}{}$$

where $A'_1 = NTr(A_1), A'_2 = NTr(A_2), B'_1 = NTr(B_1)$, and $B'_2 = NTr(B_2)$. By hypothesis we know that A'_1 and A'_2 are in the form $form_N$ and B'_1 is not rooted by a transitivity. Thus, we apply again NTr and we obtain

$$\frac{B'_1 \quad NTr\left(T \equiv \dfrac{B'_2 \quad A'_2}{t_1 \to t'}\,\mathsf{Tr}\right)}{t \to t'}\,\mathsf{Tr}$$

We distinguish the form of B_2:

- If B_2 is a tree of the form $form_N$ then T has the form $form_N$ and by hypothesis $NTr(T)$ is in form $form_N$ of type 4.
- If B_2 is a replacement with its premises of the form $form_C$, the application of NTr transforms it into a replacement with its premises of the form $form_N$ and we have a tree of type 4.
- If B_2 is a transitivity with its left premise a replacement with its premises of the form $form_C$ and its right premise a tree of form $form_C$, the application of NTr transforms the tree into another one whose left premise is a replacement whose premises are in the form $form_N$, the right premise builds a new tree in form $form_N$ with A'_2 and we obtain a tree of type 4.
- Transitivity whose left premise is a congruence and whose right premise is a replacement. By hypothesis the premises have the form $form_N$ rooted respectively by a congruence and a replacement, and the tree is of the type 4.

5. Subject reduction. Analogous to the transitivity.

□

Lemma 6. *Let T_1 and T_2 proof trees in $form_N$. Then merge (T_1, T_2) is in $form_N$.*

Proof. We apply induction over the size $n = |T_1| + |T_2|$.

Basis, $n = 2$

Since the trees cannot be rooted by a transitivity, the rule (**Merge₂**) is applied and we obtain a proof tree in form $form_N$.

Inductive step, $n > 2$

We distinguish cases over T_1.

- If it is not a transitivity, then the right premise is a tree of the form $form_N$ by hypothesis and we obtain a tree of the form $form_N$.
- If it is a transitivity, we apply (**Merge₁**). We know that the left premise is on the form $form_N$ and it is not another transitivity. By hypothesis, the new right premise of the tree is of the form $form_N$ and thus the whole tree is on the form $form_N$.

□

Finally we define the class $form_C$ which corresponds to the proof tree in *canonical* form .

Definition 9. *We say that a proof tree is in canonical form when it is in $form_C$, which is defined as the set of proof trees consisting of:*

1. *A single node corresponding to a reflexivity inference step.*
2. *A tree with a congruence step at the root whose premises are in $form_C$.*
3. *A replacement or membership applied to a term t with $arity(t) = 0$ whose premises are trees in $form_C$.*
4. *A transitivity of the form:*

$$\frac{\dfrac{F}{t_1 \to t_3}\,\mathsf{Cong} \quad \dfrac{\dfrac{F'}{t_3 \to t_4}\,\mathsf{Rep} \quad T_{t_4 \to t_2}}{t_3 \to t_2}\,\mathsf{Tr}}{t_1 \to t_2}\,\mathsf{Tr}$$

where F and F' are sequences of trees in $form_C$ and $T_{t_4 \to t_2}$ is in $form_C$ with $t_4 \neq t_2$.

5. *A transitivity of the form:*

$$\dfrac{\dfrac{F}{t_1 \to t_3}\,\mathsf{Cong} \quad \dfrac{F'}{t_3 \to t_2}\,\mathsf{Rep}}{t_1 \to t_2}\,\mathsf{Tr}$$

where F and F' are sequences of trees in $form_C$.

6. *A transitivity of the form:*

$$\dfrac{\dfrac{F}{t_1 \to t_3}\,\mathsf{Rep} \quad T_{t_3 \to t_2}}{t_1 \to t_2}\,\mathsf{Tr}$$

where F is a sequence of trees in $form_C$, $T_{t_3 \to t_2}$ in $form_C$, $arity(t_1) = 0$ and $T_{t_3 \to t_2}$ not rooted by a reflexivity.

7. *A subject reduction of the form:*

$$\dfrac{\dfrac{F}{t_1 \to t_2}\,\mathsf{Cong} \quad \dfrac{\dfrac{F'}{t_2 \to t_3}\,\mathsf{Rep} \quad T_{t_3:s}}{t_2 : s}\,\mathsf{SRed}}{t_1 : s}\,\mathsf{SRed}$$

where F and F' are sequences of trees in $form_C$ and $T_{t_3:s}$ is a tree in $form_C$.

8. *A subject reduction of the form:*

$$\dfrac{\dfrac{F}{t_1 \to t_2}\,\mathsf{Cong} \quad \dfrac{F'}{t_2 : s}\,\mathsf{Mb}}{t_1 : s}\,\mathsf{SRed}$$

where F and F' are sequences of trees in $form_C$.

9. *A subject reduction of the form:*

$$\dfrac{\dfrac{F}{t_1 \to t_2}\,\mathsf{Rep} \quad T_{t_2:s}}{t_1 : s}\,\mathsf{Tr}$$

where F is a sequence of trees in $form_C$, $T_{t_2:s}$ in $form_C$, and $arity(t_1) = 0$.

Lemma 7. *Let T a proof tree of the form $form_N$. Then $RemInf(T)$ is in canonical form, i.e. $RemInf(T)$ is in $form_C$.*

Proof. We apply induction over the size of the trees:
Basis, $n = 1$

- Reflexivity, unconditional replacement and unconditional membership. The tree does not change and we obtain a tree of type 1 and 3 respectively.

Inductive step, $n > 1$

- Congruence. By hypothesis the premises are in canonical form and we have a tree of type 2.
- Conditional replacement and membership. By hypothesis the premises are in canonical form and thus the tree, after applying (**RemInf$_1$**) we obtain a tree of type 3. respectively.
- Transitivity wit a reflexivity as left premise. In this case (**RemInf$_4$**) is applied. By hypothesis the right premise, that becomes the new tree, is in canonical form.
- Transitivity with a replacement with the arity of the lefthand side 0 as left premise. In this case by hypothesis are in canonical form and we obtain a tree of type 9.

- Transitivity with a congruence as left premise. We distinguish cases in the right premise:
  - Reflexivity. In this case (**RemInf$_5$**) is applied. By hypothesis the application of *RemInf* to the left premise, the new tree, is in canonical form.
  - Replacement with the arity of the lefthand side 0. By hypothesis the premises are in canonical form and we obtain a tree of type 6.
  - Congruence. In this case (**RemInf$_1$**) is applied, by Lemma 6 the premises are in the form $form_N$ and we can apply the induction hypothesis, obtaining a tree of type 2.
  - Transitivity.
    * Its left premise is a reflexivity. We apply (**RemInf$_3$**) and, since the tree is now smaller, the proposition holds by induction.
    * Its left premise is a replacement. We know that the right premise is in form $form_N$, so we apply (**RemInf$_6$**). If the right premise is not a reflexivity the application of *RemInf* preserves the transitivity and the replacement, while by induction the right premise is in canonical form, and we obtain a tree of type 4; if it is a reflexivity we obtain a tree of type 5.
    * Its left premise is a congruence. In that case (**RemInf$_2$**) is applied and the right premise is either a tree of the form $form_N$, a replacement or a transitivity with replacement as left premise and a tree of the form $form_N$ as right premise. In the last two cases, when *RemInf* is applied we obtain a replacement or a transitivity with a replacement as left premise and a canonical tree as right one, so the next application of *RemInf* does not modify the tree, that will have type 4 or 5. However, if this right premise is a tree of the form $form_N$ the application of *RemInf* returns a tree in canonical form, and we have to distinguish cases again. If this canonical tree is not rooted by a congruence or by a transitivity the proof is analogous to the previous one; if it is a congruence we reason like in 5, while if it is a transitivity we have to take into account that the right premise cannot be another congruence, and thus the next merge will be the last one, obtaining a canonical tree by using Lemma 6 and induction.
- Subject reduction. Analogous to transitivity.

□

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

Proof. Straightforward from Lemmas 4, 5, and 7 □

Observe that the results and definitions of this section can be extended in a natural way to colored proof trees.

6 Algorithm

Definition 10. *Let T be a confluent c-proof tree . We define the* norm *of T , represented by $\| T \|$, as the sum of the lengths of all the reduction chains that can be applied to expressions in $APT(T)$. More formally:*

$$\| T \| = \sum_{\substack{N \in APT(T) \\ N \neq root(APT(T))}} reduc(N, APT(T))$$

where reduc is the function defined in Definition 3.

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

Proof. We prove the following auxiliary hypotheses:

1. $APT(T) = APT(InsCong(T))$ for every c-proof tree T.
2. $APT(T) = APT(NTr(T))$ for every c-proof tree T.

3. $\| T \| \geq \| RemInf(T) \|$.

Then the result is established as follows: let $T' = NTr(InsCong(T))$. T' is a c-tree by Lemmas 4 and 5. Then:

$$\begin{array}{ll} \| RemInf(NTr(InsCong(T))) \| & \leq \text{(by definition 6 and by hypothesis 3)} \\ \| NTr(InsCong(T)) \| & = \text{(by hypothesis 2)} \\ \| InsCong(T) \| & = \text{(by hypothesis 1)} \\ \| T \| & \end{array}$$

Next we prove the three auxiliary hypotheses.

1. $APT(T) = APT(InsCong(T))$ for every c-proof tree T.

We prove the result by complete induction on the number of nodes of T, for the case where the inference applied at the root is (**InsCong$_1$**). The cases corresponding to (**InsCong$_2$**), (**InsCong$_3$**) are analogous.

If (**InsCong$_1$**) has been applied, then: $T = \dfrac{T_1 \dots T_m}{f(t_1, \dots, t_n) \to t}\,\mathsf{Rep}$, with $n > 0$, and $InsCong(T)$ is of the form:

$$T' = \dfrac{\dfrac{\dfrac{}{t_1 \to t_1}\,\mathsf{Rf} \;\dots\; \dfrac{}{t_n \to t_n}\,\mathsf{Rf}}{f(t_1, \dots, t_n) \to f(t_1, \dots, t_n)}\,\mathsf{Cong} \quad \dfrac{InsCong(T_1) \dots InsCong(T_m)}{f(t_1, \dots, t_n) \to t}\,\mathsf{Rep}}{f(t_1, \dots, t_n) \to t}\,\mathsf{Tr}$$

We check that $APT(T) = APT(T')$

$$APT(T) = \text{(by (}\mathbf{APT_1}\text{))} = \dfrac{APT'\left(\dfrac{T_1 \dots T_m}{f(t_1, \dots, t_n) \to t}\,\mathsf{Rep}\right)}{f(t_1, \dots, t_n) \to t} = \text{(by (}\mathbf{APT_8}\text{))} =$$

$$= \dfrac{\dfrac{APT'(T_1) \dots APT'(T_m)}{f(t_1, \dots, t_n) \to t}\,\mathsf{Rep}}{f(t_1, \dots, t_n) \to t}$$

and

$$APT(T') = \text{(by (}\mathbf{APT_1}\text{))} = \dfrac{APT'(T')}{f(t_1, \dots, t_n) \to t} = \text{(by (}\mathbf{APT_4}\text{))} =$$

$$\dfrac{APT'\left(\dfrac{\dfrac{}{t_1 \to t_1}\,\mathsf{Rf} \;\dots\; \dfrac{}{t_n \to t_n}\,\mathsf{Rf}}{f(t_1, \dots, t_n) \to f(t_1, \dots, t_n)}\,\mathsf{Cong}\right) \quad APT'\left(\dfrac{InsCong(T_1) \dots InsCong(T_m)}{f(t_1, \dots, t_n) \to t}\,\mathsf{Rep}\right)}{f(t_1, \dots, t_n) \to t}$$

Using (**APT$_5$**) and then (**APT$_2$**) in the reflexivity premises, we have

$$APT'\left(\dfrac{\dfrac{}{t_1 \to t_1}\,\mathsf{Rf} \;\dots\; \dfrac{}{t_n \to t_n}\,\mathsf{Rf}}{f(t_1, \dots, t_n) \to f(t_1, \dots, t_n)}\,\mathsf{Cong}\right) = \emptyset$$

and by (**APT$_8$**)

$$APT'\left(\dfrac{InsCong(T_1) \dots InsCong(T_m)}{f(t_1, \dots, t_n) \to t}\,\mathsf{Rep}\right) =$$

$$\dfrac{APT'(InsCong(T_1)) \dots APT'(InsCong(T_m))}{f(t_1, \dots, t_n) \to t}\,\mathsf{Rep}$$

Therefore

$$APT(T') = \dfrac{\dfrac{APT'(InsCong(T_1)) \dots APT'(InsCong(T_m))}{f(t_1, \dots, t_n) \to t}\,\mathsf{Rep}}{f(t_1, \dots, t_n) \to t}$$

And the result follows by induction hypothesis.

2 $APT(T) = APT(NTr(T))$ for every c-proof tree T. The reasoning is similar to the previous case.
3 $\| T \| \geq \| RemInf(T) \|$. In this case we reason informally skipping the lengthy formal proof. The idea is that this transformation cannot increase the norm of T because it does not introduce any new nodes in the APT, with the only exception of the application of *merge*, where a new node $t \to t''$ is introduced as consequence of a transitivity step. This new node will be in the APT if its left premise ($T_{t \to t_1}$ in (**Merge$_1$**) and $T_{t \to t'}$ in (**Merge$_2$**)) is a replacement inference (by rule (**APT$_3$**)). However, observe that in this case (**APT$_3$**) removes the left premise. Therefore we substitute in the APT a node $t \to t_1$ (or $t \to t'$) by another $t \to t''$ (respectively $t_1 \to t''$), and we know that the proof tree includes (both before and after the transformation) the proof of $t_1 \to t''$. Then it is possible to check that the norm is not increased because t'' admits less reductions than t'.

□

Lemma 8. *Let T a proof tree rooted by $t \to t'$ and e, e' terms such that $e \in t, e \neq t$. Then the result of replacing e by e' in T is a proof tree.*

Proof. We apply induction over the size n of the trees:
Basis, $n = 1$

- Reflexivity. If we substitute we obtain the same expression in both sides and hence we have another reflexivity.
- Unconditional replacement. We know that we can still apply the same equation because the lefthand side of equations are patterns, and thus the matching cannot rely on terms that are not fully reduced. Thus, the equation can be applied to the new term on the lefthand side. if the righthand side contains e is because the equation propagates the value and we have to replace it as well, while if it does not contain e there is no need of replacing any values, so in both cases we obtain a new proof tree.

Inductive step, $n > 1$

- Congruence. First, we replace e in the premises. If a premise does not contain a proof tree it remains unchanged and thus it is a proof tree, while if it contains e in its lefthand side it is a proof tree by induction hypothesis (note that it is impossible to have e only in the righthand side, because it occurred in the lefthand side of the root). Finally, the terms e that where justified by the premises are justified now by the new premises, while the premises that where propagated from the lefthand side are now propagated as e'.
- Conditional replacement. We can apply the equation for the same reasons shown in the basis, thus we have to prove that the new premises fulfill the conditions of the equation. The conditions that have got e in its lefthand side are by hypothesis proof trees once the replacement takes place, and prove the corresponding conditions because these terms are obtained from the lefthand side of the replacement (since they are colored) and thus it is necessary to change them. Similarly, the conditions that do not contain e are proof trees because they do not change and prove the same conditions than before.
- Transitivity. The left premise contains e in its lefthand side and thus we can apply the induction hypothesis. If the lefthand side of the right premise (i.e., the righthand side of the left premise) also contains we can apply the induction hypothesis again and obtain a proof tree, while if it does not contain it the premise remains unchanged and thus is a proof, obtaining again that the whole tree is a proof tree.

□

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.

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* $t_1 \notin t'_3$.

Proof. From the construction of $APT(T)$ it is easy to check by induction that every node in $APT(T)$ of the form $a \to b$ corresponds in T to either:

- The root of T (rule ($\mathbf{APT_1}$)).
- The consequence of a transitivity inference with a replacement as left premise.
- The consequence of a replacement inference which is not the left premise of a transitivity.

Since $\| T \| > 0$ from Definition 10 T contains a node different from the root

But in this case $a \to b$ cannot be the root by the point 5 of Lemma 1 and the second point of Definition 10. This proves the first point of the Lemma.

To check the second point we consider any occurrence of N_2 in T (it must occur at least once because every node in $APT(T)$ is also in T), and an occurrence of a in N_2. By Definition 10 N_2 is not the root, and by Lemma 1 we can track the occurrence of a to the father of N_2 until a certain node N_3 is reached such that either N_3 is the root or N_3 does not contain a. We check the two cases separately:

- N_3 contains a, and thus N_3 is the root. Then N_3 is either of the form $e \to e'$ with $a \in e$, $a \neq e$, $a \notin e'$ (since e' is in normal form), or of the form $e : s$ with $a \in e$.
- N_3 does not contain a.

□

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,[2] 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.

Step 6 replaces the proofs of the reduction $t_1 \to t'_1$ by reflexivity steps $t'_1 \to t'_1$. Since this transformation 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.

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 already "joined" to create a $t_1 \to t'_1$ proof.

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.

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 e we are further reducing corresponds to the term t' 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' and thus we must use a congruence inference rule to reduce it, using again a transitivity step to infer the new judgment.

Finally, these transformation make the trees to lose their canonical form.

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

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

1. *Let* $T_r = T$
2. *Loop while* $\| T_r \| > 0$
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 (closer to 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* $\| T_{rp} \| = 0$.
4. *Let* C_{er} *be a copy of the tree rooted by* N_{er} *in* T.
5. *Let* C_{ible} *be a copy of the tree rooted by* N_{ible} *in* T *and* p_{ible} *the position of* N_{ible} *in* T.
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) $t_2' = t_1$.

   $$\frac{C_{ible} \quad C_{er}}{t_2 \to t_1'}\ \mathsf{Tr}$$

   (b) $t_2' \neq t_1$.

   $$\frac{C_{ible} \quad \dfrac{C_{er}}{t_2' \to t_2'[t_1 \mapsto t_1']}\ \mathsf{Cong}}{t_2 \to t_2'[t_1 \mapsto t_1']}\ \mathsf{Tr}$$

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

Lemma 9. *Let* T *a canonical c-proof tree such that* $\| T \| > 0$. *Then the result of applying the steps 3 – 9 of Algorithm 1 to this tree is also a c-proof tree with the same root.*

Proof. We prove first the following facts:

- Given the values N_{er}, N_{ible}, C_{er}, C_{ible}, and p_0 from steps 3 – 5, any subtree $T'_{a\to b}$ of T containing N_{ible} and not containing N_{er} is a proof tree rooted by $a \to b[e \mapsto e']$ after applying steps 8 – 9 of Algorithm 1.

  To prove that, note that the change done in step 9 (changing the righthand side of an inference) affects all the nodes coming after this one in postorder:

  Reflexivity This inference rule does not generate any changes.

  Transitivity A change in the left premise affects the right one, and thus affects the father, while a change in the left one only affects the father.

  Congruence A change in a premise only affects the father.

  Replacement A change in the conditions can affect posterior conditions and the father (both due to matching conditions).

  Subject reduction Analogous to transitivity, although the father cannot be changed.

  Membership Analogous to replacement, although the father cannot be changed.

  Since the node changed in step 9 is the first one in preorder, to have a proof tree all the remaining nodes have to be changed as stated in step 8 to remain a proof tree.

  Finally, note that the step 8 changes the righthand side of the root, leading to $t \to t'[e \mapsto e']$.

- Given a proof tree $T_{a\to b}$, $e \in a$, $\| T_{a\to b} \| = 0$ then the tree obtained by applying steps 6 and 7 from Algorithm 1 to $T_{a\to b}$ is a proof tree rooted by $a[e \mapsto e'] \to b$.

Proof. By induction on the size of the tree T. When $|T| = 1$ we can have:

- A reflexivity step, with $e \to e$ as consequence. Thus we have $e = e'$ and the steps 6 and 7 return a proof tree which is constituted by a reflexivity step and $e' \to e'$ as consequence.
- An unconditional membership step. Since the specification is assumed to be sort-decreasing and the lefthand side of memberships must contain a pattern, the substitution of e by a more reduced term e' returns a proof tree.
- In that case this is the replacement with consequence $e \to e'$ and thus it is replaced by a reflexivity, which is a valid proof tree.

Assuming the application of the steps is correct for proof trees $|T| = n$, we prove it for trees with size $|T| \geq n$:

- A transitivity step. We distinguish whether it is the transitivity that will be transformed into a reflexivity step, and thus the fact follows trivially, or it has the form

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

  if $e \in t'$ then the premises are proof trees by induction hypothesis and we have the proof tree:

$$\frac{T_{t_1[e \mapsto e'] \to t'[e \mapsto e']} \quad T_{t'[e \mapsto e'] \to t_2}}{t_1[e \mapsto e'] \to t_2}\ \mathsf{Tr}$$

  In other case, the steps do not affect the right premise, the left one is a proof tree by induction hypothesis and we have the proof tree:

$$\frac{T_{t_1[e \mapsto e'] \to t'} \quad T_{t' \to t_2}}{t_1[e \mapsto e'] \to t_2}\ \mathsf{Tr}$$

- Congruence step. The result is straightforward by induction hypothesis.
- Replacement step with $t \to t'$ as consequence. If it is replaced by a reflexivity, it is trivially a proof tree. In other case, we have, for $0 \leq i \leq n$, $0 \leq j \leq m$

$$\frac{T_{u_i \to t_i} \quad T_{u_i' \to t_i} \quad T_{v_j : s_j}}{t \to t'}\ \mathsf{Rep}$$

  Since e' is in normal form with respect to the tree, we know that the t_i do not contain e. Thus, if u_i (analogously u_i') contains e, the steps return a proof tree by induction hypothesis; in other case, the steps do not affect the trees and thus remain as proof trees. We reason in the same way for the membership conditions. Now we have to prove that the new tree stands for a valid replacement step. The new term $t[e \mapsto e']$ matches the lefthand side of the equation because it is a pattern, and thus cannot depend on unreduced functions. Moreover, the replacements in the equations are also valid: the equalities, since we know that the e was reduced to e' along the condition (because the tree is confluent) we have only anticipated its reduction, and the condition will hold if and only if it held before the steps. We reason analogously, taking into account that the specification is sort decreasing, for membership conditions.
- Subject reduction step.

□

Note now that the first fact can be applied to the left premise of p_0 and the second one to its right premise, and thus after applying steps 3 – 9 of the algorithm they are still proof trees. Moreover, p_0 is not affected by the algorithm, and thus the rest of the tree is not affected by the algorithm, so we must only check p_0 is a correct transitivity inference. This is straightforward since:

- The left premise, initially rooted by $t_l \to t_l'$ is now rooted by $t_l \to t_l'[e \mapsto e']$ by the firs fact.
- The right premise, rooted by $t_r \to t_r'$, is now rooted by $t_r[e \mapsto e'] \to t_r'$ by the second fact.

□

Lemma 10. *Let T be a proof tree in canonical form s.t. $\| T \| > 0$. Then the result of applying the steps 3 - 9 of Algorithm 1 to $T_r = T$ is a proof tree T''' s.t. $\| T''' \| < \| T \|$, $root(T''') = root(T)$.*

Proof. The existence of the nodes $N = e \to e'$ and $M = t \to t'$ mentioned in the step 3 of the algorithm is ensured by Lemma 3. Since M is the first node in post-order satisfying $e \in t'$, $e \notin t$ it has to be the consequence of a replacement inference, because only replacements can introduce values that do not appear in their premises.

We first prove that $root(T''') = root(T)$ by showing that $e \notin root(T)$. It is easy to check that in this case the algorithm steps 3 – 9 will not affect $root(T)$. The node $root(T)$ can be either of the form $a \to b$ or $a : s$. Then:

- $e \notin b$, T contains $e \to e'$ and b is in normal form.
- $e \notin a$. From Lemma 3, we know that there exist at least one node with e in the righthand side but not in the lefthand side. Since $e \notin b$ is not one of these nodes and terms are colored, this cannot happen if the term already exists in the root, and thus the root does not contain e.

T''' is confluent because T was confluent and the algorithm only copies and removes nodes, without introducing any new replacement inference.

Lemma 9 proves that the transformation returns a proof tree.

Finally, we have to check that $\| T''' \| < \| T \|$. We consider the case case a) of step 9, which corresponds to $t' = e$. The case b) $t' \neq e$ is analogous. Let D be the first preorder ancestor of the node at position p_M in T verifying that it is either of the form $a : s$ or of the form $a \to b$ with $e \notin b$. Then it is easy to check that the algorithm only affects the subtree T_D, which must contain all the occurrences of N and M and in particular T_N and T_M. Therefore we need to prove that $\| T'_D \| < \| T_D \|$, with T'_D the result of applying the algorithm steps 3 - 9 to T_D.

The only steps that actually modify T_D are steps 6, 7 and 9. Replacing the nodes rooted by N by reflexivity inferences $e' \to e'$ (step 6) can only make the tree norm smaller, and in any case never larger. Replacing e by e' (step 7) cannot yield a larger norm since e' is in normal form. Finally step 9 replaces the subtree at position p_M by a new tree

$$T'_M = \frac{T_M \quad T_N}{t \to e'} \text{ (Tr)}$$

Now observe that T_D contained the subtree T_M at position p_M. This node is replaced by T'_M in T'_D. Since M was a replacement and not the left premise of a transitivity, it was part of $APT(T_D)$, and since it was of the form $t \to e$ its righthand side contributed to the norm with a value greater than 0. In T'_M the root of M is no longer part of the APT, since now is the left premise of a transitivity. Instead we have $t \to e'$, and e' does not contribute to the norm because it is in normal form. Moreover, the rest of the nodes from T'_M part of $APT(T'_D)$ will correspond either to nodes T_M (except its root) or to nodes in T_N, which were already in $APT(T_D)$. Observe that the algorithm does duplicate these nodes, because T'_M is replacing an occurrence of T_M and also because we have removed previously (step 6) all the occurrences (at least one) of T_N. Therefore $\| T'_D \| < \| T_D \|$.

□

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

Proof. The result is a straightforward consequence of Proposition 2 and Lemma 10.

□

Theorem 2. *Let T be a proof tree with invalid root. Then $APT(T)$ contains a buggy node which points out an incorrect program statement.*

Proof. Seen in [7] □

Corollary 1. *Let T be a confluent c-proof tree in canonical form s.t. $root(T)$ is invalid. Let T_r be the result of applying Algorithm 1 to T. Then $APT(T_r)$ contains a buggy node which points out an incorrect program statement.*

Proof. Straightforward consequence of Theorems 1 and 2. □

7 Concluding Remarks

One of the main criticisms of the declarative debugging tools 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 the cause of Maude functional modules, an improvement in this direction is to ensure that judgments presented to the user contain terms reduced as much as possible. We have presented a transformation that allows us to produce abbreviated proof trees fulfilling this property starting with any valid proof tree for the computation. The result is a debugging tree with questions as simple as possible without increasing the number of questions. Moreover, the theoretical results supporting the debugging technique presented in previous papers remain valid since we have proven that our transformation converts proof trees into proof trees for the same computation.

The transformations have been implemented in a working tool that can be used for debugging wrong answers in Maude that can be found at `http://maude.sip.ucm.es/debugging/`.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

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

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

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

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

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

[7] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. *Journal of Logic and Algebraic Programming*, 2011. To appear.

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

(**RemInf$_1$**)

$$RemInf\left(\begin{array}{c}\dfrac{T_{t_1\to t'_1}\dots T_{t_n\to t'_n}}{f(t_1,\dots,t_n)\to f(t'_1,\dots,t'_n)}\,\mathsf{Cong}\\[2ex]\dfrac{T_{t'_1\to t''_1}\dots T_{t'_n\to t''_n}}{f(t'_1,\dots,t'_n)\to f(t''_1,\dots,t''_n)}\,\mathsf{Cong}\\[2ex]\hline f(t_1,\dots,t_n)\to f(t''_1,\dots,t''_n)\end{array}\,\mathsf{Tr}\right)=$$

$$\frac{RemInf\left(merge\left(T_{t_1\to t'_1},\ T_{t'_1\to t''_1}\right)\right)\dots RemInf\left(merge\left(T_{t_n\to t'_n},\ T_{t'_n\to t''_n}\right)\right)}{f(t_1,\dots,t_n)\to f(t''_1,\dots,t''_n)}\,\mathsf{Cong}$$

(**RemInf$_2$**)

$$RemInf\left(\begin{array}{c}\dfrac{T_{t_1\to t'_1}\dots T_{t_n\to t'_n}}{f(t_1,\dots,t_n)\to f(t'_1,\dots,t'_n)}\,\mathsf{Cong}\\[2ex]\dfrac{\dfrac{T_{t'_1\to t''_1}\dots T_{t'_n\to t''_n}}{f(t'_1,\dots,t'_n)\to f(t''_1,\dots,t''_n)}\,\mathsf{Cong}\quad T_{f(t''_1,\dots,t''_n)\to e}}{f(t'_1,\dots,t'_n)\to e}\,\mathsf{Tr}\\[2ex]\hline f(t_1,\dots,t_n)\to e\end{array}\,\mathsf{Tr}\right)=$$

$$RemInf\left(\begin{array}{c}\dfrac{merge\left(T_{t_1\to t'_1},\ T_{t'_1\to t''_1}\right)\dots merge\left(T_{t_n\to t'_n},\ T_{t'_n\to t''_n}\right)}{f(t_1,\dots,t_n)\to f(t''_1,\dots,t''_n)}\,\mathsf{Cong}\\[2ex]RemInf\left(T_{f(t''_1,\dots,t''_n)\to e}\right)\\[2ex]\hline f(t_1,\dots,t_n)\to e\end{array}\,\mathsf{Tr}\right)$$

(**RemInf$_3$**)

$$RemInf\left(\begin{array}{c}\dfrac{F}{t\to t_1}\,\mathsf{Cong}\qquad\dfrac{\dfrac{}{t_1\to t_1}\,\mathsf{Rf}\quad T_{t_1\to t'}}{t_1\to t'}\,\mathsf{Tr}\\[2ex]\hline t\to t'\end{array}\,\mathsf{Tr}\right)=$$

$$RemInf\left(\begin{array}{c}\dfrac{F}{t\to t_1}\,\mathsf{Cong}\qquad T_{t_1\to t'}\\[2ex]\hline t\to t'\end{array}\,\mathsf{Tr}\right)$$

(**RemInf$_4$**)

$$RemInf\left(\frac{T^{\mathsf{Rf}}\quad T'}{s}\,\mathsf{R}\right)=RemInf\left(\frac{T'\quad T^{\mathsf{Rf}}}{s}\,\mathsf{Tr}\right)=RemInf(T')$$

aj any judgment, R either Tr or SRed

(**RemInf$_5$**)

$$RemInf\left(\frac{T_1\dots T_n}{s}\ \mathsf{R}\ \right)=\frac{RemInf(T_1)\dots RemInf(T_n)}{s}\,\mathsf{R}$$

aj any judgment, R any inference rule

Figure 5: Remove superfluous inferences (*RemInf*)

(**Merge$_1$**)

$$merge\left(\frac{T_{t\to t_1}\quad T_{t_1\to t'}}{t\to t'}\,\mathsf{Tr},T_{t'\to t''}\right)=\frac{T_{t\to t_1}\quad merge\,(T_{t_1\to t'},T_{t'\to t''})}{t\to t''}\,\mathsf{Tr}$$

(**Merge$_2$**)

$$merge\,(T_{t\to t'},T_{t'\to t''})=\frac{T_{t\to t'}\quad T_{t'\to t''}}{t\to t''}\,\mathsf{Tr}$$

Figure 6: Merge Trees