The Journal of Logic and Algebraic Programming 81 (2012) 851–897

Contents lists available at SciVerse ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Declarative debugging of rewriting logic specifications ☆

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

Dpto. Sistemas Informáticos y Computación, Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain

ARTICLE INFO

Article history:
Available online 9 September 2011

Keywords:
Declarative debugging
Rewriting logic
Maude
Wrong answers
Missing answers

ABSTRACT

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

© 2011 Elsevier Inc. All rights reserved.

1. Introduction

Declarative debugging [35], also known as declarative diagnosis or algorithmic debugging, is a debugging technique that abstracts the execution details, which may be difficult to follow in declarative languages, to focus on the results. We can distinguish between two different kinds of declarative debugging: debugging of *wrong answers*, applied when a *wrong* result is obtained from an initial value, which has been widely employed in the logic [38,18], functional [26,28], multi-paradigm [5,20], and object-oriented [6] programming languages; and debugging of *missing answers* [23,38,18,10,1], applied when a result is *incomplete*, which has been less studied because the calculus involved is more complex than in the case of wrong answers. Declarative debugging starts from an incorrect computation, the error symptom, and locates the code (or the absence of code) responsible for the error. To find this error the debugger represents the computation as a *debugging tree* [24], where each node stands for a computation step and must follow from the results of its child nodes by some logical inference. This tree is traversed by asking questions to an external oracle (generally the user) until a *buggy node*—a node containing an erroneous result, but whose children are all correct—is found. Hence, we distinguish two phases in this scheme: the debugging tree *generation* and its *navigation* following some suitable strategy [36].

We present here Maude DDebugger, a declarative debugger for *Maude specifications*. Maude [12] is a high-level language and high-performance system supporting both equational and rewriting logic computation. Maude modules correspond to specifications in *rewriting logic* [21], a logic that allows the representation of many models of concurrent and distributed systems. This logic is an extension of *membership equational logic* [2], an equational logic that, in addition to equations, allows

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

* Corresponding author. Address: Facultad de Informática, Universidad Complutense de Madrid, C/Prof. José García Santesmases s/n, 28040 Madrid, Spain. Tel.: +34 913947648; fax: +34 913947529.

E-mail addresses: ariesco@fdi.ucm.es (A. Riesco), alberto@sip.ucm.es (A. Verdejo), narciso@sip.ucm.es (N. Martí-Oliet), rafa@sip.ucm.es (R. Caballero).

1567-8326/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2011.06.004

one to state *membership axioms* characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules, which represent transitions in a concurrent system and can be nondeterministic. The Maude system supports several approaches for debugging Maude programs: tracing, term coloring, and using an internal debugger [12, Chapter 22]. The tracing facilities allow us to follow the execution of a specification, that is, the sequence of applications of statements that take place. The same ideas have been applied to the functional paradigm by the tracer *Hat* [11], where a graph constructed by graph rewriting is proposed as a suitable trace structure. Term coloring uses different colors to print the operators used to build a term that does not fully reduce. Finally, the Maude internal debugger allows the definition of break points in the execution by selecting some operators or statements. When a break point is found the debugger is entered, where we can see the current term and execute the next rewrite with tracing turned on. However, these tools have the disadvantage that, since they are based on the trace, show the statements applied in the order in which they are executed, and thus the user can lose the general view of the proof of the incorrect computation that produced the wrong result.

Declarative debugging of wrong answers of membership equational logic specifications was studied in [8,7], and was later extended to debugging of wrong answers in rewriting logic specifications in [30], while descriptions of the implemented system can be found in [34], where we present how to debug wrong results due to errors in the statements of the specification. In [32] we investigated how to apply declarative debugging of missing answers, traditionally associated with nondeterministic frameworks [10,23], to membership equational logic specifications. We achieve this by broadening the concept of missing answers to deal with erroneous normal forms and least sorts. Finally, we extended the calculus developed thus far in [31] to debug missing answers in rewriting logic specifications, that is, expected results that the specification is not able to compute. A description of the whole system is presented in [33].

One of the strong points of our approach is that, unlike other proposals like [10], it combines the treatment of wrong and missing answers and thus it is able to detect missing answers due to both wrong and missing statements. The state of the art can be found in [36], which contains a comparison among the algorithmic debuggers B.i.O. [3] (Believe in Oracles), a debugger integrated in the Curry compiler KICS; Buddha [27,28], a debugger for Haskell 98; DDT [9], a debugger for TOY; Freja [26], a debugger for Haskell; Hat-Delta [14], part of a set of tools to debug Haskell programs; Mercury's Algorithmic Debugger [20], a debugger integrated into the Mercury compiler; Münster Curry Debugger [19], a debugger integrated into the Münster Curry compiler; and Nude [25], the NU-Prolog Debugging Environment. We extend this comparison by taking into account the features in the latest updates of the debuggers and adding two new ones: DDJ [16], a debugger for Java

Table 1
A comparative of declarative debuggers I.

	Maude DDebugger	B.i.O.	Buddha	DDJ	DDT	Freja
Implementation language	Maude	Curry	Haskell	Java	Toy (front-end) Java (back-end)	Haskell
Target language	Maude	Curry	Haskell	Java	Toy	Haskell subset
Strategies	TD DQ	TD	TD	TD DQ DRQ SS HF MRF HD	TD DQ	TD
Database/Memoization?	NO/YES	NO/NO	NO/YES	YES/YES	NO/YES	NO/NO
Front-end	Independent prog. trans.	Independent prog. trans.	Independent prog. trans.	Independent prog. trans.	Integrated prog. trans.	Integrated compiler
Interface	APT on demand	Step count	ET	ET on demand	ET (XML/TXT)	ET
Debugging tree	Main memory on demand	Main memory on demand	Main memory on demand	Database	Main memory	Main memory
Missing answers?	YES	NO	NO	NO	YES	NO
Accepted answers	`yes no dk tr`	`yes no dk`	`yes no dk in tr`	`yes no dk tr`	`yes no dk tr`	`yes no my mn`
Tracing subexpressions?	NO	NO	NO	NO	NO	NO
ET exploration?	YES	YES	YES	YES	YES	YES
Different trees?	YES	NO	NO	YES	NO	NO
Tree compression?	NO	NO	NO	NO	NO	NO
Undo?	YES	YES	NO	YES	NO	YES
Trusting	MO/FU/FN	MO/FU/AR	MO/FU	FU	FU	MO/FU
GUI?	YES	NO	NO	YES	YES	NO
Version	2.0 (24/5/2010)	Kics 0.81893 (15/4/2009)	1.2.1 (1/12/2006)	2.4 (23/10/2010)	1.2 (29/9/2005)	(2000)

Table 2
A comparative of declarative debuggers II.

	Maude DDebugger	Hat-Delta	Mercury Debugger	Münster Curry Debugger	Nude
Implementation language	Maude	Haskell	Mercury	Haskell (front-end) Curry (back-end)	Prolog
Target language	Maude	Haskell	Mercury	Curry	Prolog
Strategies	TD DQ	HD	TD DQ SD MD	TD	TD
Database/Memoization?	NO/YES	NO/YES	NO/YES	NO/NO	YES/YES
Front-end	Independent prog. trans.	Independent prog. trans.	Independent compiler	Integrated compiler	Independent compiler
Interface	APT	ART (native)	ET on demand	ET	ET on demand
Debugging tree	Main memory on demand	File system	Main memory on demand	Main memory	Main memory on demand
Missing answers?	YES	NO	NO	NO	NO
Accepted answers	`yes no dk tr`	`yes no`	`yes no dk in tr`	`yes no`	`yes no`
Tracing subexpressions?	NO	NO	YES	NO	NO
ET exploration?	YES	YES	YES	YES	NO
Different trees?	YES	NO	NO	NO	NO
Tree compression?	NO	YES	NO	NO	NO
Undo?	YES	NO	YES	NO	NO
Trusting	MO/FU/FN	MO	MO/FU	MO/FU	FU
GUI?	YES	NO	NO	YES	NO
Version	2.0 (24/5/2010)	2.05 (22/10/2006)	Mercury 0.13.1 (1/12/2006)	AquaCurry 1.0 (13/5/2006)	NU-Prolog 1.7.2 (13/7/2004)

programs, and our own debugger, Maude DDebugger. This comparison is summarized in Tables 1 and 2, where each column shows a declarative debugger and each row a feature. More specifically:

- The implementation language indicates the language used to implement the debugger. In some cases front- and back-ends are shown: they refer, respectively, to the language used to obtain the information needed to compute the debugging tree and the language used to interact with the user.
- The target language states the language debugged by the tool.
- The strategies row indicates the different navigation strategies implemented by the debuggers. TD stands for *top-down*, that starts from the root and selects a wrong child to continue with the navigation until all the children are correct; DQ for *divide and query*, that selects in each case a node rooting a subtree half the size of the whole tree; SS for *single stepping*, that performs a post-order traversal of the execution tree; HF for *heaviest first*, a modification of top-down that selects the child with the biggest subtree; MRF for *more rules first*, another variant of top-down that selects the child with the biggest number of different statements in its subtree; DRQ for *divide by rules and query*, an improvement of divide and query that selects the node whose subtree has half the number of associated statements of the whole tree; MD for the divide and query strategy implemented by the Mercury Debugger; SD for *subterm dependency*, a strategy that allows one to track specific subterms that the user has pointed out as erroneous; and HD for the Hat-Delta heuristics.
- Database indicates whether the tool keeps a database of answers to be used in future debugging sessions, while memoization indicates whether this database is available for the current session.
- The front-end indicates whether it is integrated into the compiler or it is standalone.
- Interface shows the interface between the front-end and the back-end. Here, APT stands for the Abbreviated Proof Tree generated by Maude; ART for Augmented Redex Trail, the tree generated by Hat-Delta; ET is an abbreviation of Execution Tree; and step count refers to a specific method of the B.i.O. debugger that keeps the information used thus far into a text file.
- Debugging tree presents how the debugging trees are managed.
- The missing answers row indicates whether the tool can debug missing answers.
- Accepted answers: the different answers that can be introduced into the debugger. `yes`; `no`; `dk` (*don't know*); `tr` (*trust*); `in` (*inadmissible*), used to indicate that some arguments should not have been computed; and `my` and `mn` (*maybe yes* and *maybe no*), that behave as yes and no although the questions can be repeated if needed. More details about these debugging techniques can be found in [36,37].
- Tracing subexpressions means that the user is able to point out a subterm as erroneous.

- ET exploration indicates whether the debugging tree can be freely traversed.
- Whether the debugging tree can be built following different strategies depending on the specific situation is shown in the Different trees? row.
- Tree compression indicates whether the tool implements tree compression [14], a technique to remove redundant nodes from the execution tree.
- Undo states whether the tool provides an undo command.
- Trusting lists the trusting options provided by each debugger. MO stands for trusting modules; FU for functions (statements); AR for arguments; and FN for final forms.
- GUI shows whether the tool provides a graphical user interface.
- Version displays the version of the tool used for the comparison.

The results shown in these tables can be interpreted as follows:

Navigation strategies: Several navigation strategies have been proposed for declarative debugging [36]. However, most of the debuggers (including Maude DDebugger) only implement the basic top-down and divide and query techniques. On the other hand, DDJ implements most of the known navigation techniques (some of them also developed by the same researchers), including an adaptation of the navigation techniques developed for Hat-Delta. Among the basic techniques, only DDJ, DDT, and Maude DDebugger provide the most efficient divide and query strategy, Hirunkitti's divide and query [36].

Available answers: The declarative debugging scheme relies on an external oracle answering the questions asked by the tool, and thus the bigger the set of available answers the easier the interaction. The minimum set of answers accepted by all the debuggers is composed of the answers *yes* and *no*; Hat-Delta, the Münster Curry Debugger, and Nude do not accept any more answers, but the remaining debuggers allow some others. Other well-known answers are *don't know* and *trust*; the former, that can introduce incompleteness, allows the user to skip the current question and is implemented by B.i.O., DDJ, DDT, Buddha, Mercury, and Maude DDebugger, while the latter prevents the debugger from asking questions related to the current statement and is accepted by DDJ, DDT, Buddha, the Mercury debugger, and Maude DDebugger. Buddha and the Mercury debugger have developed an answer *inadmissible* to indicate that some arguments should not have been computed, redirecting the debugging process in this direction; our debugger accepts a similar mechanism when debugging missing answers in system modules with the answer *the term n is not a solution/reachable*, which indicates that a term in a set is not a solution/reachable, leading the process in this direction. Finally, Freja accepts the answers *maybe yes* and *maybe not*, that the debugger uses as *yes* and *not*, although it will return to these questions if the bug is not found.

Database: A common feature in declarative debugging is the use of a database to prevent the tool from asking the same question twice, which is implemented by DDJ, DDT, Hat-Delta, Buddha, the Mercury debugger, Nude, and Maude DDebugger. Nude has improved this technique by allowing this database to be used during the next sessions, which has also been adopted by DDJ.

Memory: The debuggers allocate the debugging tree in different ways. The Hat-Delta tree is stored in the file system, DDJ uses a database, and the rest of the debuggers (including ours) keep it in main memory. Most debuggers improve memory management by building the tree on demand, as B.i.O., Buddha, DDJ, the Mercury debugger, Nude, and Maude DDebugger.

Tracing subexpressions: The Mercury debugger is the only one able to indicate that a specific subexpression, and not the whole term, is wrong, improving both the answers *no* and *inadmissible* with precise information about the subexpression. With this technique the navigation strategy can focus on some nodes of the tree, enhancing the debugging process.

Construction strategies: A novelty of our approach is the possibility of building different trees depending on the complexity of the specification and the experience of the user: the trees for both wrong and missing answers can be built following either a one-step or a many-step strategy (giving rise to four combinations). While with the one-step strategy the tool asks more questions in general, these questions are easier to answer than the ones presented with the many-steps strategy. An improvement of this technique has been applied in DDJ in [17], allowing the system to balance the debugging trees by combining so called *chains*, that is, sequences of statements where the final data of each step is the initial data of the following one.

Tree compression: The Hat-Delta debugger has developed a new technique to remove redundant nodes from the execution tree, called tree compression [14]. Roughly speaking, it consists in removing (in some cases) 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 debugging information for both itself and these children. This technique is very similar to the balancing technique implemented for DDJ in [17].

Tree exploration: Most of the debuggers allow the user to freely navigate the debugging tree, including ours when using the graphical user interface. Only the Münster Curry Debugger and Nude do not implement this feature.

Trusting: Although all the debuggers provide some trusting mechanisms, they differ on the target: all the debuggers except Hat-Delta have mechanisms to trust specific statements, and all the debuggers except DDJ, DDT, and Nude can trust complete modules. An original approach is to allow the user to trust some arguments, which currently is only

supported by B.i.O. In our case, and since we are able to debug missing answers, a novel trusting mechanism has been developed: the user can identify some sorts and some operators as *final*, that is, they cannot be further reduced; with this method all nodes referring to "finalness" of these terms are removed from the debugging tree. Finally, a method similar to trusting consists in using a correct specification as an oracle to answer the questions; this approach is followed by B.i.O. and Maude DDebugger.

Undo command: In a technique that relies on the user as oracle, it is usual to commit an error and thus an undo command can be very useful. However, not all the debuggers have this command, with B.i.O., DDJ, Freja, the Mercury debugger, and Maude DDebugger being the only ones implementing this feature.

Graphical interface: A graphical user interface eases the interaction between the user and the tool, allowing him to freely navigate the debugging tree and showing all the features in a friendly way. In [36], only one declarative debugger—DDT—implemented such an interface, while nowadays four tools—DDT, DDJ, Münster Curry Debugger,[1] and Maude DDebugger—have this feature.

Errors detected: It is worth noticing that only DDT and Maude DDebugger can debug missing answers, while all the other debuggers are devoted exclusively to wrong answers. However, DDT only debugs missing answers due to nondeterminism, while our approach uses this technique to debug erroneous normal forms and least sorts.

Other remarks: An important subject in declarative debugging is scalability. The development of DDJ has taken special care of this subject by using a complex architecture that manages the available memory and uses a database to store the parts of the tree that do not fit in main memory. Moreover, the navigation strategies have been modified to work with incomplete trees. Regarding reusability, the latest version of B.i.O. provides a generic interface that allows other tools implementing it to use its debugging features. Finally, the DDT debugger has been improved to deal with constraints.

Exploiting the fact that rewriting logic is *reflective* [13], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [12, Chapter 14], a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. Therefore, we are able to generate and navigate the debugging tree of a Maude computation using operations in Maude itself. In addition, the Maude system provides another module, `LOOP-MODE` [12, Chapter 17], which can be used to specify input/output interactions with the user. However, instead of using this module directly, we extend Full Maude [12, Chapter 18], which includes features for parsing, evaluating, and pretty-printing terms, improving the input/output interaction. Moreover, Full Maude allows the specification of concurrent object-oriented systems, which can also be debugged. Thus, our declarative debugger, including its user interactions, is implemented in Maude itself.

The rest of the paper is structured as follows. Section 2 presents the preliminaries of our debugging approach. Section 3 describes our calculus while the next section explains how to transform the proof trees built with this calculus into appropriate debugging trees. Section 5 shows how to use the debugger, while Section 6 illustrates it with an example. Section 7 describes the implementation of our tool and Section 8 concludes and presents some future work. We present in Appendix A the detailed proofs of the results stated throughout the paper.

Additional examples, the source code of the tool, and other papers on the subject, including the user guide [29], where the graphical user interface for the debugger is presented, are all available from the webpage http://maude.sip.ucm.es/debugging.

2. Preliminaries

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

2.1. Membership equational logic

A *signature* in membership equational logic is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1\ldots k_n,k}\}_{(k_1\ldots k_n,k)\in K^*\times K}$ a many-kinded signature, and $S = \{S_k\}_{k\in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. The kind of a sort s is denoted by $[s]$. We write $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ to denote, respectively, the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of K-kinded variables. Intuitively, terms with a kind but without a sort represent undefined or error elements.

The atomic formulas of membership equational logic are *equations* $t = t'$, where t and t' are Σ-terms of the same kind, and *membership axioms* of the form $t : s$, where the term t has kind k and $s \in S_k$. *Sentences* are universally-quantified Horn clauses of the form $(\forall X)\, A_0 \Leftarrow A_1 \wedge \cdots \wedge A_n$, where each A_i is either an equation or a membership axiom, and X is a set of K-kinded variables containing all the variables in the A_i. A *specification* is a pair (Σ, E), where E is a set of sentences in membership equational logic over the signature Σ.

Models of membership equational logic specifications are Σ-*algebras* $\mathcal{A}$ consisting of a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1\ldots k_n,k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. Given

[1] Only available for Mac OS X.

a Σ-algebra $\mathcal{A}$ and a valuation $\sigma : X \longrightarrow \mathcal{A}$ mapping variables to values in the algebra, the meaning $[\![t]\!]^{\sigma}_{\mathcal{A}}$ of a term t is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies, under a valuation σ,

- an equation $t = t'$, denoted $\mathcal{A}, \sigma \models t = t'$, if and only if both terms have the same meaning: $[\![t]\!]^{\sigma}_{\mathcal{A}} = [\![t']\!]^{\sigma}_{\mathcal{A}}$; we also say that the equation holds in the algebra under the valuation.
- a membership $t : s$, denoted $\mathcal{A}, \sigma \models t : s$, if and only if $[\![t]\!]^{\sigma}_{\mathcal{A}} \in A_s$.

Satisfaction of Horn clauses is defined in the standard way. Finally, when terms are ground, valuations play no role and thus can be omitted. A membership equational logic specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of ground terms $[t]_E$, and where an equation or membership is satisfied if and only if it can be deduced from E by means of a sound and complete set of deduction rules [2,22].

Since the membership equational logic specifications that we consider are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness [12], their equations $t = t'$ can be oriented from left to right, $t \to t'$. Such a statement holds in an algebra, denoted $\mathcal{A}, \sigma \models t \to t'$, exactly when $\mathcal{A}, \sigma \models t = t'$, i.e., when $[\![t]\!]^{\sigma}_{\mathcal{A}} = [\![t']\!]^{\sigma}_{\mathcal{A}}$. Moreover, under those assumptions an equational condition $u = v$ in a conditional equation can be checked by finding a common term t such that $u \to t$ and $v \to t$; the notation we will use in the inference rules and debugging trees studied in Section 3 for this situation is $u \downarrow v$. Also, the notation $t =_E t'$ means that the equation $t = t'$ can be deduced from E, equivalently, that $[t]_E = [t']_E$.

2.2. Maude functional modules

Maude functional modules [12, Chapter 4], introduced with syntax `fmod ... endfm`, are executable membership equational logic specifications and their semantics is given by the corresponding initial algebra in the class of algebras satisfying the specification.

In a functional module we can declare sorts (by means of the keyword `sorts`); *subsort* relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`). Conditions, in addition to memberships and equations, can also be *matching equations* $t := t'$, whose mathematical meaning is the same as that of an ordinary equation $t = t'$ but that operationally are solved by matching the righthand side t' against the pattern t in the lefthand side, thus instantiating possibly new variables in t.

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. Kinds are *not* declared explicitly and correspond to the connected components of the subsort relation. The kind corresponding to a sort `s` is denoted `[s]`. For example, if we have sorts `Nat` for natural numbers and `NzNat` for nonzero natural numbers with a subsort `NzNat < Nat`, then `[NzNat] = [Nat]`.

An operator declaration like

```
op _div_ : Nat NzNat -> Nat .
```

is logically understood as a declaration at the kind level

```
op _div_ : [Nat] [Nat] -> [Nat] .
```

together with the conditional membership axiom

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

A subsort declaration `NzNat < Nat` is logically understood as the conditional membership axiom

```
cmb N : Nat if N : NzNat .
```

2.3. Rewriting logic

Rewriting logic extends equational logic by introducing the notion of *rewrites* corresponding to transitions between states; that is, while equations are interpreted as equalities and therefore they are symmetric, rewrites denote changes which can be irreversible.

A rewriting logic specification, or *rewrite theory*, has the form $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational specification and R is a set of *rules* as described below. From this definition, one can see that rewriting logic is built on top of equational logic, so that rewriting logic is parameterized with respect to the version of the underlying equational logic; in our case, Maude uses membership equational logic, as described in the previous sections. A rule q in R has the general conditional form[2]

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

[2] There is no need for the condition to list equations first, then memberships, and then rewrites; this is just a notational abbreviation, since they can be listed in any order.

where q is the rule label, the head is a rewrite and the conditions can be equations, memberships, and rewrites; both sides of a rewrite must have the same kind. From these rewrite rules, one can deduce rewrites of the form $t \Rightarrow t'$ by means of general deduction rules introduced in [21] (see also [4]).

Models of rewrite theories are called $\mathcal{R}$-*systems* in [21]. Such systems are defined as categories that possess a (Σ, E)-algebra structure, together with a natural transformation for each rule in the set R. More intuitively, the idea is that we have a (Σ, E)-algebra, as described in Section 2.1, with transitions between the elements in each set A_k; moreover, these transitions must satisfy several additional requirements, including that there are identity transitions for each element, that transitions can be sequentially composed, that the operations in the signature Σ are also appropriately defined for the transitions, and that we have enough transitions corresponding to the rules in R. The rewriting logic deduction rules introduced in [21] are sound and complete with respect to this notion of model. Moreover, they can be used to build initial models. Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, the initial model $\mathcal{T}_{\Sigma/E,R}$ for $\mathcal{R}$ has an underlying (Σ, E)-algebra $\mathcal{T}_{\Sigma/E}$ whose elements are equivalence classes $[t]_E$ of ground Σ-terms modulo E, and there is a transition from $[t]_E$ to $[t']_E$ when there exist terms t_1 and t_2 such that $t =_E t_1 \rightarrow^*_R t_2 =_E t'$, where $t_1 \rightarrow^*_R t_2$ means that the term t_1 can be rewritten into t_2 in zero or more rewrite steps applying rules in R, also denoted $[t]_E \rightarrow^*_{R/E} [t']_E$ when rewriting is considered on equivalence classes [21,15].

However, for our purposes in this paper, we are interested in a subclass of rewriting logic models [21] that we call *term models*, where the syntactic structure of terms is kept and associated notions such as variables, substitutions, and term rewriting make sense. These models will be used in the next section to represent the *intended interpretation* that the user had in mind while writing a specification. Since we want to find the discrepancies between the intended model and the initial model of the specification as written, we need to consider the relationship between a specification defined by a set of equations E and a set of rules R, and a model defined by possibly different sets of equations E' and of rules R'; in particular, when $E' = E$ and $R' = R$, the term model coincides with the initial model built in [21].

Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, with Σ a signature, E a set of equations, and R a set of rules, a Σ-term model has an underlying (Σ, E')-algebra whose elements are equivalence classes $[t]_{E'}$ of ground Σ-terms modulo some set of equations and memberships E' (which may be different from E), and there is a transition from $[t]_{E'}$ to $[t']_{E'}$ when $[t]_{E'} \rightarrow^*_{R'/E'} [t']_{E'}$, where rewriting is considered on equivalence classes [21,15]. The set of rules R' may also be different from R, that is, the term model is $\mathcal{T}_{\Sigma/E',R'}$ for some E' and R'. In such term models, the notion of valuation coincides with that of (ground) substitution. A term model $\mathcal{T}_{\Sigma/E',R'}$ satisfies, under a substitution θ,

- an equation $u = v$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u = v$, when $\theta(u) =_{E'} \theta(v)$, or equivalently, when $[\theta(u)]_{E'} = [\theta(v)]_{E'}$;
- a membership $u : s$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u : s$, when the Σ-term $\theta(u)$ has sort s according to the information in the signature Σ and the equations and memberships E';
- a rewrite $u \Rightarrow v$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u \Rightarrow v$, when there is a transition in $\mathcal{T}_{\Sigma/E',R'}$ from $[\theta(u)]_{E'}$ to $[\theta(v)]_{E'}$, that is, when $[\theta(u)]_{E'} \rightarrow^*_{R'/E'} [\theta(v)]_{E'}$.

Satisfaction is extended to conditional sentences as usual. A Σ-term model $\mathcal{T}_{\Sigma/E',R'}$ satisfies a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ when $\mathcal{T}_{\Sigma/E',R'}$ satisfies the equations and memberships in E and the rewrite rules in R in this sense. For example, this is obviously the case when $E \subseteq E'$ and $R \subseteq R'$; as mentioned above, when $E' = E$ and $R' = R$ the term model coincides with the initial model for $\mathcal{R}$.

2.4. Maude system modules

Maude system modules [12, Chapter 6], introduced with syntax `mod ... endm`, are executable rewrite theories and their semantics is given by the initial system in the class of systems corresponding to the rewrite theory. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`), whose conditions can be equations, matching equations, memberships, and rewrites.

The executability requirements for equations and memberships in a system module are the same as those of functional modules, namely, confluence, termination, and sort-decreasingness. With respect to rules, the satisfaction of all the conditions in a conditional rewrite rule is attempted sequentially from left to right, solving rewrite conditions by means of search; for this reason, we can have new variables in such conditions but they must become instantiated along this process of solving from left to right (see [12] for details). Furthermore, the strategy followed by Maude in rewriting with rules is to compute the normal form of a term with respect to the equations before applying a rule. This strategy is guaranteed not to miss any rewrites when the rules are *coherent* with respect to the equations [39,12]. In a way quite analogous to confluence, this coherence requirement means that, given a term t, for each rewrite of it using a rule in R to some term t', if u is the normal form of t with respect to the equations and memberships in E, then there is a rewrite of u with some rule in R to a term u' such that $u' =_E t'$.

The following section describes an example of a Maude system module with both equations and rules.

2.5. An example of system module: a maze

Given a maze, we want to obtain all the possible paths to the exit. First, we define the sorts `Pos`, `Pos?`, `List`, and `State`, that stand for positions in the labyrinth, incorrect positions (that we will use later to indicate that terms with this sort must

be rewritten to become a correct position) lists of positions, and the path traversed so far, respectively:

```
(mod MAZE is
  pr NAT .
  sorts Pos Pos? List State .
```

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

```
  subsorts Pos < Pos? List .
  op [_,_] : Nat Nat -> Pos [ctor] .
  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
```

Terms of sort `State` are lists enclosed by curly brackets, that is, `{_}` is an "encapsulation operator" that ensures that the whole state is used:

```
  op {_} : List -> State [ctor] .
```

The predicate `isSol` checks whether a list is a solution in a 8×8 labyrinth:

```
  vars X Y : Nat .
  vars P Q : Pos .
  var L : List .
  op isSol : List -> Bool .
  eq [is1] : isSol(L [8,8]) = true .
  eq [is2] : isSol(L) = false [owise] .
```

The next position is computed with rule `expand`, that extends the solution with a new position by rewriting `next(L)` to obtain a new position and then checking whether this list is correct with `isOk`. Note that the choice of the next position, that could be initially wrong, produces an implicit backtracking:

```
  crl [expand] : { L } => { L P } if next(L) => P /\ isOk(L P) .
```

The function `next`, that builds terms of the sort `Pos?`, is defined in a nondeterministic way with the rules:

```
  op next : List -> Pos? .
  rl [n1] : next(L [X,Y]) => [X, Y + 1] .
  rl [n2] : next(L [X,Y]) => [sd(X, 1), Y] .
  rl [n3] : next(L [X,Y]) => [X, sd(Y, 1)] .
```

where `sd` denotes symmetric difference on natural numbers.

`isOk(L P)` checks that the position `P` is within the limits of the labyrinth, not repeated in `L`, and not part of the wall by using an auxiliary function `contains`:

```
  op isOk : List -> Bool .
  eq isOk(L [X,Y]) = X >= 1 and Y >= 1 and X <= 8 and Y <= 8
           and not(contains(L, [X,Y])) and not(contains(wall, [X,Y])) .
  op contains : List Pos -> Bool .
  eq [c1] : contains(nil, P) = false .
  eq [c2] : contains(Q L, P) = if P == Q then true else contains(L, P) fi .
```

Finally, we define the `wall` of the labyrinth as a list of positions:

```
  op wall : -> List .
  eq wall =       [2,1]       [4,1]
                  [2,2] [3,2]             [6,2] [7,2]
                  [2,3]       [4,3] [5,3] [6,3] [7,3]

            [1,5] [2,5] [3,5] [4,5] [5,5] [6,5]       [8,5]
                                          [6,6]       [8,6]
                                          [6,7]
                                          [6,8] [7,8]       .
endm)
```

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

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

2.6. Assumptions

Since we are debugging Maude modules, they are expected to satisfy the appropriate executability requirements indicated in the previous sections. Namely, the specifications in functional modules have to be terminating, confluent, sort decreasing and, given an equation $t_1 = t_2$ *if* $C_1 \wedge \cdots \wedge C_n$, all the variables occurring in t_2 and $C_1 \ldots C_n$ must appear in t_1 or become instantiated by matching [12, Section 4.6]. While the equational part of system modules has to fulfill these requirements, rewrite rules must be coherent with respect to the equations and, given a rule $t_1 \Rightarrow t_2$ *if* $C_1 \wedge \cdots \wedge C_n$, the variables occurring in t_2 and $C_1 \ldots C_n$ must appear in t_1 or become instantiated in matching or rewriting conditions [12, Section 6.3].

One interesting feature of our tool is that the user can trust some statements, by means of labels applied to the suspicious statements. This means that the unlabeled statements are assumed to be correct, and only their conditions will generate questions. In order to obtain a nonempty abbreviated proof tree, the user must have labeled some statements (all with different labels); otherwise, everything is assumed to be correct. In particular, the wrong statement must be labeled in order to be found. Likewise, when debugging missing answers, constructed terms (terms built only with constructors, indicated with the attribute `ctor`, and also known as data terms in other contexts) are considered to be in normal form, and some of these constructed terms can be pointed out as “final” (they cannot be further rewritten). Thus, this information has to be accurate in order to find the buggy node.

Although the user can introduce a module importing other modules, the debugging process takes place in the *flattened* module. However, the debugger allows the user to trust a whole imported module.

Navigation of the debugging tree takes place by asking questions to an external oracle, which in our case is either the user or another module introduced by the user. In both cases the answers are assumed to be correct. If either the module is not really correct or the user provides an incorrect answer, the result is unpredictable. Notice that the information provided by the correct module need not be complete, in the sense that some functions can be only partially defined. In the same way, it is not required to use the same signature in the correct and the debugged modules. If the correct module cannot help in answering a question, the user may have to answer it.

Finally, all the information in the signature (sorts, subsorts, operators, and equational attributes such as assoc, comm, etc.) is supposed to be correct and will not be considered during the debugging process.

3. A calculus for debugging

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

3.1. A calculus for wrong answers

We show here a calculus to deduce judgments for reductions $e \to e'$, memberships $e : s$, and rewrites $e \Rightarrow e'$. The inference rules for this calculus, shown in Fig. 1, are an adaptation of the rules presented in [2,22] for membership equational logic and in [21,4] for rewriting logic. Remember that the notation $\theta(u_i) \downarrow \theta(u'_i)$ is an abbreviation of $\exists t_i.\theta(u_i) \to t_i \wedge \theta(u'_i) \to t_i$. As usual, we represent deductions in the calculus as *proof trees*, where the premises are the child nodes of the conclusion at each inference step. We assume that the inference labels $\mathsf{Rep}_\Rightarrow$, $\mathsf{Rep}_\to$, and Mb decorating the inference steps contain information about the particular rewrite rule, equation, and membership axiom, respectively, applied during the inference. This information will be used by the debugger in order to present to the user the incorrect fragment of code causing the error.

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

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

Figs. 2 and 3 depict the proof tree associated to this reduction, where `c` stands for `contains`, `t` for `true`, `f` for `false`, *rhs* for `1 >= 1 and 2 >= 1 and 1 <= 8 and 2 <= 8 and not(c([1,1],[1,2])) and not(c(wall,[1,2]))`, t_1 for `if [1,1] == [1,2] then t else c(nil,[1,2]) fi`, t_2 for `if f then t else c(nil,[1,2]) fi`, and each $\bigtriangledown$ abbreviates a computation not shown here. In order to obtain the result we use the transitivity inference rule, whose left premise applies the replacement rule with the equation for `isOk`, obtaining the term *rhs*, that will be further reduced in the right premise to obtain `t` by means of another transitivity step. The left child of this last node reduces all the subterms in *rhs* to `t`, while the right one just applies the usual equations for conjunctions to obtain the final result. While the first reductions in the premises of the node (•) correspond to arithmetic computations and will not been shown here, the last two are more complex. Fig. 3 describes the tree $\bigtriangledown^{1}$, that proves how one of the subterms using equations defined by the user is reduced to `t`, while the tree on its right is very similar and will not be studied in depth. The tree $\bigtriangledown^{1}$ reduces in its left child the inner subterm to `f` by traversing the list of positions (in this case the only element in the list is `[1,1]`), reducing

(**Reflexivity**)

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

(**Transitivity**)

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

(**Congruence**)

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

(**Replacement**)

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

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

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

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

(**Membership**)

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

Fig. 1. Semantic calculus for Maude modules.

$$\dfrac{\dfrac{}{\texttt{isOk([1,1][1,2])} \rightarrow \mathit{rhs}}\ \mathsf{Rep}_{\rightarrow} \qquad \dfrac{\dfrac{\dfrac{\triangledown \quad \triangledown}{\texttt{1 >= 1} \rightarrow \texttt{t}}\ \mathsf{Tr} \quad \ldots \quad \dfrac{\triangledown \quad \triangledown}{\texttt{2 <= 8} \rightarrow \texttt{t}}\ \mathsf{Tr} \quad \triangledown_1 \quad \triangledown}{(\bullet)\ \mathit{rhs} \rightarrow \texttt{t and ... and t}}\ \mathsf{Cong} \qquad \dfrac{\triangledown \quad \triangledown}{\texttt{t and ... and t} \rightarrow \texttt{t}}\ \mathsf{Tr}}{\mathit{rhs} \rightarrow \texttt{t}}\ \mathsf{Tr}}{\texttt{isOk([1,1][1,2])} \rightarrow \texttt{t}}\ \mathsf{Tr}$$

Fig. 2. Tree for the reduction of `isOk([1,1][1,2])`.

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

Fig. 3. Tree $\triangledown_1$ for `not(c([1,1],[1,2]))`.

the `if_then_else_fi` term in t_1 and then applying the equation for the empty list `nil`. Then, the right child of the root applies the predefined equation for `not` to obtain the final result.

In our debugging framework we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. This intended interpretation is a Σ-term model corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. Therefore the user expects that $\mathcal{I} \models e \Rightarrow e'$, $\mathcal{I} \models e \rightarrow e'$, and $\mathcal{I} \models e : s$ for each rewrite $e \Rightarrow e'$, reduction $e \rightarrow e'$, and membership $e : s$ computed w.r.t. the specification $\mathcal{R}$. As a term model, $\mathcal{I}$ must satisfy the following proposition:

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

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

3.2. A calculus for missing answers

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

Definition 1. A condition $\mathcal{C} \equiv C_1 \wedge \cdots \wedge C_n$ is *admissible* if, for $1 \leq i \leq n$,

- C_i is an equation $u_i = u_i'$ or a membership $u_i : s$ and
$$vars(C_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j), \text{ or}$$
- C_i is a matching condition $u_i := u_i'$, u_i is a pattern and
$$vars(u_i') \subseteq \bigcup_{j=1}^{i-1} vars(C_j), \text{ or}$$
- C_i is a rewrite condition $u_i \Rightarrow u_i'$, u_i' is a pattern and
$$vars(u_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j).$$

Definition 2. A condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$, where $\circledast$ denotes a special variable not occurring in the rest of the condition, is *admissible* if $P := t \wedge C_1 \wedge \cdots \wedge C_n$ is admissible for t any ground term.

Definition 3. A *kind-substitution*, denoted by κ, is a mapping from variables to terms of the form $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ such that $\forall_{1 \leq i \leq n} . kind(v_i) = kind(t_i)$, that is, each variable has the same kind as the associated term.

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

Definition 5. Given an atomic condition C, we say that a substitution θ is *admissible for C* if

- C is an equation $u = u'$ or a membership $u : s$ and $vars(C) \subseteq dom(\theta)$, or
- C is a matching condition $u := u'$ and $vars(u') \subseteq dom(\theta)$, or
- C is a rewrite condition $u \Rightarrow u'$ and $vars(u) \subseteq dom(\theta)$.

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

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

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

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

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

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

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

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

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

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

Fig. 4. Calculus for substitutions.

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

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(l) \rightarrow_{red} \theta(r)} \text{ Rdc}_1 \text{ if } l \rightarrow r \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \in E$$

$$\frac{t \rightarrow_{norm} t'}{f(t_1, \ldots, t, \ldots, t_n) \rightarrow_{red} f(t_1, \ldots, t', \ldots, t_n)} \text{ Rdc}_2 \text{ if } t \not\equiv_A t'$$

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

$$\frac{t \rightarrow_{red} t_1 \quad t_1 \rightarrow_{norm} t'}{t \rightarrow_{norm} t'} \text{ NTr}$$

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

Fig. 5. Calculus for normal forms and least sorts.

returned; otherwise (at least one of the terms has an incorrect sort), the kind-substitution is not a substitution and the empty set is returned.[4]

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

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

  that is, Θ is the set of substitutions that fulfill the atomic condition C and extend θ by binding the new variables appearing in C.
- Given a set of admissible substitutions Θ for an atomic condition C, $\mathcal{T}' \models \langle C, \Theta\rangle \rightsquigarrow \Theta'$ when

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

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

[4] Do not confuse, in the judgments inferring sets of substitutions, the empty set of substitutions $\emptyset$, which indicates that no substitutions fulfill the condition, with the set containing the empty substitution $\{\emptyset\}$, which indicates that the condition is fulfilled and the condition is ground.

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

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

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

Fig. 6. Calculus for solutions.

$$\frac{fulfilled(\mathcal{C}, t) \ \ t \Rightarrow_1 \{t_1, \ldots, t_k\} \ \ t_1 \rightsquigarrow_n^{\mathcal{C}} S_1 \ \ldots \ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^k S_i \ \cup \ \{t\}} \text{ Tr}_1$$

$$\frac{fails(\mathcal{C}, t) \ \ t \Rightarrow_1 \{t_1, \ldots, t_k\} \ \ t_1 \rightsquigarrow_n^{\mathcal{C}} S_1 \ \ldots \ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^k S_i} \text{ Tr}_2$$

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

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

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

$$\frac{t \rightarrow_{norm} t_1 \quad t_1 \rightsquigarrow_n^{\mathcal{C}} \{t_2\} \cup S \quad t_2 \rightarrow_{norm} t'}{t \rightsquigarrow_n^{\mathcal{C}} \{t'\} \cup S} \text{ Red}_1$$

Fig. 7. Calculus for missing answers.

- $\mathcal{T}' \models disabled(a, t)$ when the equation or membership a cannot be applied to t at the top.
- $\mathcal{T}' \models t \rightarrow_{red} t'$ when either $\mathcal{T}' \models t \rightarrow^1_{E'} t'$ or $\mathcal{T}' \models t_i \rightarrow^!_{E'} t'_i$, with $t_i \neq t'_i$, for some subterm t_i of t such that $t' = t[t_i \mapsto t'_i]$, that is, the term t is either reduced one step at the top or reduced by substituting a subterm by its normal form.
- $\mathcal{T}' \models t \rightarrow_{norm} t'$ when $\mathcal{T}' \models t \rightarrow^!_{E'} t'$, that is, t' is in normal form with respect to the equations E'.
- Given an admissible condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$, $\mathcal{T}' \models fulfilled(\mathcal{C}, t)$ when there exists a substitution θ such that $\mathcal{T}', \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ holds when $\circledast$ is substituted by t.
- Given an admissible condition $\mathcal{C}$ as before, $\mathcal{T}' \models fails(\mathcal{C}, t)$ when there exists *no* substitution θ such that $\mathcal{T}', \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ does not hold when $\circledast$ is substituted by t.
- $\mathcal{T}' \models t :_{ls} s$ when $\mathcal{T}' \models t : s$ and moreover s is the least sort with this property (with respect to the ordering on sorts obtained from the signature Σ and the equations and memberships E' defining the Σ-term model $\mathcal{T}'$).
- $\mathcal{T}' \models t \Rightarrow^{top} S$ when $S = \{t' \mid t \rightarrow^{top}_{R'} t'\}$, that is, the set S is formed by all the reachable terms from t by exactly one rewrite *at the top* with the rules R' defining $\mathcal{T}'$. Moreover, equality in S is modulo E', i.e., we are implicitly working with equivalence classes of ground terms modulo E'.
- $\mathcal{T}' \models t \Rightarrow^q S$ when $S = \{t' \mid t \rightarrow^{top}_{\{q\}} t'\}$, that is, the set S is the complete set of reachable terms (modulo E') obtained from t with one application of the rule $q \in R'$ at the top.
- $\mathcal{T}' \models t \Rightarrow_1 S$ when $S = \{t' \mid t \rightarrow^1_{R'} t'\}$, that is, the set S is constituted by all the reachable terms (modulo E') from t in exactly one step, where the rewrite step can take place anywhere in t.
- $\mathcal{T}' \models t \rightsquigarrow^{\mathcal{C}}_n S$ when $S = \{t' \mid t \rightarrow^{\leq n}_{R'} t' \text{ and } \mathcal{T}' \models fulfilled(\mathcal{C}, t')\}$, that is, S is the set of all the terms (modulo E') that satisfy the admissible condition $\mathcal{C}$ and are reachable from t in at most n steps.
- $\mathcal{T}' \models t \rightsquigarrow+^{\mathcal{C}}_n S$ as before, but with reachability from t in at least one step and in at most n steps.
- $\mathcal{T}' \models t \rightsquigarrow!^{\mathcal{C}}_n S$ when $S = \{t' \mid t \rightarrow^{\leq n}_{R'} t' \text{ and } \mathcal{T}' \models fulfilled(\mathcal{C}, t') \text{ and } t' \not\rightarrow_{R'}\}$, that is, now the terms (modulo E') in S are *final*, meaning that they cannot be further rewritten.

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

- Rule PatC computes all the possible substitutions that extend θ and satisfy the matching of the term t_2 with the pattern t_1 by first computing the normal form t' of t_2, obtaining then all the possible kind-substitutions κ that make t' and $\theta(t_1)$ equal modulo axioms (indicated by $\equiv_A$), and finally checking that the terms assigned to each variable in the kind-substitutions have the appropriate sort with $adequateSorts(\kappa)$. The union of the set of substitutions thus obtained constitutes the set of substitutions that satisfy the matching.
- Rule AS_1 checks whether the terms of the kind-substitution have the appropriate sort to match the variables. In this case the kind-substitution is a substitution and it is returned.
- Rule AS_2 indicates that, if any of the terms in the kind-substitution has a sort bigger than the required one, then it is not a substitution and thus the empty set of substitutions is returned.
- Rule MbC_1 returns the current substitution if a membership condition holds.
- Rule MbC_2 is used when the membership condition is not satisfied. It checks that the least sort of the term is not less than or equal to the required one, and thus the substitution does not satisfy the condition and the empty set is returned.
- Rule EqC_1 returns the current substitution when an equality condition holds, that is, when the two terms can be joined.
- Rule EqC_2 checks that an equality condition fails by obtaining the normal forms of both terms and then examining that they are different.
- Rewrite conditions are handled by rule RlC. This rule extends the set of substitutions (where we use the juxtaposition of substitutions to express composition) by computing all the reachable terms that satisfy the pattern (using the relation $t \rightsquigarrow_n^{\mathcal{C}} S$ explained below) and then using these terms to obtain the new substitutions.
- Finally, rule SubsCond computes the extensions of a set of admissible substitutions for C $\{\theta_1, \ldots, \theta_n\}$ by using the rules above with each of them.

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

- Rule Dsb indicates when an equation or membership a cannot be applied to a term t. It checks that there are no substitutions that satisfy the matching of the term with the lefthand side of the statement and that fulfill its condition. Note that we check the conditions from left to right, following the same order as Maude and making all the substitutions admissible.
- Rule Rdc_1 reduces a term by applying one equation when it checks that the conditions can be satisfied, where the matching conditions are included in the equality conditions. While in the previous rule we made explicit the evaluation from left to right of the condition to show that finally the set of substitutions fulfilling it was empty, in this case we only need one substitution to fulfill the condition and the order is unimportant.
- Rule Rdc_2 reduces a term by reducing a subterm to normal form (checking in the side condition that it is not already in normal form).
- Rule Norm states that the term is in normal form by checking that no equations can be applied at the top considering the variables at the kind level (which is indicated by $\ll_K^{top}$) and that all its subterms are already in normal form.
- Rule NTr describes the transitivity for the reduction to normal form. It reduces the term with the relation $\rightarrow_{red}$ and the term thus obtained then is reduced to normal form by using again $\rightarrow_{norm}$.
- Rule Ls computes the least sort of the term t. It computes a sort for its normal form (that has the least sort of the terms in the equivalence class) and then checks that memberships deducing smaller sorts cannot be applied.

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

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

- Rule Rf_1 indicates that when only zero steps can be used and the current term fulfills the condition, the set of reachable terms consists only of this term.
- Rule Rf_2 complements Rf_1 by defining the empty set as result when the condition does not hold.
- Rule Fulfill checks whether a term satisfies a condition. The premises of this rule check that all the atomic conditions hold, taking into account that it starts with a matching condition with a hole that must be filled with the current

term and thus proved with the premise $\theta(P) \downarrow t$ (the rest of the matching conditions are included in the equality conditions). Note that when the condition is satisfied we do not need to check *all* the substitutions, but only to verify that there exists *one* substitution that makes the condition true.
- To check that a term does not satisfy a condition, it is not enough to check that there exists a substitution that makes it fail; we must make sure that there is no substitution that makes it true. This is indicated by rule Fail, which uses the rules shown in Fig. 4 to prove that the set of substitutions that satisfy the condition (where the first set of substitutions is obtained from the first matching condition filling the hole with the current term) is empty. Note that, while rule Fulfill provides the positive information indicating that a condition is fulfilled, this one provides the negative information, proving that the condition does not hold.

Now we introduce in Fig. 7 the rules defining the relation $t \rightsquigarrow_n^{\mathcal{C}} S$ when the bound n is greater than 0, which can be understood as searches in *zero or more* steps:

- Rules Tr_1 and Tr_2 show the behavior of the calculus when at least one step can be used. First, we check whether the condition holds (rule Tr_1) or not (rule Tr_2) for the current term, in order to introduce it in the result set. Then, we obtain all the terms reachable in one step with the relation $\Rightarrow_1$, and finally we compute the reachable solutions from these terms constrained by the same condition and the bound decreased by one step. The union of the sets obtained in this way and the initial term, if needed, corresponds to the final result set.
- Rule Stp shows how the set for one step is computed. The result set is the union of the terms obtained by applying each rule *at the top* (calculated with $t \Rightarrow^{top} S$) and the terms obtained by rewriting the arguments of the term one step. This rule can be straightforwardly adapted to the more general case in which the operator f has some *frozen* arguments (i.e., that cannot be rewritten); the implementation of the debugger makes use of this more general rule.
- How to obtain the terms by rewriting at the top is explained by rule Top, which specifies that the result set is the union of the sets obtained with all the possible applications of each rule in the program. We have restricted these rules to those whose lefthand side, with the variables considered at the kind level, matches the term, represented with notation $q \ll_K^{top} t$, where q is the label of the rule and t the current term.
- Rule Rl uses the rules in Fig. 4 to compute the set of terms obtained with the application of a single rule. First, the set of substitutions obtained from matching with the lefthand side of the rule is computed, and then it is used to find the set of substitutions that satisfy the condition. This final set is used to instantiate the righthand side of the rule to obtain the set of reachable terms. The kind of information provided by this rule corresponds to the information provided by the substitutions; if the empty set of substitutions is obtained (negative information) then the rule computes the empty set of terms, which also corresponds with negative information proving that no terms can be obtained with this rewrite rule; analogously when the set of substitutions is nonempty (positive information). This information is propagated through the rest of the inference rules justifying why some terms are reachable while others are not.
- Finally, rule Red_1 reduces the reachable terms in order to obtain their normal forms. We use this rule to reproduce Maude behavior, first the normal form of the term is computed and then the rules are applied.

This calculus is correct in the sense that the derived judgments with respect to the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ coincide with the ones satisfied by the corresponding initial model $\mathcal{T}_{\Sigma/E,R}$, i.e., for any judgment φ, φ is derivable in the calculus if and only if $\mathcal{T}_{\Sigma/E,R} \models \varphi$. Detailed proofs of all the results are available in Appendix A.

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

Once these rules are defined, we can build the tree corresponding to the search result shown in Section 2.5 for the maze example. We recall that we have defined a system to search a path out of a labyrinth but, given a concrete labyrinth with an exit, the program is unable to find it:

```
Maude> (search {[1,1]} =>* {L:List} s.t. isSol(L:List) .)
search in MAZE :{[1,1]} =>* {L:List}.

No solution.
```

First of all, we have to use a concrete bound to build the tree. It must suffice to compute all the reachable terms, and in this case the least of these values is 4. We have depicted the tree in Fig. 8, where we have abbreviated the equational condition `{L:List}` $:= \circledast \wedge$ `isSol(L:List) = true` by $\mathcal{C}$ and `isSol(L:List) = true` by `isSol(L)`. The leftmost tree justifies that the search condition does not hold for the initial term (this is the reason why Tr_2 has been used instead of Tr_1) and thus it is not a solution. Note that first the substitutions from the matching with the pattern are obtained (`L` $\mapsto$ `[1,1]` in this case), and then these substitutions are used to instantiate the rest of the condition, that for this term does not hold, which is proved by $\bigtriangledown_2$. The next tree shows the set of reachable terms in one step (the tree $\bigtriangledown_3$, explained below, computes the terms obtained by rewrites at the top, while the tree on its right shows that the subterms cannot be further rewritten)

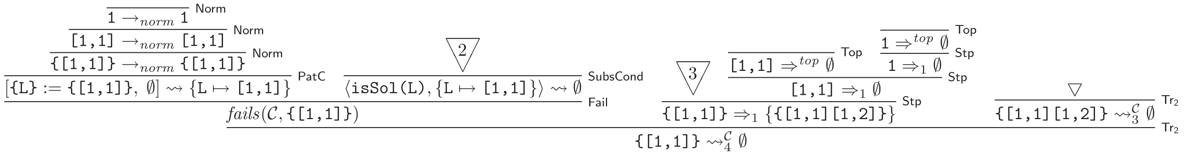

Fig. 8. Tree for the maze example.

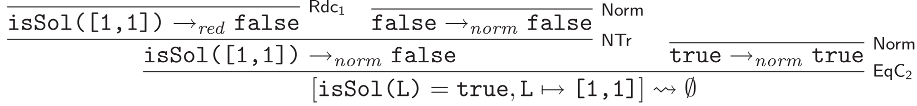

Fig. 9. Tree $\bigtriangledown^2$ for the search condition.

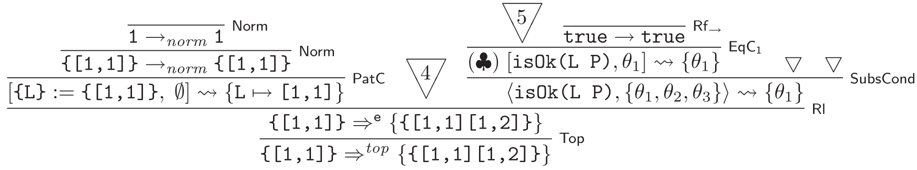

Fig. 10. Tree $\bigtriangledown^3$ for the applications at the top.

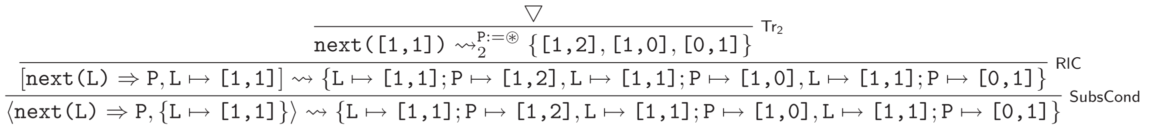

Fig. 11. Tree $\bigtriangledown^4$ for the first condition of `expand`.

and finally the rightmost tree, that has a similar structure to this one and will not be studied in depth, continues the search with the bound decreased in one step.

The tree $\bigtriangledown^2$ shows why the current list is not a solution (i.e., the tree provides the negative information proving that this fragment of the condition does not hold). The reason is that the function `isSol` is reduced to `false`, when we needed it to be reduced to `true`.

The tree labeled with $\bigtriangledown^3$ is sketched in Fig. 10. In this tree the applications of all the rules whose lefthand side matches the current term (`{[1,1]}`) are tried. In this case only the rule `expand` (abbreviated by `e`) can be used, and it generates a list with the new position `[1,2]`; the tree $\bigtriangledown^4$ is used to justify that the first condition of `expand` holds and extends the set of substitutions that fulfill the condition thus far to the set $\{\theta_1, \theta_2, \theta_3\}$, where $\theta_1 \equiv$ `L` $\mapsto$ `[1,1]; P` $\mapsto$ `[1,2]`, $\theta_2 \equiv$ `L` $\mapsto$ `[1,1]; P` $\mapsto$ `[1,0]`, and $\theta_3 \equiv$ `L` $\mapsto$ `[1,1]; P` $\mapsto$ `[0,1]`. The substitution θ_1 also fulfills the next condition, `isOk(L P)`, which is proved with the rule EqC_1 in (♣) (where $\bigtriangledown^5$ is the proof tree shown in Fig. 2, proving that the condition holds), while the substitutions θ_2 and θ_3 fail; the trees $\bigtriangledown$ proving it are analogous to the one shown in Fig. 9. This substitution θ_1 is thus the only one inferred in the root of the tree, where the node (♣) provides the positive information proving why the substitution is obtained and its siblings ($\bigtriangledown$) the negative information proving why the other substitutions are not in the set.

The tree $\bigtriangledown^4$, shown in Fig. 11, is in charge of inferring the set of substitutions obtained when checking the first condition of the rule `expand`, namely `next(L) => P`. The condition is instantiated with the substitution obtained from matching the term with the lefthand side of the rule (in this case `L` $\mapsto$ `[1,1]`) and, since it is a rewrite condition, the set of reachable terms is used to extend this substitution, obtaining a set with three different substitutions (that we previously abbreviated as θ_1, θ_2, and θ_3).

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

- Rules Rf_3 and Rf_4 are applied when the set of reachable terms in one step is empty (that is, when the term is final). They check whether the term, in addition to being final, fulfills the condition in order to insert it in the result set when appropriate.
- Rule Rf_5 specifies that, if the term is not final but no more steps are allowed, then the set of reachable final terms is empty.
- Rule Tr_3 shows the transitivity for this kind of search. Since the term is not final, it is not necessary to check whether it fulfills the condition.

$$\frac{fulfilled(\mathcal{C}, t) \quad t \Rightarrow_1 \emptyset}{t \leadsto !^{\mathcal{C}}_n \{t\}} \; \mathsf{Rf}_3$$

$$\frac{fails(\mathcal{C}, t) \quad t \Rightarrow_1 \emptyset}{t \leadsto !^{\mathcal{C}}_n \emptyset} \; \mathsf{Rf}_4$$

$$\frac{t \Rightarrow_1 S}{t \leadsto !^{\mathcal{C}}_0 \emptyset} \; \mathsf{Rf}_5 \quad S \neq \emptyset$$

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

$$\frac{t \to t_1 \quad t_1 \leadsto !^{\mathcal{C}}_n \{t_2\} \cup S \quad t_2 \to t'}{t \leadsto !^{\mathcal{C}}_n \{t'\} \cup S} \; \mathsf{Red}_2$$

$$\frac{}{t \leadsto +^{\mathcal{C}}_0 \emptyset} \; \mathsf{Rf}_6$$

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

Fig. 12. Calculus for final and one or more steps searches.

- Rule Red_2 reduces the reachable final terms in order to obtain their normal forms.
- If only zero steps are available in searches where at least one is required, the empty set is obtained, which is indicated in rule Rf_6.
- When at least one step can be used we apply rule Tr_4, that indicates that one step is used, and then the relation for zero or more steps is used with the results in order to obtain the final solutions.

The correctness of these inference rules with respect to the initial model $\mathcal{T}_{\Sigma/E,R}$ is proved in the following theorem:

Theorem 2. *The calculus of Fig. 12 is correct.*

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

Proposition 2. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, C an atomic condition, θ an admissible substitution, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If adequateSorts$(\kappa) \leadsto \Theta$, $[C, \theta] \leadsto \Theta$, or $\langle C, \Theta\rangle \leadsto \Theta'$ can be deduced using the rules from Fig. 4 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models$ adequateSorts$(\kappa) \leadsto \Theta$, $\mathcal{T}_{\Sigma/E',R'} \models [C, \theta] \leadsto \Theta$, and $\mathcal{T}_{\Sigma/E',R'} \models \langle C, \Theta\rangle \leadsto \Theta'$, respectively.*

Proposition 3. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and φ a judgment deduced with the inference rules* Dsb, Rdc_2, *or* NTr *from Fig. 5 from premises that hold in $\mathcal{T}_{\Sigma/E',R'}$. Then also $\mathcal{T}_{\Sigma/E',R'} \models \varphi$.*

Proposition 4. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If $t \leadsto^{\mathcal{C}}_0 S$ can be deduced using rules* Rf_1 *or* Rf_2 *from Fig. 6 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \leadsto^{\mathcal{C}}_0 S$.*

Proposition 5. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, n a natural number, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If $t \leadsto^{\mathcal{C}}_n S$ or $t \Rightarrow_1 S$ can be deduced by means of the rules in Fig. 7 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \leadsto^{\mathcal{C}}_n S$ or $\mathcal{T}_{\Sigma/E',R'} \models t \Rightarrow_1 S$, respectively.*

Proposition 6. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, n a natural number, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If a statement $t \leadsto !^{\mathcal{C}}_n S$ or $t \leadsto +^{\mathcal{C}}_n S$ can be deduced by means of the rules in Fig. 12 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \leadsto !^{\mathcal{C}}_n S$ or $\mathcal{T}_{\Sigma/E',R'} \models t \leadsto +^{\mathcal{C}}_n S$, respectively.*

Observe that these soundness propositions cannot be extended to the Ls, Fulfill, Fail, Top, and Rl inference rules, where the soundness of the conclusion depends not only on the calculus but also on the specification, which could be wrong.

4. Debugging trees

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

4.1. Debugging with proof trees

As explained in the previous sections, we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. This intended interpretation is a Σ-term model corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. We will say that a judgment is *valid* when it holds in the intended interpretation $\mathcal{I}$, and *invalid* otherwise. Our goal is to find a buggy node (an invalid node with all its children correct) in any proof tree T rooted by the initial error symptom detected by the user. This could be done simply by asking the user questions about the validity of the nodes in the tree according to the following *top-down* strategy:

Input: A tree T with an invalid root.
Output: A buggy node in T.
Description: Consider the root N of T. There are two possibilities:

- If all the children of N are valid, then finish pointing out N as buggy.
- Otherwise, select the subtree rooted by any invalid child and recursively use the same strategy to find the buggy node.

Proving that this strategy is complete is straightforward by using induction on the height of T. As an easy consequence, the following result holds:

Proposition 7. *Let T be a proof tree with an invalid root. Then there exists a buggy node $N \in T$ such that all the ancestors of N are invalid.*

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

- Given a statement $A \Leftarrow C_1 \wedge \cdots \wedge C_n$ (where A is either an equation $l = r$, a membership $l : s$, or a rule $l \Rightarrow r$) and a substitution θ, the *statement instance* $\theta(A) \Leftarrow \theta(C_1) \wedge \cdots \wedge \theta(C_n)$ is *wrong* when all the atomic conditions $\theta(C_i)$ are valid in $\mathcal{I}$ but $\theta(A)$ is not.
- Given a rule $l \Rightarrow r \Leftarrow C_1 \wedge \cdots \wedge C_n$ and a term t, the rule has a *wrong instance* if the judgments $[l := t, \emptyset] \rightsquigarrow \Theta_0$, $[C_1, \Theta_0] \rightsquigarrow \Theta_1, \ldots, [C_n, \Theta_{n-1}] \rightsquigarrow \Theta_n$ are valid in $\mathcal{I}$ but the application of Θ_n to the righthand side does not provide all the results expected for this rule.
- Given a condition $l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ and a term t, if $[l := t, \emptyset] \rightsquigarrow \Theta_0, [C_1, \Theta_0] \rightsquigarrow \Theta_1, \ldots, [C_n, \Theta_{n-1}] \rightsquigarrow \emptyset$ are valid in $\mathcal{I}$ (meaning that the condition does not hold for t) but the user expected the condition to hold, then we have a *wrong search condition instance*.
- Given a condition $l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ and a term t, if there exists a substitution θ such that $\theta(l) \equiv_A t$ and all the atomic conditions $\theta(C_i)$ are valid in $\mathcal{I}$, but the condition is not expected to hold, then we also have a *wrong search condition instance*.
- A statement or condition is *wrong* when it admits a wrong instance.
- Given a term t, there is a *missing equation for t* if t is not expected to be in normal form and none of the equations in the specification are expected to be applied to it.
- A specification has a *missing equation* if there exists a term t such that there is a missing equation for t.
- Given a term t, there is a *missing membership for t* if t is an expected normal form such that the computed least sort of t is not the expected one and none of the membership axioms in the specification are expected to be applied to it.
- A specification has a *missing membership* if there exists a term t such that there is a missing membership for t.
- Given a term t, there is a *missing rule for t* if all the rules applied to t at the top lead to judgments $t \Rightarrow^{q_i} S_{q_i}$ valid in $\mathcal{I}$ but the union $\bigcup S_{q_i}$ does not contain all the reachable terms from t by using rewrites at the top.
- A specification has a *missing rule* if there exists a term t such that there is a missing rule for t.

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

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

Table 3
Errors detected by the proof trees.

$\mathsf{Rep}_{\rightarrow}$	Wrong equation
$\mathsf{Rep}_{\Rightarrow}$	Wrong rule
Mb	Wrong membership
Rdc_1	Wrong equation
Norm	Missing equation
Ls	Missing membership
Fulfill	Wrong search condition
Fail	Wrong search condition
Top	Missing rule
Rl	Wrong rule

1. *N corresponds to the consequence of an inference rule in the first column of Table 3.*
2. *The error associated to N can be obtained from the inference rule as shown in the second column of Table 3.*

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

4.2. Abbreviated proof trees

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

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

- Questions associated to nodes with reductions are improved (rules ($\mathbf{APT}_2$), ($\mathbf{APT}_3$), ($\mathbf{APT}_5$), ($\mathbf{APT}_6$), and ($\mathbf{APT}_7$)) by asking about normal forms instead of asking about intermediate states. For example, in rule ($\mathbf{APT}_2$) the error associated to $t_1 \rightarrow t_2$ is the one associated to $t_1 \rightarrow t'$, which is not included in the APT. We have chosen to introduce $t_1 \rightarrow t_2$ instead of simply $t_1 \rightarrow t'$ in the APT as a pragmatic way of simplifying the structure of the APTs, since t_2 is obtained from t' and hence likely simpler.
- The rule ($\mathbf{APT}_4$) deletes questions about rewrites *at the top* of a given term (that may be difficult to answer due to matching modulo) and associates the information of those nodes to questions related to the set of reachable terms in one step with rewrites in any position, that are in general easier to answer.
- It creates, with the variants of the rules ($\mathbf{APT}_8$) and ($\mathbf{APT}_9$), two different kinds of tree, one that contains judgments of rewrites with several steps and another that only contains rewrites in one step. The one-step debugging tree strictly follows the idea of keeping only nodes corresponding to relevant information. However, the many-steps debugging tree also keeps nodes corresponding to the *transitivity* inference rules. The user will choose which debugging tree (one-step or many-steps) will be used for the debugging session, taking into account that the many-steps debugging tree usually leads to shorter debugging sessions (in terms of the number of questions) but with likely more complicated questions. The number of questions is usually reduced because keeping the transitivity nodes for rewrites gives some parts of the debugging tree the shape of a balanced binary tree (each transitivity inference has two premises, i.e., two child subtrees), and this allows the debugger to efficiently use the divide and query navigation strategy. On the contrary, removing the transitivity inferences for rewrites (as rules ($\mathbf{APT}_8^o$) and ($\mathbf{APT}_9^o$) do) produces flattened trees where this strategy is no longer so efficient. On the other hand, in rewrites $t \Rightarrow t'$ and searches $t \rightsquigarrow_n^C S$ appearing as the conclusion of a transitivity inference rule, the judgment can be more complicated because it combines several

$$(\mathbf{APT}_1)\quad APT\left(\frac{T_1 \ldots T_n}{\varphi}\,\mathsf{R}_1\right) = \frac{APT'\left(\frac{T_1 \ldots T_n}{\varphi}\,\mathsf{R}_1\right)}{\varphi}$$

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

$$(\mathbf{APT}_3)\quad APT'\left(\frac{\frac{T_1\ \ldots\ T_n}{t \to t''}\,\mathsf{Rdc1}\quad T'}{t \to t'}\,\mathsf{NTr}\right) = \left\{\frac{APT'(T_1)\ \ldots\ APT'(T_n)\ APT'(T')}{t \to t'}\,\mathsf{Rdc1}\right\}$$

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

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

$$(\mathbf{APT}_6)\quad APT'\left(\frac{T \quad \frac{T_1 \ldots T_n}{\varphi'}\,\mathsf{R}_1 \quad T'}{\varphi}\,\mathsf{Red}_{\mathsf{i}}\right) = \left\{\frac{APT'(T)\ APT'(T_1)\ \ldots\ APT'(T_n)\ APT'(T)}{\varphi}\,\mathsf{R}_1\right\}$$

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

$$(\mathbf{APT}^o_8)\quad APT'\left(\frac{T_1 \quad T_2}{t_1 \Rightarrow t_2}\,\mathsf{Tr}_{\Rightarrow}\right) = APT'(T_1) \bigcup APT'(T_2)$$

$$(\mathbf{APT}^m_8)\quad APT'\left(\frac{T_1 \quad T_2}{t_1 \Rightarrow t_2}\,\mathsf{Tr}_{\Rightarrow}\right) = \left\{\frac{APT'(T_1)\ APT'(T_2)}{t_1 \Rightarrow t_2}\,\mathsf{Tr}_{\Rightarrow}\right\}$$

$$(\mathbf{APT}^o_9)\quad APT'\left(\frac{T_1 \ldots T_n}{\varphi}\,\mathsf{Tr}_{\mathsf{j}}\right) = APT'(T_1) \bigcup \ldots \bigcup APT'(T_n)$$

$$(\mathbf{APT}^m_9)\quad APT'\left(\frac{T_1 \ldots T_n}{\varphi}\,\mathsf{Tr}_{\mathsf{j}}\right) = \left\{\frac{APT'(T_1)\ \ldots\ APT'(T_n)}{\varphi}\,\mathsf{Tr}_{\mathsf{j}}\right\}$$

$$(\mathbf{APT}_{10})\quad APT'\left(\frac{T_1 \ldots T_n}{\varphi}\,\mathsf{R}_2\right) = \left\{\frac{APT'(T_1) \ldots APT'(T_n)}{\varphi}\,\mathsf{R}_2\right\}$$

$$(\mathbf{APT}_{11})\quad APT'\left(\frac{T_1 \ldots T_n}{\varphi}\,\mathsf{R}_1\right) = APT'(T_1) \bigcup \ldots \bigcup APT'(T_n)$$

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

$1 \le i \le 2 \quad 1 \le j \le 4 \quad \varphi, \varphi'$ any judgment

Fig. 13. Transforming rules for obtaining abbreviated proof trees.

inferences. The user must balance the pros and cons of each option, and choose the best one for each debugging session.

- The rule (**APT**$_{11}$) removes from the tree all the nodes which are not associated with relevant information, since the rule (**APT**$_{10}$) keeps the relevant information and the rules are applied in order. We remove, for example, nodes related to judgments about sets of substitutions, disabled statements, and rewrites with a concrete rule. Moreover, it removes trivial judgments, like the ones related to reflexivity or congruence, from the tree.
- Since the *APT* is built without computing the associated proof tree, it reduces the time and space needed to build the tree.

We can state the correctness and completeness of the debugging technique based on *APT*s:

Theorem 3. *Let T be a finite proof tree representing an inference in the calculus of Figs. 1, 4, 5, 6, 7, and 12 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root of T is invalid in $\mathcal{I}$. Then:*

- *APT(T) contains at least one buggy node (completeness).*
- *Any buggy node in APT(T) has an associated wrong statement, missing statement, or wrong condition in $\mathcal{R}$ according to Table 3 (correctness).*

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

The trees in Figs. 14–17 depict the (one-step) abbreviated proof tree for the maze example, where $\mathcal{C}$ stands for `{L:List}:=` ⊛ ∧ `isSol(L:List)`, P_1 for `[1,1]`, L_1 for `[1,1][1,2]`, L_2 for `[1,1][1,0]`, L_3 for `[1,1][0,1]`, `t` for `true`, `f` for false,

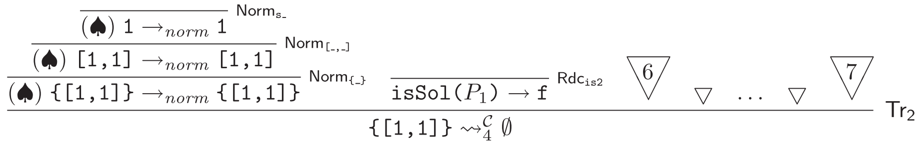

Fig. 14. Abbreviated proof tree for the maze example.

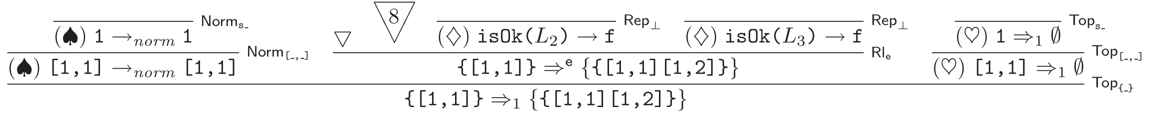

Fig. 15. Abbreviated tree $\overset{6}{\bigtriangledown}$.

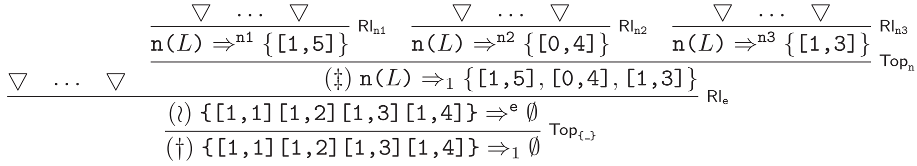

Fig. 16. Abbreviated tree $\overset{7}{\bigtriangledown}$.

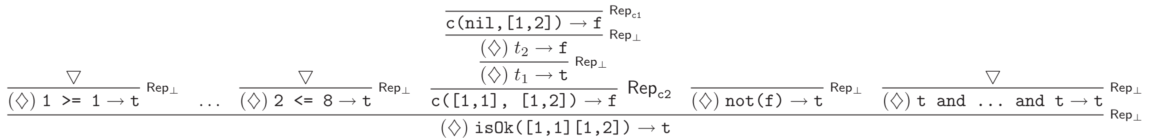

Fig. 17. Abbreviated proof tree $\overset{8}{\bigtriangledown}$.

`n` for `next`, `e` for `expand`, and *L* for `[1,1][1,2][1,3][1,4]`. We have also extended the information in the labels with the operator or statement associated to the inference. More concretely, the tree in Fig. 14 abbreviates the tree in Fig. 8; the first two premises in the abbreviated tree stand for the first premise in the proof tree (which includes the tree in Fig. 9), keeping only the nodes associated with relevant information according to Proposition 8: Norm, with the operator associated to the reduction, and Rdc_1, with the label of the associated equation. The tree $\overset{6}{\bigtriangledown}$, shown in Fig. 15, abbreviates the second premise of the tree in Fig. 8 as well as the trees in Figs. 10 and 11; it only keeps the nodes referring to normal forms, searches in one step, that are now associated to the rule Top, each of them referring to a different operator (the operator `s_` is the successor constructor for natural numbers), and the applications of rules (Rl) and equations ($\mathsf{Rep}_{\rightarrow}$). Note that the equation describing the behavior of `isOk` has not got any label, which is indicated with the symbol $\perp$; we will show below how the debugger deals with these nodes. The tree $\overset{7}{\bigtriangledown}$, presented in Fig. 16, shares these characteristics and only keeps nodes related to one-step searches and application of rules. The tree $\overset{8}{\bigtriangledown}$ abbreviates the proof tree for the reduction shown in Fig. 2, where the important result of the abbreviation is that all replacement inferences are related now to reductions to normal form, thus easing the questions that will be asked to the user.

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

- Statements can be trusted in several ways: non labeled statements, which include the predefined functions, are always trusted (i.e., the nodes marked with ($\diamondsuit$) in Figs. 15 and 17 will be discarded by the debugger); statements and modules can be trusted before starting the debugging process; and statements can also be trusted on the fly.
- A correct module can be given before starting a debugging session. By checking the correctness of the judgments against this module, correct nodes can be deleted from the tree.
- Constructed terms (that is, terms built only with constructors, defined by means of the `ctor` attribute) of certain sorts or built with some operators can be considered *final*, which indicates that they cannot be further rewritten. For example, we could consider terms of sorts `Nat` and `List` (and hence its subsort `Pos`) to be final and thus the nodes marked with ($\heartsuit$) in Fig. 15 would be removed from the tree.
- Moreover, we consider that constructed terms are in normal form and thus they are automatically removed from the tree. For example, the nodes marked with ($\spadesuit$) in Figs. 14 and 15 will be removed from the debugging tree.

Table 4
Available commands I.

Command	Effect	When
Trusting		
`(set debug select on .)`	Activates trusting	Before starting the debugging
`(set debug select off .)`	Deactivates trusting	Before starting the debugging
`(debug select LABELS .)`	Suspects of `LABELS`	Before starting the debugging
`(debug deselect LABELS .)`	Trusts `LABELS`	Before starting the debugging
`(debug include MODULES .)`	Suspects of `MODULES`	Before starting the debugging
`(debug exclude MODULES .)`	Trusts `MODULES`	Before starting the debugging
`(debug include eqs MODULES .)`	Suspects of the equations in `MODULES`	Before starting the debugging
`(debug exclude eqs MODULES .)`	Trusts the equations in `MODULES`	Before starting the debugging
`(debug include mbs MODULES .)`	Suspects of the memberships in `MODULES`	Before starting the debugging
`(debug exclude mbs MODULES .)`	Trusts the memberships in `MODULES`	Before starting the debugging
`(debug include rls MODULES .)`	Suspects of the rules in `MODULES`	Before starting the debugging
`(debug exclude rls MODULES .)`	Trusts the rules in `MODULES`	Before starting the debugging
`(correct module MODULE-NAME .)`	Sets the correct module	Before starting the debugging
`(set bound BOUND .)`	Sets the bound for the correct module	Before starting the debugging
`(delete correct module .)`	Deletes the correct module	Before starting the debugging
`(set final select on .)`	Activates "final" trusting	Before starting the debugging
`(set final select off .)`	Deactivates "final" trusting	Before starting the debugging
`(final select SORTS .)`	Sorts `SORTS` are final	Before starting the debugging
`(final deselect SORTS .)`	Sorts `SORTS` are not final	Before starting the debugging
Tree options		
`(one-step tree .)`	Selects the one-step tree for wrong rewrites	Before starting the debugging
`(many-steps tree .)`	Selects the many-steps tree for wrong rewrites	Before starting the debugging
`(one-step missing tree .)`	Selects the one-step tree for missing rewrites	Before starting the debugging
`(many-steps missing tree .)`	Selects the many-steps tree for missing rewrites	Before starting the debugging
`(solutions prioritized on .)`	Prioritizes questions about solutions	Before starting the debugging
`(solutions prioritized off .)`	Does not prioritize questions about solutions	Before starting the debugging
Strategies		
`(top-down strategy .)`	Switches to the top-down strategy	At any time
`(divide-query strategy .)`	Switches to the divide and query strategy	At any time

5. Using the debugger

We introduce in this section how to create and navigate the debugging tree.

5.1. Creating the debugging tree

We describe in this section how to start the debugging process, describing the commands that must be used before creating the debugging tree and the different commands to create it.

The debugger is initiated in Maude by loading the file `dd.maude` (available from http://maude.sip.ucm.es/debugging), which starts an input/output loop that allows the user to interact with the tool. Then, the user can enter Full Maude modules and commands, as well as commands for the debugger. Tables 4 and 5 present a summary of the commands explained below.

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

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

Once this mode is activated, the user can select and deselect statements by using [5]

```
(debug select LABELS .)
(debug deselect LABELS .)
```

where `LABELS` is a list of statement labels separated by spaces.

Moreover, all the labels in statements of a flattened module can be selected or deselected with the commands

```
(debug include MODULES .)
(debug exclude MODULES .)
```

where `MODULES` is a list of module names separated by spaces.

The selection mode can be switched off by using the command

```
(set debug select off .)
```

[5] Although these labels, as well as the set of labels from a module and the final sorts below, can be selected and deselected with the corresponding modes switched off, they will have effect only when the corresponding modes are activated.

Table 5
Available commands II.

Command	Effect	When
Debugging		
`(debug [in M :] INIT-T -> WRNG-T .)`	Starts the debugging for wrong reductions	At any time
`(debug [in M :] INIT-T : WRNG-S .)`	Starts the debugging for wrong sort inferences	At any time
`(debug [in M :] INIT-T =>* WRNG-T .)`	Starts the debugging for wrong rewrites	At any time
`(missing [in M :] INIT-T -> ERR-NF .)`	Starts the debugging for incomplete normal forms	At any time
`(missing [in M :] INIT-T : ERR-LST-S .)`	Starts the debugging for bigger than expected least sorts	At any time
`(missing [[dpt]] [in M :] T =>* P [s.t. C] .)`	Starts the debugging for incomplete sets in zero or more steps	At any time
`(missing [[dpt]] [in M :] T =>+ P [s.t. C] .)`	Starts the debugging for incomplete sets in one or more steps	At any time
`(missing [[dpt]] [in M :] T =>! P [s.t. C] .)`	Starts the debugging for incomplete sets of final terms	At any time
Answers		
`(yes .)`	The judgment is correct	D&Q strategy
`(N : yes .)`	The `N`th judgment is correct	TD strategy
`(all : yes .)`	All the judgments are correct	TD strategy
`(no .)`	The judgment is incorrect	D&Q strategy
`(N : no .)`	The `N`th judgment is incorrect	TD strategy
`(trust .)`	Trusts the statement associated with the current judgment	D&Q strategy
`(N : trust .)`	Trusts the statement associated with the `N`th judgment	TD strategy
`(I is wrong .)`	The `I`th element is not reachable	D&Q strategy
`(N : I is wrong .)`	The `I`th element of the `N`th judgment is not reachable	TD strategy
`(I is not a solution .)`	The `I`th element is not a solution	D&Q strategy
`(N : I is not a solution .)`	The `I`th element of the `N`th judgment is not a solution	D&Q strategy
`(its sort is final .)`	The sort of the current term is final	D&Q strategy
`(N : its sort is final)`	The sort of the term in the `N`th judgment is final	TD strategy
`(don't know .)`	Skips the current judgment	D&Q strategy
`(undo .)`	Returns to the previous state	At any time

In a similar way, it is also possible to indicate that some terms are final, that is, that they cannot be further rewritten:

- By using the value `final` in the attribute `metadata` of an operator declaration, that indicates that the terms built with this operator at the top are final.
- By selecting a set of final sorts. In this case, constructed terms having one of these sorts (or having a subsort of these sorts) are considered final.
- On the fly, as will be explained below.

In the first two cases, the user must activate the final sorts mode with the command

```
(set final select on .)
```

While the attribute `metadata` must be written in the Maude file, final sorts can be selected/deselected with the commands

```
(final select SORTS .)
(final deselect SORTS .)
```

where `SORTS` is a list of sort identifiers separated by spaces.
This option can be switched off with the command

```
(set final select off .)
```

A module with only correct definitions can be used to reduce the number of questions. In this case, it must be indicated before starting the debugging process with the command

```
(correct module MODULE-NAME .)
```

and can be deselected with the command

```
(delete correct module .)
```

Since rewriting is not assumed to terminate, a bound, which is 42 by default, is used when searching in the correct module and can be set with the command

```
(set bound BOUND .)
```

where `BOUND` is either a natural number or the constant `unbounded`. Note that if it is 0 the correct module will not be used for rewrites, while if it is `unbounded` the correct module is assumed to be terminating.

When debugging wrong rewrites, two different trees can be built: one whose questions are related to one-step rewrites and another whose questions are related to several steps. The user can switch between these trees, before starting the debugging process, with the commands

```
(one-step tree .)
(many-steps tree .)
```

the first of which is the default one.

In the same way, when debugging missing answers we distinguish between trees whose nodes are related to sets of terms obtained with one (the default case) or many steps. The user can select them with the commands

```
(one-step missing tree .)
(many-steps missing tree .)
```

When debugging missing answers, the user can prioritize questions related to the fulfillment of the search condition from questions involving the statements defining it. This option, switched off by default, can be activated with the command

```
(solutions prioritized on .)
```

and can be switched off again with

```
(solutions prioritized off .)
```

The debugging process for wrong answers is started with the commands

```
(debug [in MODULE-NAME :] INITIAL-TERM -> WRONG-TERM .)
(debug [in MODULE-NAME :] INITIAL-TERM : WRONG-SORT .)
(debug [in MODULE-NAME :] INITIAL-TERM =>* WRONG-TERM .)
```

for wrong reductions, memberships, and rewrites, respectively. `MODULE-NAME` is the module where the computation took place; if no module name is given, the current module is used by default. Similarly, we start the debugging of missing answers with the commands

```
(missing [in MODULE-NAME :] INITIAL-TERM -> ERR-NORMAL-FORM .)
(missing [in MODULE-NAME :] INITIAL-TERM : ERR-LEAST-SORT .)
(missing [[depth]] [in MODULE-NAME :] INITIAL-TERM =>* PATTERN [s.t. CONDITION] .)
(missing [[depth]] [in MODULE-NAME :] INITIAL-TERM =>+ PATTERN [s.t. CONDITION] .)
(missing [[depth]] [in MODULE-NAME :] INITIAL-TERM =>! PATTERN [s.t. CONDITION] .)
```

where the first command debugs erroneous normal forms, the second one erroneous least sorts, and the remaining ones refer to incomplete sets found when using search. More specifically, the third command specifies a search in zero or more steps, the fourth command in one or more steps, and the last one only checks final terms. The `depth` argument indicates the bound in the number of steps allowed in the search, and it is considered unbounded when omitted, while `MODULE-NAME` has the same behavior as in the commands above.

5.2. Navigating the debugging tree

We describe in this section how the debugging tree created with the commands described in the previous section is traversed. The debugging tree can be navigated by using two different strategies, namely, *top-down* and *divide and query*, the latter being the default one. The user can switch between them at any moment by using the commands

```
(top-down strategy .)
(divide-query strategy .)
```

In the divide and query strategy, each question refers to one judgment that can be either correct or wrong. The different answers are transmitted to the debugger with the answers

```
(yes .)
(no .)
```

If the question asked is too difficult, the user can avoid answering with [6]

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

To know the appropriate answer, we briefly describe the different kinds of questions asked by the debugger, defining for each of them when they are considered correct and describing the additional answers that can be used in each specific case. The possible questions are related to:

[6] Notice that in the current version of the debugger the question will not be asked again, thus this answer can lead to incompleteness.

Reductions: When a term t has been reduced by using equations to another term t', the debugger asks questions of the form "Is this reduction correct? $t \rightarrow t'$." These judgments are correct if the user expected t to be fully reduced to t' by using the equational part (equations and memberships) of the module.

In addition to the general answers, when the question corresponds to the application of a specific statement (either a equation, like in this case, a membership, or a rule), instead of just answering `yes`, we can also *trust* the statement on the fly if we decide the bug is not there. To trust the current statement we answer `(trust .)`.

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

Memberships: When a sort s is inferred for a term t, the debugger prompts questions of the form "Is this membership correct? $t : s$." These judgments are correct if the expected least sort of t is a subsort of s or s itself.

Least sorts: When the judgment refers to the least sort ls of a term t, the tool makes questions of the form "Did you expect t to have least sort ls?." In this case, the judgment is correct if the intended least sort of t is exactly ls.

Rewrites in one step: When a term t is rewritten into another term t' in only one step, the debugger asks questions of the form "Is this rewrite correct? $t \Rightarrow_1 t'$," where t' has already been fully reduced by using equations. This judgment is correct if the user expected to obtain t' from t modulo equations with only one rewrite.

Rewrites in several steps: When a term t is rewritten into another one t' after several rewrite steps, the debugger shows the question "Is this rewrite correct? $t \Rightarrow^+ t'$," where t' is fully reduced. This question is only prompted if the user selects the many-steps tree for wrong answers. This judgment is correct if t' is expected to be reachable from t.

Final terms: When a term t cannot be further rewritten, the debugger asks "Did you expect t to be final?." This judgment is correct if the user expected that no rules can be applied to t.

Additional information for this question can be given by answering `(its sort is final .)`, that indicates to the debugger that all the constructed terms with the same sort as this term are final.

Solutions: When a term t fulfills the search condition, the debugger shows questions of the form "Did you expect t to be a solution?." This judgment is correct if t is one of the intended solutions. In the same way, if a term does not fulfill the search condition the debugger asks "Did you expect t not to be a solution?," that is correct if t is not one of the expected solutions.

Reachable terms in one step: When all the possible applications of each rule in the current specification to a term t lead to a set of terms $\{t_1, \ldots, t_n\}$, with $n > 0$, the debugger prompts the question "Are the following terms all the reachable terms from t in one step? $t_1, \ldots, t_n$." This judgment is correct if all the expected terms from t in one step constitute the set $\{t_1, \ldots, t_n\}$.

In this case, if one of the terms is not reachable, the user can point it out with the answer `(I is wrong .)` where `I` is the index of the wrong term in the set. With this answer the debugger focuses on debugging this wrong judgment. This answer can also be used for reachable terms with one rule and in several steps.

Reachable terms with one rule: Given a term t and a rule r, when all the possible applications of r to t produce a set of terms $\{t_1, \ldots, t_n\}$, the debugger presents questions of the form "Are the following terms all the reachable terms from t with one application of the rule r? $t_1, \ldots, t_n$." This judgment is correct if all the expected reachable terms from t with one application of r form the set $\{t_1, \ldots, t_n\}$. When $n = 0$ the debugger prompts questions of the form "Did you expect that no terms can be obtained from t by applying the rule r?," that is correct if the rule r is not expected to be applied to t.

Reachable terms in several steps: Given an initial term t, a condition c, and a bound in the number of steps n, when all the terms reachable in at most n steps from t that fulfill c are $t_1, \ldots, t_m$, with $m > 0$, the debugger makes the following distinction:

- If the condition c defines the initial condition of the search, the tool asks questions of the form "Are the following terms all the possible solutions from t in n steps? $t_1, \ldots, t_m$," where the bound is omitted if it is unbounded. This judgment is correct if all the solutions that the user expected to obtain from t in at most n steps constitute the set $\{t_1, \ldots, t_m\}$. If $m = 0$ the debugger asks questions of the form "Did you expect that no solutions are reachable from t in n steps?," where the bound is again omitted if it is unbounded. In this case, the judgment is correct if no solutions were expected from t in at most n steps.

In this case, if one of the solutions is reachable but it should not fulfill the search condition, the user can indicate it with `(I is not a solution .)`, where `I` is the index of the term that should not be in the set. With this answer the user indicates that the definition of the search condition is erroneous and the debugger centers on it to continue the process.

- If the condition c has been obtained from a rewrite condition $t' \Rightarrow p$, then c is just a matching condition with the pattern p, and n is unbounded. In this case, the questions have the form "Are the following terms all the reachable terms from t

that match the pattern p? $t_1, \ldots, t_m$." This judgment is correct if all the terms that should be obtained from t and match the pattern p constitute the set $\{t_1, \ldots, t_m\}$. When $m = 0$ the questions have the form "Did you expect that no terms matching the pattern p can be obtained from t?," that is correct if t is expected to be final or all the terms reachable from t are not expected to match p.

These questions are only asked if the many-steps tree for missing answers is used.

In case the top-down strategy is selected, several questions will be displayed in each step. The user can then introduce answers of the form `(N : answer .)`, where `N` is the index of the question and `answer` is the same answer that would be used in the divide and query strategy for this question. Moreover, as a shortcut to answer `(yes .)` to all the questions, the debugger provides the answer

```
(all : yes .)
```

Finally, we can return to the previous state in both strategies by using the command

```
(undo .)
```

5.3. Recommendations

We recommend following some tips to ease the questions asked during the debugging process:

- It is usually more complicated to answer questions related to many steps (both in wrong and missing answers) than questions related to one step. Thus, if a specification is complex it is better to debug it with a one-step tree.
- There are some sorts that are usually final, such as `Bool` and `Nat`, so identifying them as final can avoid several tedious questions.
- If an error is found using a complex initial term, this error can probably be reproduced with a simpler one. Using this simpler term leads to easier debugging sessions.
- When facing a problem with both wrong and missing answers, it is usually better to debug the wrong answers first, because questions related to them are usually easier to answer and fixing them can also solve the missing answers problem.
- When a question is related to a set of reachable terms that contains some wrong terms, it is recommended to point out one of these terms as erroneous instead of indicating the whole set as wrong.
- When using the top-down navigation strategy, several questions are prompted. To point out one as erroneous or all of them as valid will shorten the debugging process, while pointing out one question as correct usually only eases the current set of questions. Thus, to indicate that a question is valid is only recommended for extremely complicated or large sets of questions.

If the user follows these tips and uses the trusting mechanisms it is possible to debug very large specifications, because:

- Specifications are assumed to be structured, and usually the module being debugged imports several other auxiliary modules. These modules should have been debugged before testing the current one, and thus they can be trusted (maybe some complex functions from these auxiliary modules can be suspicious).
- Specific reductions/sort inferences/rewrites usually do not apply every statement in the specification, but a small subset of them. From this point of view, debugging a large specification should not be harder than debugging a smaller one.
- The debugger assists the user through the computation, making the debugging process easier than checking by hand thousands of statements and than traversing the trace without any guide.

6. A debugging session

We describe in this section how to debug the maze example shown in Section 2.5. We recall that we have specified a module to search a path out of a labyrinth but, given a concrete labyrinth with an exit, the program is unable to find it. We start the debugging process with the command:

```
Maude> (missing {[1,1]} =>* {L:List} s.t. isSol(L:List) .)
```

With this command the debugger builds a debugging tree for missing answers in zero or more steps with the questions about solutions not prioritized, and navigated with the default divide and query strategy. The first question is:

```
Did you expect {[1,1][1,2][1,3][1,4]} to be final?
Maude> (no .)
```

Since we expected to reach the position `[2,4]` from `[1,4]`, this state should be rewritten and thus it is not final. The next question is:

```
Is this reduction (associated with the equation c2) correct?

contains([2,1][4,1][2,2][3,2][6,2][7,2][2,3][4,3][5,3][6,3][7,3][1,5][2,5][3,5][4,5][5,5]
         [6,5][8,5][6,6][8,6][6,7][6,8][7,8],[1,3]) -> false

Maude> (yes .)
```

That is, the debugger asks whether it is correct that the position `[1,3]` is not included in the wall. We answer that it is correct and the next question is:

```
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4]) in one step?

1 [1,5]
2 [1,3]
3 [0,4]

Maude> (no .)
```

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

```
Did you expect [1,4] to be final?

Maude> (yes .)
```

The answer is `yes` because we have not defined rules for positions, thus they cannot evolve. The following series of questions are:

```
Did you expect [1,3] to be final?

Maude> (yes .)

Did you expect [1,2] to be final?

Maude> (yes .)

Did you expect [1,1][1,2][1,3][1,4] to be final?

Maude> (yes .)
```

We use the same reasoning about final terms to answer these questions. The next questions are:

```
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4])
with one application of the rule n2 ?

1 [0,4]

Maude> (yes .)

Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4])
with one application of the rule n3 ?

1 [1,3]

Maude> (yes .)

Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4])
with one application of the rule n1 ?

1 [1,5]

Maude> (yes .)
```

All these questions are related to the appropriate application of certain rules; these rules move the last position of the list to the left, up, and down, and thus they are correct. With this information, the debugger is able to find the bug, prompting:

```
The buggy node is:
next([1,1][1,2][1,3][1,4]) =>1 {[1,5], [1,3], [0,4]}

Either the operator next needs more rules or the conditions of the current rules are not written
in the intended way.
```

In fact, if we check the code we realize that we forgot to define the rule that specifies movements to the right. We must add the rule:

```
  rl [next4] : next(L [X,Y]) => [X + 1, Y] .
```

However, we noticed that this session required us to answer a lot of similar questions. We can enhance the behavior of the debugger by using features such as selection of final terms on the fly. For example, when the fourth question is prompted:

```
Did you expect [1,4] to be final?

Maude> (its sort is final .)

Terms of sort Pos are final.
```

we can indicate that not only this term, but all the terms with its sort (not necessarily as least one, that is, subsorts are also checked) are final. With this answer the debugging tree is pruned, and the next question is:

```
Did you expect [1,1][1,2][1,3][1,4] to be final?

Maude> (its sort is final .)

Terms of sort List are final.
```

We use this answer again, although in this case it does not reduce the number of questions. As before, the debugger finishes with the same three questions as above.

Although the number of questions has been reduced, we still face some questions that we would like to avoid about final terms. To do this, we can activate the final selection mode before starting the debugging:

```
Maude> (set final select on .)
Final select is on.
```

Once this mode is active, we can point out the sorts of the terms that will not be rewritten. Note that terms whose least sort is a subsort of the sorts selected will also be considered as final. For example, we consider in our specification the sorts `Nat` and `List` as final, which implicitly indicates that the sort `Pos`, subsort of `List`, is also final:

```
Maude> (final select Nat List .)

Sorts List Nat are now final.
```

Moreover, since we know that the rules `next1`, `next2`, and `next3` are correct, we can avoid questions about them by pointing out that the rest of the statements are suspicious with the commands:

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

Debug select is on.

Maude> (debug select is1 is2 c1 c2 expand .)

Labels c1 c2 expand is1 is2 are now suspicious.
```

Once these options are introduced, we can start the debugging process with the same command as before:

```
Maude> (missing {[1,1]} =>* {L:List} s.t. isSol(L:List) .)

Are the following terms all the reachable terms from {[1,1][1,2][1,3]} in one step?

1 {[1,1][1,2][1,3][1,4]}

Maude> (yes .)
```

```
Are the following terms all the reachable terms from {[1,1][1,2]} in one step?

1 {[1,1][1,2][1,3]}

Maude> (yes .)
```

Given the labyrinth's limits and wall, we must go down in both cases to find the exit. The next question selected by the debugger is:

```
Did you expect that no terms can be obtained from {[1,1][1,2][1,3][1,4]}
by applying the rule expand ?

Maude> (no .)
```

As we know, the list of positions should evolve to find the exit. The debugger asks now:

```
Is this reduction (associated with the equation c2) correct?

contains([2,1][4,1][2,2][3,2][6,2][7,2][2,3][4,3][5,3][6,3][7,3][1,5][2,5]
         [3,5][4,5][5,5][6,5][8,5][6,6][8,6][6,7][6,8][7,8],[1,3]) -> false

Maude> (trust .)
```

We realize now that the equation `c2` is simple enough to be trusted, although we pointed it out as suspicious at the beginning of the session. We use the command `trust` and the following question is prompted:

```
Is this reduction (associated with the equation c1) correct?

contains(nil,[1,5]) -> false

Maude> (trust .)
```

We consider that this equation can also be trusted. Finally, the debugger detects the problem with the next answer:

```
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4]) in one step?

1 [1,5]
2 [1,3]
3 [0,4]

Maude> (no .)

The buggy node is:
next([1,1][1,2][1,3]) =>1 {[1,4], [1,2], [0,3]}

Either the operator next needs more rules or the conditions of the current rules are not written
in the intended way.
```

Although in this example we have used the default divide and query navigation strategy, it is also possible to use the top-down one by using:

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

In this case we reduce the number of questions by considering that the sorts `Nat` and `List` are final and that the suspicious statements are the equations defining the solution, `is1` and `is2`:

```
Maude> (set final select on .)

Final select is on.

Maude> (final select Nat List .)

Sorts List Nat are now final.

Maude> (set debug select on .)

Debug select is on.

Maude> (debug select is1 is2 .)
Labels is1 is2 are now suspicious.
```

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

```
Maude> (missing { [1,1] } =>* { L:List } s.t. isSol(L:List) .)

Question 1 :
Did you expect {[1,1]} not to be a solution?

Question 2 :
Are the following terms all the reachable terms from {[1,1]} in one step?

1 {[1,1][1,2]}

Question 3 :
Did you expect {[1,1][1,2]} not to be a solution?

Question 4 :
Are the following terms all the reachable terms from {[1,1][1,2]} in one step?

1 {[1,1][1,2][1,3]}

Question 5 :
Did you expect {[1,1][1,2][1,3]} not to be a solution?

Question 6 :
Are the following terms all the reachable terms from {[1,1][1,2][1,3]} in one step?

1 {[1,1][1,2][1,3][1,4]}

Question 7 :
Did you expect {[1,1][1,2][1,3][1,4]} not to be a solution?

Question 8 :
Did you expect {[1,1][1,2][1,3][1,4]} to be final?

Maude> (8 : no .)
```

The eighth question (corresponding to the root of the tree in Figure 16, marked with (†)) is erroneous because position `[2,4]` is reachable from `[1,4]` and it is free of wall, so we do not expect this term to be final. The following questions are:[7]

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

Question 2 :
Is Pos? the least sort of next([1,1][1,2][1,3][1,4]) ?

Question 3 :
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4]) in one step?

1 [1,5]
2 [1,3]
3 [0,4]

Maude> (3 : no .)
```

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

```
The buggy node is:
next([1,1][1,2][1,3][1,4]) =>1 {[1,5], [1,3], [0,4]}
```

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

```
Either the operator next needs more rules or the conditions of the current rules are not written
in the intended way.
```

Many more examples are available at http://maude.sip.ucm.es/debugging/.

7. Implementation

We show here how the ideas described in the previous sections are implemented. This implementation is done in Maude itself by means of its reflective capabilities, which allow us to use Maude terms and modules as data [12, Chapter 14]. Sections 7.1 and 7.2 describe the tree construction stage, where the abbreviated proof trees are constructed. The interaction with the user is explained in Section 7.3.

The complete code of the tool is contained in the file `dd.maude`, available at http://maude.sip.ucm.es/debugging/.

7.1. *Debugging trees definition*

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

The parameterized module that describes the behavior of the tree receives the theory `TRIV` (that simply requires a sort `Elt`) as parameter. We use lists of natural numbers to identify (the position of) each node. General trees are defined by means of the constructor `tree`, composed of some contents (received from the theory), the size of the tree, and a `Forest`, which in turn is a list of trees:

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

 sorts Tree Forest .
 subsort Tree < Forest .

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

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

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

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

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

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

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

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

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

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

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

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

- The set of reachable terms in one step:

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

- The set of reachable terms by applying one rule:

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

- The set of reachable terms when many rewrite steps are used. In this case we also keep the bound, the pattern, the condition and a Boolean value indicating whether this search corresponds to the initial one, and thus these terms are the reachable *solutions* from the initial one, or corresponds to a search due to a rewrite condition:

```
op _~>[_]{_}s.t._&_[_] : Term Bound TermList Term Condition Bool
                         -> Judgment [ctor format (d b o d d d d d d d d d d d d d)] .
```

We use the operator `sol` to indicate (the Boolean value in the fourth argument) whether a term (the first argument) matches the pattern given as second argument and fulfills the condition given as third argument. When the questions about solutions are prioritized these nodes are frozen and are expanded on demand, so it has a Boolean value (the fifth argument) indicating whether the node has been already expanded. Finally, the last Boolean value indicates whether this term is a solution of the initial search condition or it is a solution of a rewrite condition:

```
op sol : Term Term Condition Bool Bool Bool -> Judgment [ctor format (b o)] .
```

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

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

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

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

We use this module to create a view from the `TRIV` theory and we obtain our debugging trees by instantiating the module `TREE` above with this view:

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

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

7.2. Debugging trees construction

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

7.2.1. Debugging trees for wrong reductions and memberships

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

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

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

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

The function `createForest2` checks first whether the current term is of the form `if T1 then T2 else T3 fi`. In this case, the debugger evaluates `T1` and then, depending on the result, it evaluates either `T2` or `T3` following the same evaluation strategy as Maude:[8]

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

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

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

The function `applyEq` tries to apply (at the top) one equation,[9] by using the *replacement* rule from Fig. 1, with the constraint that we cannot apply equations with the `otherwise` attribute if other equations can be applied. To apply an equation we check whether the term we are trying to reduce matches the lefthand side of the equation and its conditions are fulfilled. If this happens, we obtain a substitution (from both the matching with the lefthand side and the conditions) that we can apply to the righthand side of the equation. Note that, if we can obtain the transition in the correct module, the forest is not computed:

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

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

For example, the equations without the `otherwise` attribute as applied as follows:

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

[8] Note that it is possible to obtain neither `true` nor `false` when evaluating the condition. In this case, both branches will be evaluated and the term thus obtained (which is not fully evaluated) used in the rest of the computation, possibly leading to a missing answer.

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

```
     not owise?(AtS) /\
     sameKind(OM, type(OM, L), type(OM, T)) /\
     SB := metaMatch(OM, L, T, C, 0) /\
     R' := substitute(OM, R, SB) /\
     F := conditionForest(substitute(OM, C, SB), OM, TM, CM, QS)
          createForest(OM, TM, CM, R', T', QS) .
```

where we distinguish with the function `in?(AtS,QS)` whether the equation is trusted (the attribute set does not contain a label or the label is contained in the set `QS` of trusted labels) to generate the node.

7.2.2. Debugging trees for wrong rewrites

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

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

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

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

```
op createRewTree : Module Maybe{Module} Term Term QidSet TreeType Bound -> Maybe{Tree} .
eq createRewTree(OM, CM, T, T', QS, os, B) = oneStepTree(OM, CM, T, T', QS, B) .
eq createRewTree(OM, CM, T, T', QS, ms, B) = manyStepsTree(OM, CM, T, T', QS, B) .
```

The function `oneStepTree` creates a complete debugging tree with only one-step rewrites in its nodes. It puts the complete judgment as the root of the tree, computes the tree for the reduction from the initial term to normal form with the function `createForest` from Section 7.2.1, and then computes the rest of the tree with the function `oneStepForest`. This corresponds to a concrete application of the *equivalence class* inference rule from Fig. 1:

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

`oneStepForest` computes the trace of a rewrite with the predefined function `metaSearchPath` and uses it to generate a debugging tree by using `trace2forest`, which generates a forest of one-step rewrites by extracting each step of the trace and creating its corresponding tree:

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

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

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

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

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

many-steps tree from the trace is built with the function `trace2tree`, that traverses the trace and creates a balanced tree from the forest of leaves obtained from it:

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

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

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

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

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

7.2.3. Debugging trees for missing answers

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

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

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

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

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

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

The debugging tree for incomplete sets of reachable terms is built with the function `createMissingTree`, that receives:

- the module where the terms should be found,
- a correct module (possibly `undefMod`),
- the initial term, the pattern,
- the condition to be fulfilled,
- the bound in the number of rewrites for *wrong* rewrites,
- the number of steps that can be given in the search,
- the search type,
- the type of tree to be built (one-step or many-steps) for both wrong and missing answers,
- the set of suspicious labels,
- the set of final sorts,
- a Boolean value indicating whether the search introduced by the user was unbounded, and
- a Boolean value pointing out whether the questions about solutions are prioritized.

The forest is generated with an auxiliary function `createMissingForest` that receives, in addition to the values above, a Boolean value indicating whether the forest currently built corresponds to the initial search or to a search due to a rewrite condition, which is `true` in the first case. Once the tree has been built, the questions associated with terms that the user has declared as final are pruned with `cleanTree*`:

```
op createMissingTree : Module Maybe{Module} Term Term Condition Bound Bound SearchType
                       TreeType TreeType QidSet Bool QidSet Bool Bool -> Tree .

ceq createMissingTree(M, CM, T, PAT, C, BW, BM, ST, TTW, TTM, QS, BFS, FS, UB?, SP) =
      contract(tree(T ~>[B'] {clean(extractTerms(F))} s.t. PAT & C [true],
                                        1 + getOffspring*(F), F))
 if TM := deleteSuspicious(M, QS) /\
    T' := getTerm(metaReduce(M, T)) /\
    F := cleanTree*(M, BFS, FS, createForest(M, TM, CM, T, T', QS, strat?(M))
                           createMissingForest(labeling(M), TM, CM, T', PAT,
                                   C, BW, BM, ST, TTW, TTM, QS, FS, UB?, SP, true)) /\
    B' := if UB? then unbounded else BM fi .
```

If the tree to be built cannot evolve (the bound is `0`) and zero or more steps can be used, then we use the function `solutionTree` to create a tree that proves whether the condition is satisfied or not:

```
op createMissingForest : Module Module Maybe{Module} Term Term Condition Bound Bound SearchType
                         TreeType TreeType QidSet QidSet Bool Bool Bool -> Forest .
eq createMissingForest(OM, TM, CM, T, PAT, C, BW, 0, zeroOrMore, TTW, TTM, QS,
                       FS, UB?, SP, FST) =
       solutionTree(OM, TM, CM, T, PAT, C, BW, zeroOrMore, TTW, TTM, QS, FS, SP, FST) .
```

When the terms can still evolve (the bound is greater than `0`), we compute all the possible reachable terms in exactly one step with the function `oneStepMissingTree` and evolve each of them with `createMissingForest*`. The solutions obtained are gathered with `extractTerms`, while we check whether the current term is a valid solution with the function `solveCondition`. Finally, if the tree selected by the user is for many-steps transitions we create a root for the generated forest specifying the number of steps, while if we want one-step transitions only the forest is returned:

```
ceq createMissingForest(OM, TM, CM, T, PAT, C, BW, s(N'), zeroOrMore, TTW, TTM,
                        QS, FS, UB?, SP, FST) =
            if TTM == os then RF
            else tree(T ~>[B'] {TL''} s.t. PAT & C [FST], 1 + getOffspring*(RF), RF)
            fi
 if tree(T =>1 {TL}, N, F) := oneStepMissingTree(OM, TM, CM, T, QS, FS, BW, zeroOrMore,
                                                 TTW, TTM, SP) /\
    F' := createMissingForest*(OM, TM, CM, TL, PAT, C, BW, N', zeroOrMore,
                                    TTW, TTM, QS, FS, UB?, SP, FST) /\
    TL' := if solveCondition(OM, T, PAT, C) then T
                                           else empty fi /\
    TL'' := clean((extractTerms(F'), TL')) /\
    CF := solutionTree(OM, TM, CM, T, PAT, C, BW, zeroOrMore, TTW, TTM, QS, FS, SP, FST) /\
    RF := CF tree(T =>1 {TL}, N, F) F' /\
    B' := if UB? then unbounded else s(N') fi .
```

7.3. The debugger environment

We implement our system on top of Full Maude, a language that extends Maude with support for object-oriented specification and advanced module operations [12, Part II]. The implementation of Full Maude includes code for parsing user input and pretty-printing; storing modules, theories, and views; and transforming object-oriented modules into system modules.

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

```
fmod DD-SIGNATURE is
 including FULL-MAUDE-SIGN .
 op debug_. : @Bubble@ -> @Command@ .
 op missing_. : @Bubble@ -> @Command@ .
 ...
endfm
```

This signature is included in the meta-module `GRAMMAR` to obtain the grammar `DD-GRAMMAR`, that allows us to parse both Full Maude modules and commands together with the debugger commands:

```
fmod META-DD-SIGN is
 inc META-FULL-MAUDE-SIGN .
 inc UNIT .
 op DD-GRAMMAR : -> FModule [memo] .
 eq DD-GRAMMAR = addImports((including 'DD-SIGNATURE .), GRAMMAR) .
 ...
endfm
```

The module DD-COMMAND-PROCESSING is in charge of processing the commands dealing with suspicious statements, final sorts, and the debugging commands:

```
fmod DD-COMMAND-PROCESSING is
  pr COMMAND-PROCESSING .
  pr META-DD-SIGN .
  pr MISSING-ANSWERS-TREE .
  pr SEARCH-TYPE .
  pr PRINT .
```

For example, the parsing of the debugging command for wrong answers returns a tuple containing the generated tree, the module where the computation took place, the set of suspicious statements, and a list of quoted identifiers indicating the errors that occurred during the parsing:

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

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

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

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

After finding out the kind of the debugging command (reduction, membership, or rewrite) and if a module name has been selected by the command, the function procDebug builds the appropriate tree by using the functions createTree and createRewTree explained in Section 7.2:

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

The persistent state of Full Maude's system is given by a single object of class DatabaseClass, which maintains the database of the system. We extend the Full Maude system by defining a subclass of DatabaseClass inheriting its behavior and adding new attributes to it:

```
mod DD-DATABASE-HANDLING is
 inc DATABASE-HANDLING .
 pr DD-COMMAND-PROCESSING .
 pr TREE-PRUNING .
 pr DIVIDE-QUERY-STRATEGY .
 pr LIST{DDState} .
 pr LIST{Answer} .
 sort DDDatabaseClass .
 subsort DDDatabaseClass < DatabaseClass .
 op DDDatabase : -> DDDatabaseClass [ctor] .
```

The new attributes include, for example:

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

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

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

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

- the set of labels considered `suspicious`:

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

- the set of final sorts:

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

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

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

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

When a correct module expression is introduced, `correct-module` keeps the associated module if it exists, and shows an error message otherwise:

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

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

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

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

```
crl [top-down-traversal-no] :
    < O : DDDC | input : ('_:'no'.['token[T]]), strategy : td, tree : PT,
                 previousStates : LS, answers : LA, state : waiting, AtS >
 => < O : DDDC | input : nilTermList, strategy : td, tree : PT',
                 previousStates : LS < nil, PT, td >,
                 answers : LA getAnswer(PT', wrong), state : computing, AtS >
 if UPT := removeUnknownChildren(PT) /\
    N := downNat*(T) /\
    N > 0 /\
    N <= size(getForest(UPT, nil)) /\
    PT' := getSubTree(UPT, sd(N, 1)) .
```

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

The rule `missing-wrong` is used when, while debugging missing answers with the divide and query strategy, the user points out that a certain term is not reachable. The rule checks that the current question is related to an inference of a set of terms with `setInference?` and that the selected question points to one of these terms, and then creates the debugging tree for wrong answers with `createRewTree`:

```
crl [missing-wrong] :
    < O : DDDC | input : ('_is'wrong'.['token[T]]), strategy : dq, tree : PT,
                 current : NL, previousStates : LS, answers : LA, state : waiting,
                 currentSuspicious : QS, bound : BND, module : M, correction : MM,
                 currentTTW : TT, AtS >
 => < O : DDDC | input : nilTermList, strategy : dq, tree : PT',
                 current : NL, previousStates : LS < NL, PT, dq >,
                 answers : LA getAnswer(getSubTree(PT, NL), wrong),
                 state : computing, currentSuspicious : QS, bound : BND,
                 module : M, correction : MM, currentTTW : TT, AtS >
 if N := downNat*(T) /\
    setInference?(getContents(PT, NL)) /\
    N > 0 /\
    N <= numTermsInRootSet(getSubTree(PT, NL)) /\
    T1 := getFirstTerm(getSubTree(PT, NL)) /\
    T2 := getWrongTerm(getSubTree(PT, NL), N) /\
    PT' := createRewTree(labeling(M), MM, T1, T2, QS, TT, BND) .
```

When the divide and query strategy is selected and the user decides to trust a statement, the current subtree is deleted and the resulting tree is pruned in order to delete the nodes associated with the trusted statement:

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

In the divide and query strategy, when the user indicates that the sort of a certain term is final on the fly the rule `sort-final` is applied. It checks that the question is related to final terms with the function `finalQuestion?` and then prunes all the tree with the function `pruneFinalSort`:

```
crl [sort-final] :
    < O : DDDC | input : ('its'sort'is'final'..@Command@), output : nil,
                 tree : PT, current : NL, module : M, state : waiting, AtS >
 => < O : DDDC | input : nilTermList, output : ('\n '\b 'Terms 'of 'sort '\o Ty
                                               '\b 'are 'final. '\o '\n),
                 tree : PT', current : NL, module : M, state : computing, AtS >
 if finalQuestion?(getContents(PT, NL)) /\
    T := getFirstTerm(getSubTree(PT, NL)) /\
    Ty := getType(metaReduce(M, T)) /\
    PT' := pruneFinalSort(M, Ty, PT) .
```

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

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

The module `DD` manages the introduction of data by the user and the output of the debugger's answers. Full Maude uses the input/output facility provided by the `LOOP-MODE` module [12, Chapter 17], which consists of an operator `[_,_,_]` with an input stream (the first argument), an output stream (the third argument), and a state (given by its second argument):

```
mod DD is
 inc DD-DATABASE-HANDLING .
 inc LOOP-MODE .
 inc META-DD-SIGN .
 op o : -> Oid .
 --- State for LOOP mode:
 subsort Object < State .
 op init-debug : -> System .

 rl [init] :
    init-debug
 => [nil, < o : DDDatabase | input : nilTermList, output : nil, init-state >, dd-banner] .
```

The rule `in` below parses the data introduced by the user, which appears in the first argument of the loop, in the module `DD-GRAMMAR` and introduces it in the `input` attribute if it is correctly built:

```
crl [in] :
    [QIL, < O : X@Database | input : nilTermList, Atts >, QIL']
 => [nil,
     < O : X@Database | input : getTerm(metaParse(DD-GRAMMAR, QIL, '@Input@)), Atts >,
     QIL']
 if QIL =/= nil /\
    metaParse(DD-GRAMMAR, QIL, '@Input@) : ResultPair .
```

The rule `out` is in charge of printing the messages from the debugger by moving the data in the `output` attribute to the third component of the loop:

```
 rl [out] :
    [QIL, < O : X@Database | output : (QI QIL'), Atts >, QIL'']
 => [QIL, < O : X@Database | output : nil, Atts >, (QIL'' QI QIL')] .
endm
```

8. Conclusions and future work

We have presented in this paper a declarative debugger for Maude specifications. The debugging trees used in the debugging process are obtained from an abbreviation of a proper calculus whose adequacy for debugging has been proved. This work comprises our previous work on wrong [30,8,34] and missing answers [32,31], and provides a powerful and complete debugger for Maude specifications. Moreover, we also provide a graphical user interface that eases the interaction with the debugger and allows one to traverse the debugging tree with more freedom [29,33]. The tree construction, its navigation, and the user interaction (excluding the GUI) have all been implemented in Maude itself. For more information, see http://maude.sip.ucm.es/debugging.

We plan to add new navigation strategies like the ones shown in [36] that take into account the number of different potential errors in the subtrees, instead of their size. Moreover, the current version of the tool allows the user to introduce a correct but maybe incomplete module in order to shorten the debugging session. We intend to add a new command to introduce *complete* modules, which would greatly reduce the number of questions asked to the user. Finally, we also plan to create a test generator to test Maude specifications and debug the erroneous tests with the debugger.

Appendix A. Proofs

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

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

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

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

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

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

- $adequateSorts(\kappa) \rightsquigarrow \Theta$ is correct. Given a kind-substitution κ, when it has the variables of the appropriate sorts only the rule $\mathsf{SubsCond}$ can be applied and the set containing κ is returned. If the matching fails, AS_2 has to be applied and the empty substitution set is returned, being the judgment correct.
- $[C, \theta] \rightsquigarrow \Theta$ is correct. We distinguish subcases over the different kinds of conditions:
  - $C \equiv t_1 = t_2$. Since we work with admissible conditions, we know that $\theta(t_1)$ and $\theta(t_2)$ are ground, and thus the only possible substitution that can be included in Θ is θ. If the condition is fulfilled only rule EqC_1 can be used, and $\{\theta\}$ is returned, which is correct. Otherwise, only EqC_2 can be used, returning now the empty set which is again correct.
  - $C \equiv t_1 := t_2$. We assume that $\theta(t_2) \rightarrow_{norm} t'$ so, given the complete set of kind-substitutions, we restrict them to those that are substitutions, thus returning the correct set.
  - $C \equiv t : s$. Like in equational conditions, $\theta(t)$ is ground and the resulting set can only contain θ. If the condition is fulfilled only MbC_1 can be applied and the set obtained is correct. Analogously, if the condition does not hold, only MbC_2 can be used and the correct result is the empty set.
  - $C \equiv t_1 \Rightarrow t_2$. We assume that the set of reachable terms from $\theta(t_1)$ that match $\theta(t_2)$ is correct, and thus by definition the set computed by rule RlC, the only one applicable here, is correct.
- $\langle C, \Theta \rangle \rightsquigarrow \Theta'$ is correct. The only rule that deals with this judgment is $\mathsf{SubsCond}$. Assuming the premises correct, the conclusion is also correct.
- $disabled(e, t)$ is correct. The only rule that deals with this judgment is Dsb. Assuming the premises correct there are no substitutions satisfying the conditions and making the lefthand side of the equation or membership match the term, so it cannot be applied and the judgment is correct.
- $t \rightarrow_{red} t'$ is correct. In this case two rules can be used: Rdc_1 and Rdc_2. The first one covers reductions at the top, while the second one covers reductions on the subterms, thus dealing with all possibilities. Assuming the premises correct, in the first case we verify that one step is used because it corresponds to the application of one equation, while in the second one we check with the side condition that at least one step is used and thus the judgment is correct.
- $t \rightarrow_{norm} t'$ is correct. The rules that deal with this case are Norm and NTr, that distinguish whether the term is already in normal form or can be further reduced. In the first case if we assume the premises correct then the term is in normal form and then the same term has to be returned. In the second case, assuming the premises correct and a confluent specification, the conclusion is correct.
- $fulfilled(\mathcal{C}, t)$. This judgment is correct when there exists a substitution that makes $\mathcal{C}$ with the hole $\circledast$ filled by t hold. Rule $\mathsf{Fulfill}$, the only one that can be used to prove this predicate, states this fact and thus the judgment is correct.
- $fails(\mathcal{C}, t)$. This judgment is correct when $\mathcal{C}$ with t filling its hole $\circledast$ cannot be satisfied. Since the only rule that can be used for this predicate is Fail and the premise indicates that the set of substitutions that fulfill the condition is empty, the judgment is correct.
- $t \Rightarrow^{q} S$. This judgment is only computed with rule Rl. By hypothesis, all the substitutions that fulfill the conditions and make t match the lefthand side of the rule are in Θ_k, thus by definition the union of the application of all the substitutions in Θ_k to the lefthand side of the rule generate the set we are looking for and the judgment is correct.
- $t \Rightarrow^{top} S$. This judgment is only computed with rule Top. First, we notice that the rules in $\{q_1, \ldots, q_l\}$ are the only ones that can be applied to t (it does not match the lefthand side of the rest of the rules) and thus the correctness is not affected by this selection. We know by hypothesis that each S_i, the set of reachable terms obtained from t with the rule q_i, is correct and hence the union of all these sets is by definition the set of reachable terms by rewriting at the top and the judgment is correct.
- $t \Rightarrow_1 S$. This judgment is only computed with rule Stp. By hypothesis, we know that S_t contains the set of reachable terms obtained by rewriting t at the top, while S_i contains the reachable terms in one step from t_i. Since the set of reachable terms in one step from t is the union of the terms obtained by one rewriting at the top and the set created by substituting each subterm by all the reachable terms in one step from it, the judgment is correct.

- $t \rightsquigarrow_n^{\mathcal{C}} S$. For this judgment, rule Red_1 can always be applied. Since we work with a coherent theory, the set of reachable terms from both t and t_1 are the same, while t_2 and t' are in the same equivalence class and thus are equal modulo E.
  When $n = 0$, rules Rf_1 or Rf_2 are used and the result is straightforward.
  If $n > 0$ and the term fulfills the condition, rule Tr_1 is applied. Since the condition holds, the result set must contain t, that is added in the conclusion of the rule. Moreover, the terms $t_1, \ldots, t_k$ are the reachable terms from t in exactly one step, while S_i is the set of reachable terms from t_i in zero or more steps, that is, the union of the S_i is the set of reachable terms in at least one step and at most n, and thus the union of this set with the singleton set $\{t\}$ creates a correct set for this judgment. Analogously, when $n > 0$ and the condition does not hold, rule Tr_2 is applied. □

Theorem 2. *The calculus of Fig. 12 is correct.*

Proof.
- $t \rightsquigarrow!_n^{\mathcal{C}} S$. For this judgment, rule Red_2 can always be applied. Since we work with a coherent theory, the set of reachable terms from both t and t_1 are the same, while t_2 and t' are equal modulo E.
  When $n = 0$, rules Rf_3, Rf_4, and Rf_5 can be used. If t is not final only Rf_5 can be used and, since no more steps are allowed, the empty set of results is returned, which is correct by definition. If t is final we have to check whether the term fulfills the condition; if the condition holds only Rf_3 can be used and hence the singleton set consisting of the term is returned, while if the condition fails Rf_4 is applied and the empty set is returned. In both cases the result is correct by definition.
  When $n > 0$ rules Rf_3, Rf_4, and Tr_3 can be used. If the term is final, Rf_3 and Rf_4 are applied and the result holds as in the previous case. If the term is not final, then Tr_3 is applied; the terms $t_1, \ldots, t_k$ are the reachable terms from t in exactly one step, while S_i is the set of reachable terms from t_i in zero or more steps, that is, the union of the S_i is the set of reachable terms in at least one step and at most n and, since the current term cannot be a solution because it is not final, the judgment is correct.
- $t \rightsquigarrow+_n^{\mathcal{C}} S$. We distinguish cases over n:
  When $n = 0$, only rule Rf_6 can be applied; since the judgment requires at least one step, the set of reachable terms is empty by definition.
  When $n > 0$, rule Tr_4 is applied. Since $t \to t'$ and the specification is coherent, we know that the set of reachable terms from both t and t' is the same; the terms $t_1, \ldots, t_k$ are the reachable terms from t in exactly one step, while S_i is the set of reachable terms from t_i in zero or more steps (note that the judgments in the premises are different from the one in the conclusion), that is, the union of the S_i is the set of reachable terms in at least one step and at most n and hence the judgment is correct. □

Proposition 2. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, C an atomic condition, θ an admissible substitution, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If adequateSorts$(\kappa) \rightsquigarrow \Theta$, $[C, \theta] \rightsquigarrow \Theta$, or $\langle C, \Theta \rangle \rightsquigarrow \Theta'$ can be deduced using the rules from Fig. 4 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models$ adequateSorts$(\kappa) \rightsquigarrow \Theta$, $\mathcal{T}_{\Sigma/E',R'} \models [C, \theta] \rightsquigarrow \Theta$, and $\mathcal{T}_{\Sigma/E',R'} \models \langle C, \Theta \rangle \rightsquigarrow \Theta'$, respectively.*

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

- EqC_1 From the premises we deduce that $[\theta(t_1)]_{E'} = [\theta(t_2)]_{E'}$, that is, the condition is satisfied with the current substitution θ. Since θ already binds all the variables in the condition, it cannot be extended and θ itself is the result.
- EqC_2 From the premises we deduce that $[\theta(t_1)]_{E'} \neq [\theta(t_2)]_{E'}$, thus the condition fails and there is no substitution that could satisfy it.
- PatC We know that $[\theta(t_2)]_{E'} = [t']_{E'}$ and that matching conditions can have variables in its lefthand side that are not bound in θ. Thus, the substitution is extended with all the substitutions θ' that match t' and, since t' is equal (modulo E') to $\theta(t_2)$ by hypothesis, these are all the substitutions that satisfy the condition.
- AS_1 We know that the terms in the kind-substitution have the adequate sort, so it is a substitution.
- AS_2 When one term in the kind-substitution has an incorrect sort the match fails.
- MbC_1 We know that the condition is fulfilled and θ binds all the variables, therefore it cannot be extended and the single substitution that verifies the condition is θ itself.
- MbC_2 Similarly to EqC_2, we know by hypothesis that the condition does not hold, thus there is no substitution able to satisfy it and the empty set of substitutions is computed.
- RlC In this case θ can be extended because rewrite conditions can contain new variables in their righthand side. We assume that S contains all the terms reachable from $\theta(t_1)$ that match the pattern t_2, and then use it to extend θ with all the substitutions θ' that bind the new variables in t_2 to match the terms in S, obtaining by definition all the substitutions that verify the condition.
- $\mathsf{SubsCond}$ We assume that, for each θ_i, $1 \leq i \leq n$, we obtain the set of substitutions S_i that extend $[C, \theta_i]$. By definition, $\langle C, \{\theta_1, \ldots, \theta_n\} \rangle$ computes the set of substitutions that extend any $[C, \theta_i]$, i.e., the union of the S_i, thus the inference is sound. □

Proposition 3. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and φ a judgment deduced with the inference rules Dsb, Rdc_2, or NTr from Fig. 5 from premises that hold in $\mathcal{T}_{\Sigma/E',R'}$. Then also $\mathcal{T}_{\Sigma/E',R'} \models \varphi$.*

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

- Dsb If the matching with the lefthand side and the conditions cannot be satisfied, then it is straightforward to see that the statement cannot be applied.
- Rdc_2 The substitution of a subterm by its normal form is correct if the normal form is correct.
- NTr Since the specification is confluent, we can use any equations to evolve a term and then compute the normal form from this new term. □

Proposition 4. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If $t \rightsquigarrow_0^{\mathcal{C}} S$ can be deduced using rules Rf_1 or Rf_2 from Fig. 6 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow_0^{\mathcal{C}} S$.*

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

- Rf_1 We know by hypothesis that the term t fulfills the condition thus, by definition, the set of reachable terms in zero steps is the singleton set with t as single element.
- Rf_2 In a similar way to the case above, if the condition does not hold with the term t, then the set of reachable terms is empty. □

Proposition 5. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, n a natural number, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If $t \rightsquigarrow_n^{\mathcal{C}} S$ or $t \Rrightarrow_1 S$ can be deduced by means of the rules in Fig. 7 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow_n^{\mathcal{C}} S$ or $\mathcal{T}_{\Sigma/E',R'} \models t \Rrightarrow_1 S$, respectively.*

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

- Tr_1 We know that the condition is fulfilled by t, that t in exactly one step is rewritten to the set $\{t_1, \ldots, t_k\}$, and that each of these terms is rewritten in at most n steps to $S_1, \ldots, S_k$. Since $\{t_1, \ldots, t_k\}$ have been obtained in one step, the terms in $S_1, \ldots, S_k$ have been computed in at most $n + 1$ steps and in at least 1 step. Since we are looking for the solutions in zero or more steps, we have to compute the union of these sets with the set of reachable terms in zero steps, that in this case is the singleton set containing the term t itself, because we are assuming it fulfills the condition. Thus, the inference is sound.
- Tr_2 Analogous to the case above.
- Stp We assume that all the possible rewrites in exactly one step at the top of $f(t_i)$, $0 \leq i \leq m$, lead to the set S_t and that all the reachable terms in exactly one step of each subterm t_i form the set S_i. By definition, all the reachable terms in exactly one step is the union of the set of all the terms obtained by rewrites at the top and the sets built by substituting each subterm by each reachable term from it (only one subterm is substituted at the same time), so the inference is sound.
- Red_1 Since we know that $t \to t_1$, by coherence the same reachable terms are obtained from t and t_1. Moreover, since $t_2 =_{E'} t'$ we can substitute t_2 by t' and the set remains unchanged. □

Proposition 6. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, n a natural number, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If a statement $t \rightsquigarrow !_n^{\mathcal{C}} S$ or $t \rightsquigarrow +_n^{\mathcal{C}} S$ can be deduced by means of the rules in Fig. 12 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow !_n^{\mathcal{C}} S$ or $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow +_n^{\mathcal{C}} S$, respectively.*

Proof.

- Rf_3 In this case we know that the term fulfills the condition and that it is final, so by definition the set of final reachable terms consists exactly of the term itself.
- Rf_4 If the term is final but it does not satisfy the condition, then the set of reachable states is empty by definition.
- Rf_5 If no more steps can be used and the term is not final, the set of reachable terms is empty by definition.
- Tr_3 We know that the term is not final, so we can split the search into two different searches, one in one step that leads to $\{t_1, \ldots, t_k\}$, and another in n steps from these terms, that we know generate the sets $S_1, \ldots, S_k$. Thus, the result is the union of these sets.
- Red_2 Analogous to Red_1 in Proposition 5.
- Rf_6 By definition the relation requires at least one step, thus if only zero steps are available the result is the empty set.
- Tr_4 First, we know that $t \to t'$, hence, by coherence, the same reachable terms are obtained from t and t'. Again, we distinguish the first step of the search, that leads to $\{t_1, \ldots, t_k\}$, and the next n steps. Since the terms in this second phase of the search have already evolved one step, the single requirement is to fulfill the condition, and thus the union of the sets obtained with the relation for zero or more steps has to be the result. □

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

1. *N corresponds to the consequence of an inference rule in the first column of Table 3.*
2. *The error associated to N can be obtained from the inference rule as shown in the second column of Table 3.*

Proof. The first item is a straightforward consequence of Propositions 1, 2, 3, 4, 5, and 6: N buggy means N invalid with all its children valid, and these are the only possible inference rules at N.

For the second property we study each inference rule separately:

- $\mathsf{Rep}_{\rightarrow}$ In this case the associated equation is wrong as a direct consequence of having a wrong statement instance: N is invalid in $\mathcal{I}$, while the previous conditions, which state the validity of the statements in the equation condition instance, correspond to the premises of the $\mathsf{Rep}_{\rightarrow}$ inference rule (see Fig. 1), which are valid in $\mathcal{I}$ because N is buggy.
- $\mathsf{Rep}_{\Rightarrow}$ and Mb Analogous to the case above.
- Rdc_1 In this case it is possible to have an erroneous result when the conditions hold. The reason is that the equation can be wrong, and thus we would have a wrong equation instance.
- Norm If the conclusion of this rule is erroneous but its premises hold this means that the specification does not have all the required equations, that is, an error in this node is associated with a missing equation.
- Ls Similarly to the case above, if the conclusion of this rule is wrong while its premises hold this means that the specification lacks some membership, that is, an error in this node is associated with a missing membership.
- $\mathsf{Fulfill}$ If this node is buggy then there exists a substitution that satisfies the condition but the condition should not hold, thus we have a wrong condition. In this case the condition in the buggy node is pointed out as the error in the specification.
- Fail In this case the set of substitutions that fulfill the condition is empty but the condition should hold, so the node is associated with a wrong condition. As in the case above, the error in the specification is related to the condition in the buggy node.
- Top When this node is buggy all the possible rules have been applied at the top and their results are correct, but the union of these terms does not lead to all the intended reachable terms by rewriting the term at the top, so this node is related to a missing rule. In this case, we will point to the operator at the top of the term in the lefthand side of the buggy node as incompletely defined.
- Rl The nodes computing the set of substitutions that fulfill the condition of the rule are correct, but once the righthand side of the rule is instantiated with these substitutions there are reachable terms in the intended interpretation that are not in this set. Thus, in this case the buggy node is associated with a wrong rule and the rule applied in the node is pointed out as buggy. □

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

(a) If T contains only one node, then $APT'(T) = \{T\}$.
(b) There is a $T' \in APT'(T)$ such that T' has an invalid root.

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

The second item can be proved by induction on the number of nodes of T, which we denote as $n(T)$. If $n(T) = 1$ the property is straightforward from the part (a) above because $T \in APT'(T)$. If $n(T) > 1$ we distinguish cases depending on the rule for APT' that can be applied at the root of T:

- If it is either $(\mathbf{APT}_2)$, $(\mathbf{APT}_3)$, $(\mathbf{APT}_4)$, $(\mathbf{APT}_5)$, $(\mathbf{APT}_6)$, $(\mathbf{APT}_7)$, $(\mathbf{APT}_8^m)$, $(\mathbf{APT}_9^m)$, or $(\mathbf{APT}_{10})$ the result holds directly because the result is a singleton set with the same invalid root (in the case of $(\mathbf{APT}_7)$ an equivalent root).
- If it is $(\mathbf{APT}_8^o)$, $(\mathbf{APT}_9^o)$, or $(\mathbf{APT}_{11})$ by Proposition 8 N has some invalid child, which corresponds to the root of some premise T_i. By the induction hypothesis, there is some $T' \in APT'(T_i)$ with invalid root. And by observing the rules of Fig. 13 it can be checked that every subtree T_i of the root of T verifies $APT'(T_i) \subseteq APT'(T)$. Then $T' \in APT'(T)$. □

Theorem 3. *Let T be a finite proof tree representing an inference in the calculus of Figs. 1, 4, 5, 6, 7, and 12 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root of T is invalid in $\mathcal{I}$. Then:*

- *$APT(T)$ contains at least one buggy node (completeness).*
- *Any buggy node in $APT(T)$ has an associated wrong statement, missing statement, or wrong condition in $\mathcal{R}$ according to Table 3 (correctness).*

Proof. We prove each item separately:

- $APT(T)$ contains at least one invalid node, since its root is the root of T, and any debugging tree containing an invalid node contains a buggy node by Proposition 7.
- First we observe that the root of $APT(T)$ cannot be buggy, because if it is invalid then it has an invalid child (Lemma 1(b)). Therefore any buggy node must be part of $APT'(T)$ (the premise in (**APT**$_1$)).
  Let N be a buggy node occurring in $APT'(T)$. Then N is the root of some tree T_N, subtree of some $T' \in APT'(T)$. By the structure of the APT' rules this means that there is a subtree T' of T such that $T_N \in APT'(T')$. We prove that N has an associated wrong statement in S by induction on the number of nodes of T', $n(T')$.
  If $n(T') = 1$ then T' contains only one node and $APT'(T') = \{T'\}$ by Lemma 1(a). Then the only possible buggy node is N, which means that N is also buggy in T and that the associated fragment of code is wrong by Proposition 8.
  If $n(T') > 1$ we examine the APT rule applied at the root of T':
  (**APT**$_2$) Then T' is of the form

$$\cfrac{\cfrac{T_1 \dots T_n}{e_1 \to e'}\text{Rep}_\to \qquad T''}{e_1 \to e_2}\text{Tr}_\to$$

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

$$\frac{APT'(T_1) \dots APT'(T_n)\; APT'(T'')}{e_1 \to e_2}\text{Rep}_\to$$

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

(**APT**$_3$) In this case T' has the form

$$\cfrac{\cfrac{T_1 \quad \dots \quad T_n}{t \to_{red} t''}\text{Rdc}_1 \qquad T}{t \to_{norm} t'}\text{NTr}$$

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

$$\frac{APT'(T_1) \quad \dots \quad APT'(T_n) \quad APT'(T)}{t \to_{norm} t'}\text{Rdc}_1$$

By Proposition 8 we know that N cannot be buggy in T', thus either $t \to_{red} t''$ or the root of T is invalid. However, if the root of T were invalid we know by Lemma 1 that the set obtained with APT' would contain a tree with an invalid root and then N cannot be buggy. Therefore, $t \to_{red} t''$ is invalid but, for the same reason as before, $T_1 \dots T_n$ cannot be invalid, so it is also buggy in T' and by Proposition 8 the rule label Rdc_1 has associated a wrong equation. Since this same label has been now assigned to N, the buggy node in the abbreviated proof tree has an associated wrong equation.

(**APT**$_4$) In this case T' has the form

$$\cfrac{\cfrac{T_1 \quad \dots \quad T_n}{t \Rightarrow^{top} S'}\text{Top} \qquad T'_1 \quad \dots \quad T'_n}{t \Rightarrow_1 S}\text{Stp}$$

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

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

By Proposition 5 we know that N cannot be buggy in T', thus either of $t \Rightarrow^{top} S'$ or the root of one of $T'_1 \dots T'_n$ is invalid. However, if the root of one of the trees $T'_1 \dots T'_n$ were invalid we know by Lemma 1 that the set obtained with APT' would contain a tree with an invalid root and then N cannot be buggy. Therefore, $t \Rightarrow^{top} S'$ is invalid but, for the same reason as before, $T_1 \dots T_n$ cannot be invalid, so it is also buggy in T' and by Proposition 8 the rule

label Top has associated a missing rule. Since this same label has been now assigned to N, the buggy node in the abbreviated proof tree has an associated missing rule.

(**APT**$_5$) **and** (**APT**$_6$) Analogous to the previous cases.

(**APT**$_7$) T' has the form

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

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

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

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

(**APT**$_8^o$), (**APT**$_9^o$), (**APT**$_{11}$) Then $T_N \in APT'(T_i)$ for some child subtree T_i of the root of T' and the result holds by the induction hypothesis.

(**APT**$_8^m$) We check that actually this rule cannot be applied to produce a buggy node and therefore must not be considered here. If (**APT**$_8^m$) is applied then T' must be of the form

$$\frac{T_1 \quad T_2}{e_1 \Rightarrow e_2} \mathsf{Tr}_{\Rightarrow}$$

N is $e_1 \Rightarrow e_2$ and T_N is

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

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

(**APT**$_9^m$) Analogous to the previous case.

(**APT**$_{10}$) We present the proof for the inference rule Fulfill, with the other cases being analogous. T' has the form

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

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

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

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

References

[1] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, S. Lucas, Abstract diagnosis of functional programs, in: M. Leuschel (Ed.), Logic Based Program Synthesis and Transformation, Lecture Notes in Computer Science, vol. 2664, Springer, 2002, pp. 1–16.
[2] A. Bouhoula, J.P. Jouannaud, J. Meseguer, Specification and proof in membership equational logic, Theoret. Comput. Sci. 236 (2000) 35–132.
[3] B. Braßel, The debugger B.I.O., 2008. Available from: <http://www-ps.informatik.uni-kiel.de/currywiki/tools/oracle_debugger>.
[4] R. Bruni, J. Meseguer, Semantic foundations for generalized rewrite theories, Theoret. Comput. Sci. 360 (2006) 386–414.
[5] R. Caballero, A declarative debugger of incorrect answers for constraint functional-logic programs, in: S. Antoy, M. Hanus (Eds.), Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, WCFLP 2005, Tallinn, Estonia, ACM Press, 2005, pp. 8–13.
[6] R. Caballero, C. Hermanns, H. Kuchen, Algorithmic debugging of Java programs, in: F.J. López-Fraguas (Ed.), Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain, Electronic Notes in Theoretical Computer Science, vol. 177, Elsevier, 2007, pp. 75–89.
[7] R. Caballero, N. Martí-Oliet, A. Riesco, A. Verdejo, Declarative debugging of membership equational logic specifications, in: P. Degano, R. De Nicola, J. Meseguer (Eds.), Concurrency, Graphs and Models. Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday, Lecture Notes in Computer Science, vol. 5065, Springer, 2008, pp. 174–193.
[8] R. Caballero, N. Martí-Oliet, A. Riesco, A. Verdejo, A declarative debugger for Maude functional modules, in: G. Roşu (Ed.), Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Electronic Notes in Theoretical Computer Science, vol. 2383, Elsevier, 2009, pp. 63–81.

[9] R. Caballero, M. Rodríguez-Artalejo, DDT: a declarative debugging tool for functional-logic languages, in: Y. Kameyama, P.J. Stuckey (Eds.), Proceedings of the Seventh International Symposium on Functional and Logic Programming, FLOPS 2004, Nara, Japan, Lecture Notes in Computer Science, vol. 2998, Springer, 2004, pp. 70–84.
[10] R. Caballero, M. Rodríguez-Artalejo, R. del Vado Vírseda, Declarative diagnosis of missing answers in constraint functional-logic programming, in: J. Garrigue, M.V. Hermenegildo (Eds.), Proceedings of the Nineth International Symposium on Functional and Logic Programming, FLOPS 2008, Ise, Japan, Lecture Notes in Computer Science, vol. 4989, Springer, 2008, pp. 305–321.
[11] O. Chitil, Y. Luo, Structure and properties of traces for functional programs, in: I. Mackie (Ed.), Proceedings of the Third International Workshop on Term Graph Rewriting, TERMGRAPH 2006, Electronic Notes in Theoretical Computer Science, vol. 176, Elsevier, 2007, pp. 39–63.
[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All about Maude: a high-performance logical framework, vol. 4350, Springer, 2007.
[13] M. Clavel, J. Meseguer, M. Palomino, Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic, Theoret. Comput. Sci. 373 (2007) 70–91.
[14] T. Davie, O. Chitil, Hat-Delta: one right does make a wrong, in: Seventh Symposium on Trends in Functional Programming, TFP 06.
[15] N. Dershowitz, J.P. Jouannaud, Rewrite systems, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, North-Holland, 1990, pp. 243–320.
[16] D. Insa, J. Silva, An algorithmic debugger for Java, in: M. Lanza, A. Marcus (Eds.), Proceedings of the 26th IEEE International Conference on Software Maintenance, ICSM 2010, IEEE Computer Society, 2010, pp. 1–6.
[17] D. Insa, J. Silva, A. Riesco, Balancing execution trees, in: V.M. Gulías, J. Silva, A. Villanueva (Eds.), Proceedings of the 10th Spanish Workshop on Programming Languages, PROLE 2010, Ibergarceta Publicaciones, 2010, pp. 129–142.
[18] J.W. Lloyd, Declarative error diagnosis, New Gener. Comput. 5 (1987) 133–154.
[19] W. Lux, Münster Curry User's Guide, 0.9.10 ed., 2006.
[20] I. MacLarty, Practical Declarative Debugging of Mercury Programs, Master's Thesis, University of Melbourne, 2005.
[21] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoret. Comput. Sci. 96 (1992) 73–155.
[22] J. Meseguer, Membership algebra as a logical framework for equational specification, in: F. Parisi-Presicce, Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT 1997, Tarquinia, Italy, June 3–7, 1997, Selected Papers, Lecture Notes in Computer Science, vol. 1376, Springer, 1998, pp. 18–61.
[23] L. Naish, Declarative diagnosis of missing answers, New Gener. Comput. 10 (1992) 255–286.
[24] L. Naish, A declarative debugging scheme, J. Funct. Logic Programming 1997 (1997).
[25] L. Naish, P.W. Dart, J. Zobel, The NU-Prolog debugging environment, Technical Report 88/31, Department of Computer Science, University of Melbourne, Australia, June 1989. Available from: <http://www.cs.mu.oz.au/lee/papers/nude/>.
[26] H. Nilsson, How to look busy while being as lazy as ever: the implementation of a lazy functional debugger, J. Funct. Programming 11 (2001) 629–671.
[27] B. Pope, Declarative debugging with Buddha, in: V. Vene, T. Uustalu (Eds.), Advanced Functional Programming – Fifth International School, AFP 2004, Lecture Notes in Computer Science, vol. 3622, Springer, 2005, pp. 273–308.
[28] B. Pope, A Declarative Debugger for Haskell, Ph.D. Thesis, The University of Melbourne, Australia, 2006.
[29] A. Riesco, A. Verdejo, R. Caballero, N. Martí-Oliet, A Declarative Debugger for Maude Specifications – User Guide, Technical Report SIC-7-09, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2009. Available from: <http://maude.sip.ucm.es/debugging>.
[30] A. Riesco, A. Verdejo, R. Caballero, N. Martí-Oliet, Declarative debugging of rewriting logic specifications, in: A. Corradini, U. Montanari (Eds.), Recent Trends in Algebraic Development Techniques, WADT 2008, Lecture Notes in Computer Science, vol. 5486, Springer, 2009, pp. 308–325.
[31] A. Riesco, A. Verdejo, N. Martí-Oliet, Declarative debugging of missing answers for Maude specifications, in: C. Lynch (Ed.), Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, Leibniz International Proceedings in Informatics, vol. 6, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 277–294.
[32] A. Riesco, A. Verdejo, N. Martí-Oliet, Enhancing the debugging of Maude specifications, in: P.C. Ölveczky (Ed.), Proceedings of the Eighth International Workshop on Rewriting Logic and its Applications, WRLA 2010, Lecture Notes in Computer Science, vol. 6381, Springer, 2010, pp. 226–242.
[33] A. Riesco, A. Verdejo, N. Martí-Oliet, A complete declarative debugger for Maude, in: M. Johnson, D. Pavlovic (Eds.), Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology, AMAST 2010, Lecture Notes in Computer Science, vol. 6486, Springer, 2011, pp. 216–225.
[34] A. Riesco, A. Verdejo, N. Martí-Oliet, R. Caballero, A declarative debugger for Maude, in: J. Meseguer, G. Roşu (Eds.), Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology, AMAST 2008, Lecture Notes in Computer Science, vol. 5140, Springer, 2008, pp. 116–121.
[35] E.Y. Shapiro, Algorithmic Program Debugging, ACM Distinguished Dissertation, MIT Press, 1983.
[36] J. Silva, A comparative study of algorithmic debugging strategies, in: G. Puebla (Ed.), Logic-Based Program Synthesis and Transformation, Lecture Notes in Computer Science, vol. 4407, Springer, 2007, pp. 143–159.
[37] J. Silva, Debugging Techniques for Declarative Languages: Profiling, Program Slicing, and Algorithmic Debugging, Ph.D. Thesis, Technical University of Valencia, June, 2007.
[38] A. Tessier, G. Ferrand, Declarative diagnosis in the CLP scheme, in: P. Deransart, M.V. Hermenegildo, J. Maluszynski (Eds.), Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl Project), Lecture Notes in Computer Science, vol. 1870, Springer, 2000, pp. 151–174.
[39] P. Viry, Equational rules for rewriting logic, Theoret. Comput. Sci. 285 (2002) 487–517.