Using Big-Step and Small-Step Semantics in Maude to Perform Declarative Debugging*

Adrián Riesco

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Madrid, Spain
ariesco@fdi.ucm.es

Abstract. Declarative debugging is a semi-automatic debugging technique that abstracts the execution details to focus on results. This technique builds a debugging tree representing an incorrect computation and traverses it by asking questions to the user until the error is found. In previous works we have presented a declarative debugger for Maude specifications. Besides a programming language, Maude is a semantic framework where several other languages can be specified. However, our declarative debugger is only able to find errors in Maude specifications, so it cannot find bugs on the programs written on the languages being specified. We study in this paper how to modify our declarative debugger to find this kind of errors when defining programming languages using big-step and small-step semantics, two generic approaches that allow to specify a wide range of languages in a natural way. We obtain our debugging trees by modifying the proof trees obtained from the semantic rules. We have extended our declarative debugger to deal with this kind of debugging, and we illustrate it with an example.

Keywords: Declarative debugging, Big-step semantics, Small-step semantics, Maude.

1 Introduction

Declarative debugging is a semi-automatic debugging technique that abstracts the execution details to focus on results. It has been widely used in logic [10,16], functional [11,12], multi-paradigm [2,8], and object-oriented [3] programming languages. The declarative debugging scheme consists of two steps: during the first one a tree representing the erroneous computation is built, while during the second one this tree is traversed by asking questions to an external oracle (usually the user) until the bug is found.

The operational semantics of a programming language can be defined in different ways [6]. One approach, called *big-step* or *evaluation semantics*, consists of defining how the final results are obtained. The complementary approach,

* Research supported by MICINN Spanish project *StrongSoft* (TIN2012-39391-C04-04) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

M. Codish and E. Sumii (Eds.): FLOPS 2014, LNCS 8475, pp. 52–68, 2014.

small-step or *computation semantics*, defines how each step in the computation is performed.

Maude [4] is a high-level language and high-performance system supporting both equational and rewriting logic computation. Maude modules correspond to specifications in *rewriting logic* [9], a logic that allows the representation of many models of concurrent and distributed systems. This logic is an extension of *membership equational logic* [1], an equational logic that, in addition to equations, allows the statement of *membership axioms* characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules that represent transitions in a concurrent system and can be nondeterministic.

In [18] (and in the extended version in [17]) the big-step and small-step semantics for several programming languages and its translation to Maude is presented. These papers show that big-step and small-step semantics can be easily specified in Maude by using a method called *transitions as rewrites*. This approach translates the inferences in the semantics into rewrite rules, that is, the lefthand side of the rule stands for the expression before being evaluated and the righthand side for the reached expression. The premises are specified analogously by using rewrite conditions.

Our declarative debugger for Maude specifications is presented in [14]. This debugger uses the standard calculus for rewriting logic[1] to build the debugging trees, which are used to locate bugs in equations, membership axioms, and rules. However, when a programming language is specified in Maude we cannot debug the language but only the semantics. That is, the previous version of our debugger could point out some rules as buggy (e.g. the rule in charge of executing functions) but not the specific constructs of the language being specified (e.g. the specific function going wrong in our program). We present here an improvement of this debugger to locate the user-defined functions responsible for the error when big-step or small-step semantics are used to define the programming languages. It is based on the fact these semantics contain a small number of rules that represent evaluation of functions in the specified language, so they can be isolated to extract the applied function, hence revealing an error in the program. Note that the information about the rules in charge of evaluating functions will be different in every semantics, and hence they must be provided by the user. We will see a more detailed example in the next section, showing the difference between these two kinds of debugging.

The rest of the paper is organized as follows: Section 2 describes the standard approach to big-step and small-step semantics and how to represent the semantics of programming languages in Maude following them. Section 3 develops the relation between the proof trees obtained with these approaches and the debugging trees used by our declarative debugger. Section 4 presents our declarative debugger by means of examples. Finally, Section 5 concludes and outlines some lines of future work.

[1] In fact, we extended the calculus to debug new kinds of errors. However, this extension of the debugger is not relevant for the present work.

Syntactic categories:

D in *Decl*	e in *Exp*	be in *BExp*	F in *FunVar*	
op in Op	bop in *BOp*	n in *Num*	x in *Var*	bx in *BVar*

Definitions:

$D ::= F(x_1, \ldots, x_k) \Longleftarrow e \mid F(x_1, \ldots, x_k) \Longleftarrow e, D, k \geq 0$
$op ::= + \mid - \mid * \mid div$
$bop ::= And \mid Or$
$e ::= n \mid x \mid e\ op\ e \mid let\ x\ =\ e\ in\ e \mid If\ be\ Then\ e\ Else\ e \mid F(e_1, \ldots, e_k), k \geq 0$
$be ::= bx \mid T \mid F \mid be\ bop\ be \mid Not\ be \mid Equal(e, e)$

Fig. 1. Syntax for Fpl

2 Preliminaries

We present in this section the basic notions used throughout the rest of the paper. First, we briefly describe a simple functional language and introduce its big-step and small-step semantics. Then, we present Maude and outline how to specify the semantics from the previous sections. The example in this section has been extracted from [6], while the translation to Maude follows [17]. Finally, we compare this approach with standard trace-debuggers.

2.1 Fpl, A Simple Functional Language

The Fpl language [6] is a simple functional language with arithmetic and Boolean expressions, let expressions, if conditions, and function definitions. The syntax for Fpl, presented in Figure 1, indicates that declarations are mappings, built with $\Longleftarrow$, between function definitions and their bodies; an expression can be either a number, a variable, two expressions combined with an arithmetic operator, a let expression, an if condition, or a function call; and a Boolean expression can be either a Boolean variable, true, false, two Boolean expressions combined with a Boolean operator, the negation of a Boolean expression, or an equality between two arithmetic expressions.

In order to execute programs, we will also need an environment ρ mapping variables to values. We will use the syntax $D, \rho \vdash e$ for our evaluations, indicating that the expression e is evaluated by using the definitions in D and the environment ρ. We explain now the different semantics to reach the final value.

Big-Step Semantics. Big-step semantics evaluates a term written in our Flp language to its final value, evaluating in the premises of each rule the auxiliary values. That is, this semantics will be used to infer judgements of the form $D, \rho \vdash e \Rightarrow_B v$, with D the function definitions, ρ an (initially empty) environment, e and expression, and v a value. For example, the rule for executing a function call is defined as follows:

$$(\mathsf{Fun_{BS}})\ \frac{D, \rho \vdash e_i \Rightarrow_B v_i \qquad D, \rho[v_i/x_i] \vdash e \Rightarrow_B v}{D, \rho \vdash F(e_1, \ldots, e_n) \Rightarrow_B v}$$

where $1 \leq i \leq n$ and $F(x_1, \ldots, x_n) \Longleftarrow e \in D$.

That is, the arguments are evaluated in the premises, and then the variables, obtained from the definition of the function on the function definitions, are bound to these values in order to evaluate the body of the function. The value thus obtained is the one returned by the rule.[2]

Small-Step Semantics. In contrast to big-step semantics, small-step semantics just try to represent each step performed by the program to reach the final value. We will use in this case judgements of the form $D, \rho \vdash e \Rightarrow_S e'$, and hence the expression may need several steps to reach its final value. For example, the rules for evaluating a function call with this semantics would be:

$$(\mathsf{Fun_{SS_1}})\ \frac{D, \rho \vdash e_i \Rightarrow_B e_i'}{D, \rho \vdash F(e_1, \ldots, e_i, \ldots, e_n) \Rightarrow_B F(e_1, \ldots, e_i', \ldots, e_n)}$$

$$(\mathsf{Fun_{SS_2}})\ \frac{}{D, \rho \vdash F(v_1, \ldots, v_n) \Rightarrow_B e[v_1/x_1, \ldots, v_n/x_n]}$$

where $F(x_1, \ldots, x_n) \Longleftarrow e \in D$.

That is, they first evaluate all the arguments to their final values and then continue by evaluating the body of the function. Note that this semantics just shows the result of applying one step; since this is very inconvenient for execution purposes, we will define in the next section its reflexive and transitive closure.

2.2 Maude

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

To specify our semantics in Maude we first define its syntax in the `SYNTAX` module. This module contains the sort definitions for all the categories:

[2] Note that we are using *call-by-value* parameter passing; a modification of the rule could also define the behavior for *call-by-name*.

```
sorts Var Num Op Exp BVar Boolean BOp BExp FunVar VarList NumList
      ExpList Prog Dec .
```

Note that we need to define lists explicitly. It also defines some subsorts, e.g. the one stating that a variable or a number are specific cases of expressions:

```
subsort Var Num < Exp .
```

Then it defines the syntax for each sort. For example, variables are defined by using the operator `V`, which receives a `Qid` (a term preceded by a quote) as argument; function names, of sort `FunVar`, are built by the operator `FV`, which also receives a `Qid`; operators are defined as constants; and let-expressions, if-expressions, and function calls are defined by using the operators below, where underscores are placeholders, `ctor` indicates that the operator is a constructor, and the `prec` attribute indicates its precedence:

```
op V : Qid -> Var [ctor] .
op FV : Qid -> FunVar [ctor] .
ops + - * : -> Op [ctor] .
op let_=_in_ : Var Exp Exp -> Exp [ctor prec 25] .
op If_Then_Else_ : BExp Exp Exp -> Exp [ctor prec 25] .
op _(_) : FunVar ExpList -> Exp [ctor prec 15] .
```

Once this module is defined, we have others that use equations to define the behavior of the basic operators, such as addition; another to define the environment (mapping variables to values); and another one for dealing with substitutions. These modules are required by the ones in charge of defining the semantics. Following the idea of *transitions as rewrites* [17] outlined in the introduction, we can specify inference rules by using conditional rules, being the body of the rule the conclusion of the inference and the conditions, written as rewrite conditions, the premises. In this way, we can write the ($\mathsf{Fun}_{\mathsf{BS}}$) rule as:

```
crl [FunBS] : D, ro |- FV(elist) => v
 if D, ro |- elist => vlist /\
    FV(xlist) <= e & D' := D /\
    D, ro[vlist / xlist] |- e => v .
```

that is, given the set of definitions `D`, the environment for variables `ro` and a function `FV` applied to a list of expressions `elist`, the function is evaluated to the value `v` if (i) the expressions `elist` are evaluated to the list of values `vlist`; (ii) the body of the function `FV` stored in `D` is `e` and the function parameters are `xlist`; and (iii) the body, where the variables in `xlist` are substituted by the values in `vlist`, is evaluated to `v`.

We define the small-step semantics in another module. The rule `FunSS1` indicates that, if the list of expressions applied to a function `FV` has not been evaluated to values yet, then we can take any of these expressions and replace it by a more evolved one:

```
crl [FunSS1] : D, rho |- FV(elist,e,elist') => FV(elist,e',elist')
 if D, rho |- e => e' .
```

The rule `FunSS2` indicates that, once all the expressions have been evaluated into values, we can look for the definition of `FV` in the set of definitions `D`, substitute the parameters by the given values, and then reduce the function to the body:

```
crl [FunSS2] : D, rho |- FV(vlist) => e[vlist / xlist]
 if FV(xlist)<= e & D' := D .
```

Moreover, we can also define the reflexive and transitive closure, which will be required to reach final values. These rules are defined by using a different operator `_|=_`, which distinguishes between single steps and the closure in order to avoid infinite computations. The rule `zero` indicates that a value is reduced to itself:

```
rl  [zero] : D, ro |= v => v .   *** no step
```

The rule `more` indicates that, if the expression `e` has not reached a value then we can first perform a small step to reach a newer expression `e'`, which will further evaluated using this reflexive and transitive closure until it reaches a value:

```
crl [more] : D, ro |= e => v
 if not (e :: Num) /\
    D, ro |- e => e' /\          *** one step
    D, ro |= e' => v .           *** all the rest
```

However, note that this distinction is only necessary from the executability point of view, and hence these rules can be understood as:

$$\text{(zero)}\ \frac{}{D, \rho \vdash v \Rightarrow_S v}$$

$$\text{(more)}\ \frac{D, \rho \vdash e \Rightarrow_S e' \qquad D, \rho \vdash e' \Rightarrow_S v}{D, \rho \vdash e \Rightarrow_S v}$$

Using any of these semantics we can execute programs written in our Fpl language. For example, we can define in a constant `exDec` the Fibonacci function and use a wrong addition function, which is implemented as "times":

```
eq exDec =
  FV('Fib)(V('x)) <= If Equal(V('x), 0) Then 0
                     Else If Equal(V('x), 1) Then 1
                          Else FV('Add)(FV('Fib)(V('x) - 1),
                                        FV('Fib)(V('x) - 2)) &
  FV('Add)(V('x), V('y)) <= V('x) * V('y) .
```

and use the big-step semantics to execute `Fib`(2), obtaining 0 as result:

```
Maude> (rew exDec, mt |= FV('Fib)(2) .)
rewrite in BIG-STEP : exDec, mt |- FV('Fib)(2)
result Num : 0
```

$$\dfrac{\dfrac{}{D, id \vdash 2 \Rightarrow_B 2}\,(\mathsf{CRN}) \quad \dfrac{\dfrac{\bigtriangledown_1 \quad \bigtriangledown_2}{D, \rho \vdash \texttt{x == 0} \Rightarrow_B \texttt{F}}\,(\mathsf{EqR2}) \quad \dfrac{\bigtriangledown_3 \quad \bigtriangledown_4}{D, \rho \vdash \texttt{If x == 1} \ldots \Rightarrow_B 0}\,(\mathsf{IfR2})}{D, \rho \vdash \texttt{If x == 0} \ldots \Rightarrow_B 0}\,(\mathsf{IfR2})}{D, id \vdash \texttt{Fib}(2) \Rightarrow_B 0}\,(\mathsf{FunBS})$$

where proof tree $\bigtriangledown_4$ is defined as:

$$\dfrac{\dfrac{\dfrac{\bigtriangledown \quad \bigtriangledown}{D, \rho \vdash \texttt{Fib}(\texttt{x} - 1) \Rightarrow_B 1}\,(\mathsf{FunBS}) \quad \dfrac{\bigtriangledown \quad \bigtriangledown}{D, \rho \vdash \texttt{Fib}(\texttt{x} - 2) \Rightarrow_B 0}\,(\mathsf{FunBS})}{D, \rho \vdash \texttt{Fib}(\texttt{x} - 1), \texttt{Fib}(\texttt{x} - 2) \Rightarrow_B 1, 0}\,(\mathsf{ExpLR}) \quad \bigtriangledown}{D, \rho \vdash \texttt{Add}(\texttt{Fib}(\texttt{x} - 1), \texttt{Fib}(\texttt{x} - 2)) \Rightarrow_B 0}\,(\mathsf{FunBS})$$

Fig. 2. Proof tree for `Fib(2)` evaluated by using big-step semantics

This result is erroneous, and its associated proof tree, partially depicted in Figure 2, has 37 nodes. In this figure D stands for the declarations shown above, ρ for $\texttt{x} \mapsto 2$, and we have simplified the syntax to improve the readability. The labels for the rules that we have not shown are straightforward: `CRN` evaluates a value to itself, `IfR2` evaluates an if statement when the condition is false, `EqR` evaluates an equality to false, and `ExpLR` evaluates a list of expressions. The tree $\bigtriangledown_1$ abbreviates the tree for the evaluation of `x` to 2, $\bigtriangledown_2$ the evaluation of 0 to itself, $\bigtriangledown_3$ evaluates `x == 1`, and the rest of $\bigtriangledown$'s just continue with the computation following the same ideas presented above.

Similarly, the evaluation of `Fib`(2) using small-step semantics returns the following result:

```
Maude> (rew exDec2, mt |= FV('Fib)(2) .)
rewrite in COMPUTATION : exDec2, mt |= FV('Fib)(2)
result Num : 0
```

The proof tree for this case is shown in Figure 3, where D stands for the definitions, e_1 for the definition of `Fib` applied to 2, and e_2 for this definition once we have substituted the condition by `F`. We start with a transitivity step, which has a function application as left child, which replaces the call to `Fib` by the body of the function. This value is used to keep looking for the final result with another `more` rule, which evaluates the condition in this `If` expression by means of an (IfR1) inference rule. It then continues with another `more` rule, where the $\bigtriangledown$'s stand for trees similar to the ones shown here.

If we try to use the previous version of our debugger to debug this problem it will indicate that the rule `FunBS` is buggy for the big-step semantics, while `FunSS2` will be pointed out as buggy for small-step semantics, although they are correctly defined. This happens because these rules are in charge of applying the functions (in this case `Fib` and `Add`) defined by the user, but they cannot distinguish between different calls. We will show how to improve the debugger to point out the specific user-defined function responsible for the error in the next sections.

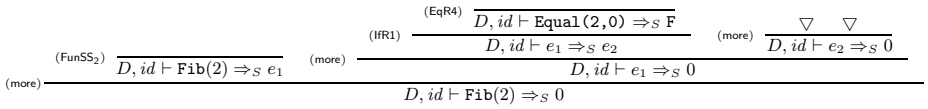

Fig. 3. Proof tree for `Fib(2)` evaluated by using small-step semantics

2.3 Related Work

We compare in this section our approach with the best known debugging method for any programming language: tracing with breakpoints. Although the trace is easy to use and highly customizable, a declarative debugger provides more clarity and simplicity of usage. In a trace-debugger, programmers must set some breakpoints where they want to stop the execution. From those points they can proceed step by step, checking whether the results of the functions or the arguments and bindings are the expected ones. If they skip the evaluation of a function but they discover it returns a wrong value, they have to restart the session to enter and debug its code. The advantage of the declarative debugger is that, starting only from an expression returning a wrong value, it finds a buggy function by simply asking about the results of the functions in the computation, avoiding low-level details. It focuses on the intended meaning of functions, which is something very clear to programmers (or debuggers), and the navigation strategies saves them from choosing what functions check and in what order as with breakpoints in the trace-debugger.

Moreover, note that throughout the paper we talk about functions as potential sources of errors because they depend on the user code and have an expected result (so we can make questions about them). However, we can modify the granularity of the errors discovered by the debugger by pointing out more specific rules as potentially erroneous. For example, we could consider that a loop in an imperative language is a "computational unit" and hence it has meaning by itself (as considered when enhancing declarative debugging in [7]). In this way, the debugger would ask questions related to loops (it could also ask questions about functions, if they are included). That is, we can configure the debugger to ask questions as specific as we want, so we could also examine the code inside functions, just as a trace-debugger.

Moreover, our approach is quite useful when prototyping a new programming language, since it would not have any debugging mechanism a priori. That is, the present work provides a debugger *for free* for any programming language specified in Maude, while the trace would only provide the execution of Maude equations and rules.

3 Declarative Debugging Using the Semantics

We present in this section the relation between the proof trees obtained by using the operational semantics in the previous section and the debugging trees that should be used to debug them.

3.1 Preliminaries

Declarative debugging requires an *intended interpretation*, which corresponds to the model the user had in mind while writing the program, to locate the bug. This interpretation depends on the programming language, and hence we cannot define it a priori. For this reason, we present the assumptions that must be fulfilled by the calculus and the information provided by the user before starting the debugging process:

- There is a set S of rules whose correctness depends on the code of the program being debugged. That is, we can distinguish between the inference rules executing the user code (e.g. the rules defining function call), which will be contained in S, and the rest of rules defining the operational details (e.g. the rules defining the execution order). If the inference rules are correctly implemented, only the execution of rules in S may lead to incorrect results.
- The user must provide this set, which will fix the granularity of the debugging process. The rest of the rules will be assumed to work as indicated by the semantics (i.e. no errors can be detected through them).
- The user knows the fragment of code being executed by each rule, assuming the premises are correct.

Example 1. The obvious candidate for the set S in our functional language is $S = \{(\mathsf{FUN}_{\mathsf{BS}})\}$ for big-step semantics (respectively, $S = \{(\mathsf{FUN}_{\mathsf{SS}_2})\}$ for small-step semantics). This rule is in charge of evaluating a function, and thus we can indicate that, when an error is found, the responsible is F, the name given in the rule to the function being evaluated.

Corollary 1. *As a consequence of the second restriction, given a calculus with a set of inference rules R, a set of rules S fulfilling the restrictions above, a set of rules $U \subseteq R$ such that $U \cap S = \emptyset$, a model $\mathcal{M}$ of the calculus, an intended interpretation $\mathcal{I}$, and a judgement j which is the consequence of any inference rule in U, we have $\mathcal{M} \models j \iff \mathcal{I} \models j$.*

We are interested in the judgements whose correctness may differ between the model and the intended interpretation. Given a model of the calculus, an intended interpretation $\mathcal{I}$, and a judgement j such that $\mathcal{M} \models j$ we will say that a judgement j is valid if $\mathcal{I} \models j$ and invalid otherwise. The basic declarative debugging scheme tries to locate a *buggy node*, that is, an invalid node with all its children valid. Regarding buggy nodes in the proof trees defined above, it is important to take into account the following property:

Proposition 1. *Let N be a buggy node in some proof tree in the given calculus, $\mathcal{I}$ an intended interpretation, and S the set of rules indicated by the user.*

1. *N corresponds to the consequence of an inference rule in S.*
2. *The error associated to N is the one indicated by the user to the rule in N.*

Proof. The first item is a straightforward consequence of Corollary 1. The second item is also straightforward from the third condition required on the set S.

3.2 Declarative Debugging with Big-Step Semantics

Instead of using the proof trees obtained from the calculus for declarative debugging, we will use an abbreviation to remove all the nodes that do not provide debugging information. We call this abbreviation APT_{bs}, from Abbreviated Proof Tree for big-step semantics. APT_{bs} is defined by using the set of rules S indicated by the user as follows:

$$
\begin{aligned}
APT_{bs}\left((\mathsf{R})\,\frac{T}{j}\right) &= (\mathsf{R})\frac{APT'_{bs}(T)}{j} \\
APT'_{bs}\left((\mathsf{R})\,\frac{T_1 \ldots T_n}{j}\right) &= \left\{(\mathsf{R})\frac{APT'_{bs}(T_1)\ \ldots\ APT'_{bs}(T_n)}{j}\right\},\ (\mathsf{R}) \in S \\
APT'_{bs}\left((\mathsf{R})\,\frac{T_1 \ldots T_n}{j}\right) &= APT'_{bs}(T_1) \bigcup \ldots \bigcup APT'_{bs}(T_n),\ (\mathsf{R}) \notin S
\end{aligned}
$$

The basic idea of the transformation is that we keep the initial evaluation and the evaluation performed by rules in S, while the rest of evaluations are removed from the tree. We show that this transformation is appropriate:

Theorem 1. *Let T be a finite proof tree representing an inference in the given calculus. Let $\mathcal{I}$ be an intended interpretation for this calculus such that the root N of T is invalid in $\mathcal{I}$. Then:*

(a) $APT_{bs}(T)$ contains at least one buggy node (completeness).
(b) Any buggy node in $APT_{bs}(T)$ has an associated error, according to the information given by the user.

Proof. We prove the items separately:

(a) By induction on the height of $APT_{bs}(T)$.
(Base case). If $height(APT_{bs}(T)) = 1$ then $APT_{bs}(T)$ only contains the root, which is invalid by hypothesis, and hence buggy.
(Inductive case). When $height(APT_{bs}(T)) = k$ then $APT_{bs}(T)$ contains a buggy node. If $height(APT_{bs}(T)) = k + 1$ we distinguish whether the root if buggy or not. If it is buggy then we have found the buggy node. Otherwise, it has at least one child node which is invalid and it contains a buggy node by hypothesis.
(b) Note first that the root is buggy iff $(\mathsf{R}) \in S$. In fact, it is easy to see that, by Proposition 2, if $(\mathsf{R}) \notin S$ then at least one of its child nodes is invalid in $\mathcal{I}$, and hence the APT'_{bs} function will contain an invalid tree, preventing the root from being buggy. Once this is stated, we realize that APT'_{bs} only keeps the inference rules in S and, since a buggy node has all its children valid and the user has assured that it can indicate the source of the error, the result holds.

where the auxiliary results are proved as follows:

Proposition 2. *Given a proof tree T, a set S of rules given by the user, an intended interpretation $\mathcal{I}$ such that the root N of T is invalid, and $\{T_1, \ldots, T_n\} = APT'_{bs}(T)$, then $\exists\ T_i$, $1 \leq i \leq n$, and N' the root of t_i such that $\mathcal{I} \not\models N'$.*

$$\dfrac{\dfrac{\dfrac{}{D,\rho \vdash \texttt{Fib(x}-1\texttt{)} \Rightarrow_B 1}\,(\mathsf{Fun_{BS}}) \quad \dfrac{}{D,\rho \vdash \texttt{Fib(x}-2\texttt{)} \Rightarrow_B 0}\,(\mathsf{Fun_{BS}})}{D,\rho \vdash \texttt{Add(Fib(x}-1\texttt{),Fib(x}-2\texttt{))} \Rightarrow_B 0}\,(\mathsf{Fun_{BS}})}{D, id \vdash \texttt{Fib}(2) \Rightarrow_B 0}\,(\mathsf{Fun_{BS}})$$

Fig. 4. Abbreviated proof tree for `Fib`(2)

Proof. We proceed by induction on the height of T.

Base case. If $height(T) = 1$ then, because of the restrictions on the set S, we have $(\mathsf{R}) \in S$, with (R) the rule used to infer N, and the result holds.

Inductive case. When $height(T) = k$ we have an invalid tree w.r.t. $\mathcal{I}$ in the set $APT'_{bs}(T)$. If $height(T) = k$, then we distinguish whether the rule (R) used in the root is in S. If it is, then the result holds trivially, since the transformation will keep this tree. Otherwise, we know that there is a child node that is invalid (since we restrictions in the set S prevent this node from being buggy), and we can use the induction hypothesis to check that the result holds.

Example 2. We can apply these rules, using the set S from Example 1, to the proof tree presented in Figure 2, obtaining the tree in Figure 4. Note that the 37 nodes in the proof tree have been reduced to 4 nodes in the abbreviated one.

The major weakness of big-step semantics when used for declarative debugging resides in the fact that evaluating terms whose subterms have not been fully reduced, as shown in Figure 4. This element makes the debugging process more complicated because it forces the user to think about the expected results for the subterms before considering whether the current computation is correct or not.

To solve this problem, we propose to use the single-stepping navigation strategy [15], which starts asking from the leaves, discarding the correct ones until an invalid one (and hence buggy, since leaves have no children) is found. This strategy allows us to substitute subterms by the appropriate values when asking questions, given this property is assured by the user:

Proposition 3. *Given $\mathcal{I} \models t \Rightarrow t'$, we have*

$$\mathcal{I} \models f(t_1, \ldots, t, \ldots, t_n) \Rightarrow r \iff \mathcal{I} \models f(t_1, \ldots, t', \ldots, t_n) \Rightarrow r$$

That is, the user must make sure that the semantics works, for the rules he has selected, by first reducing the subterms/arguments and then applying the rules for the reduced term. Note that these reduced terms are just a specific case of the ones indicated by the user for his intended semantics, and thus are included in $\mathcal{I}$. Since the structure of the tree does not change it is easy to see that completeness holds. Regarding soundness, it only holds if we traverse the tree by checking the correctness of the inferences for the subterms before checking the correctness of the whole term (otherwise we might discard the real source

$$\dfrac{\dfrac{\dfrac{}{(\diamondsuit)\ D,\rho \vdash \texttt{Fib}(1) \Rightarrow_B 1}(\mathsf{Fun_{BS}}) \quad \dfrac{}{(\heartsuit)\ D,\rho \vdash \texttt{Fib}(0) \Rightarrow_B 0}(\mathsf{Fun_{BS}})}{(\clubsuit)\ D,\rho \vdash \texttt{Add}(1,0) \Rightarrow_B 0}(\mathsf{Fun_{BS}})}{D, id \vdash \texttt{Fib}(2) \Rightarrow_B 0}(\mathsf{Fun_{BS}})$$

Fig. 5. Simplified abbreviated proof tree for `Fib`(2)

of the error). For this reason, we combine this transformation with the single-stepping navigation strategy [15], which performs exactly this traversal.

Using this simplification, we would obtain the tree in Figure 5, where all the subterms have been reduced. Although this change might seem trivial in this simple example, the benefits are substantial when more complex programs are debugged. Notice also that Proposition 3 may not hold in some cases, e.g. in lazy languages where the arguments are not evaluated until they are required. In this case we will follow the standard approach, asking about subterms not fully reduced and using the top-down or divide and query navigation strategies, which are more efficient in general that single-stepping.

3.3 Declarative Debugging with Small-Step Semantics

In contrast to the big-step semantics above, the small-step semantics applies a single evaluation step, making the debugging very similar to the step-by-step approach. To avoid this problem we place transitivity nodes in such a way that (i) the debugging tree becomes as balanced as possible, which improves the behavior of the navigation strategies and (ii) the questions in the debugging tree refer to final results, making the questions easier to answer. The tree transformation for this semantics, similar to the one in [14], takes advantage of transitivity rules while keeping the correctness and completeness of the technique. Thus, in this case the APT_{ss} function is defined as:

$$APT_{ss}\left((\mathsf{R})\ \frac{T}{j}\right) = (\mathsf{R})\frac{APT'_{ss}(T)}{j}$$

$$APT'_{ss}\left((\mathsf{Tr})\frac{(\mathsf{R})\ \dfrac{T_1 \ldots T_n}{j'}\ T}{j}\right) = \left\{(\mathsf{R})\frac{APT'_{ss}(T_1)\ldots APT'_{ss}(T_n)\ APT'_{ss}(T)}{j}\right\},\ (\mathsf{R}) \in S$$

$$APT'_{ss}\left((\mathsf{R})\ \frac{T_1 \ldots T_n}{j}\right) = \left\{(\mathsf{R})\frac{APT'_{ss}(T_1)\ldots APT'_{ss}(T_n)}{j}\right\},\ (\mathsf{R}) \in S$$

$$APT'_{ss}\left((\mathsf{R})\ \frac{T_1 \ldots T_n}{j}\right) = APT'_{ss}(T_1) \bigcup \ldots \bigcup APT'_{ss}(T_n),\ (\mathsf{R}) \notin S$$

That is, when we have a transitivity step whose left premise is a rule pointed out by the user, then we keep the "label" of the inference (which indicates that

we will locate the error in the lefthand side of this node, which is the same as the one in the premise) in the transitivity step, that presents the final value.

Theorem 2. *Let T be a finite proof tree representing an inference in the given calculus. Let $\mathcal{I}$ be an intended interpretation for this calculus such that the root N of T is invalid in $\mathcal{I}$. Then:*

(a) $APT_{ss}(T)$ contains at least one buggy node (completeness).
(b) Any buggy node in $APT_{ss}(T)$ has an associated error, according to the information given by the user.

Proof. We prove each item separately:

(a) Analogous to the proof for Theorem 1(a).
(b) We first use Proposition 4 below to check that, if the root is buggy, then it is associated to an inference rule $(\mathsf{R}) \in S$ and the result holds by using the assumed conditions on the rules of this set. Any other buggy node must be obtained from $APT'_{ss}(T)$.
Let N be a buggy node occurring in $APT'_{ss}(T)$. Then N is the root of some tree T_N, subtree of some $T' \in APT'_{ss}(T)$. By the structure of the APT'_{ss} rules this means that there is a subtree T' of T such that $T_N \in APT'_{ss}(T')$. We prove that the inference rule used to obtain N is in the set S by induction on the number of nodes of T', $|T'|$.
If $|T'| = 1$ then T' contains only one node and $APT'_{ss}(T') = \{T'\}$, due to the restrictions in the set S. Since only the second APT'_{ss} rule can be applied in this case the rule has been kept and it is associated to the error specified by the user.
If $|T'| > 1$ we examine the APT'_{ss} rule applied at the root of T':
- In the first case, T' is of the form

$$\dfrac{\dfrac{\dfrac{T_1 \quad \ldots \quad T_n}{j'}\,(\mathsf{R}) \qquad T''}{j}\,(\mathsf{Tr}),\ \text{with } (\mathsf{R}) \in S$$

Hence $N \equiv j$ and T_N is

$$\dfrac{APT'_{ss}(T_1) \quad \ldots \quad APT'_{ss}(T_n) \quad APT'_{ss}(T'')}{j}\,(\mathsf{R})$$

Since N is buggy in T_N it is invalid w.r.t. $\mathcal{I}$. However, a transitivity step cannot be buggy in T', since it does not depend on the code written by the user. For this reason either j' or the root of T'' are invalid. But the root of T'' cannot be buggy, since Proposition 4 indicates that the APT'_{ss} applied to a tree with an invalid root produces a set where at least one tree has an invalid root, which would prevent N from being buggy. Therefore j' is invalid. Moreover, the roots of $T_1 \ \ldots \ T_n$ must be valid, since otherwise N would not be buggy. Hence, j' is buggy in T' and the user is able to point out the error. Therefore, and given that in a transitivity rule the lefthand side of the root is the same as the lefthand side of the left premise, the user can identify the error in j.

- In the second case the conclusion of the inference is kept and, given that N is buggy, all the premises are valid and hence the user can identify the error.
- In the third case, $T_N \in APT'_{ss}(T_i)$ for some child subtree T_i of the root T' and the result holds by the induction hypothesis.

where the auxiliary results are proved as follows:

Proposition 4. *Given a proof tree T, a set S of rules given by the user, an intended interpretation $\mathcal{I}$ such that the root N of T is invalid, and $\{T_1, \ldots, T_n\} = APT'_{ss}(T)$, then $\exists\ T_i$, $1 \leq i \leq n$, and N' the root of t_i such that $\mathcal{I} \not\models N'$.*

Proof. Analogous to Proposition 2.

The tree obtained by using this abbreviation is equal to the one shown in Figure 5, although in this case the ($\mathsf{Fun_{BS}}$) inferences are ($\mathsf{Fun_{SS_2}}$) inferences obtained by applying the first equation for APT'_{SS}. Finally, note that, if the user does not want to use this simplification, we can still use the APT transformation for big-step and the result from Theorem 1 to perform the debugging in a "normal" tree, which will present a step-by-step-like debugging session.

4 Debugging Session

The debugger is started by loading the file `dd.maude` available at `http://maude.sip.ucm.es/debugging/`. It starts an input/output loop where commands can be introduced by enclosing them into parentheses.

Once we have introduced the modules specifying the semantics, we can introduce the set S rule by rule as follows:

```
Maude> (intended semantics FunBS culprit FV:FunVar .)
The rule FunBS has been added to the intended semantics.
If buggy, FV in the lefthand side will be pointed out as erroneous.
```

This command introduces the rule `FunBS` into the set S indicating that, when the buggy node is found, the responsible for the error will be the value matching the variable `FV`. Now we can select the single-stepping navigation strategy and start the debugging session for big-step, which reduces the subterms by default:

```
Maude> (single-stepping strategy .)
 Single-stepping strategy selected.
Maude> (debug big step semantics exDec, mt |- FV('Fib)(2) => 0 .)
Is D, V('x) = 2 |- FV('Fib)(1) evaluated to 1 ?
Maude> (yes .)
```

The first question (where we have replaced the definitions by `D` to simplify the presentation) corresponds to the node marked as ($\diamondsuit$) in Figure 5. Since we expected this result we have answered `yes`, so the subtree is removed and the next question corresponds to the node marked as ($\heartsuit$):

```
Is D, V('x) = 2 |- FV('Fib)(0) evaluated to 0 ?
Maude> (yes .)
```

This result was also expected, so we have answered `yes` again, and this subtree is also removed. The next question, which is related to the node marked as (♣), corresponds to an erroneous evaluation, so we answer `no`. Now, this node becomes an invalid node with all its children valid, and hence it reveals an error in the specification:

```
Is D, V('x) = 2 |- FV('Add)(1,0) evaluated to 1 ?
Maude> (no .)

The buggy node is:
The term D, V('x) = 2 |- FV('Add)(1, 0) has been evaluated to 0
The code responsible for the error is FV('Add)
```

That is, the debugger indicates that the function `Add` has been wrongly implemented in the Fpl language. We can debug the program in a similar way using the small-step semantics. In this case we have to introduce, in addition to the set of rules responsible for errors as shown above, the set of rules required for transitivity:

```
 Maude> (intended semantics FunSS2 culprit FV:FunVar .)
 The rule FunSS2 has been added to the intended semantics.
 If buggy, the FV:FunVar in the lefthand side will be pointed out
as erroneous.

 Maude> (transitivity rules more .)
 The rules more have been introduced as rules for transitivity.
```

We can now start the debugging session for small-step as follows:

```
 Maude> (debug small step semantics exDec2, mt |= FV('Fib)(2) => 0 .)
 Is D, mt |= FV('Add)(1, 0) evaluated to 0 ?
```

The answer is `no`, which allows the debugger to find the error in `Add`:

```
The buggy node is:
The term D, mt |= FV('Add)(1, 0) has been evaluated to 0
The code responsible for the error is FV('Add)
```

5 Concluding Remarks and Ongoing Work

We have presented in this paper a methodology to use declarative debugging on programming languages defined using big-step and small-step semantics in Maude. It uses the specific features of each semantics to improve the questions asked to the user. The big-step semantics can present terms with all the subterms in normal form, while the small-step semantics use the transitivity rule

to present the final results. This approach has been implemented in a Maude prototype extending the previous declarative debugger for Maude specifications. The major drawback of this approach consists in relying on the user for most of the results, that depend on the set of rules chosen for debugging. Although this is unfortunate, we consider it is necessary to build a tool as general as the one presented here.

Thus far we have checked the adequacy of the approach for small functional and imperative languages, so we also plan to study its behavior with more complex languages. Moreover, we want to extend our declarative debugger to work with K definitions. The K framework [5] is a rewrite-based executable semantic framework where programming languages and applications can be defined. However, K performs intermediate transformations to the rules defining the semantics and thus it is difficult to reason about them.

Another interesting subject of future work would consist of studying how declarative debugging works for languages with parallelism. We could use the search engine provided by Maude to look for paths leading to errors, and then use the path leading to the errors to build the debugging tree, thus providing a simple way to combine verification and debugging. We also plan to extend the possible answers in this kind of debugging. We are specifically interested in implementing a *trust* answer that removes from the tree all the subtrees rooted by the expression being trusted. Finally, a prototype for generating test cases based on the semantics specified in Maude has been proposed in [13]. It would be interesting to connect both tools, in order to debug the failed test cases.

References

1. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236, 35–132 (2000)
2. Caballero, R.: A declarative debugger of incorrect answers for constraint functional-logic programs. In: Antoy, S., Hanus, M. (eds.) Proc. of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, WCFLP 2005, Tallinn, Estonia, pp. 8–13. ACM Press (2005)
3. Caballero, R., Hermanns, C., Kuchen, H.: Algorithmic debugging of Java programs. In: López-Fraguas, F. (ed.) Proc. of the 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain. Electronic Notes in Theoretical Computer Science, vol. 177, pp. 75–89. Elsevier (2007)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
5. Şerbănuţă, T.F., Roşu, G.: K-Maude: A rewriting based tool for semantics of programming languages. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 104–122. Springer, Heidelberg (2010)
6. Hennessy, M.: The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics. John Wiley & Sons (1990)
7. Insa, D., Silva, J., Tomás, C.: Enhancing declarative debugging with loop expansion and tree compression. In: Albert, E. (ed.) LOPSTR 2012. LNCS, vol. 7844, pp. 71–88. Springer, Heidelberg (2013)

8. MacLarty, I.: Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne (2005)
9. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
10. Naish, L.: Declarative diagnosis of missing answers. New Generation Computing 10(3), 255–286 (1992)
11. Nilsson, H.: How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. Journal of Functional Programming 11(6), 629–671 (2001)
12. Pope, B.: A Declarative Debugger for Haskell. PhD thesis, The University of Melbourne, Australia (2006)
13. Riesco, A.: Using semantics specified in Maude to generate test cases. In: Roychoudhury, A., D'Souza, M. (eds.) ICTAC 2012. LNCS, vol. 7521, pp. 90–104. Springer, Heidelberg (2012)
14. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: Declarative debugging of rewriting logic specifications. Journal of Logic and Algebraic Programming 81(7-8), 851–897 (2012)
15. Shapiro, E.Y.: Algorithmic Program Debugging. ACM Distinguished Dissertation. MIT Press (1983)
16. Tessier, A., Ferrand, G.: Declarative diagnosis in the CLP scheme. In: Deransart, P., Małuszyński, J. (eds.) DiSCiPl 1999. LNCS, vol. 1870, pp. 151–174. Springer, Heidelberg (2000)
17. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. Technical Report 134-03, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid (2003)
18. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. Journal of Logic and Algebraic Programming 67, 226–293 (2006)