Using big-step and small-step semantics in Maude to perform declarative debugging

Adrián Riesco

Universidad Complutense de Madrid, Madrid, Spain

FLOPS 2014, Kanazawa
June 4, 2014

Preliminaries: Declarative debugging

- Declarative debugging is a semi-automatic debugging technique.
- It abstracts the execution details to focus on results.
- The declarative debugging scheme consists of two steps:
  - A tree representing the erroneous computation is built.
  - This tree is traversed by asking questions to an external oracle, usually the user, until the bug is found.
- The search finishes when the tool finds a wrong node with all its children correct, the buggy node.
- This node is associated with the erroneous source code.

Preliminaries: Operational semantics

- The operational semantics of a programming language can be defined in different ways.
- One approach, called big-step or evaluation semantics, consists of defining how the final results are obtained.
- The complementary approach, small-step or computation semantics, defines how each step in the computation is performed.

Preliminaries: Maude

- Maude is a high-level language and high-performance system supporting both equational and rewriting logic computation.
- Maude modules correspond to specifications in rewriting logic.
- This logic is an extension of membership equational logic, 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.

Preliminaries: Operational semantics in Maude

- Big-step and small-step semantics for several programming languages have been specified in Maude.
- These 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.
- 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.

Preliminaries: Operational semantics in Maude

- That is, the inference rule

$$r_1\,\dfrac{r_2\,\dfrac{}{f(a) \rightarrow b} \qquad r_3\,\dfrac{}{g(b) \rightarrow c}}{a \rightarrow c}$$

- would be represented in Maude as

```
crl [r1] : a => c
 if f(a) => b /\
    f(b) => c .
```

Preliminaries: Declarative debugging in Maude

- Our previous debugger uses the standard calculus for rewriting logic to build debugging trees.
- These trees are used to locate bugs in equations, membership axioms, and rules.
- However, when a programming language is specified in Maude we can only debug the semantics.
- That is, the previous version of our debugger could point out some rules as buggy but not the specific constructs of the language being specified.
- We present here an improvement of this debugger to locate the user-defined functions responsible for the error.
- It is based on the fact these semantics contain a small number of rules that represent evaluation of functions in the specified language.
- Hence they can be isolated to extract the applied function, hence revealing an error in the program.

Preliminaries: Fpl

Definitions:

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

- We will use a simple functional language, Fpl, to introduce big-step and small-step semantics.
- It has Boolean expressions, let expressions, if conditions, and function definitions.
- Function declarations are mappings, built with $\Longleftarrow$, between function definitions and their bodies.

Preliminaries: Fpl

- In order to execute programs, we also need an environment ρ mapping variables to values.
- We 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.

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.
- This semantics will be used to infer judgements of the form $D, \rho \vdash e \Rightarrow_B v$.
- For example, the rule for executing a function call is defined as follows:

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

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

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

Small-step semantics

- In contrast to big-step semantics, small-step semantics just try to represent each step.
- We 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:

$$(\mathsf{Fun}_{\mathsf{SS}_1})\ \frac{D, \rho \vdash e_i \Rightarrow_S e'_i}{D, \rho \vdash F(e_1, \ldots, e_i, \ldots, e_n) \Rightarrow_S F(e_1, \ldots, e'_i, \ldots, e_n)}$$

$$(\mathsf{Fun}_{\mathsf{SS}_2})\ \frac{}{D, \rho \vdash F(v_1, \ldots, v_n) \Rightarrow_S e[v_1/x_1, \ldots, v_n/x_n]}$$

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

- Note that this semantics just shows the result of applying one step.
- Since this is very inconvenient for execution purposes, we will define later its reflexive and transitive closure.

Maude

- Maude modules are executable rewriting logic specifications.
- We introduce Maude syntax by defining the semantics of Fpl.
- 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:

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

- 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 .
```

Maude

- We use operators to indicate how values of these sorts are built:

```
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;
    - to define the environment (mapping variables to values); and
    - deal with substitutions.
- These modules will be used by the one in charge of the semantics.

Maude

- Following the idea of transitions as rewrites, we specify inference rules by using conditional rules.
- In this way, we can write the (Fun_{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:

  1. the expressions `elist` are evaluated to the list of values `vlist`;
  2. the body of the function `FV` stored in `D` is `e` and the function parameters are `xlist`; and
  3. the body, where the variables in `xlist` are substituted by the values in `vlist`, is evaluated to `v`.

Maude

- 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, and then reduce the function to the body:

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

Maude

- We can also define the reflexive and transitive closure.
- These rules are defined by using a different operator _|=_.
- The rule zero indicates that a value is reduced to itself:

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

- The rule more is in charge of performing at least one step:

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

- This distinction is only necessary from the executability point of view.
- They can be understood as:

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

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

Maude

- 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) .
```

- If we execute `Fib(2)` using the big-step semantics, the following result is obtained:

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

Big-step tree

- The following tree represents this computation.

$$\dfrac{\dfrac{}{D, id \vdash 2 \Rightarrow_B 2}\;(\text{CRN}) \qquad \dfrac{\dfrac{\bigtriangledown 1 \quad \bigtriangledown 2}{D, \rho \vdash \mathtt{x\ ==\ 0} \Rightarrow_B \mathtt{F}}\;(\text{EqR2}) \qquad \dfrac{\bigtriangledown 3 \quad \bigtriangledown 4}{D, \rho \vdash \mathtt{If\ x\ ==\ 1} \ldots \Rightarrow_B 0}\;(\text{IfR2})}{D, \rho \vdash \mathtt{If\ x\ ==\ 0} \ldots \Rightarrow_B 0}\;(\text{IfR2})}{D, id \vdash \mathtt{Fib}(2) \Rightarrow_B 0}\;(\text{FunBS})$$

where proof tree $\bigtriangledown_4$ is defined as:

$$\dfrac{\dfrac{\dfrac{\bigtriangledown \quad \bigtriangledown}{D, \rho \vdash \mathtt{Fib}(\mathtt{x} - 1) \Rightarrow_B 1}\;(\text{FunBS}) \qquad \dfrac{\bigtriangledown \quad \bigtriangledown}{D, \rho \vdash \mathtt{Fib}(\mathtt{x} - 2) \Rightarrow_B 0}\;(\text{FunBS})}{D, \rho \vdash \mathtt{Fib}(\mathtt{x} - 1), \mathtt{Fib}(\mathtt{x} - 2) \Rightarrow_B 1, 0}\;(\text{ExpLR}) \qquad \bigtriangledown}{D, \rho \vdash \mathtt{Add}(\mathtt{Fib}(\mathtt{x} - 1), \mathtt{Fib}(\mathtt{x} - 2)) \Rightarrow_B 0}\;(\text{FunBS})$$

Small-step tree

- Similarly, the evaluation of $\texttt{Fib}(2)$ 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
```

- This computation is shown in the following tree:

$$
{\scriptstyle(\mathsf{more})}\dfrac{
{\scriptstyle(\mathsf{FunSS}_2)}\dfrac{}{D, id \vdash \texttt{Fib}(2) \Rightarrow_S e_1}
\quad
{\scriptstyle(\mathsf{more})}\dfrac{
{\scriptstyle(\mathsf{IfR1})}\dfrac{{\scriptstyle(\mathsf{EqR4})}\dfrac{}{D, id \vdash \texttt{Equal(2,0)} \Rightarrow_S \texttt{F}}}{D, id \vdash e_1 \Rightarrow_S e_2}
\quad
{\scriptstyle(\mathsf{more})}\dfrac{\bigtriangledown \quad \bigtriangledown}{D, id \vdash e_2 \Rightarrow_S 0}
}{D, id \vdash e_1 \Rightarrow_S 0}
}{D, id \vdash \texttt{Fib}(2) \Rightarrow_S 0}
$$

(Previous) Debugging results

- 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.
  - The rule `FunSS2` is buggy for the small-step semantics.
- However, they are correctly defined.
- This happens because these rules are in charge of applying the functions (`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.

Declarative debugging using the semantics

- Declarative debugging requires an intended interpretation, which corresponds to the model the user had in mind while writing the program.
- This interpretation depends on the programming language, and hence we cannot define it a priori.
- For this reason, we require some assumptions:

1. There is a set S of rules whose correctness depends on the code of the program being debugged.
    - We can distinguish between the inference rules executing the user code (e.g. function call), which will be contained in S, and the rest of rules defining the operational details (e.g. execution order).
    - If the inference rules are correctly implemented, only the execution of rules in S may lead to incorrect results.
2. 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.
3. The user knows the fragment of code being executed by each rule.

Declarative debugging using the semantics

Example

The obvious candidate for the set S in our functional language is $S = \{(\mathsf{FUN_{BS}})\}$ for big-step semantics (respectively, $S = \{(\mathsf{FUN_{SS_2}})\}$ 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.

Declarative debugging with big-step semantics

- We will use an abbreviation to remove all the nodes that do not provide debugging information.
- We call it 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:

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

- The basic idea is that we keep the initial evaluation and the evaluation performed by rules in S, while the rest of evaluations are removed.

Declarative debugging with big-step semantics

- The major weakness of big-step semantics resides in the fact that evaluating terms whose subterms have not been fully reduced.
- To solve this problem, we propose to use the single-stepping navigation strategy, 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.

Proposition

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

- The user must make sure that the semantics works, for the rules he has selected, by first reducing the subterms.

Declarative debugging with big-step semantics

- Using this simplification we obtain the following tree, where all the subterms have been reduced.

$$
(\mathsf{Fun_{BS}})\,\dfrac{(\mathsf{Fun_{BS}})\,\dfrac{(\mathsf{Fun_{BS}})\,\dfrac{}{D,\rho \vdash \mathtt{Fib}(1) \Rightarrow_B 1} \qquad (\mathsf{Fun_{BS}})\,\dfrac{}{D,\rho \vdash \mathtt{Fib}(0) \Rightarrow_B 0}}{D,\rho \vdash \mathtt{Add}(1,0) \Rightarrow_B 0}}{D, id \vdash \mathtt{Fib}(2) \Rightarrow_B 0}
$$

- Notice also that Proposition 1 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 standard navigation strategies.

Declarative debugging with small-step semantics

- 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
  1. The debugging tree becomes as balanced as possible.
  2. The questions in the debugging tree refer to final results.
- The tree transformation for this semantics takes advantage of transitivity:

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

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

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

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

Declarative debugging with small-step semantics

- 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 in the transitivity step, that presents the final value.
- This label indicates that we will locate the error in the lefthand side of this node, which is the same as the one in the premise.
- Using this transformation, we obtain the following tree:

$$\dfrac{\dfrac{\dfrac{}{D,\rho \vdash \texttt{Fib}(1) \Rightarrow_S 1}(\mathsf{Fun}_{SS_2}) \quad \dfrac{}{D,\rho \vdash \texttt{Fib}(0) \Rightarrow_S 0}(\mathsf{Fun}_{BS})}{D,\rho \vdash \texttt{Add}(1,0) \Rightarrow_S 0}(\mathsf{Fun}_{SS_2})}{D, id \vdash \texttt{Fib}(2) \Rightarrow_S 0}(\mathsf{Fun}_{SS_2})$$

Debugging session

- We have implemented this methodology in Maude.
- The debugger is started by loading the file `dd.maude`.
- 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.
```

Debugging session

- This command introduces the rule FunBS into the set S.
- It indicates 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 corresponds to the evaluation of Fib(1).
- Since we expected this result we have answered yes.

Debugging session

- The next question corresponds to the evaluation of Fib(0):

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

- This result was also expected, so we have answered yes again.
- The next question corresponds to an erroneous evaluation, so we answer no.
- The corresponding 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 0 ?
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.

Concluding remarks

- We have presented how 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.
- The big-step semantics can present terms with the subterms in normal form.
- The small-step semantics use the transitivity rule to present the final results.

Concluding remarks

- 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.

Ongoing work

- We also plan to study the behavior of the tool with more complex languages.
- We want to extend our declarative debugger to work with K definitions.
- The K framework 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.

Ongoing work

- 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.
- In this way we would provide 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 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 developed.
- It would be interesting to connect both tools, in order to debug the failed test cases.