Declarative debugging of missing answers in Maude specifications

Adrián Riesco Alberto Verdejo Narciso Martí-Oliet

Facultad de Informática, Universidad Complutense de Madrid, Spain

September 11, 2009

Motivation

- The Maude system supports several approaches for debugging Maude programs: tracing, term coloring, and using an internal debugger based on the trace.
- These techniques have as a disadvantage that they show to the user the application of equations without making explicit the relation between them.
- The user can loose the general view of the proof.
- Moreover, these debugging techniques can inform about a concrete computation, but cannot give details about other computations (e.g. if they are possible).

- Declarative debugging is a semi-automatic technique that starts from an incorrect computation (error symptom) and locates a program fragment responsible for the error.
- The declarative debugging scheme uses a debugging tree as a logical representation of the computation.
- Each node in the tree represents the result of a computation.
- This tree is navigated by asking questions to an external oracle.
- In previous works we developed a declarative debugger for wrong answers.
- We present now a declarative debugger for missing answers.

`http://maude.sip.ucm.es/debugging`

The Maude system

http://maude.cs.uiuc.edu

- 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 (MEL).
  - Maude functional modules correspond to specifications in MEL.
  - They specify equations, that must be confluent and terminating.
  - In addition to equations, they allow the statement of membership axioms characterizing the elements of a sort.
- Rewriting logic extends MEL by adding rewrite rules.
  - Rules have to be coherent with equations, but they are not required to be either confluent or terminating.
  - Maude system modules correspond to specifications in rewriting logic.

Example: System module

- Given a labyrinth, we want to obtain all the possible ways that allow us to reach the exit.

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

[1,5]                   [4,5] [5,5]
[1,6]                   [3,6] [5,6]
[1,7]                   [3,7] [5,7]
```

- First, we define the sorts `Pos`, `List`, and `State`:

```
(mod MAZE is
  protecting NAT .

  sorts Pos List State .
```

- Terms of sort `Pos` are just pairs of natural numbers:

```
  op [_,_] : Nat Nat -> Pos [ctor] .
```

- Since a particular case of list is the unitary list, we define `Pos` to be a subsort of `List`:

```
  subsort Pos < List .
```

- The constructors for `List` are the empty list and juxtaposition of lists:

```
op nil : -> List [ctor] .
op __ : List List -> List [ctor assoc id: nil] .
```

- Terms of sort `State` are lists enclosed by curly brackets.

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

- We assume a 6×7 labyrinth with the exit in the position `[6,7]`. The predicate `isSol` checks if a list is a solution:

```
vars X Y : Nat .
var P P' : Pos .
var L : List .

op isSol : List -> Bool .
eq [is1] : isSol(L [6,7]) = true .
eq [is2] : isSol(L) = false [owise] .
```

- The next position is computed with the rule `expand`, generating new positions with `next(L)` and checking them with `isOk`.

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

- The operation `next` is defined in a non-deterministic way:

```
op next : List -> Pos .

rl [next1] : next(L [X,Y]) => [X, Y + 1] .
rl [next2] : next(L [X,Y]) => [X + 1, Y] .
```

- `isOk(L P)` checks that the position is within the labyrinth, not repeated, and not part of the wall:

```
op isOk : List -> Bool .
eq isOk(L [X,Y]) = X >= 1 and Y >= 1 and X <= 6 and Y <= 7
                   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(P L, P') = if P == Q then true
                                        else contains(L, P) fi .
```

- Finally, we define the `wall` of the labyrinth:

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

- Now, we can rewrite the initial state `{[1,1]}` to see the behavior of the specification:

```
Maude> (rew {[1,1]} .)
rewrite in MAZE :
  {[1,1]}
result State :
  {[1,1][1,2][1,3][1,4][1,5][1,6][1,7][2,7][3,7][4,7][5,7][6,7]}
```

Wrong answers

- We use the inference rules of MEL and rewriting logic to build proof trees.
- These rules allow to deduce statements for reductions, memberships, and rewrites.
- An abbreviation of these trees, called APT, that reduces and simplifies the questions is used as debugging tree.
- We assume that the inference labels for replacements and memberships contain information about the particular statement applied during the inference.
- We assume the existence of an intended interpretation $\mathcal{I}$ of the given rewrite theory.
- The intended interpretation corresponds to the model that the user had in mind while writing the specification of the rewriting system.
- A statement $e \Rightarrow e'$ (respectively $e \longrightarrow e'$, $e : s$) is valid when it holds in the intended interpretation, and invalid otherwise.

Membership equational logic inference rules

Reflexivity

$$\frac{}{e \longrightarrow e}\ Rf_{\rightarrow}$$

Transitivity

$$\frac{e_1 \longrightarrow e' \qquad e' \longrightarrow e_2}{e_1 \longrightarrow e_2}\ Tr_{\rightarrow}$$

Congruence

$$\frac{e_1 \longrightarrow e_1' \quad \ldots \quad e_n \longrightarrow e_n'}{f(e_1, \ldots, e_n) \longrightarrow f(e_1', \ldots, e_n')}\ Cong_{\rightarrow}$$

Subject Reduction

$$\frac{e \longrightarrow e' \quad e' : s}{e : s}\ SRed$$

Replacement

$$\frac{\{\theta(u_i) \downarrow \theta(u_i')\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(e) \longrightarrow \theta(e')}\ Rep_{\rightarrow}$$

if $e \longrightarrow e' \Leftarrow \bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j$

Membership

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

if $e : s \Leftarrow \bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j$

Rewriting logic inference rules

Reflexivity

$$\frac{}{e \Rightarrow e}\ Rf_{\Rightarrow}$$

Transitivity

$$\frac{e_1 \Rightarrow e' \qquad e' \Rightarrow e_2}{e_1 \Rightarrow e_2}\ Tr_{\Rightarrow}$$

Congruence

$$\frac{e_1 \Rightarrow e'_1 \quad \ldots \quad e_n \Rightarrow e'_n}{f(e_1, \ldots, e_n) \Rightarrow f(e'_1, \ldots, e'_n)}\ Cong_{\Rightarrow}$$

Equivalence Class

$$\frac{e \longrightarrow e' \quad e' \Rightarrow e'' \quad e'' \longrightarrow e'''}{e \Rightarrow e'''}\ EC$$

Replacement

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \ \{\theta(v_j) : s_j\}_{j=1}^{m} \ \{\theta(w_k) \Rightarrow \theta(w'_k)\}_{k=1}^{l}}{\theta(e) \Rightarrow \theta(e')}\ Rep_{\Rightarrow}$$

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

Debugging the labyrinth

- We recall that, if we rewrite the state `{[1,1]}` in our labyrinth specification, we obtain a path that goes through the walls:

```
Maude> (rew {[1,1]} .)
rewrite in MAZE :
  {[1,1]}
result State :
  {[1,1][1,2][1,3][1,4][1,5][1,6][1,7][2,7][3,7][4,7][5,7][6,7]}
```

- Although we could debug this behavior, it is usually easier to debug smaller examples.
- We debug how the first encounter with the wall (position `[1,3]`) is reached.
- We start the debugging process with the following command:

```
Maude> (debug {[1,1]} =>* {[1,1][1,2][1,3]} .)
```

- With this command a one-step tree is built, that will be navigated with the default divide and query strategy.

- The first question asked by the debugger is:

```
Is this rewrite (associated with the rule expand) correct?

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

Maude> (no .)
```

- As we said, the position `[1,3]` contains part of the wall, and the position is not legal. The next question is:

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

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

Maude> (yes .)
```

- In this case the transition is correct because the list does not contain the position `[1,3]`

- The next questions are similar:

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

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

Maude> (no .)

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

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

Maude> (yes .)

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

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

Maude> (yes .)
```

- With this information, the debugger is able to infer the location of the bug:

```
The buggy node is:
contains([2,1][4,2][5,2][1,3][2,3][3,3][5,3][4,5][5,5]
         [3,6][5,6][3,7][5,7],[1,3]) -> false
with the associated equation: c2
```

- In fact, if we check the definition of the equation

```
eq [c2] : contains(P L, P') = if P == P' then true
                                else contains(L, P) fi .
```

should have P' instead of P in the recursive call, so we fix it as follows:

```
eq [c2] : contains(P L, P') = if P == P' then true
                                else contains(L, P') fi .
```

- Now that we have fixed the specification, we can look for the exit of the labyrinth by using the `search` command:

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

No solution.
```

Missing answers

- In a non-deterministic framework, a term can be rewritten to different terms.
- In this case, it is possible that not all the solutions expected by the user are reached.
- Each of these lost terms is a *missing answer*.
- The previous rules do not justify why a solution is not reached.
- We have defined a calculus that allows to infer, given a term, a condition, and a number of steps, the *complete* set of terms that satisfy the condition and are reachable in, at most, the given number of steps.
- That is, it explains why the terms in the set are in fact included in this set, and why the rest of terms are not included in the set.

Missing answers: Rules

$$\frac{fulfilled(\mathcal{C},t)\ \ t \Rightarrow_1 \{t_1,\ldots,t_k\}\ \ t_1 \rightsquigarrow_n^{\mathcal{C}} S_1\ \ \ldots\ \ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i \cup \{t\}}\ Tr_1$$

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

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

$$\frac{l := t \rightsquigarrow \Theta_0 \quad \langle C_1,\Theta_0\rangle \rightsquigarrow \Theta_1\ \cdots\ \langle C_k,\Theta_{k-1}\rangle \rightsquigarrow \Theta_k}{t \Rightarrow^{q} \bigcup_{\forall\,\theta \in \Theta_k} \{\theta(r)\}}\ Rl$$

if $q : l \Rightarrow r \Leftarrow C_1 \wedge \ldots \wedge C_k$

Missing answers: Rules II

$$\frac{t \to t_1 \quad t_1 \rightsquigarrow_n^{\mathcal{C}} \{t_2\} \cup S \quad t_2 \to t'}{t \rightsquigarrow_n^{\mathcal{C}} \{t'\} \cup S} \; Red_1$$

$$\frac{fails(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \emptyset} \; Rf_2$$

$$\frac{P := t \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0 \rangle \rightsquigarrow \Theta_1 \; \cdots \; \langle C_k, \Theta_{k-1} \rangle \rightsquigarrow \emptyset}{fails(\mathcal{C}, t)} \; Fail$$

if $\mathcal{C} \equiv P := \circledast \wedge C_1 \; \wedge \; \ldots \; \wedge \; C_k$

Missing answers: Rules III

$$\frac{\theta(t_1) \downarrow \theta(t_2)}{\theta(t_1 = t_2) \rightsquigarrow \{\theta\}} \; EqC_1$$

$$\frac{\theta(t_1) \rightarrow nf(\theta(t_1)) \quad \theta(t_2) \rightarrow nf(\theta(t_2))}{\theta(t_1 = t_2) \rightsquigarrow \emptyset} \; EqC_2 \quad \text{if } nf(\theta(t_1)) \not\equiv_A nf(\theta(t_2))$$

$$\frac{\theta(t_1) \rightsquigarrow_{n+1}^{t_2 := \circledast} S}{\theta(t_1 \Rightarrow t_2) \rightsquigarrow \{\theta'\theta \mid \theta'(\theta(t_2)) \in S\}} \; RlC$$

$$\text{if } n = min(x \in \mathbb{N} : \forall i \geq 0 \; (\theta(t_1) \rightsquigarrow_{x+i}^{\mathcal{C}} S))$$

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

- With this calculus, we can detect missing answers due to:
  - A wrong statement, that is, an error in the definition of an equation, a membership, or a rewrite rule.
  - An error in the formulation of the condition.
  - A missing rule.
- We do not use these trees directly, but an abbreviation that we call APT following the notation introduced for wrong modules.
- This transformation preserves the completeness and correctness of the debugging process while shortens and eases it.
  - Nodes related to inferences about sets of substitutions are removed.
  - In the same way, the APT associates the information about rewrites at top, that are deleted, to the rewrites in one step in any position.
  - It creates two different kinds of trees, one that contains inferences in several steps and another that only contains inferences in one step.
  - The APT reduces the time and space needed to compute the debugging tree.

$$(\mathbf{APT}_1) \quad APT\left(\frac{T_1 \ldots T_n}{af}R_1\right) = \frac{APT'(T_1) \ \ldots \ APT'(T_n)}{af}R_1$$

$$(\mathbf{APT}_2^o) \quad APT'\left(\frac{T_1 \ldots T_n}{af}Tr_i\right) = APT'(T_1) \ \ldots \ APT'(T_n)$$

$$(\mathbf{APT}_2^m) \quad APT'\left(\frac{T_1 \ldots T_n}{af}Tr_i\right) = \frac{APT'(T_1) \ \ldots \ APT'(T_n)}{af}Tr_i$$

$$(\mathbf{APT}_3) \quad APT'\left(\frac{\frac{T_1 \ \ldots \ T_n}{t \Rightarrow^{top} S'}Top \ T_1 \ldots Tm}{t \Rightarrow_1 S}Rep\right) = \left\{\frac{APT'(T_1) \ \ldots \ APT'(T_n) \ APT'(T_1) \ \ldots \ APT'(T_m)}{t \Rightarrow_1 S}Top\right\}$$

$$(\mathbf{APT}_4) \quad APT'\left(\frac{T \ \frac{T_1 \ldots T_n}{af'}R_1 \ T'}{af}Red_j\right) = \frac{APT'(T) \ APT'(T_1) \ \ldots \ APT'(T_n) \ APT'(T)}{af}R_1$$

$$(\mathbf{APT}_5) \quad APT'\left(\frac{T_1 \ldots T_n}{af}R_2\right) = \frac{APT'(T_1) \ \ldots \ APT'(T_n)}{af}R_2$$

$$(\mathbf{APT}_6) \quad APT'\left(\frac{T_1 \ldots T_n}{af}R_1\right) = APT'(T_1) \ \ldots \ APT'(T_n)$$

R_1 any inference rule R_2 *Fail*, *Fulfill*, *Ls*, *Top* or *Rl* $1 \leq i \leq 4$ $1 \leq j \leq 2$ af, af' any inference

Features: wrong answers

Since the debugging of missing answers uses the calculus of wrong answers, we enumerate first its main characteristics:

- The user can select a module containing only correct statements. By checking the correctness of the inferences with respect to this module the debugger can reduce the number of questions.
- Only labeled statements generate nodes in the debugging tree.
- Moreover, a subset of these statements can be selected, instead of using the whole set.
- Statements can be trusted "on the fly."
- Two different trees can be built, one for one-step rewrites and another for many steps rewrites.

Features: missing answers

The current version of our debugger has the following characteristics:

- It supports different kinds of searches:
  - In zero or more steps.
  - In one or more steps.
  - Of final terms.
- Like the tree for wrong answers, two different trees can be built, one for one-step searches and another for many steps searches.
- Once the tree has been calculated, it can be traversed with two different strategies: top-down and divide and query.
- It allows to shorten the debugging process by selecting terms that are final, i.e., that will never be rewritten:
  - Using the attribute `metadata "final"`.
  - Selecting the sorts with all their terms final.
  - The sort of a concrete term can be declared final "on the fly."

- Given a set of reachable solutions, we can
  - Decide if it is complete (debug missing answers).
  - Point out a term that is not reachable (switch to wrong answers).
  - Point out a reachable term that is reachable but it is not a solution (switch to wrong answers).
- The user can prioritize questions asking whether a term is a solution or not before questions about the correctness of the statements defining the condition.
  - In general, the user has in mind the terms he expects to be solutions, while deciding the correctness of concrete computations could require more reflection.
- The user can answer "don't know" to the questions, avoiding difficult questions but introducing incompleteness.
- The debugging tree is computed on demand, improving the required time and space.

Assumptions

The debugged modules are supposed to fulfill some requirements:

- Functional modules are expected to satisfy the executability requirements: specifications have to be terminating, confluent, and sort decreasing.
- Rules are assumed to be coherent with the equations.
- In order to obtain a nonempty abbreviated proof tree, the user must have labeled some statements; otherwise, everything is assumed to be correct.
- The buggy statement must be labeled in order to be found.
- When not all the statements are labeled, correctness and completeness are conditioned by the goodness of the labeling for which the user is responsible.
- The information provided by the correct module need not be complete, in the sense that some functions can be only partially defined.
- The `ctor` attribute has to be used in the constructor operators in order to define final sorts.
- The information supplied about final terms has to be accurate.

A debugging session

- The system we have described allows us to debug the maze example shown before.
- We recall that if we try to find the exit of a labyrinth, Maude was unable to reach it.

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

No solution.
```

- First, we prioritize the questions about the validity of solutions with the command

```
Maude> (solutions prioritized on .)

Solutions are prioritized.
```

- Then, we declare the sorts `Nat` and `List` to be final. Note that, since `Pos` is a subset of `List`, terms of this sort must also be final.

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

Final select is on.

Maude> (final select Nat List .)

Sorts List Nat are now final.
```

- We start the debugging process with the command

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

- With this command, the tool builds the abbreviated one-step proof tree, and the default divide and query traversal selects the following question

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

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

Maude> (yes .)
```

- Since the other possible steps are either part of the wall, or positions already used, or lead outside the labyrinth, this path is the only one possible and we answer `(yes .)`

- With this answer the subtree with this node as root is deleted and the next question are prompted:

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

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

Maude> (yes .)

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

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

Maude> (yes .)
```

- Again, the answers are `yes` because the other positions are not correct.

- The next question is related to reductions:

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

isOk([1,1][1,2][2,2][3,2][3,3]) -> false

Maude> (trust .)
```

- We have used the command `trust` for wrong answers to point out that all the reductions associated with the equation `ok` are correct.
- The next questions is:

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

Maude> (no .)
```

- Since we expected to reach the position `[3,1]`, the answer is `no`.

- The next questions are:

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

contains([1,1][1,2][2,2][3,2],[4,2]) -> false

Maude> (trust .)

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

1 [3,3]
2 [4,2]

Maude> (no .)
```

- While the first question is related again to equations, the second one shows all the terms reached in one step from the given term.
- Since we expected to reach the positions in the 4 possible directions, the answer is no.

- The next questions refer to the correction of the application of some concrete rules:

```
Are the following terms all the reachable terms from
next([1,1][1,2][2,2][3,2]) with one application of the rule next2 ?

1 [4,2]

Maude> (yes .)

Are the following terms all the reachable terms from
next([1,1][1,2][2,2][3,2]) with one application of the rule next1 ?

1 [3,3]

Maude> (yes .)
```

- These answers allow the debugger to find the buggy node, showing

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

The operator next is not completely defined.
```

- In fact, if we check the rules for `next` we realize that two rules specifying movements up and to the left and are missing:

```
rl [next3] : next(L [X,Y]) => [X, sd(Y, 1)] .
rl [next4] : next(L [X,Y]) => [sd(X, 1), Y] .
```

Top-down strategy

- It is also possible to use the top-down navigation strategy:

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

Top-down strategy selected.
```

- We can avoid questions about final term by activating 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:

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

Sorts List Nat are now final.
```

- We can also trust some statements, for example we consider that only the statements defining the condition are suspicious:

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

Debug select is on.

Maude> (debug select is1 is2 .)

Labels is1 is2 are now suspicious.
```

- Now we start the debugging process with the same command than before:

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

- The first questions are related to the validity of the children of the root:

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

Maude> (8 : no .)
```

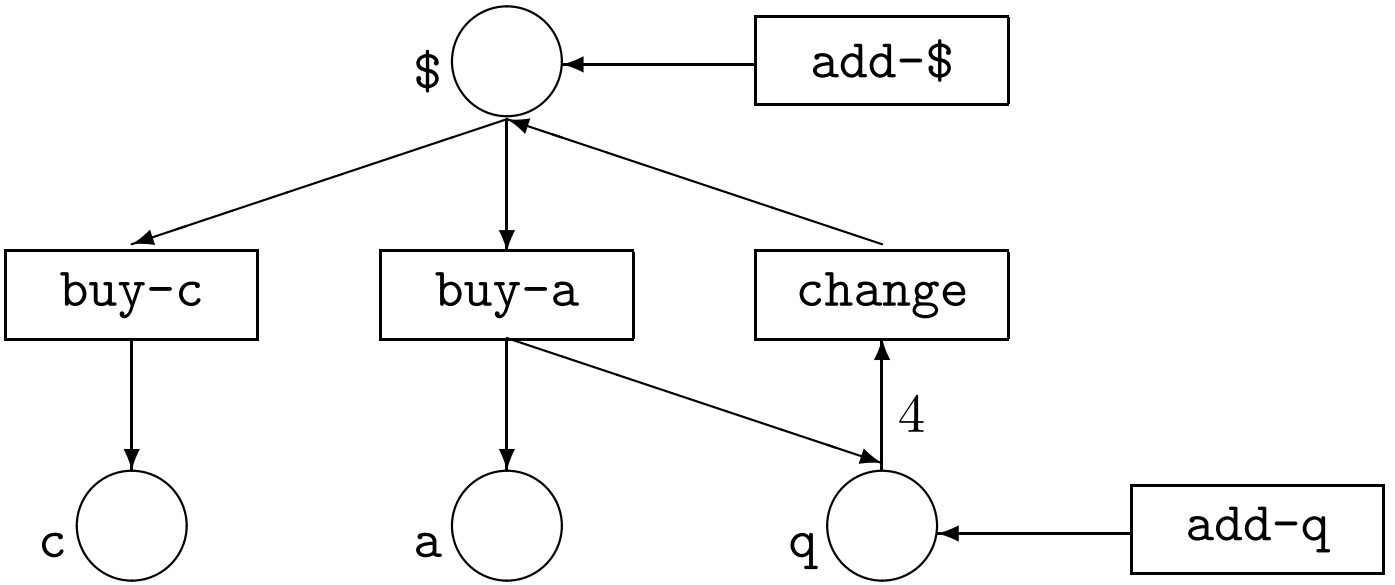
- The node associated to the 8th question is selected as current one and the next questions refer to the validity of its children:

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

1 [3,3]
2 [4,2]

Maude> (1 : no .)
```

- With these answers the debugger can detect the problem:

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

The operator next is not completely defined.
```

Example: Vending machine



- We specify the vending machine in the usual way

```
(mod VENDING-MACHINE is
 protecting NAT .
 sorts Coin Item Marking .
 subsorts Coin Item < Marking .

 ops $ q : -> Coin [format (r! o)] .
 ops a c : -> Item [format (b! o)] .

 op null : -> Marking .
 op __ : Marking Marking -> Marking [assoc comm id: null] .

 var M : Marking .

 rl [add-q] : M => M q .
 rl [add-$] : M => M $ .
 rl [buy-c] : $ => c .
 rl [buy-a] : $ => c q .
 rl [change] : q q q q => $ .
```

- Plus a function num$ that counts the number of dollars in a Marking

```
  op num$ : Marking -> Nat .
  eq [n$1] : num$($ M) = s(num$(M)) .
  eq [n$2] : num$(M) = 1 [owise] .
endm)
```

- If we try to find the terms with exactly two dollars in four steps, all the terms found have only one.
- We use the many steps strategy to debug the tree.

```
Maude>  (many-steps missing tree .)

Many-steps tree selected when debugging missing answers.

Maude> (missing [4] $ =>+ M:Marking s.t. num$(M:Marking) = 2 .)
```

- The first question prompted by the debugger is:

```
Are the following terms all the reachable solutions from
$ $ in at most 3 steps?

1  $ c c q q
2  $ c c q
3  $ c c
4  $ c q q q
5  $ c q q
6  $ c q
7  $ c

Maude> (1 is not a solution .)
```

- Although the set is clearly incomplete, we realize that the seven terms are not solutions. Thus, we answer that the first term is reachable but *it is not* a solution.

- With this information the debugger switches the current debugging tree to the tree justifying why this term is a solution, and asks

```
Is this reduction (associated with the equation n$2) correct?

num$(c c q q) -> 1

Maude> (no .)

The buggy node is:
num$(c c q q) -> 1
with the associated equation: n$2
```

- With just one more question the debugger is able to detect an error in the equations defining the condition. In this case, the function should return `0` instead of `1` if there are no dollars in the term.

- However, if we check if the program works correctly once this bug has been fixed, we realize that the terms containing apples are missing.
- Thus, we introduce the same debugging command and another debugging session starts.

```
Are the following terms all the reachable solutions from $ $
in at most 3 steps?

1  $ $ c q q
2  $ $ c q
3  $ $ c
4  $ $ q q q
5  $ $ q q
6  $ $ q
7  $ $

Maude> (no .)
```

- The next question is

```
Are the following terms all the reachable solutions from $ $ $
in at most 2 steps?

1  $ $ c q q
2  $ $ c q
3  $ $ c

Maude> (1 is wrong .)
```

- We could answer no because the set is incomplete, but we have decided to point out the first term as not reachable.
- In general, questions about wrong computations are easier and can lead to different bugs.

- Finally, with the following question we finish the debugging

```
Is this rewrite (associated with the rule buy-a) correct?

$ =>1 c q

Maude> (no .)

The buggy node is:
$ =>1 c q
with the associated rule: buy-a
```

- In this case, the cause of the missing answer was not a missing rule, but a wrong one.

Conclusions and ongoing work

- We have presented a declarative debugger that allows to debug wrong and missing answers in different ways.
- An important advantage of this kind of debuggers is the help provided in locating the buggy statements, assuming the user answers correctly the corresponding questions.
- To debug missing answers we have developed a calculus that allows to infer a set of terms justifying the presence of the terms in the set as well as the absence of all the other terms.
- Our debugging trees are an abbreviation of these trees, that shorten and ease the questions presented to the user.
- Since these trees have been obtained from a proof tree in a suitable semantic calculus, we can prove the correctness and completeness of the debugging technique.

- We plan to develop a calculus of "missing" answers for functional modules.
- We want to improve our prototype in order to detect errors due to errors in patterns in conditions with matchings.
- We also want to improve the "don't know" answer, in order to allow the user to answer the questions again if the tree reaches a state with insufficient information.
- Since one of the requirements of this kind of debuggers is the interaction with an oracle, that typically is the user, one of the principal aspects that must be improved is the user interface.

http://maude.sip.ucm.es/debugging