A complete declarative debugger for Maude

System demonstration

Adrián Riesco Alberto Verdejo Narciso Martí-Oliet

Facultad de Informática, Universidad Complutense de Madrid, Spain

June 24, 2010, Québec, AMAST 2010

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).
  - Sorts are grouped into equivalence classes called *kinds*.
  - 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.

Motivation

- 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.
- Since these trees have been obtained from a proof tree in a suitable semantic calculus, we can prove the correctness and completeness of the technique.

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

Declarative debugging & Maude

Symptoms detected:

- Wrongs answers: given an initial term we obtain either a wrong term (due to a reduction or a rewrite) or a wrong membership computation.
  $2 + 2 \rightarrow 5$
  $5 : Bool$
- Missing answers: in functional modules they correspond to not completely reduced normal forms and bigger than expected least sorts:
  $2 + 2 \rightarrow_{norm} 0 + 4$
  $5 :_{ls} Nat$ (should be $NzNat$)
  In system modules a missing answer is a term that should be reachable in at most n steps fulfilling some condition and that the system does not compute:
  $search\ coin \Rightarrow^* N\ s.t.\ true \rightsquigarrow \{0, 1\}$

Causes:

- Wrongs statements.
- Missing statements.
- Wrongs search conditions.

Questions

Reductions "Is this reduction correct? $t \longrightarrow t'$"

Normal forms "Is t in normal form?"

Memberships "Is this membership correct? $t : s$"

Least sorts "Did you expect t to have least sort ls?"

Rewrites in one step "Is this rewrite correct? $t \Rightarrow_1 t'$"

Rewrites in several steps "Is this rewrite correct? $t \Rightarrow^+ t'$"

Final terms "Did you expect t to be final?"

Solutions "Did you expect t to be a solution?"

Reachable terms in one step "Are the following terms all the reachable terms from t in one step? $t_1, \ldots, t_n$"

Reachable terms with one rule "Are the following terms all the reachable terms from t with one application of the rule r? $t_1, \ldots, t_n$"

Reachable terms in several steps

- "Are the following terms all the possible solutions from t in n steps? $t_1, \ldots, t_m$"
- "Are the following terms all the reachable terms from t that match the pattern p? $t_1, \ldots, t_m$"

Features

The main features of the debugger are:

- It debugs wrong reductions, sort inferences, and rewrites.
- It also debugs missing answers.
- The error attributed to those errors are wrong and missing statements, and wrong search conditions.
- The correctness of the inferences can be checked with respect to a correct module, reducing the number of questions.
- Moreover, a subset of these statements can be selected, instead of using the whole set.
- 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."

Features II

The main features of the debugger are:

- Statements can be trusted "on the fly."
- Two different trees can be built, one for one-step rewrites and another for many steps rewrites.
- Two different navigation strategies.
- The user can answer "don't know" to the questions, avoiding difficult questions but introducing incompleteness.
- The user can answer "undo" to return to the previous state.
- The debugging tree is computed on demand, improving the required time and space.
- Constructed terms are considered to be already in normal form.
- We provide a graphical user interface.

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.

Missing answers

- Traditionally, missing answers have been studied in nondeterministic contexts, where a term can be rewritten to different terms.
- In the Maude case they correspond to terms the user expected to reach from an initial one but the system cannot compute.
- It is possible to generalize the concept of missing answer to terminating and confluent frameworks, considering it indicates an *incomplete* result.
- In Maude, these errors correspond to correct reductions that do not reach the *normal form* and to correct sort inferences that do not compute the *least sort*.
- We have defined a calculus that allows to infer the normal form and the least sort of a term, and the complete set of reachable terms given a bound in the number of rewrites and the condition to be fulfilled.
- It explains why the term where reduced (respectively the sort inferred, the term included in the set) but also why it was not further reduced (respectively why it does not have a lesser sort, why a term is not included in the set).
- This calculus extends the calculus for wrong answers.

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.

DEMO

Conclusions

- We have presented a declarative debugger that allows to debug both wrong and missing answers.
- An important advantage of this kind of debuggers is the help provided in locating the buggy statements.
- We have developed a formal calculus that extends the standard calculus for rewriting logic, allowing to infer normal forms, least sorts and sets of reachable terms.
- To debug missing answers we have developed a calculus that allows to infer normal forms and least sorts.
- 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 semantic calculus, we have proved the correctness and completeness of the debugging technique.

Ongoing work

- We 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.
- We are implementing new navigation strategies to take into account the number of different potential errors in the subtrees, instead of their size.
- We plan to implement a test case generator, that will allow to test the correctness of a Maude specification and invoke the debugger in case something incorrect is found.

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