"ALEXANDRU IOAN CUZA" UNIVERSITY OF IAȘI
FACULTY OF COMPUTER SCIENCE

Program Equivalence : Relational Separation Logic interactive prover implemented in Maude

ANDREI-ALIN CORODESCU

Session: *July, 2018*

Scientific Coordinator
Conf. Dr. Ciobâcă Ștefan

Contents

1 Introduction

The present paper describes the development of an interactive tool for reasoning how to programs are related, based on studied and previously used theoretical concepts and technologies which facilitate the implementation.

The tool represents an implementation of Hoare Logic - which allows formal reasoning about a program - , along with 2 of its extensions, namely the Separation Logic (named Separation Logic from now on) and Relational Separation Logic [1] (named Relational Logic from now on). The 2 extensions simplify the Hoare Logic proofs, mainly using the "*" connector, allowing for local reasoning of effects of statements in a program . The tool has been implemented in Maude, a high performance logical framework with powerful metalanguage applications which facilitate the implementations of executable environments for logics.

The tool is built as a CLI which helps [2] [3] [4] [5] [6] [7] [8] [9] with reasoning how two programs are related using Relational Separation Logic specifications. As a consequence of the dependency of Relational Separation Logic on Separation Logic, proofs about single programs using the latter are also supported by the tool. The tool has been developed with extensibility in mind, the main desired extensions being concurrent programs support and automatic proofs.

The rest of the paper is organized as follows:

- **Section 1** will describe the main personal contributions to the realization of this project
- **Section 2** will present the problem this project aims to solve
- **Section 3** will shortly present some other projects related to ours
- **Section 4** will briefly touch upon the theoretical foundations of this project, together with the technologies used to implement them
- **Section 5**, the main section of this paper, will present our approach to solving the problem, discuss shortcomings of the current solution and how it can be improved in the future. All the elements of our project will be presented in an individual sub-section

2 Contributions

Personal contributions to the realization of the project :

- Modelled the Relational Logic and Separation Logic using Maude equational and rewriting logic specifications .
- Developed an interactive tool for reasoning about program behaviour using the aforementioned logics.
- Automation of some tasks which makes the tool more convenient to use .
- Examples of formal proofs done using the tool

3 Description of the problem

The problem this project is aiming to solve is related to formal reasoning about the execution of code, mainly focusing on how two programs are related to each other (most often the relation to be proven is equivalence)

Comparing programs or code fragments and studying their equivalence is part of every software engineer's activities when they are testing an alternative implementation for an existing solution, fixing bugs, launching new product versions, etc . Naturally, for every process completed manually there are efforts being made in order to make it more efficient, less error-prone and, in the end, automate the process all together. Once such a task is automated in software engineering, it can be included in the flow of any research or development phase of a product. An example benefiting from a formal proof of program equivalence is compiler optimization, where the optimized code needs to be equivalent to the input one .

This project aims to lay the foundations of a tool which facilitates formal reasoning on the way two programs are related with a focus on extensibility and automation of tasks.

4 Previous work

Previous work related to the topic has been done mostly in terms of Separation Logic based tools, with Relational Separation Logic not being treated as much.

A notable example which also uses the same framework as this project, namely Maude, is the Java+ITP[4] tool, which enables analysis of Java programs using Separation Logic. The implementation relies heavily on Maude's ITP (iterative theorem prover).

5 Theoretical Foundations and technologies

This section will briefly introduce the concepts on which the project is based, together with the technologies used to implement it, along with a few references to resources which explain them in depth.

5.1 Separation Logic

Separation Logic allows for reasoning about the effects of code on the program state in a formal manner. The main abstraction used in separation logic is the **Hoare Triple**

$$\{P\}\ C\ \{Q\}$$

where P denotes the a condition satisfied by state of the program before the execution of the command C and Q denotes another condition satisfied by state of the program after the execution of C.

Good references for reading on separation logic include :

- [2] - for a good introduction and some interesting usages
- [3] - for a more in depth explanation, including the mathematical fundaments of the logic

5.2 Relational Separation Logic

Relational Separation Logic, which is the central concept of our project builds upon the Separation Logic to reason about how to programs are related, by using the concept of a **Hoare Quadruple**

$$\{R\}\ \begin{matrix} C_1 \\ C_2 \end{matrix}\ \{T\}$$

R represents a **relation** between the two program states holding before the execution of commands C_1 and C_2 respectively while T represents a **relation** which holds after the execution of the two commands.

Relational Separation Logic is presented in detail in [1].

5.3 Maude

Maude is a framework based on rewriting logic which allows for natural representations of a wide range of applications, including other logics, which made it a perfect fit for our case. Maude allows for short and simple, yet clever and self-explanatory solutions to problems.

Good references for Maude language include:

- Maude Primer [2] which was written for Maude 2.0.1 but still applies to the latest versions of Maude is a good starting point for learning Maude, as it introduces the language in a more informal, friendlier way
- Maude Manual [7] which presents the whole Maude system in depth, along with the mathematical foundations
- All about Maude book [10] which includes everything contained in the manual and presents some of the more relevant tools implemented in Maude .

6 Relational Separation Logic Interactive Prover

This section will include details related to the implementation and usage of the Relational Separation Logic prover and highlight a few of Maude's features which made it the a language fit for the purpose of this project.

The information presented in this section assumes the reader's familiarity with Maude, especially the META-LEVEL module and with the underlying theoretical concepts implemented by the prover (Separation Logic and Relational Separation Logic)

6.1 Modelling the Separation Logics

Thanks to Maude's ability to naturally represent logics, the modelling of the Separation Logic and Relational Separation Logic is mostly reduced to translating the logical formulae into appropriate syntax for Maude, making the representational distance (the amount of information lost when representing

This figure will explain how we translated the proof rules into Maude syntax

a theoretical concept into a programming language)[7] almost non existent.

Each proof rule in the Separation Logic and Relational Separation Logic has been translated into a Maude rewrite rule.
The prover is centered around the concept of **Goals** which can be one of the three things:

- Hoare Quadruple - corresponding to Relational Separation Logic
- Hoare Triple - corresponding to Separation Logic
- Implication between relations / assertions

The proof rules are interpreted in a bottom-up manner, meaning that applying a rule to a goal matching the conclusion of a rule will generate new goals consisting of the hypothesis elements of the rule.

6.1.1 Sort Hierarchy

This section will describe the sorts defined in Maude to handle the constructs of both Separation logics.

6.2 Language Grammar and executable semantics

This section will describe how we defined our language in Maude [and how it is possible to create executable semantics for this language]

6.3 Interaction by Maude LOOP-MODE

Maude's LOOP-MODE module is the entry point for any interactive Maude application, being the only way to store state between commands. In this chapter we will discuss how we handle input and output , the structure of a state and the role every element plays in the execution of the prover.

We will make a distinction between **Loop state**, which specific to Maude's LOOP-MODE module and the **Prover state** which is specific to our use-case, and holds the necessary data related to the state of the proof we are currently working on. The **Prover state** is encapsulated within **Loop state**.

$$[\mathtt{INPUT},\ \mathtt{PROVER} - \mathtt{STATE},\ \mathtt{OUPTUT}]$$

Figure 1: Loop state

6.4 State of the program

Following the examples provided together with the Maude Manual [7], our state (Figure 2) is composed multiple sections modified accordingly during program execution .

- **Action** - represents the current action to be executed by the prover, more details about supported actions are included in the following sections.
- **Root Goal** - represents the end-goal of this demonstration
- **Current Goal** - the goal being treated currently - all actions will be applied to this goal
- **Goal Stack** - Stack of goals necessary to be proven in order to prove the root goal. The current goal is the last element popped from the stack and new elements are generated and pushed by the application of proof rules to the current goal.
- **Proven Goals** - List of goals which have been already been proven or axioms. Each goal will be first checked to see if it matches any goal in this list to prevent repeating the same proofs.
- **Staging output** - contains the output generated by the current action, before being passed to the Loop state. This staging mechanism enables each action (user or system) to generate output independent of the other actions.

$$< \mathtt{Action}, \mathtt{RootGoal}, \mathtt{CurrentGoal}, \mathtt{GoalStack}, \mathtt{ProvenGoals}, \mathtt{StagingOutput} >$$

Figure 2: Prover State

6.5 Input and output handling

To understand how our prover handles input and output, it is important to note that the flows of the LOOP-MODE module is as follows:

1. User input is placed in the `INPUT` member of the loop state
2. Maude engine applies all the rewritings possible, resulting in a new loop state.

   This is the step where all the logic of the prover goes into, in the form of rewriting rules which modify the prover state.
3. The member `OUTPUT` of this new loop state is taken and displayed at the console.

There are two main rules used for handling input and output :

1. `in` rule, which is responsible for parsing the user input and modifying the action in the prover state accordingly

```
crl [in] :
--- [ INPUT, <PROVER_STATE>, OUTPUT]
[QIL, < noRule ; RG ; G ; GS ; GL ; nil > , QIL']
=>
if T:ResultPair? :: ResultPair
--- If parsing succeeded, place the parsed action in
--- the prover state
then [nil, < downTerm(getTerm(T:ResultPair?), noAction) ; RG
    ; G ; GS ; GL ; nil >, QIL']
--- If the parsing failed, display an error
else [nil, < noRule ; RG ; G ; GS ; GL ; nil >, 'ERROR QIL]
fi
--- Try to parse the input as an term of sort Action
if QIL =/= nil /\ T:ResultPair? := metaParse(upModule('
    PROVER-GRAMMAR, false), QIL, 'Action) .
```

Listing 1: in rule

2. `out` rule, which takes the staging output from the prover state and appends it to the currently existing output in the loop state

```
crl [out] :
[QIL, < A ; RG ; G ; GS ; GL ;  QIL' >, QIL'']
=> [QIL, < A ; RG ; G ; GS ; GL ; nil >, QIL''  QIL']
if QIL' =/= nil .
```

Listing 2: out rule

6.6 Prover execution flow

In this section we will describe the flow of a demonstration made using our prover. A diagram presenting how the prover works is presented in Figure 3

1. The user issues the command `loop init .` in the Maude console. This will initialize the loop state.
2. Now the user will have to specify the goal it wants to prove, by issuing the command

   (`prove` G)

   , where G is the representation of the goal using the syntax defined for our logics. The command will populate the required fields of the prover state (**RootGoal**, **GoalStack**) with G.
   Please note that the command is enclosed within parentheses, this being a requirement of Maude LOOP-MODE [7]. All the prover specific commands need to be enclosed in parenthesis. The previous command, `loop init .` was not enclosed because it is a Maude specific command, not something we want to pass as input to our prover.
3. The next phase involves applying the logic proof rules to the goals in the **GoalStack** to generate sub-goals and prove them. This phase lasts until there are no more goals left in the **GoalStack**, meaning that the **RootGoal** has been proven.

Figure 3: Prover Execution Diagram

6.6.1 User Actions

User actions are the main entry point for user interaction and they describe how the user intends to process the current goal.

- Rule application action - indicates the application of a specific rule to the current goal, the action can be parametrized or not, depending on the rule. These actions are named following the pattern :

  [SL] + **apply** + *RuleName*

Figure 4: Examples of user actions which control the application of rules

With **SL** being added only when applying Separation Logic rules, to distinguish between the Relational Separation Logic rules with the same name.

- `ok` action - admits the current goal
- `showGoal` action - prints the current goal to the console - it does not modify the state of the prover

6.6.2 System Actions

System actions were implemented to mimic the behaviour of a "chain of responsibility" design pattern from object-oriented programming. The system actions are all applied in order before asking for user input. System actions are the main component through which our prover can be enriched with other features in the future, because integrating a new system action is simple. At the moment, the system actions are:

- `check` - marks the current goal to be checked against previously proven goals or axioms
- `implication` - marks the current goal to be run through the implication prover - in case the goal is an implication, try to prove it automatically
- `stop` - when there are no more sub-goals to be proven, the prover should stop and indicate the success of proving the root goal

6.7 Automated processes

This section will describe the functional parts of the prover which have been automated to make it more user-friendly and even prove sub-goals of the main demonstration by itself. Since the automation of the prover is part of the future goals of this project, we will also include a few notes on how we might approach problems related to this goal, even though they have not been implemented yet.

6.7.1 Automatic matching of axioms and previously proven goals

6.7.2 Automatic demonstration of implications

6.8 User Interface

Since our project aims to simplify the demonstrations of Separation Logic / Relational Separation Logic formulae, using an interactive prover, the user interface plays a major role in the success of the project - the easier it is to use and understand, the more chances of success it has. This idea has been presented in [2], when discussing key elements of tools related to program verification.

Maude provides a simple, yet powerful system for controlling the output formatting of the CLI through which we aim to make the demonstrations more readable and user-friendly.

For a more advanced user interface or to expose the prover's functionalities through a programming interface, a wrapper written in more feature-rich language through which to control the Maude environment is probably the best choice. The concept is exemplified the Maude plug-in for Eclipse, **citation needed** which enables writing of Maude code inside the Eclipse IDE, and the plug-in communicates with the underlying Maude environment.

6.9 Future directions

7 Conclusions

References

[1] Hongseok Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3):308–334, April 2007.

[2] Peter W. O'Hearn. A primer on separation logic (and automatic program verification and analysis). In *Software Safety and Security*, 2012.

[3] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[4] Ralf Sasse and José Meseguer. Java+itp: A verification tool based on hoare logic and algebraic semantics. *Electronic Notes in Theoretical Computer Science*, 176(4):29 – 46, 2007. Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006).

[5] Traian Florin Şerbănuţă, Grigore Roşu, and José Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305 – 340, 2009. Special issue on Structural Operational Semantics (SOS).

[6] Theodore McCombs. Maude 2.0 primer, August 2003.

[7] Manuel Clavel, Francisco Duran, Steven Eker, Santiago Escobar, Narciso Patrick Lincoln, Marti-Olietand Jose Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.7.1)*, July 2016.

[8] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73 – 155, 1992.

[9] James Brotherston, Nikos Gorogiannis, and Rasmus L Petersen. A generic cyclic theorem prover. In *Asian Symposium on Programming Languages and Systems*, pages 350–367. Springer, 2012.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag, Berlin, Heidelberg, 2007.