**"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

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 by persons 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 as a task is automated in software engineering, it can be included in the flow of any research or development phase. 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 .

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

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] argue 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 language supported by the tool and a brief description of the logics utilized for reasoning .

- Section 2 will describe in detail the building process of the tool and how

it can be used, with an accent on the features offered by the Maude language.

- Section 3 will present some of the research done on subjects related to the theme of the paper and discuss how those could be integrated into the current solution.

- Section 4 will present the conclusions of the paper, regarding both the theoretical and technical aspects of the paper.

# 2  Contributions

Personal contributions to the realization of the project :

- Modelled the Relational Logic and Separation Logic using Maude equational and rewriting logic specifications .

- Executable semantics of a simple programming language using Maude

- 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  Theoretical Foundations

In this section we will briefly present the theoretical aspects on which the project is based.

## 3.1  Language

The prover supports expressing programs in a simple, imperative language, commonly used throughout papers [3] [2] related to the subject of program verification, including the one describing Relational Separation Logic [1], around which this project is based.

### 3.1.1  Storage Model

A state of in our storage model is defined by a pair consisting of a *Store* and a *Heap* .

Assuming that all the variables usable in programs from the set `Vars` and the set of positive natural numbers is denoted by `PosNats`:

A *Store* is defined as:

$$S : \mathtt{Vars} \rightarrow \mathtt{PosNats}$$

A *Heap* represents a mapping from the `PosNats` to `Integers`

$$H : \mathtt{PosNats} \rightarrow \mathtt{Integers}$$

More informally, the *Store* holds the value of the variables while the *Heap* maps the active memory cells during a program execution to their contents.

### 3.1.2  Syntax and Semantics of the language

| | |
|---|---|
| Integer Expressions | $E ::= x \mid Integer \mid E \ \mathtt{plus} \ E \mid E \ \mathtt{times} \ E \mid E \ \mathtt{minus} \ E$ |
| Boolean Expressions | $B ::= false \mid true \mid B \ \&\& \ B \mid B \ \|\| \ B \mid B -> B \mid B <=> B \mid !B \mid E \ \mathtt{ge} \ E \mid E \ \mathtt{le} \ E \mid E \ \mathtt{eqs} \ E$ |
| Commands | $C ::= x := alloc(E) \mid x := [E] \mid [E] := E \mid free(E) \mid x := E \mid C; C \mid if \ B \ then \ C \ else \ C \mid while \ B \ do \ C \ od$ |

Figure 1: Syntax of the language

The syntax of the language is presented in Figure 1 . It represents an adapted subset of the language presented in the Relational Separation Logic paper [1] .

**Semantics of the language**   The semantics of the previously defined language are standard but a few clarifications are necessary to point out how our language constructs interact with the storage model, as well as a few differences from regular programming languages:

- Verbosity of operators - most operators are replaced by their literal names ( + becomes `plus` etc.); we have opted for this approach to avoid complication when implementing the language in Maude, because most of the operators are already defined in Maude for its built-in types, and overloading them could cause conflicts since we are basing our own defined types on Maude's primitives.

  $B$ `ge` $B$ translates to the usual $>=$ operator

  $B$ `le` $B$ translates to the usual $<=$ operator

  $B$ `eqs` $B$ translates to the usual $==$ equality comparison operator

  $B -> B$ denotes implication between boolean expressions

  $B <=> B$ denotes equivalence between boolean expressions

  The rest of the operators are mostly similar to their C++ or Java counterparts and their semantics are self explanatory

- Memory allocation / deallocation:

  $x := alloc(E)$ allocates a new cell in the memory, initializes it with the value of $E$ and stores its address in the variable $x$

  $x := free(E)$ deallocates the cell at the address equal to the value of $E$

- Working with variables and memory:

  $x := [E]$ reads the contents of the memory cell at address $E$ and stores the value in the variable $x$

  $[E] := E\prime$ updates the contents of the memory cell at address $E$ with the value of the expression $E\prime$

  $x := E$ updates the value of the variable $x$ with the value of the expression $E$ - note that this command does not modify the heap in any way, as it usually happens with regular programming languages when updating the value of a variable

## 3.2   Hoare Logic

Hoare Logic is a formal system which allows us to rigorously reason about the behaviour of programs. The main component of Hoare Logic is the Hoare Triple, defined as follows:

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

where $P$ denotes the state of the program before the execution of the command $C$ and $Q$ denotes the state of the program after the execution of $C$.

The role of a Hoare triple is to capture how a command or set of commands modify the state of a program in a formal manner.

$P$ and $Q$, called precondition and postcondition, respectively can be specified using an assertion language . Figure 2 represents the assertion language, defined in relation with the language syntax:

---

Assertions    $P ::= B \mid P \Rightarrow P \mid P \wedge P \mid P \vee P \mid \neg P \mid \exists x.P$

Figure 2: Syntax of the assertion language

---

At the base of proofs based on Hoare Logic are a set of axioms and proof rules which can be applied to Hoare triples, presented in Figure 3

## 3.3   Separation Logic

Separation Logic is an extension of Hoare Logic which simplifies reasoning about program execution by introducing ways of localising parts of the proof : draw conclusions based on a subset of the program state, the subset chosen such that it contains only the parts relevant to the command specified in the analysed Hoare triple.

The main feature of Separation Logic is the **separating conjunction** $*$ which allows to split the assertion into parts which refer to disjoint areas of the heap. Through the $*$ operator, Separation Logic enables reasoning about the effects of commands only at a local level, by splitting the assertions into a section which is relevant to the current command and "saving" the rest for subsequent commands.

Separation Logic introduces new elements to the assertions language - Figure 4:

Empty Statement Axiom

$$\overline{\{P\} \; \texttt{skip} \; \{P\}}$$

Assignment Axiom

$$\overline{\{P[E/x]\} \; \texttt{x := E} \; \{P\}}$$

Composition Rule

$$\frac{\{P\}S\{Q\} \quad \{Q\}T\{R\}}{\{P\} \; \texttt{S;T} \; \{Q\}}$$

Consequence Rule

$$\frac{P \Rightarrow P\prime \quad \{P\prime\}C\{Q\prime\} \quad Q \Rightarrow Q\prime}{\{P\} \; C\{Q\}}$$

Auxiliary Variable Elimination Rule

$$\frac{\{P\}C\{Q\}}{\{\exists x.P\} \; C \; \{\exists x.Q\}} \; x \notin Free(C)$$

Conditional Rule

$$\frac{\{B \wedge P\}S\{Q\} \quad \{\neg B \wedge P\}T\{Q\}}{\{P\}\texttt{if B then S else T fi}\{Q\}}$$

While Rule

$$\frac{\{P \wedge B\}S\{P\}}{\{P\} \; \texttt{while B do S od} \; \{\neg B \wedge P\}}$$

Figure 3: Hoare Logic Rules

**Semantics of the new elements**

- $E \mapsto E\prime$ means that the heap has an active cell at address $E$, with the content $E\prime$, and the cell is the **only** active cell in the heap

- $P * Q$ holds for heap $h$ iff $\exists h_0, h_1$ such that $h_0 * h_1 = h$ and $P$ holds for $h_0$ and $Q$ holds for $h_1$

- $emp$ denotes a heap without any active cells

7

$$\text{Assertions} \quad P ::= ...(specified\ previously) \mid E \mapsto E \mid P * P \mid emp$$

Figure 4: Syntax of the assertion language

**Inference Rules for Separation Logic**  Separation Logic introduces a new rule, called Frame Rule, which makes use of the $*$ operator to allow local reasoning :

Frame Rule

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ Modifies(C) \cup Free(R) = \emptyset$$

## 3.4   Relational Separation Logic

Relational Separation Logic builds upon the foundations layed by the Separation Logic by extending it to facilitate analysing how two programs are related.

**Hoare Quadruple**  The main abstraction introduced by the Relational Separation Logic is the Hoare Quadruple, which express the effects of commands on the states of the two programs.

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

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

Since relational separation language reasons about two programs simultaneously, it employs a storage model in which both programs share a single *Store* but have independent *Heaps*. This implies that the programs need to use **disjoint** parts of the same *Store*. The benefit of such a model is that we can use variables in Relations which aren't used by the two programs directly to relate them.

**Semantics of the Relation Language**  The semantics of the Relation Language are derived from the semantics of the Assertion Language, and in the following section I will present the newly introduced elements.

Relation    $R ::= Same \mid Emp \mid \begin{pmatrix} P_1 \\ P_2 \end{pmatrix} \mid R_1 * R_2 \mid B \mid R_1 \Rightarrow R_2 \mid R_1 \wedge R_2 \mid$
$R_1 \vee R_2 \mid \neg R \mid \exists x.R \mid \forall x.R$

Figure 5: Syntax of the relation language

- *Same* denotes that the heaps associated with the 2 programs are identical

- *Emp* is the Relational equivalent of the *emp* assertion; specifies two empty heaps

- $\begin{pmatrix} P_1 \\ P_2 \end{pmatrix}$ is the building block of the Relational separation logic: it allows to construct a relation between programs by putting together 2 assertions, $P_1$ holding for the first program and $P_2$ holding for the second program

- $R_1 * R_2$ represents the relational separating conjunction and it keeps a similar meaning to the Separation Logic one ; it separates two relations referring to **disjoint** parts of the heaps of the two programs.

Most of the rules are adapted from the Separation Logic ones, and the link between the two logics is given by the **Embedding Rule**, which allows to transform a Relational Separation Logic Hoare Quadruple into two Separation Logic Hoare Triples. The rules are presented in Figure 6.

Loop:

$$\frac{R \Rightarrow (B \Leftrightarrow B\prime) \qquad \{R \wedge B\}^{C}_{C\prime} R}{\{R\} \begin{smallmatrix} \texttt{while B do C od} \\ \texttt{while B\prime do C\prime od} \end{smallmatrix} \{R \wedge \neg B\}}$$

Conditional Statement

$$\frac{R \Rightarrow (B \Leftrightarrow B\prime) \qquad \{R \wedge B\}^{C_1}_{C_1\prime}\{S\} \qquad \{R \wedge \neg B\}^{C_2}_{C_2\prime}\{S\}}{\{R\} \begin{smallmatrix} \texttt{if B then C}_1 \texttt{ else C}_2 \texttt{ fi} \\ \texttt{if B\prime then C}_1\prime \texttt{ else C}_2\prime\texttt{fi} \end{smallmatrix} \{S\}}$$

Sequencing Rule

$$\frac{\{R\}^{C_1}_{C_1\prime}\{S\} \qquad \{S\}^{C_2}_{C_2\prime}\{T\}}{\{R\}^{C_1;C_2}_{C_1\prime;C_2\prime}\{T\}}$$

Frame Rule

$$\frac{\{R\}^{C}_{C\prime}\{S\}}{\{R * T\}^{C}_{C\prime}\{S * T\}}$$

Embedding rule

$$\frac{\{P\}C\{Q\} \qquad \{P\prime\}C\prime\{Q\prime\}}{\{\binom{P}{P\prime}\}^{C}_{C\prime}\left\{\binom{Q}{Q\prime}\right\}}$$

Consequence Rule

$$\frac{R \Rightarrow R_1 \qquad \{R_1\}^{C}_{C\prime}\{S_1\} \qquad S \Rightarrow S_1}{\{R\}^{C}_{C\prime}\{S\}}$$

Conjunction Rule

$$\frac{\{R_1\}^{C}_{C\prime}\{S_1\} \qquad \{R_2\}^{C}_{C\prime}\{S_2\}}{\{R_1 \wedge R_2\}^{C}_{C\prime}\{S_1 \wedge S_2\}}$$

Auxiliary Variable Elimination

$$\frac{\{R\}^{C}_{C\prime}\{S\}}{\{\exists x.R\}^{C}_{C\prime}\{\exists x.S\}}$$

Figure 6: Relational Separation Logic Proof Rules

## 3.5 Rewriting Logic

Rewriting logic is both a semantic and logical framework [10] in which, via **term rewriting**, changes within systems (concurrent or not) and other logics can be expressed naturally. The latter represents the motivation for choosing rewriting logic for our project, as well as extensibility in the future, in terms of concurrent program analysis.

One of the strong points of rewriting logic is that it reduces the **representational distance**, as it was named in [10], of the systems modelled using it. Representational distance is how much detail is lost in the translation process between semantic representation and implementation. This feature will be highlighted in the following sections, describing the Maude representation of the Relational Separation Logic and an executable semantic specification of the language supported by the prover. As we can see, the Relational Separation Logic has been represented naturally using rewrite rules and the translation process between representation and implementation usually involves just syntactic differences.

A rewrite theory is a triplet :

$$\mathcal{R} = (\Sigma, E, R)$$

where $\Sigma$ represent the "environment" - in our case, $\Sigma$ represents all the symbols which can appear when representing a Hoare Quadruple and their corresponding operators , $E$ represents a set of equations denoting algebraic identities for $\Sigma$ and $R$ represents a set of rewrite rules, possibly conditional which take the form of $t \leftarrow t\prime \ [condition]$, where $t$ and $t\prime$ are terms.

# 4 Relational Separation Logic Interactive Prover

This chapter 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.

## 4.1 Maude - General Overview

Maude is a programming language which is based on Rewriting Logic, discussed in the previous section. The main features used for this project which set Maude apart are :

- **Expressiveness**

  Maude allows the user to model a wide range of applications through user-defined data types and operators

  The principles upon which Maude was built make the implementations be close to the problem they are trying to solve. This forces programmers to come up with clever, often simple and self-explanatory solutions.

- **Simplicity**

  The previously mentioned expressiveness is augmented by the simplicity of the Maude language, which incorporates just a few keywords and commands,and the user is then tasked with building solutions using the underlying system (and, if desired, using pre-defined modules).

- **Metalanguage applications**

  Maude has a powerful, built-in way of analysing and controlling Maude's execution environment at runtime via the META-LEVEL module. Throughout this project we make heavy use of the meta language capabilities of Maude to control the flow of the prover execution. This approach also has the benefit of adding extensibility to it.

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 [11] which includes everything contained in the manual and presents some of the more relevant tools implemented in Maude .

# 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] José Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7):721 – 781, 2012. Rewriting Logic and its Applications.

[11] 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.

14