"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:

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

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

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

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	$E ::= x \mid Integer \mid E$ `plus` $E \mid E$ `times` $E \mid E$ `minus` E
Boolean Expressions	$B ::= false \mid true \mid B \;\&\&\; B \mid B \mid\mid B \mid B -> B \mid B <=> B \mid !B \mid E$ `ge` $E \mid E$ `le` $E \mid E$ `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

3.3 Separational Logic

3.4 Relational Separation Logic

3.5 Rewriting Logic

4 Relational Separation Logic Interactive Prover

Ordinea subsectiunilor mai poate fi schimbata ulterior. Capitolul cel mai semnificativ din lucrare, va contine toate detaliile de implementare ale solutiei curente

4.1 Maude - General Overview

4.2 Executable semantics of the language

4.3 Modelling the Separation Logics

4.4 Interaction by Maude LOOP-MAUDE

4.5 Prover execution flow

4.6 Automated processes

4.6.1 Automatic matching of axioms and previously proven goals

4.6.2 Automatic demonstration of implications

4.7 User Interface

4.8 Future directions

5 Additional Research

5.1 Automatic prover

5.2 Concurrent programs extension

5.3 Java+ITP

6 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.