ALEXANDRU IOAN CUZA UNIVERSITY OF IAȘI
FACULTY OF COMPUTER SCIENCE

Program Equivalence: An Interactive Relational Separation Logic Prover Implemented in Maude

ANDREI-ALIN CORODESCU

Session: *July, 2018*

Scientific Coordinator
Conf. Dr. Ciobâcă Ștefan

Contents

Abstract

Lucrarea de față descrie procesul de dezvoltare al unui utilitar interactiv utilizat în analiza formală a echivalenței programelor, folosindu-se de Logica Separatoare Relațională [1] și Logica Separatoare [2] [3], utilizând Maude [4] drept cadru de dezvoltare. Componenta interactivă se îmbină cu o componentă automatizată, prin care, părți din demonstrație vor fi realizate automat. Lucrarea este orientată pe rezolvarea problemei prin intermediul funcționalităților oferite de Maude și va include secvențe de cod ce vor ilustra conceptele cheie legate de modelarea și aplicarea fundamentelor teoretice utilizate pe parcursul proiectului.

Introduction

In the present paper I describe the development of an interactive tool for reasoning about how two programs are related, in a formal manner.

The tool is an implementation of Hoare Logic, which allows formal reasoning about a program, along with two of its extensions, namely the Separation Logic and Relational Separation Logic [1]. The two extensions simplify the Hoare Logic proofs, mainly using the "*" connector, allowing for local reasoning about the effects of statements in a program. I have implemented the tool in Maude, a high performance logical framework with powerful meta-language applications that facilitate the implementation of executable environments for logics.

The tool is built as a CLI that helps with reasoning about 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 paper is organized as follows:

- **Section 1** describes the main personal contributions to the realization of this project;
- **Section 2** presents the problem this project aims to solve;
- **Section 3** shortly presents some other projects related to this one;

- **Section 4** briefly touches upon the theoretical foundations of this project, together with the technologies used to implement them;
- **Section 5**, the main section of this paper, presents my approach to solving the problem, and discuss how it can be improved in the future.

1 Contributions

- Modelling the Relational Separation Logic and Separation Logic using Maude equational and rewriting logic specifications;
- Developing an interactive tool for reasoning about program behaviour using the aforementioned logics;
- Automation of tasks that make the tool more convenient to use;
- Example of formal proof performed using the tool - Appendix A

Throughout the development of the prover, I have learned how to use a different, unfamiliar technology, that is a good fit for modelling theoretical concepts, which were also new to me. This learning by applying process properly introduced me to both the technology and the logics involved in formally verifying program equivalence.

2 Description of the Problem

Comparing programs or code fragments and studying their equivalence is part of the activities of every software engineer, 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. These goals can be achieved by creating an easy to use and reliable formal method of verifying program equivalence.

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 [1].

The problem this project aims to solve is reducing the gap between theoretical and practical aspects of formal proofs for program equivalence, by creating an executable environment for Relational Separation Logic in which the user can prove the equivalence of programs. Through the creation of such a tool, the theoretical concepts become more accessible, easier to apply and understand. Making formal verification methods more accessible leads to an overall increase in the quality of software, by replacing the existing methods of verifying equivalence, that are often prone to errors, with more robust ones.

The difficult aspects of creating this tool include:

- Modelling the theoretical concepts in a programming language;
- Approaching computationally-hard problems required for Relational Separation Logic proofs;
- Understanding the complexities of Relational Separation Logic and creating an user experience that facilitates the application of the logic.

3 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[5] tool, which enables analysis of Java programs using Separation Logic. The implementation relies heavily on Maude's ITP (inductive theorem prover).

Several other tools are presented or referenced in [2].

4 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.

4.1 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 prover before the execution of the command C and Q denotes another condition satisfied by state of the prover 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.

4.2 Relational Separation Logic

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

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

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

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

4.3 Maude

Maude [4] is a framework based on rewriting logic [6] 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:

- The 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;
- The Maude Manual [7], which presents the whole Maude system in depth, along with the mathematical foundations;

- "All about Maude" book [8], which includes everything contained in the manual and presents some of the more relevant tools implemented in Maude.

5 Relational Separation Logic Interactive Prover

This section includes details related to the implementation and usage of the Relational Separation Logic prover and highlights a few of the features provided by Maude that made it a language fit for the purpose of this project.

The information presented in this section assumes a degree familiarity with Maude, especially the META-LEVEL module, and with the underlying theoretical concepts implemented by the prover (Separation Logic and Relational Separation Logic). References for these can be found in the previous section.

This section includes snippets of Maude code, but since Maude is a very expressive and self-explanatory language, I have not included a complete presentation of every language construct defined for the scope of this project. The ideas presented are accompanied by some examples and key features of the implementation. For a more detailed presentation, please refer to the code itself.

5.1 Language

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

5.1.1 Storage Model

A state of in our storage model is defined by a pair consisting of a `Store` and a `Heap`.
The set of program variables is `Vars`. 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` and is defined as:

$$\texttt{H} : \texttt{PosNats} \rightarrow \texttt{Integers}.$$

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

5.1.2 Syntax and Semantics of the language

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

Figure 1: Syntax of the language

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

The semantics of the operators 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.). I have opted for this approach to avoid syntactic 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 the primitives of Maude.

  B `ge` B translates to the usual $\geq$ operator

  B `le` B translates to the usual $\leq$ 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 self-explanatory.

- Memory allocation / deallocation:

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

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

  $x := E$ 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.

Maude Modelling of the Language: We define sorts corresponding to language elements and their associated operators:

- `AExp` for Integer expressions in the specification. To capture the first two possible forms of Integer expressions in Figure 1, the AExp sort is a super sort of `Int` in the standard `INT` module and `Id`, which is the sort defined for variables;

- `BExp` for the Boolean Expressions type in the specification. It is a super sort of `Bool` of standard module `BOOL`;

- `Command` for programs.

An example of a translation from specification to Maude code is presented in Listing 1. Note that we have opted to skip the attributes associated with the operators, as they are irrelevant for the purpose of this example.

```
op while_do_od : BExp Command -> Command [...] .
op _:=_ : Id AExp -> Command [...] .

```

Listing 1: Example of operator translation to Maude code

Executable semantics of the language, in Maude Maude also supports specifying executable semantics for the language using rewriting logic [9]. A proof of concept for a language similar to the one used by the prover is present in the code of the project (**simpleLanguage.maude**), but it is out of scope for the presentation of the prover. An executable environment for the programming language used in the prover is a valid candidate for future improvements.

5.2 Modelling the Separation Logics

Thanks to the ability of Maude to naturally represent logics, the modelling of 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 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 and assertions.

The proof rules are interpreted in a bottom-up manner, meaning that, applying a rule to a goal, which must match the conclusion of the used rule, will generate new goals consisting of the hypothesis: Listing 2 presents how a Separation Logic rule has been translated into Maude syntax and Listing 3 presents the same process for the Relational Separation Logic counterpart of the rule.

The Separation Logic rule of Consequence:

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

Is represented in Maude syntax as:

```
rl [SeparationLogic-Consequence] : {P} C {Q} => ((P => P1) <>
  ({P1} C {Q1})) <> (Q1 => Q) [nonexec] .
```

Listing 2: Separation Logic Consequence rule

The Relational Separation Logic rule of Consequence:

$$\frac{R \Rightarrow R_1 \qquad \{R_1\} \genfrac{}{}{0pt}{}{C}{C\prime} \{S_1\} \qquad S_1 \Rightarrow S}{\{R\} \genfrac{}{}{0pt}{}{C}{C\prime} \{S\}}$$

is represented in Maude syntax as:

```
rl [Consequence] : { R } C1 -- C2  { S } => ((R => R1) <> ({R1} C1 -- C2 {S1})) <> (S1 => S) [nonexec] .

```

Listing 3: Relational Separation Logic Consequence rule

The `<>` operator is used to separate sub-goals generated by the rule.
The `--` operator is used to separate the two snippets of code mentioned in the **Hoare Quadruple**.

All the rules are given the attribute `[nonexec]` because we want them to be applied in a controlled manner, at the meta-level. Some of the rules, including the examples, are not even valid executable Maude rules, because they make use of variables before binding them For example, in the Separation Logic rule of Consequence, `P1` is used in a rule before it is bound. At the meta-level, however we can manually bind values to those variables, as we will show when discussing how we apply these rules to goals.

5.2.1 Sort Hierarchy

Much in the same manner as we modelled the language grammar, Relational Separation Logic and Separation Logic have been modelled by defining sorts and declaring operators using these sorts. The main point of reference for our modelling has been [1], for both logics. Some examples of translations from specification to code are included below.

- `Assertion` is the sort used to handle Separation Logic assertions. It is a super sort of the `BExp` sort defined as part of our language;
- `HoareTriple` is the sort for Separation Logic **Hoare Triples** (Listing 4):

  ```
  op {_}_{_} : Assertion Command Assertion -> HoareTriple .
  ```

  Listing 4: Hoare Triple constructor operator

- `Relation` is the sort for Relational Separation Logic relations. As with `Assertion`, it is a super sort of the `BExp` sort of our language. Because both the `Relation` and `Assertion` are super sorts of the same sort, Maude will display a few warnings which are not affecting functionality, but in the future, creating a separate `BExp` sort for both the `Assertion` and `Relation` sorts would be a better option;

- `HoareQuadruple` denotes the concept with the same name, in the context of Relational Separation Logic .
  Listing 5 presents language constructs relevant for Relational Separation Logic.

```
--- Operator which creates a relation from two assertions
op [_;_] : Assertion Assertion -> Relation .
--- Operator which constructs a Hoare Quadruple
op {_}_--_{_} : Relation Command Command Relation ->
    HoareQuadruple .
```

Listing 5: Examples of Relational Separation Logic specific constructs

The other operators for the Assertion and Relation languages have been defined in the same manner as the programming language operators, following their presentation in [1].

5.3 Interaction by Maude LOOP-MODE

Our Relational Separation Logic prover is implemented as an interactive CLI program, based on the LOOP-MODE module of Maude, which is the entry point for any interactive Maude application. The LOOP-MODE module provides the only way to store a 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 is specific to Maude, 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** (see Figure 2).

5.4 State of the Prover

Following the examples in the Maude Manual [7], our prover state (Figure 2) is composed of multiple sections modified accordingly during the execution of the prover:

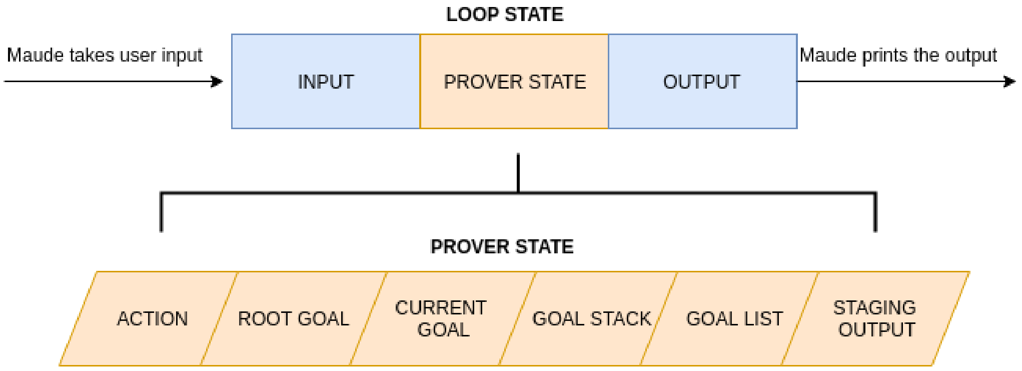


Figure 2: Loop and Prover State structure

- `Action` - represents the current action to be executed by the prover;
- `Root Goal` - represents the end-goal of the proof;
- `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 that have already been proven, or axioms. Each goal will first be 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 rewriting rule to generate output independent of the others.

5.5 Input and Output Handling

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

1. User input is placed in the `INPUT` member of the loop state;
2. Maude applies all the possible rewritings, resulting in a new loop state. All the logic of the prover is implemented here, in the form of rewriting rules which modify the prover state;

3. The `OUTPUT` member of the loop state is taken and displayed at the console.

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

1. The `in` rule (Listing 6), which is responsible for parsing 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 6: The `in` rule

2. The `out` rule (Listing 7), 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 7: The `out` rule

5.6 Prover Execution Flow

This section describes the flow of a Relational Separation Logic proof using our prover. The most important parts of the execution of the prover are controlled through the first member of a prover state, `Action`, which specifies the next step in the processing of the current goal. There are two types of actions: `System Actions` and `User Actions`. The diagram showing 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.

   This command will also populate the `ProvenGoals` list with axioms.

   Please note that the command is enclosed within parentheses, this being a requirement of the 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. Once a goal is popped from the `GoalStack`, it can be replaced by sub-goals generated by proof rules, matched with an proven goal / axiom, be proven automatically or be admitted manually. This phase lasts until there are no more goals left in the `GoalStack`, meaning that the `RootGoal` has been proven.

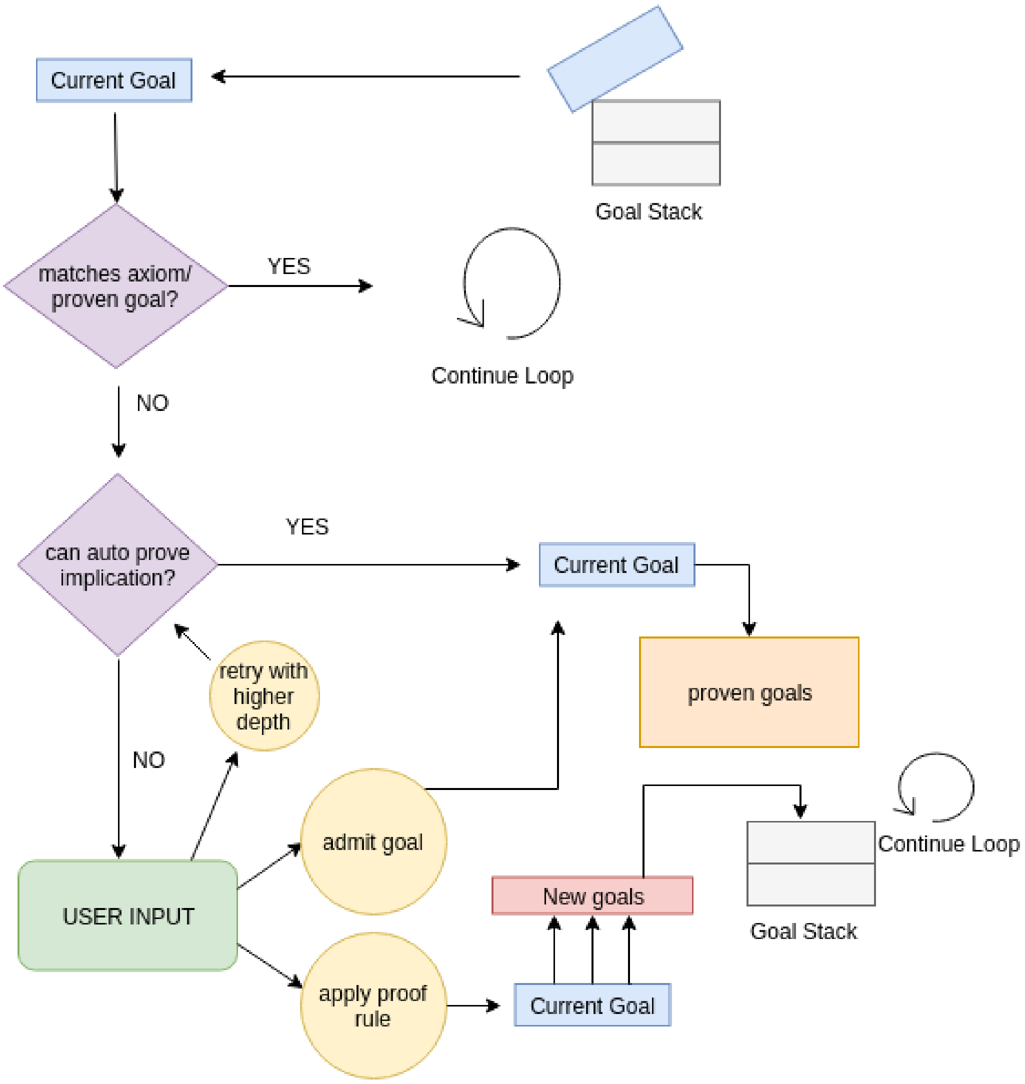


Figure 3: Prover Execution Diagram

5.6.1 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:

  $$[\texttt{SL}] + \texttt{apply} + \textit{RuleName},$$

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

```
1   crl [applyConsequence] : < applyConsequence R1 -- S1 ; RG ; G ; GS ; GL ; nil > =>
2    if SubGoals? :: Result4Tuple then
3    if downTerm(getTerm(SubGoals?), emptyGL) :: GoalList then
4    < noAction ; RG ; noGoal ; downTerm(getTerm(SubGoals?), emptyGL) <> GS ; GL ; 'applied ' 'consequence ' 'rule '\n >
5    else
6    < noAction ; RG ; G ; GS ; GL ; 'did ' 'not ' 'apply ' 'consequence ' 'rule '\n >
7    fi
8    else
9    < noAction ; RG ; G ; GS ; GL ; 'error ' 'consequence ' 'rule '\n >
10   fi
11  if SubGoals? := metaXapply(upModule('PROVER-INTERFACE, false), upTerm(G), 'Consequence, 'R1:Relation <- upTerm(R1) ; 'S1:Relation <- upTerm(S1) , 0, unbounded, 0).
```

Listing 8: Rule Application Action

The example given in Listing 8 illustrates all the core concepts involved in the application of proof rules to goals:

  - The rewrite rule is applied to a **Prover state** that has the Action field equal to `applyConsequence R1 -- S1`, where `R1` and `S1` are parameters to be passed to the proof rule;
  - The Consequence rule is applied at the meta level, using the `metaXapply` operator (line 11) from the `META-LEVEL` module. To this operator we also pass the values of the parameters (in their meta representation) with which to bind variables appearing in the `Consequence` rule;

  - The result is then checked for errors to see if the `metaXapply` operator has not been applied correctly (line 8), or if the term returned is not actually of the required type (line 3);
  - If the result corresponds to what the prover was expecting, we modify the state by adding the newly generated goals to the `GoalStack`, by moving down the meta-hierarchy - parsing meta-representation of terms;
  - The last thing to note is an example of how the rules generate output independently of others. In the last field of the `Prover State`, as `StagingOutput`, there will be a list of quoted identifiers which will eventually get moved to the loop state and then outputted to the user.

  All other actions which control the application of proof rules are written in a similar manner;

- `ok` action - admits the current goal;

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

- `auto x` action - instructs the prover to attempt to automatically prove an implication, by searching for a solution up to a depth of `x` rewrites. This command will do nothing if the current goal is not an implication. More details about the automated implication proving are included in the following section

5.6.2 System Actions

System actions 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. After a system action completes execution, it will decide the next action to be placed in the prover state. System actions are the main component through which our prover can be enriched with other features in the future. 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.

5.7 Automated Parts of the Relational Separation Logic Prover

This section describes 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.

5.7.1 Variable Substitution

In both sections describing automated processes we will interpret the equality operator between variables and replace, textually, one variable with the other in `Assertions`, `Relations` and `Commands`. The definition of the operator for `Assertions` is presented in Listing 9. The operator is defined recursively, with the base case being treated using the `owise` equational attribute. A similar approach is used for all the definitions of this operator.

```
op replace : Assertion Id Id -> Assertion .
eq replace(P * Q, X, Y) = replace(P, X, Y) * replace(Q, X , Y) .
eq replace(P /\ Q, X, Y) = replace(P, X, Y) /\ replace(Q, X, Y) .
eq replace(P \/ Q, X, Y) = replace(P, X, Y) \/ replace(Q, X, Y) .
eq replace(emp, X, Y) = emp .
eq replace(E |-> E2, X, Y) = replace(E, X, Y) |-> replace(E2, X, Y) .
eq replace(P, X, Y) = P [owise] .
```

Listing 9: Replace operator definition for assertions

5.7.2 Automatic Recognition of Axioms and Previously Proven Goals

The prover is capable of automatically recognizing goals matching an axiom or a previously proven goals, for a better user experience. It achieves this goal by using `metaXmatch` operator included in the standard `META-LEVEL` module, which determines if two terms match.

In terms of previously proven goals, the process is straight-forward, as it will compare terms in a literal manner (since they do not contain any variable).

Axioms are modelled as terms with variables in them, allowing them to be matched by any term for which all variables in the axiom can be bound. For example, the axiom

$$\{E \mapsto -\}[E] := F\{E \mapsto F\}.$$

is represented as a term of sort `HoareTriple`, but with free variables (Listing 10):

```
  {E:AExp |-> _ } [E:AExp] := F:AExp {E:AExp |-> F:AExp}
```

Listing 10: Example Axiom

However, Maude does not allow for variables to appear for the first time on the right hand side of a rewrite rule which needs to be used without meta-level computations (problem present when modelling Separation Logic as well). We had to find a workaround for this problem and the solution we came up with was to write the axiom in its meta-representation, and then use the `downTerm` operator to bring it down in the meta-hierarchy (Listing 11).

```
rl [proveGoal] : < prove(HQ) ; RG ; G ; GS ; GL ; nil > =>
< noAction ; HQ ; noGoal ; HQ <> emptyGL ;
downTerm('`{_`}_`{_`}['_|->_['E:AExp,'_:AExp],'`[_`]:=_['E:AExp,'F:AExp],'_|->_['E:AExp,'F:AExp]] , emptyGL)
--- ... [other axioms omitted for clarity]
; 'begin '\n > .
```

Listing 11: Initialization of Prover state

Before attempting to match terms we also interpret the equality between variables using the `replace` operator (Listing 13) introduced previously, to reduce the `Goal` to a simpler form, which might match to an axiom (Listing 12).

```
ceq anyMatches(G, G1 <> GL) = G1
---if G := G1 .
if metaXmatch(upModule('PROVER-INTERFACE, false),
upTerm(G1),
--- Try to reduce G as much as possible before matching
getTerm(metaRewrite(upModule('SYMBOLIC-EXECUTION, false), upTerm(G), unbounded)),
nil, 0, unbounded, 0) :: MatchPair .

```

Listing 12: Equation reducing the goal before attempting to match it

```
rl {P * (emp /\ X eqs Y)} X1 := E {Q} =>
if X1 =/= X then
{replace(P, X, Y)} replace(X1 := E, X, Y) {replace(Q, X, Y)}
else
{replace(P, X, Y)} X1 := replace(E, X, Y) {Q}
fi .
```

Listing 13: Rule intepreting variable equality

5.7.3 Automatic Proof of Implications

Automatic proof of implications is one of the most challenging aspects of this project and this section will present our incipient approach to this problem.

We have modelled implication proof rules as rewrite rules in Maude (example in Listing 14), in order to make use of its `metaSearchPath` operator, which will search for a series of rewrites which transform a term into another. We used this functionality to try and solve implications

$$R \Rightarrow S$$

by searching for a series of rewrites from `R => S` to `true`.

```
rl R:Relation * ( S:Relation \/ T:Relation ) => (R:Relation * S:Relation) \/ (R:Relation * T:Relation) .
--- Example for the usage of the replace operator
rl R * (Emp /\ X eqs Y) => replace(R, X , Y) .
rl (R:Relation => R:Relation) => true .
```

Listing 14: Example of implication rewrite rules

Most proof rules are of the form $R \Leftrightarrow S$ and thus can be naturally expressed in our prover using rewrite rules. However, when the proof rules involve just a one way implication, such as $Emp \Rightarrow Same$, if we were to model it simply as the first rule in the Listing 14, an implication, `Same => Emp` for example, could be wrongly considered true, because Maude would rewrite the `Emp` to `Same` and `Same => Same` evaluates to `true`. To prevent this from happening, we have constrained one-way implications by providing the entire context in which they can be rewritten:

```
rl ((R * Emp) => S) => ((R * Same) => S) .
```

We have opted for the approach to search from the term `R => S` to `true` (Listing 15) instead of searching from `R` to `S` because there are implications in which it is beneficial to use information from both sides in order to prove it. A simple example would be `R => true` which should evaluate to `true` - the rewrite rule corresponding to this implication is used in the proof at Appendix A - but if we were to search from `R` to `true`, the search will probably fail.

```
op tryImplicationProving : Goal Int -> Trace? .
eq tryImplicationProving(RR:Relation => TT:Relation , depth:Int) =
    metaSearchPath(upModule('IMPLICATION-PROVER, false), upTerm(
   RR:Relation => TT:Relation),upTerm(true), nil , '* , depth:Int
   , 0) .
```

Listing 15: Equation using the metaSearchPath operator

If the implication has been proven automatically, the proof is correct with respect to the rewrite rules specified in the **implication-prover.maude** file. The approach however has an exponential running time since it searches exhaustively throughout the state space for a solution.

The second parameter in Listing 15 - line 1, controls the depth up to which the search goes. The prover will try automatically to search a solution within a depth of 3 rewrites, but the user can manually specify searching for a solution using the `auto x` command, where `x` is the depth up to which Maude should try to search the solution (our experiments show that searches up to a depth around 12 should finish use a reasonable amount of resources and finish in under 10 minutes - this observation of course depends on the implication and the number of possible rewrites starting from it). The found path is presented to the user if the implication has been proven.

Almost all the implications in Appendix A can be automatically proven, the exceptions being the ones for which the compute resources required by the prover are too high, since they require a depth higher than 15 - those are manually admitted using the `ok` command.

In the future, one might consider using Maude internal strategies [7] to control the rewriting and searching in the implication proving process, to make it follow heuristics and thus be more effective.

Inductive predicates - the prover currently supports defining inductive predicates only by modifying the underlying Maude code (Listing 16). We have defined a single predicate, used in the Relational Separation Logic paper [1] :

```
op List : Id -> Relation .
```

Listing 16: Predicate syntax definition

In terms of the semantics of the `List` predicate, we have added rules to the **implication-prover.maude** file, to be considered when rewriting implications (Listing 17).

```
rl [ (X:Id |-> E:AExp) * ((X:Id plus 1) |-> Y:Id) ; (X:Id |-> E:
    AExp) * ((X:Id plus 1) |-> Y:Id) ] * List(Y:Id) => List(X) .
```

Listing 17: Example of rule interpreting the List predicate during implication proving

Cyclist [10] [11] is an automatic entailment prover for Separation Logic which we plan to integrate with our prover for Separation Logic implications and eventually extend to Relational Separation Logic, since it provides a generic framework. Since Maude does not easily support external commands, the idea would be to modify Maude's C++ code and add a hook for the `[special]` operator attribute which would invoke that hook, which will call the external command for Cyclist.

5.8 User Interface

Since our project aims to enable proofs 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.

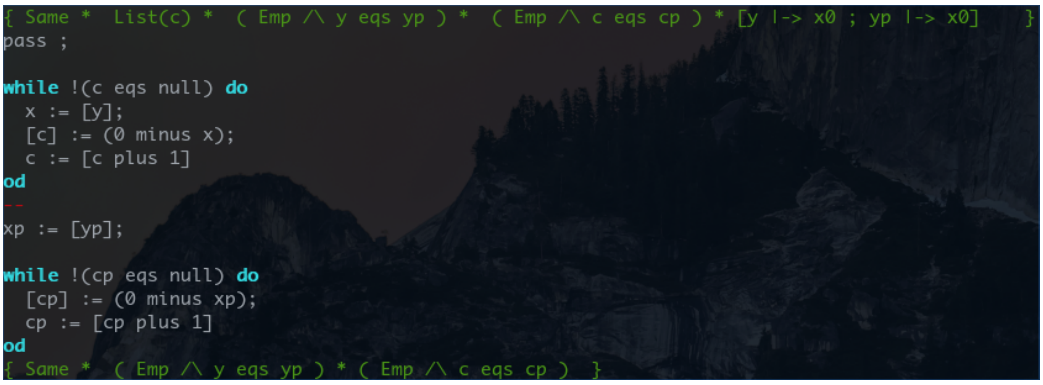
Maude provides a simple, yet powerful system for controlling the output formatting of the CLI through which we aim to make the proofs more readable and user-friendly.

The Maude features we used for this prover's user interface are:

- The `format` attribute of operators, which allows the programmer to control the style of the output when a term that makes use of the respective operator is printed to the console. It allows specifying modifiers for each possible white space in an operator, modifiers that will remain active until reset or until overriden. More information on the `format` attribute can be found in the Maude Manual [7].

  A sample output of the prover is presented in Figure 4, showcasing the manipulation of white spaces (newlines, indentation and spaces), usage of colours to distinguish between different elements - Relations are coloured in green, while the analysed code has its keywords highlighted using cyan. The separator between the two pieces of code is also coloured in red;

- The `metaPrettyPrint` operator in the `META-LEVEL` module, which returns a term as a list of quoted identifiers to be printed, formatted respecting the `format` attributes defined for the operators used in the term. An example is presented in Listing 18, which presents the rule used to take the top element of the **GoalStack** for processing, and writes the formatted output of the goal to the **StagingOutput** field.



```
{ Same *  List(c) *  ( Emp /\ y eqs yp ) *  ( Emp /\ c eqs cp ) * [y |-> x0 ; yp |-> x0]    }
pass ;

while !(c eqs null) do
  x := [y];
  [c] := (0 minus x);
  c := [c plus 1]
od
--
xp := [yp];

while !(cp eqs null) do
  [cp] := (0 minus xp);
  cp := [cp plus 1]
od
{ Same *  ( Emp /\ y eqs yp ) * ( Emp /\ c eqs cp )  }
```

Figure 4: Formatted output of the prover

```
crl [takeFirstGoalFromList] : < noAction ; RG ; noGoal ; G <> GS
    ; GL ; nil >
=>
< check ; RG ; G ; GS ; GL ; 'current ' 'goal '  '  'is '\n
    metaPrettyPrint(upModule('PROVER-INTERFACE, false), upTerm(G)
    ) '\n >
 if G =/= noGoal .

```

Listing 18: Rewrite rule making use of metaPrettyPrint

All throughout the prover calls to `metaPrettyPrint` have been made in order to inform the user of what the prover is doing and how.

For a more advanced user interface or to expose the functionalities of the prover 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 by the Maude plug-in for Eclipse (see [12]) which enables writing of Maude code inside the Eclipse IDE, and the plug-in communicates with the underlying Maude environment. The project also includes a generic plug-in which allows any Java program to communicate with Maude.

Conclusions and Future Work

The development of the Relational Separation Logic prover helped me get a deeper understating of it, along with an introduction to other theoretical concepts used in program analysis and programming language semantics, such as Separation Logic and Rewriting Logic.

While the two Separation Logics used throughout the paper are powerful tools for analyzing the behavior of program and are relatively easy to understand, it takes time to become familiar and use them at their full potential. In my opinion, the logics are better suited for studying small snippets of code and not entire programs, because the demonstrations can get cumbersome and end up spending a lot of time on focusing on irrelevant parts of the program instead of its key sections.

Maude has been interesting to work because it required an entirely different mindset than the mainstream programming languages. Once grasping the concepts around Maude which was built, it was easy to come up with simple, clever solutions to the encountered problems.
As it is described on it's website [4], Maude is a powerful logical framework which allows modelling of other logics in it; it proved to be exactly that, enabling us to represent the logics in a natural manner, which made the implementation much less error-prone and allowed us to focus on the relevant part of this project, namely how to apply the logics instead of focusing on how to properly implement them.

While being a powerful and interesting language to work with, being fairly specialized and not very widespread, guidance and help are limited to the manual [7] and primer [13]. This obstacle resulted in significant time spent on searching for solutions to otherwise trivial problems, especially when starting using the language, because the source of the problem is usually unclear to a beginner.

Overall, this whole project has been an interesting introduction to both theoretical concepts regarding program analysis and new, different technologies.

Future directions of developing this project include support for concurrent programs and reducing the amount of required user input by automation. Immediate improvements which can be implemented for the prover include:

- The ability to trace back the progress of a proof, and retry from intermediary states. This can be achieved by replacing the **GoalStack** with a directed graph structure in which nodes are **Goals** and arcs represent dependencies between the **Goals**.

- Implement a symbolic execution system for relations and assertions in order to be able to handle more complex proofs. The foundations of such a system are present in the current implementation, which interprets the equality between variables. For example, arithmetic expressions are a valid candidate for this system: in the current implementation, the expression `x plus (1 plus 2)` is different from the `x plus 3`.

- Improvements to the automatic proof of implications, with two of the options discussed in the Subsection 5.7.3

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] Maude site. `http://maude.cs.illinois.edu/w/index.php?title=The_Maude_System`, (accessed 20.06.2018).

[5] 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).

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

[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] 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.

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

[10] 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.

[11] Cyclist prover. `http://www.cyclist-prover.org/`, (accessed 20.06.2018).

[12] Maude development tools. `http://mdt.sourceforge.net/`, (accessed 20.06.2018).

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

Appendices

A Example of Proof Using the Prover

This appendix contains all the necessary commands to be run in a Maude console in order to run an example demonstration using the prover. The proven example is detailed in the [1] paper.

```
load prover-interface.maude
loop init .
(prove(
{
((
Same
*
List(c)
*
[ y |-> x0 ; yp |-> x0 ]
)
*
(y eqs yp /\ Emp))
*
(c eqs cp /\ Emp)
}
pass ;
while (! (c eqs null))
do
x := [y] ;
[c] := 0 minus x ;
c := [c plus 1]
od
--
xp := [yp] ;
while (! (cp eqs null))
do
[cp] := 0 minus xp ;
cp := [cp plus 1]
od
{
(Same
*
(y eqs yp /\ Emp))
*
(c eqs cp /\ Emp)
}
)
)
(applySequencing ( Same * List(c) * [ y |-> x0 ; yp |-> x0 ] * (y eqs yp /\ Emp) * (c eqs cp /\ Emp) * (xp eqs x0 /\ Emp) ) )
(applyFrameRule (Same * List(c) * (y eqs yp /\ Emp) * (c eqs cp /\ Emp)) )
(
applyConsequence
( [ (y |-> x0) ; (yp |-> x0) ] )
--
( [ (y |-> x0) ; ((yp |-> x0) * (xp eqs x0 /\ emp)) ] )
)
(applyEmbeddingRule)
(
applyConsequence
( Same * List(c) * (y eqs yp /\ Emp) * (xp eqs x0 /\ Emp) * (c eqs cp /\ Emp) * [y |-> x0 ; yp |-> x0] )
--
( (Same * List(c) * (y eqs yp /\ Emp) * (xp eqs x0 /\ Emp) * (c eqs cp /\ Emp) * [y |-> x0 ; yp |-> x0]) /\ !(!(c eqs null)) )
)
(applyLoop)
(applyConsequence
([(c0 |-> -) * ((c0 plus 1) |-> c1) * (y |-> x0) * (c eqs c0 /\ emp)
; (c0 |-> -) * ((c0 plus 1) |-> c1) * (xp eqs x0 /\ emp ) * (cp eqs c0 /\ emp) ]
* List(c1) * Same * (y eqs yp /\ Emp) * [emp ; yp |-> x0])
--
```

```
([(c0 |-> (0 minus x0)) * ((c0 plus 1) |-> c1) * (y |-> x0) * (c eqs c1 /\ emp)
; (c0 |-> (0 minus x0)) * ((c0 plus 1) |-> c1) * (xp eqs x0 /\ emp ) * (cp eqs c1 /\ emp) ]
* List(c1) * Same * (y eqs yp /\ Emp) * [emp ; yp |-> x0])
)
(ok)
(applyFrameRule (List(c1) * Same * (y eqs yp /\ Emp) * [emp ; yp |-> x0]))
(applyEmbeddingRule)
(SLapplySequencing ((c0 |-> (0 minus x0)) * ((c0 plus 1) |-> c1) * (y |-> x0) * (emp /\ c eqs c0)))
(SLapplySequencing ((c0 |-> -) * ((c0 plus 1) |-> c1) * (y |-> x0) * (emp /\ c eqs c0) * (emp /\ x eqs x0)))
(SLapplyFrameRule ((c0 |-> -) * ((c0 plus 1) |-> c1) * (emp /\ c eqs c0)))
(SLapplyFrameRule ( (y |-> x0) * ((c0 plus 1)|-> c1) ) )
(SLapplyFrameRule ( (y |-> x0) *  (c0 |->(0 minus x0)) ) )
(SLapplySequencing ((c0 |-> (0 minus x0)) * ((c0 plus 1) |-> c1) * (xp eqs x0 /\ emp) * (emp /\ cp eqs c0)))
(SLapplyFrameRule ( (c0 plus 1)|-> c1 ))
(SLapplyFrameRule  ((emp /\ (xp eqs x0) ) * (c0 |->(0 minus x0))) )
(ok)
(auto 9)
```