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 programelor, folosindu-se de Logica Separatoare Relațională [1] și Logica Separatoare [2] [3], utilizând Maude [10] drept cadru de dezvoltare. Componenta interactivă se îmbină cu o componentă automatizată, prin care părți din demonstrație vor fi realizate automat. Lucrarea este orientata pe rezolvarea problemei prin intermediul funcționalităților oferite de Maude și va include secvențe de cod ce vor ilustra conceptele cheie legate de modelarea si aplicarea fundamentelor teoretice utilizate pe parcursul proiectului.

Introduction

The present paper describes the development of an interactive tool for reasoning how two programs are related, based theoretical concepts from the field of formal program analysis 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 and Relational Separation Logic [1]. The 2 extensions simplify the Hoare Logic proofs, mainly using the "*" connector, allowing for local reasoning about the 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 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 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, and discuss how it can be improved in the future.

1 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 tasks, which makes the tool more convenient to use .
- Example of formal proof done using the tool - Appendix A

2 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 [1] .

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

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[4] 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 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].

4.3 Maude

Maude [10] is a framework based on rewriting logic [11] 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 [12] 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 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 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). References for these can be found in the previous

section.

This section will include snippets of Maude code, but since Maude is a very expressive and self-explanatory language, I have not included of thorough presentation of every language construct defined for the scope of this project. Ideas presented will be accompanied by some examples and key parts of the implementation but 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 $Heap$.
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:

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

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

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

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.

5.1.2 Syntax and Semantics 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]
.

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

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

Maude Modelling of the language Maude makes the representation of programming languages grammar a straight forward and natural process, by defining sorts corresponding to language elements and their associated operators. The sorts defined in Maude for the grammar are the following :

- `AExp` - abbreviation for Arithmetic Expression - equivalent 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` of `INT` standard module and `Id`, which is the sort defined for variables.
- `BExp` - abbreviation for Boolean Expressions - equivalent for the Boolean Expressions type in the specification. It is a super sort of `Bool` of standard module `BOOL`.
- `Command` sort maintains the same name as in the specification.

An example of operator 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.

```
1 op while_do_od : BExp Command -> Command [...].
2 op _:=_ : Id AExp -> Command [...].
3
```

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 [8]. 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 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 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, which must match the conclusion of the used rule, will generate new goals consisting of the hypothesis: Figure 2 presents how a Separation Logic rule has been translated into Maude syntax and Figure 3 presents the same process for the Relational Separation Logic counterpart of the rule.

Consequence Rule

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

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

Figure 2: Example of a translation to Maude syntax of a Separation Logic proof rule

The <> operator is used to separate sub-goals generated by the rule.

Consequence Rule

$$\frac{R \Rightarrow R_1 \quad \{R_1\}\, \overset{C}{C'}\, \{S_1\} \quad S_1 \Rightarrow S}{\{R\}\, \overset{C}{C'}\, \{S\}}$$

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

Figure 3: Example of a translation to Maude syntax of a Relational Separation Logic proof rule

The `--` operator is used to separate the two snippets of code mentioned in the **Hoare Quadruple**.

All the rules are marked with the attribute `[nonexec]` because we want them to be applied in a controlled manner, at the meta-level. Some of the rules, the examples included, are not even valid executable Maude rules, because they make use of variables before binding them (for example `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 exemplify 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, the Relational Separation Logic [1] and Separation Logic [3] [2] have been modeled 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 associated with the concept with the same name, in a Separation Logic context.

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

Listing 4: Hoare Triple constructor operator

- `Relation` is the sort associated with Relational Separation Logic relations. As it was the case with `Assertion`, it is a super sort of the `BExp` sort of our language grammar. 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` is again associated with the concept with the same name, in the Relational Separation Logic context.

```
--- 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

Maude's LOOP-MODE module is the entry point for any interactive Maude application, being 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 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**.

5.4 State of the Prover

Following the examples provided together with the Maude Manual [7], our state (Figure 5) is composed multiple sections modified accordingly during program execution:

[INPUT, PROVER − STATE, OUPTUT]

Figure 4: Loop state

- `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 rewriting rule to generate output independent of the others.

< Action, RootGoal, CurrentGoal, GoalStack, ProvenGoals, StagingOutput >

Figure 5: Prover State

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 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 6: 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 7: out rule

5.6 Prover execution flow

In this section we will describe the flow of a demonstration made using our prover. The most important parts of the prover's execution 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`, colored in the figure in purple and `User Actions`, colored in light-yellow. The diagram presenting how the prover works is presented in Figure 6

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 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.

Goal Graph One of the improvements which could make the user experience much better for our prover would be the capability to backtrack in the proof and try another approach for the same goal. The initial idea for this prover was to have a `GoalGraph` instead of `GoalStack` such that when the user applies a rule to a goal, the sub goals are represented as nodes linked to the goal from which they originated, making it possible to trace back the progress to intermediary states. This approach would also make

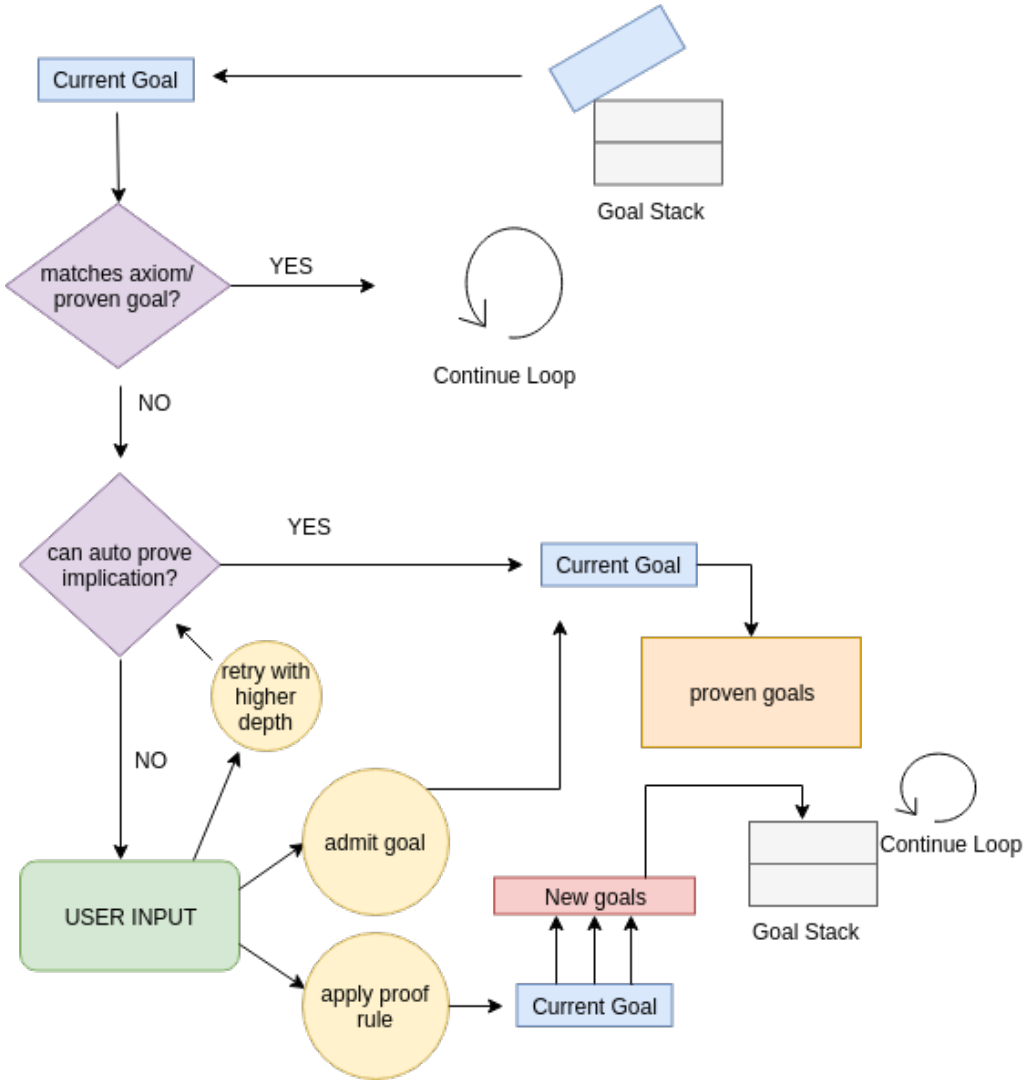

Figure 6: Prover Execution Diagram

the `ProvenGoals` a lot more useful by increasing the chance of running into an already proven goal and the ability to mark complex goals as proven. At the moment, the only goals which can be used from the `ProvenGoals` are those admitted manually by the user, since the other proven goals already matched axioms.

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** which 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 a 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 there were some errors when applying the `metaXapply` operator [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 independent of other applications. 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 the 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 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. 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 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.

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 matching 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 with any term for which all variables in the axiom can be bound. For example, the axiom

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

will be represented as a term of sort `HoareTriple`, but with free variables:

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

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 the Separation Logics as well). We had to find a work around 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 10)

```
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 10: Initialization of Prover state

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

```
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 11: 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 12: Rule intepreting variable equality

5.7.3 Automatic demonstration of implications

Automatic demonstration of implications is one of the most challenging aspects of this paper 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 13), 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 13: 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 13, 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` 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 14: Equation using the metaSearchPath operator

In its current form, the prover does not offer any guarantees, except for the certainty that 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 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 running time of the prover is 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. We have defined a single predicate, used in the Relational Separation Logic paper [1] :

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

Listing 15: 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.

```
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 16: Example of rule interpreting the List predicate during implication proving

Cyclist [9] [13] is an automatic entailment prover for Separation Logic which we planned 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. We have however focused on other aspects of our prover, and left this feature for the future.

5.8 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.

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 which makes use of the respective operator is printed to the console. It allows specifying modifiers for each possible white space in an operator, modifiers which 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 7, 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 it's keyword 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 17, which presents the rule

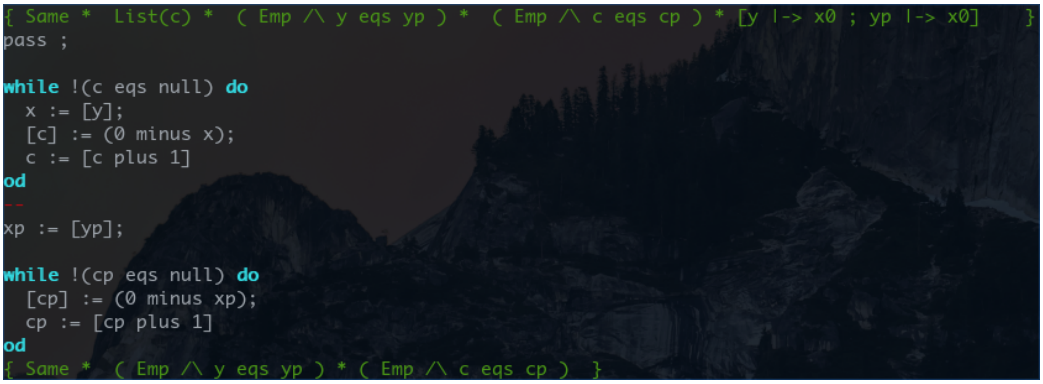

Figure 7: Formatted output of the prover

used to take the top element of the **GoalStack** for processing, and writes the formatted output of the goal to the **StagingOutput** field.

```
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 17: 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 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 by the Maude plug-in for Eclipse, [14] 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

The development of the Relational Separation Logic prover helped us 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.

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 [10], 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 primer [6] and the manual [7]. 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. The mix of the two resulted in the built prover, which could be improved in the future, both from an implementation standpoint (features discussed throughout the chapters) and by refining the theoretical concepts on which it is based (concurrency extensions for logics) to easily accommodate real-world usage.

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

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

[12] 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.

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

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

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.

```
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)
```

```
66 (applyFrameRule (List(c1) * Same * (y eqs yp /\ Emp) * [emp ; yp
   |-> x0]))
67 (applyEmbeddingRule)
68 (SLapplySequencing ((c0 |-> (0 minus x0)) * ((c0 plus 1) |-> c1)
   * (y |-> x0) * (emp /\ c eqs c0)))
69 (SLapplySequencing ((c0 |-> -) * ((c0 plus 1) |-> c1) * (y |-> x0
   ) * (emp /\ c eqs c0) * (emp /\ x eqs x0)))
70 (SLapplyFrameRule ((c0 |-> -) * ((c0 plus 1) |-> c1) * (emp /\ c
   eqs c0)))
71 (SLapplyFrameRule ( (y |-> x0) * ((c0 plus 1)|-> c1) ) )
72 (SLapplyFrameRule ( (y |-> x0) *  (c0 |->(0 minus x0)) ) )
73 (SLapplySequencing ((c0 |-> (0 minus x0)) * ((c0 plus 1) |-> c1)
   * (xp eqs x0 /\ emp) * (emp /\ cp eqs c0)))
74 (SLapplyFrameRule ( (c0 plus 1)|-> c1 ))
75 (SLapplyFrameRule  ((emp /\ (xp eqs x0) ) * (c0 |->(0 minus x0)))
     )
76 (ok)
77 (auto 9)
```