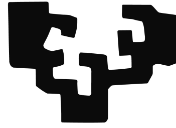

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Bachelor's Degree in Informatics Engineering
Computation

Bachelor's End Project

Verification of Concurrent Programs in Dafny

Author
Jon Mediero Iturrioz

informatika fakultatea
facultad de informática

2017

Abstract

This report documents the Bachelor's End Project of Jon Mediero Iturrioz for the Bachelor in Informatics Engineering of the UPV/EHU. The project was made under the supervision of Francisca Lucio Carrasco.

The project belongs to the domain of formal methods. In the project a methodology to prove the correctness of concurrent programs called Local Rely-Guarantee reasoning is analyzed. Afterwards, the methodology is implemented over Dagny automatic program verification tool, which was introduced to me in the Formal Methods for Software Developments optional course of the fourth year of the bachelor.

In addition to Local Rely-Guarantee reasoning, in the report Hoare logic, Separation logic and Variables as Resource logic are explained, in order to have a good foundation to understand the new methodology. Finally, the Dafny implementation is explained, and some examples are presented.

Acknowledgments

First of all, I would like to thank to my supervisor, Paqui Lucio Carrasco, for all the help and orientation she has offered me during the project.

Lastly, but not least, I would also like to thank my parents, who had supported me through all the stressful months while I was working in the project.

Contents

Abstract i

Contents v

1 Introduction 1

1.1 Objectives 3

1.2 Work plan 3

1.3 Content 4

2 Foundations 5

2.1 Hoare Logic 5

2.1.1 Assertions 5

2.1.2 Programming language 7

2.1.3 Inference rules 9

2.1.4 Recursion 11

2.2 Separation Logic 13

2.2.1 Programming language 14

2.2.2 Assertion language 17

2.2.3 Inference rules 21

2.3 Variables as Resource 26

2.3.1 State 27
2.3.2 Assertion language 27
2.3.3 Inference rules 30
2.4 Dafny 34
2.4.1 Functions and predicates 35
2.4.2 Lemmas 37
2.4.3 Algebraic data types 40

3 Proving the correctness of concurrent programs **43**
3.1 Local Rely-Guarantee reasoning 44
3.1.1 Programming language 45
3.1.2 Assertion language 47
3.1.3 Inference rules 51

4 Implementing LRG in Dafny **59**
4.1 Implementation 59
4.1.1 Expressions 60
4.1.2 Binary expressions 61
4.1.3 Programming language 62
4.1.4 Assertion language 64
4.1.5 Inference rules 67
4.2 Project structure 70
4.3 Proving methodology 71

5 Examples **73**
5.1 Example of a sequential program 73
5.2 Example of a concurrent program 76

6 Conclusions and future work **81**

6.1 Conclusions 81

6.2 Future work 83

Appendices

A Project organization **87**

Bibliography **89**

1. CHAPTER

Introduction

As a consequence of the lowering of the barrier to entry to programing, thanks to the creation of easy to use programming languages and due to an almost universal access to learning resources over the internet, the number of people who code has greatly increased over the last fifteen years. For most of this coders, programming is just a tool to be used occasionally when they need to automate a task, or to process great volumes of data or to work complex scientific calculations. On the other hand, for software developers, which profession consist specifically on creating programs to be used by other people, programming is the central element of the toolbox. Being the use of programming by these two collectives quite different, it is logical to think the methodology might also differ. Indeed, while regular programmers begin to program directly without any preparation, software developers design, specify the use cases, and create models before beginning to code a single line. Additionally they set the conventions to be used, guidelines to be followed and methodologies to apply, in order to ensure multiple developers can collaborate effectively. But at the end of the day, when programs have errors (bugs) and problems arise, the two collectives use the same approach: they analyze the error, modify the code, and test again. This process is iterative, repeating the cycle until no error is found.

The problem is that testing is not capable of proving the absence of errors, since not all possibilities are tried, and can only give us a limited assurance of the correctness of the program. For casual programmers, this is not a big problem, as programs are usually short lived, and have a low level of complexity. But for software developers, creating program with millions of lines of code, to be distributed all over the world, having a correct program is a critical condition.

To give an answer to that problem, formal methods where invented, a methodology where program behaviour is specified using logic, and the implementation is proved to conform the specification mathematically.

Formal methods where first proposed by C.A.R Hoare in 1969, when he publised *An Axiomatic basis for Computer Programming* [11]. In this paper, Hoare proposes a simple logic, that is nowadays know as *Hoare logic*, to prove the correctness of simple imperative programs. This publication opened the door to plenty of research on the area, even though industry adoption of formal method was not very high.

An specially important advance in the proliferation of formal methods, was the 1980s, when the firs proof assistants as Coq [3] and Isabelle [16] were created. Theorem provers allowed to use computers to verify the proofs done manually or directly over them were correct. Thanks to that, bigger programs could be verified and the human errors were detected early and corrected. From them, proof assistants have improved, and nowadays automatic theorem proves exist, which are capable of proving themselves the theorem they receive as parameters. Human interaction is still needed for complex proofs.

In parallel, many new methodologies were created, to allow proving correctness of programs making use of dynamic memory, or even to allow reasoning about concurrent programs.

In the recent years, many of the advances on the field have been combined, and complex things as operative systems have been verified: seL4 [13].

In this work, we are going to explain and implement Local Rely-Guarantee (LRG) reasoning [9], a logic to verify the correctness of shared memory concurrent programs, over the Dafny [14] automatic program verification tool. The objective is to allow making this proofs more easy using a computer based approach instead of doing it manually.

Unlike for actor based or message based concurrency, were thread interaction is limited to specific points were messages are send or received, in shared memory concurrency one thread might interfere with others at any time. This is why to verify shared memory concurrency conventional techniques as Hoare logic are not enough, since this logics has no way to represent neither take into account this interferences, and LRG like systems are used.

We discovered LRG while reading about LiLi [15] which was itself discovered when reading about CertikOS [10], an architecture to build certified concurrent kernels. LiLi is a

methodology to prove concurrence related properties as deadlock adn starvation freedom, and was mentioned in the CertikOS as it was necessary to prove all the system calls of the OS eventually return. LiLi is build ovar LRG. At the begining of the project we asses the possibility to do the project about LiLi instead of LRG and explain RLG in the foundations, but we decided against it because of time concerns.

1.1 Objectives

One of the project's objectives is to explore the suitability of Dafny as a tool to implement logics, as conventionally proof assistants as Coq or Isabelle have been used to that purpose. Being Dafny a higher level tool capable of proving many simple theorems by itself, we hope that the implementation will be easier, and will allow more future implementations.

The second objective is to explore the capabilities of LRG logic, implementing it and using it to prove some examples.

1.2 Work plan

The project has been carried between January 2017 and June 2017. The work was divided in two stages:

- In the first stage many technical publications were read in order to learn enough about modern formal verification techniques as Separation Logic [17] and Rely-Guarantee reasoning [12] to allow a good understanding of LRG. Additionally, in order to understand the problems might arise in concurrent programs, literature about concurrent programming has also been studied. Finally, some chapters of the Software Foundations course were followed, to learn how to implement logics over proof assistants.
- In the second stage, LRG was implemented over Dafny, and after that we tried to use the implementation to prove the correctness of some programs..

The documentation phase began in January, and continued all over the project duration. The implementation phase was started in March and most of this report was written between May and June.

In order to report progress, plan the work for the following days and solve problems, a weekly meeting with the project director was scheduled. The meeting were at Tuesday morning and with few exceptions, the schedule was fulfilled. In June some additional meeting were made in order to prepare the report and the project presentation.

1.3 Content

The document is organized as follows:

- In chapter 2 we present all the foundation necessary to understand the project, which include Hoare logic, Separation logic and Variables as Resource in Hoare logic.. Additionally, a general description of Dafny language is included.
- In chapter 3 we present the LRG logic.
- In chapter 4 we explain how the LRG logic was implemented over Dafny.
- In chapter 5 we present the proofs to example programs.
- Finally, in chapter 6 final conclusions and possible improvements are discused.

2. CHAPTER

Foundations

2.1 Hoare Logic

Hoare Logic is a formal verification system used to prove the correctness of imperative computer programs. It was first proposed by C. A. R. Hoare in 1969 in his *An Axiomatic Basic for Computer Programming* publication [11].

2.1.1 Assertions

In Hoare Logic, assertions are used to specify the state of the program in different points of the execution. Assertions are logical expressions witch relate the values of the program variables to concrete values or between them. Assertions are attached to concrete points of the program code, and must be true for any correct execution of the program (considering a correct execution an execution beginning from a valid state).

For example, in the next program:

```
{true}
i := x + 2;
{i = x+2}
z := 7;
{z = 7 ∧ i = x+2}
a := i*z + 2;
{a = 7*x+16}
```

there are 4 different assertions. The first one is true to assert that any initial state is considered valid. If instead of true the assertions value was $x \geq 2$, only executions beginning with an x value greater or equal to 2 would be considered valid.

The rest of the assertions give information about the state of the program after the execution of some code. An important detail to take into account is that assertions are not required to be complete, this is, it is not mandatory for assertions to record all the state of the program at one point. Because of that, some variables and relations might disappear from the assertions when their value is no longer considered necessary or interesting for the proof. This is the case of the last assertion, where there is no longer information about the values of z and i variables.

The assertions language used in Hoare Logic is the language of first order logic. This means the assertions language accepts:

- terms: variables, constants and functions
- atoms: predicates over terms
- logical connectors: $\wedge$, $\vee$, $\neg$, $\rightarrow$ and $\leftrightarrow$
- quantifiers over variables: $\forall$ and $\exists$

Usually, for Hoare Logic, the type of the constants is integer or boolean and the basic mathematical operators, including the relational ones, are defined ($+$, $-$, $*$, $/$, $<$, $=$, ...).

It is possible to add new functions and predicates to the language defining them in a straightforward way or recursively. For example, we might define the *isEven* function using the modulo operator

$$isEven(x) \stackrel{\text{def}}{=} x \mod 2 = 0$$

or recursively:

$$isEven(x) \stackrel{\text{def}}{=} \begin{cases} \text{true}, & x = 0 \\ \text{false}, & x = 1 \\ isEven(x-2), & x > 1 \end{cases}$$

The assertions of Hoare Logic are evaluated over a store. The store is a partial function which maps variable names to terms, and is used to represent the concrete state of a

program in a given point. In order to evaluate an assertion, all the free variables should be replaced with the values of the store.

2.1.2 Programming language

Even if it can be used to reason over complex programming languages, Hoare Logic is usually defined over a minimum language including only the most basic operations.

In this section, we are going to define one of those minimum language, to be used in the following examples and explanations.

Assuming *n* is an integer constant and *x* is a variable name, the grammar of the language is the following:

```
   E ::= n | x | E op_E E
op_E ::= ’+’ | ’-’ | ’*’

   B ::= ’true’ | ’false’ | B op_B B | E op_C E | ’not’ B
op_B ::= ’and’ | ’or’
op_C ::= ’=’ | ’!=’ | ’<=’ | ’=>’ | ’<’ | ’>’

   C ::= ’skip’
       | x ’:=’ E
       | C’;’C
       | ’if’ ’(’ B ’)’ ’{’ C ’}’ ’else’ ’{’ C ’}’
       | ’while’ ’(’ B ’)’ ’{’ C ’}’
```

For the sake of simplicity, all the variables in this programming language are going to be of integer type, and they are not going to have any upper or lower limit.

As we mentioned earlier, the program status is represented using an store, i.e. a partial function mapping variable names to values. When evaluating expressions (E) and boolean expressions (B), the variable names are replaced by the corresponding value, and the resulting mathematical expression is evaluated using the usual semantics of integer and logical operators.

The assignment (`x ’:=’ E`) is the only statement who changes the store, updating or creating the value of x variable with e. The rest of the statements have the usual semantics.

To represent changes in the s store, we are going to use the following syntax $s[x \leftarrow v]$, where x is the variable name and v is the new value for the variable. We are going to be using this operator quite often in the following chapters, so it may be useful to define it formally.

For any function $f : T \rightarrow T'$ where T and T' might be any type, $\forall x \in T$ and $\forall y \in T'$:

$$f[x \leftarrow y](z) \stackrel{\text{def}}{=} \begin{cases} y, & z = x \\ f(z), & \text{otherwise} \end{cases}$$

We can formalize the operational semantics of the language using a standard small-step semantics. Having the store $s : \mathit{Var} \rightarrow \mathbb{Z}$, $s(x)$ will give the current value of the variable x. Taking that into account, the value of an expression E or boolean expression B in some state s, denoted $[\![E]\!]_s$ and $[\![B]\!]_s$ respectively, are recursively defined as follows.

$$\begin{aligned} [\![n]\!]_s &= n \\ [\![x]\!]_s &= s(x) \\ [\![E_1 \ op_E \ E_2]\!]_s &= [\![E_1]\!]_s \ [\![op_E]\!] \ [\![E_2]\!]_s \\ [\![E_1 \ op_C \ E_2]\!]_s &= [\![E_1]\!]_s \ [\![op_C]\!] \ [\![E_2]\!]_s \\ [\![B_1 \ op_B \ B_2]\!]_s &= [\![B_1]\!]_s \ [\![op_B]\!] \ [\![B_2]\!]_s \\ [\![\texttt{not} \ B]\!]_s &= \neg [\![B]\!]_s \end{aligned}$$

where $[\![op_E]\!]$ is the standard semantic of operator op_E on integers, $[\![op_C]\!]$ is the standard semantic of operator op_C on integers and $[\![op_B]\!]$ is the natural semantic of operator op_B on booleans.

The semantics of statements are defined using the following relation $s, C \rightsquigarrow s', C'$, which means that in state s executing one step of the statement C leads to the state s' and the remaining statement C'. The semantics of the statements are defined by the following

rules:

$$\overline{s, x := e \rightsquigarrow s[x \leftarrow [\![e]\!]_s], \texttt{skip}}$$

$$\overline{s, (\texttt{skip}\,; C) \rightsquigarrow s, C}$$

$$\frac{s, C_1 \rightsquigarrow s', s'_1}{s, (C_1\,; C_2) \rightsquigarrow s', (C'_1\,; C_2)}$$

$$\frac{[\![b]\!]_s}{s, \texttt{if (} b \texttt{) \{ } C_1 \texttt{ \} else \{ } C_2 \texttt{ \}} \rightsquigarrow s, C_1} \quad \frac{\neg[\![b]\!]_s}{s, \texttt{if (} b \texttt{) \{ } C_1 \texttt{ \} else \{ } C_2 \texttt{ \}} \rightsquigarrow s, C_2}$$

$$\frac{[\![b]\!]_s}{s, \texttt{while (} b \texttt{) \{ } C \texttt{ \}} \rightsquigarrow s, (C; \texttt{while } b \texttt{ do } C)} \quad \frac{\neg[\![b]\!]_s}{s, \texttt{while (} b \texttt{) \{ } C \texttt{ \}} \rightsquigarrow s, \texttt{skip}}$$

2.1.3 Inference rules

In order to prove to correctness of a computer program using Hoare Logic we first have to specify the program behaviour using the following triple:

$$\{\varphi\}\ C\ \{\psi\}$$

where C is the full program (a sequence of statements: C), φ is the precondition and ψ the postcondition.

The meaning of the triple is the following: if the execution of the program begins in a state satisfying φ and the program ends, the final state will satisfy the ψ.

Proving the triplet is correct is what we call to prove the *partial correctness* of the program. It is important to note that in order to prove the *total correctness* of the program, it is also necessary to prove that for any initial state satisfying φ the program ends in a finite number of steps.

The Hoare inference rules can be used to prove partial correctness and with a little extension also total correctness.

This are the Hoare inference rules for partial correctness:

$$\overline{\{P\}\ \texttt{skip}\ \{P\}} \qquad \text{(Empty statement axiom)}$$

This ruler asserts that a `skip` statement does not change the program state.

$$\frac{}{\{P[E/x]\}\ x := E\ \{P\}} \qquad \text{(Assigment axiom)}$$

where $P[E/x]$ is the assertion P where all the free occurrences of x have been replaced by the expression E. This rule says that, as after the assignment, any property was fulfilled by E is going to be fulfilled by x. For example:

$$\{y+1 \geq 5\}\ \texttt{x := y + 1}\ \{x \geq 5\}$$

$$\frac{\{P\}\ C_1\ \{Q\} \quad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1\,;C_2\ \{R\}} \qquad \text{(Rule of composition)}$$

This rule asserts that if the postcondition of the first statement and the precondition of the second statement are equal, the two statement can be chained maintaining the first precondition and the second postcondition.

$$\frac{P \Rightarrow P' \quad \{P'\}C\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ C\ \{Q\}} \qquad \text{(Consequence rule)}$$

This rule asserts that strengthening a precondition or weakening a postcondition preserves the correctness. This rule is very useful as many of the rules require some of the pre or postconditions to be identical.

$$\frac{\{P \wedge B\}\ C_1\ \{Q\} \quad \{P \wedge \neg B\}\ C_2\ \{Q\}}{\{P\}\ \texttt{if (}\ B\ \texttt{) \{}\ C_1\ \texttt{\} else \{}\ C_2\texttt{\}}\{Q\}} \qquad \text{(Rule of condition)}$$

This rule ensures that if the postcondition Q is the postcondition of both of the branches of the `if` Q is also the postcondition of all the `if`.

$$\frac{\{I \wedge B\}\ C\ \{I\}}{\{I\}\ \texttt{while (}\ B\ \texttt{) \{}\ C\ \texttt{\}}\ \{I \wedge \neg B\}} \qquad \text{(While rule)}$$

In this rule, the I predicate is called the loop invariant. The loop invariant is a condition is meet before entering the loop, and maintained both while the loop is executing and just after the loop is finished. The inference rule, asserts that if the loop invariant is maintained by the loop body when the loop condition is true, after finishing the loop both the invariant and the negation of the loop condition will be true.

In order to prove program termination, it is necessary to make a little extension of the

inference rules. The only changing rule is the while rule, as it is the only place where a program might get stuck.

In order to prove termination, it is necessary to find an expression which value is strictly decreasing for each iteration. The expression must be defined for all the iterations and the strictly decreasing relation for the expression we choose must be well founded. This means, it is not possible to decrease the value infinitely, hence there is a finite maximum number of steps.

Formally the while termination rule is

$$\frac{\{I \wedge B \wedge v = V\}\ C\ \{I \wedge v \prec V\} \quad I \wedge B \rightarrow v \geq \xi}{\{I\}\ \texttt{while (} B \texttt{) \{ } C \texttt{ \}}\ \{I \wedge \neg b\}} \qquad \text{(while termination rule)}$$

where v is the variant, V is a fresh logical variable, and $\prec$ is a well founded relation with smallest value ξ. For example, if v ranges in the natural numbers, $<$ is a well founded relation, as there is no element smaller than 0 to continue decreasing the value of v infinitely.

2.1.4 Recursion

Hoare rules can be easily extended to allow proving correctness and termination of recursive programs.

First of all, it is necessary to extend the programming language to allow function calls:

```
C ::= ...
    | x ':=' f '(' e ',' ... ',' e ')'
```

where f is a name of a function. For the sake of simplicity we have omitted the function definitions from the grammar, giving instead this generic example:

```
function f(x1, x2, ..., xn) return z
{P}
{
   ... code goes here ...
}
{Q}
```

Functions also have a precondition (P) and a postcondition (Q). The precondition is usually used to limit the acceptable parameters of the function, and the postcondition to relate the returned value z with the input parameters $x_1,...,x_n$. In order the precondition to work as expected, it is necessary to assume the input parameters are read only (in particular they can not appear in the left side of an assignment). Without this assumption, it will be possible for a correct function to change the parameter values to conform the postcondition instead of returning a conforming value.

The new inference rule needed to prove programs using functions is the following:

$$\frac{\{P\}\ z := f(x_1,...,x_n)\ \{Q\}}{\begin{array}{c}\{R \wedge P[E_1,...,E_n/x_1,...,x_n]\}\\ x := f(E_1,...,E_n)\\ \{Q[x,E_1,...,E_n/z,x_1,...,x_n] \wedge R\}\end{array}} \quad \text{(call rule)}$$

where P and Q are the precondition and postcondition respectively. The condition of the rule means that the function is correct under this conditions, $P[E_1,...,E_n/x_1,...,x_n]$ means replacing all arguments with the expression values in the precondition and $Q[x,E_1,...,E_n/z,x_1,...,x_n]$ means replacing all the arguments and the z result with the expression values and x variable. R is any additional assertion, where x variable is not present. This R is conserved, as the function calls in this programming language do not have any side effect.

Intuitively, this rule means that if we meet the function preconditions before the call, the function postcondition will be true after the call.

In case we are using this rule to prove the correctness of a recursive function, we can assume the function meets the pre and postconditions for the recursive calls. If making this assumptions we can prove the function is correct, and the function has a correct base case, the prove is finished. What we are doing with the assumption is equivalent to the proving by induction used in mathematics. The base case (the one without any recursive call) is the basis, and the recursive case makes use of the inductive hypothesis (the precondition and postcondition) to perform the inductive step.

In a similar way to what happened in standard Hoare Logic, some additional proving is needed in order to ensure program termination, hence total correctness. As in the case of the while termination rule, it is necessary to found a strictly decreasing variant expression. Each recursive function call must decrease this expression. In order the decreasing relation

to be well founded, a minimum value must exist, and the precondition of the function must ensure the variant expression is bigger or equal.

For example, if we have a program to calculate the factorial:

```
function fact(x) return z
{x ≥ 0}
{
   if (x <= 1) {
      z := 1;
   } else {
      tmp := fact(x-1);
      z := x * tmp;
   }
}
{z = x!}
```

we can use the parameter x as variant, as in all recursive calls is reduced $(x-1)$ and has a minimum value of 0 enforced by the precondition.

2.2 Separation Logic

One of the biggest problems of conventional Hoare Logic, is that it does not allow the specification of programs that manipulate dynamically allocated storage or made use any kind of pointers (which is quite common to allow access to shared data or to create complex data structures). If we limit ourselves to the use of pure functional languages, were functions do not have any side effect and changes to data structures always return a copy of the structure with the desired modification, conventional Hoare Logic might be enough. The runtime of the programming language is the one who takes the responsibility of allocating memory and implementing the data structures in a save way, freeing us of the need to do it ourself and verify we do it correctly. But in case we want to use an imperative programming language, implement complex data structures and manage memory ourselves, we need additional tools.

Separation Logic [18] is an extension of Hoare Logic to allow exactly that, verifying programs with pointers and dynamic memory. In order to do so, first of all, the programming language is extended with new statements to reserve, read, write and free dynamic memory. The state of the program is also extended, as in addition to local variables we now

have memory addresses with values. Finally, new assertions and inference rules are added to reason about the new commands.

Even if it is not the original publication about Separation logic the assertions and inference rules in this section are taken from [17].

2.2.1 Programming language

The programming language defined in the Hoare Logic section has not any operation related to dynamic memory. As a consequence, the grammar and semantics should be extended to add the new functionality. There are many different ways to add memory operations to a language, but in our case we are going to use the same approach used in most of the literature regarding to Separation Logic, which consists on adding four new statements to the code.

It follows the grammar of the new statements:

```
C ::= ...
    | x ':=' 'cons' '(' E ',' ... ',' E ')'
    | x ':=' '[' E ']'
    | '[' E ']' ':=' E
    | 'dispose' E
```

For defining the semantics of the new operations, it is necessary to extend the program state. The dynamic memory of the program is going to be represented using a partial function from addresses to values. To allow doing pointer arithmetic addresses are going to be represented using positive integers and (as the variables of the language) the values of the memory are going to be also unbounded integers. So the heap function signature is $h : \mathbb{N} \to \mathbb{Z}$.

The state of the program is a pair consisting of a store s and a heap (dynamic memory) h, represented as (s,h).

Having defined the state of the program, we can proceed to define the semantics of the new statements. For all the original constructs of the language, the semantics remain unchanged, as they do not access the heap, the only difference being that where there was a

simple store s now we have the pair (s,h).

$$\frac{}{(s,h),x := E \rightsquigarrow (s[x \leftarrow [\![E]\!]_s],h),\texttt{skip}}$$

$$\frac{}{(s,h),(\texttt{skip};C) \rightsquigarrow (s,h),C}$$

$$\frac{(s,h),C_1 \rightsquigarrow (s',h),s'_1}{(s,h),(C_1\,;C_2) \rightsquigarrow (s',h),(C'_1\,;C_2)}$$

$$\frac{[\![B]\!]_s}{(s,h),\texttt{if (}B\texttt{) \{}C_1\texttt{\} else \{}C_2\texttt{\}} \rightsquigarrow (s,h),C_1}$$

$$\frac{\neg[\![B]\!]_s}{(s,h),\texttt{if (}B\texttt{) \{}C_1\texttt{\} else \{}C_2\texttt{\}} \rightsquigarrow (s,h),C_2}$$

$$\frac{[\![B]\!]_s}{s,\texttt{while (}B\texttt{) \{}C\texttt{\}} \rightsquigarrow s,(C;\texttt{while (}B\texttt{) \{}C\texttt{\}})}$$

$$\frac{\neg[\![B]\!]_s}{(s,h),\texttt{while (}B\texttt{) \{}C\texttt{\}} \rightsquigarrow (s,h),\texttt{skip}}$$

The first of the new statements is the allocation statement. This statement reserves a continuous range of n elements in the heap memory, where n is the length of the argument list, and initializes each of this memory cells with the values of the arguments. Formally we can define the statement using small step semantics:

$$\frac{}{(s,h),x\texttt{:= cons(}e_1,\ldots,E_n\texttt{)} \rightsquigarrow (s[x \leftarrow l],h[l \leftarrow [\![E_1]\!]_s]\ldots[l+n-1 \leftarrow [\![E_n]\!]_s]),\texttt{skip}}$$

where $l,\ldots,l+n-1 \in \mathbb{N} - \mathrm{dom}(h)$.

The second statement is the lookup statement. It reads the value of the memory cell $[\![E]\!]_s$ and assigns it to variable x. Unlike all the previous statements, this one might fail if the memory address we are trying to access is not in the domain of h, for which a new state **abort** is introduced, as a terminal state for memory faults. Formally:

$$\frac{[\![E]\!]_s \in \mathrm{dom}(h)}{(s,h),x\texttt{:= [}E\texttt{]} \rightsquigarrow (s[x \leftarrow h([\![E]\!]_s)],h),\texttt{skip}}$$

$$\frac{[\![E]\!]_s \notin \mathrm{dom}(h)}{(s,h),x\texttt{:= [}E\texttt{]} \rightsquigarrow \textbf{abort}}$$

The third statement is the mutation statement, which assigns the value of $[\![E_v]\!]_s$ a new value to the memory address $[\![E_a]\!]_s$ (where E_v is the expression to the right and E_a is the expression to the left). As the memory read, this statement might abort if the memory

address in not in the domain of h.

$$\frac{[\![E]\!]_s \in \mathrm{dom}(h)}{(s,h),\,\texttt{[}E_a\texttt{]} := E_v \rightsquigarrow (s,h[[\![E_a]\!]_s \leftarrow [\![E_v]\!]_s]),\texttt{skip}}$$

$$\frac{[\![E]\!]_s \notin \mathrm{dom}(h)}{(s,h),\,\texttt{[}E_a\texttt{]} := E_v \rightsquigarrow \textbf{abort}}$$

Finally, the fourth one is the deallocation statement. It removes the value $[\![E]\!]_s$ from the domain of h, this is, it frees the memory in $[\![E]\!]_s$. As the previous statements this one aborts if the memory address in not in the domain of h.

$$\frac{[\![E]\!]_s \in \mathrm{dom}(h)}{(s,h),\texttt{dispose}\ E \rightsquigarrow (s,h \restriction (\mathrm{dom}(h) - [\![e]\!]_s),\texttt{skip}}$$

$$\frac{[\![E]\!]_s \notin \mathrm{dom}(h)}{(s,h),\texttt{dispose}\ E \rightsquigarrow \textbf{abort}}$$

where $f \restriction U$ stands for function projection on a subset U of the domain of f.

Example

The following program uses the language defined for separation logic. The program creates an integer array in memory and reverses its content using a while loop.

```
array := cons(2, 5, 7, 4, 8, 6, 5);
len := 7;
i := 0;
while (i*2+1 < len) {
   t0 := [array + i];
   t1 := [array + len - 1 - i];
   [array + i] := t1;
   [array + len - 1] := t0;

   i := i + 1;
}
...

i := 0;
while (i < len) {
   dispose (array + i);
}
```

2.2.2 Assertion language

Since the state of the program is composed now by the store and the heap, the original assertion language is not powerful enough to reason about the new programs. That is why the assertion language is extended.

All the original assertions are maintained, which means that we can continue to use all the available constructs of the first order logics to reason about local variables. But in case we need to specify properties of the values on the heap, or relate the content of the store with the heap, we are going to use new constructs.

For defining the new assertion language, we are going to use some specific symbols to represent different kinds of assertions. The symbol B represent an standard Hoare Logic assertion, to be evaluated over s and which always returns a boolean value.

$$[\![B]\!]_s = \{\text{true}, \text{false}\}$$

The symbols E and F represent integer expressions, which are also evaluated over s (in a similar way of the expressions in the programming language).

$$[\![E]\!]_s = \mathbb{Z} \text{ and } [\![F]\!]_s = \mathbb{Z}$$

Finally, P and Q represent full separation logic assertions, that can be created combining the previous assertions with some new operators:

$$\begin{aligned} P, Q ::= {} & B \mid E \mapsto F \\ & \mid \text{false} \mid P \Rightarrow Q \\ & \mid \forall x.P \mid \text{emp} \\ & \mid P * Q \mid P -\!\!* \; Q \\ & \mid P -\!\!\circledast \; Q \end{aligned}$$

For defining the semantics of this operations we first define some notations:

- $h \perp h'$ means the fact that the domains of h and h' are disjoint, which can be formally represented by: $\text{dom}(h) \cap \text{dom}(h') = \varnothing$.

- $h \uplus h'$ represent the union of functions h and h' with disjoint domains, and it is undefined if the domains overlap.

$s,h \models P$ is a satisfaction judgement which says that the assertion P holds for the state s,h, where the semantics are defined as follows:

$$
\begin{array}{ll}
s,h \models B & \text{iff } [\![B]\!]_s = true \\
s,h \models E \mapsto F & \text{iff } \mathrm{dom}(h) = \{[\![E]\!]_s\} \wedge h([\![E]\!]_s) = [\![F]\!]_s \\
s,h \models \text{false} & \text{never} \\
s,h \models P \Rightarrow Q & \text{iff } s,h \models P \ \Rightarrow\ s,h \models Q \\
s,h \models \forall x.P & \text{iff } \forall v \in \mathbb{Z}(s[x \leftarrow v],h \models P) \\
s,h \models \text{emp} & \text{iff } \mathrm{dom}(h) = \emptyset \\
s,h \models P * Q & \text{iff } \exists h_0,h_1.\ h_0 \perp h_1 \wedge h_0 \uplus h_1 = h \wedge s,h_0 \models P \wedge s,h_1 \models Q \\
s,h \models P -\!\!* Q & \text{iff } \forall h'(h \perp h' \wedge s,h' \models P \ \Rightarrow\ s,h \uplus h' \models Q) \\
s,h \models P -\!\!\circledast Q & \text{iff } \exists h'(h \perp h' \wedge s,h' \models P \wedge s,h \uplus h' \models Q)
\end{array}
$$

In other words:

- $s,h \models B$ is a normal Hoare Logic assertion which meaning is maintained evaluating it independently to h.

- $s,h \models E \mapsto F$ asserts that the heap is formed by a single memory cell, with address $[\![E]\!]_s$ and value $[\![F]\!]_s$.

- $s,h \models \text{emp}$ asserts that the heap is empty.

- $s,h \models P * Q$ asserts that the heap can be divided in two non overlapping regions of memory one of them satisfying P and the other satisfying Q.

- $s,h \models P -\!\!* Q$ asserts that adding an independent memory region which satisfies P to our heap h creates a new heap satisfying Q.

- $s,h \models P -\!\!\circledast Q$ asserts that exists an independent memory region satisfying P, which if added to our heap h satisfies Q.

Even if all the logical connectors ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$) and quantifications ($\forall$ and $\exists$) are defined at the Hoare Logic assertion level, it is necessary to redefine them at this more general level. $\Rightarrow$, $\forall$ and false are defined above and the rest of the operators can be formed

as a combination of this ones.

$$\begin{aligned}
\neg P &\stackrel{\text{def}}{=} P \Rightarrow \text{false} \\
P \vee Q &\stackrel{\text{def}}{=} \neg P \Rightarrow Q \\
P \wedge Q &\stackrel{\text{def}}{=} \neg(\neg P \vee \neg Q) \\
\exists x.P &\stackrel{\text{def}}{=} \neg \forall x(\neg P) \\
true &\stackrel{\text{def}}{=} \neg \text{false}
\end{aligned}$$

In order to make the use of the assertion language more easy, some abbreviations are also defined:

$$\begin{aligned}
E \mapsto - &\stackrel{\text{def}}{=} \exists x(E \mapsto x) \text{ where } x \text{ is not free in } E \\
E \hookrightarrow F &\stackrel{\text{def}}{=} E \mapsto F * \text{true} \\
E \mapsto E_1, \ldots, E_n &\stackrel{\text{def}}{=} E \mapsto E_1 * \cdots * E + n - 1 \mapsto E_n
\end{aligned}$$

The first one is used when the value pointed by a memory address is not important. The second one is used when, in addition to $[\![E]\!]_s$, the heap might contain more memory addresses. Finally, the third one allows to specify consecutive memory address and value pairs in a concise way, which is going to be extensively used, for defining data structures and arrays.

Examples

Some simple assertions:

1. The value of the memory address x is greater than 5:

$$\exists v(x \mapsto v \wedge v > 5)$$

2. y points to a contiguous memory region of 7 elements:

$$y \mapsto -, -, -, -, -, -, -$$

3. y points to a contiguous memory region of 77 elements:

$$\forall n (0 \leq n < 77 \Rightarrow (y+n) \mapsto -)$$

4. y points to a contiguous memory region of 5 elements, x to a contiguous memory region of 3 elements and both regions do not intercede:

$$(y \mapsto -,-,-,-,-) * (x \mapsto -,-,-)$$

In order to write more complex assertions, the use of inductive predicates is encouraged. They are specially useful to reason about data structures. As most of the data structures need a way to represent there are no more elements, the negative pointers (which are invalid addresses) will represent the end. Note that in most real world computer and programming languages, a single value `NULL` is used instead of all the negative values. The typical value of `NULL` is 0.

The following predicate says that the data structure pointed by x is a singly linked list and is the only element in heap. The inductive predicate takes as an argument an address (a natural number) and returns a full separation logic assertion which can be evaluated over a store and heap to know if the property holds. Thus, the signature is $isList : \mathbb{N} \rightarrow P$.

$$isList(x) \stackrel{\text{def}}{=} \begin{cases} \text{emp}, & x < 0 \\ \exists v, p (x \mapsto v, p \; * \; isList(p)), & x \geq 0 \end{cases}$$

Each entry of the linked list is composed by an integer value v and the pointer p to the next element of the list.

If we want to reason about the elements of the list, it is convenient to represent the list using algebraic data types, for example:

```
data List = Nil | Cons Int List
```

, to reason over the values of the data type, and to check if the values of the data type correspond with the values of the linked list. We present an overloaded $isList : \mathbb{N} \times List \rightarrow$

P, which checks also the correspondence between the two lists:

$$isList(x,l) \stackrel{\text{def}}{=} \begin{cases} \text{emp}, & x < 0 \land l = \text{Nil} \\ \text{false}, & x \geq 0 \land l = \text{Nil} \\ \text{false}, & x < 0 \land \exists h,t(l = \text{Cons } h\ t) \\ \exists v,p,t(x \mapsto v,p \ast l = \text{Cons } v\ t \ast isList(p,t)), & x \geq 0 \land \exists h,t(l = \text{Cons } h\ t) \end{cases}$$

$s,h \models isList(x,l)$ evaluates to true if x is a valid linked list with content l.

Other data structure easily representable is the binary tree:

$$isTree : \mathbb{N} \to P$$
$$isTree(x) \stackrel{\text{def}}{=} \begin{cases} \text{emp}, & x < 0 \\ \exists v,l,r(x \mapsto v,l,r \ast isTree(l) \ast isTree(r)), & x \geq 0 \end{cases}$$

Note that both in the tree and in the linked list cycles are not accepted as we have used the operator $\ast$ instead of $\land$.

2.2.3 Inference rules

As in the case of the programming language and the assertions, the inference rules for separation logic are also an extension of the Hoare rules.

To explain the new rules, it is necessary to understood that Hoare triples has not exactly the same meaning in separation logic as in Hoare Logic. Is separation logic the $\{P\}C\{Q\}$ triple asserts that if program C executes from a state satisfying P precondition the program is not going to abort (because of an erroneous memory access), the program will terminate and the resulting state will satisfy Q.

The original rules of Hoare Logic are maintained exactly as in the previous section, hence we are going to avoid to repeat them here.

The new rules are the following ones:

$$\frac{}{\{x = m\}\ x\texttt{:= cons(}E_1,\ldots,E_n\texttt{)}\ \{x \mapsto E_1[m/x],\ldots,E_n[m/x]\}} \quad \text{(Allocation rule)}$$

The allocation rule is quite straightforward, as after executing a `cons` statement with an empty heap the result is always going to be a heap where the only elements are consecutive memory cells with the values of $E_1, \dots, E_n$. Note we need to replace x occurrences with m in the expressions, as x's value changes after the execution of the command.

$$\frac{}{\{E_1 \mapsto E_2 \wedge x = m\}\ x := [E_1]\ \{x = E_2 \wedge E_1[m/x] \mapsto v\}} \quad \text{(Lookup rule)}$$

The lookup rule asserts that after the statement the value of x will be equal to $h([\![E_1]\!]_s) = E_2$. As in the case of the previous rule, we have to replace the appearences of x with m in order to maintain the validity of the expression.

$$\frac{}{\{E_1 \mapsto -\}\ [E_1] := E_2\ \{E_1 \mapsto E_2\}} \quad \text{(Update rule)}$$

The update rule asserts that if address $[\![E_1]\!]_s$ exist in the heap, after the statement the value in memory cell $[\![E_1]\!]_s$ will be $[\![E_2]\!]_s$.

$$\frac{}{\{E \mapsto -\}\ \texttt{dispose}\ E\ \{\mathrm{emp}\}} \quad \text{(Deallocation rule)}$$

The deallocation rule asserts that if address $[\![E]\!]_s$ exist and is the only element in the heap, after the command the heap will be empty.

$$\frac{\{P\}\ C\ \{Q\}}{\{\exists x.P\}\ C\ \{\exists x.Q\}}\ x \notin \mathrm{free}(C) \quad \text{(Auxiliary Variable Elimination)}$$

This rule allows to eliminate the existential quantifier over x when proving the correctness of C.

$$\frac{\{P\}\ C\ \{Q\}}{(\{P\}\ C\ \{Q\})[E_1, \dots, E_k/x_1, \dots, x_k]} \quad \begin{array}{l} \mathrm{free}(P,C,Q) \subseteq \{x_1, \dots, x_k\} \wedge \\ \forall i (1 \leq i \leq k \wedge x_i \in \mathrm{mod}(x) \Rightarrow \\ \forall j (1 \leq j \leq k \wedge i \neq k \Rightarrow E_i \notin \mathrm{free}(E_k))) \end{array} \quad \text{(Substitution rule)}$$

where $\mathrm{free}(P,C,Q) = \mathrm{free}(P) \cup \mathrm{free}(C) \cup \mathrm{free}(Q)$. This rule allows to replace the free variables in P, C and Q with expressions if for each x_i modified in C, E_i is not free in the remaining expressions.

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\quad \mathrm{mod}(C) \cap \mathrm{free}(R) = \varnothing \quad \text{(Frame rule)}$$

where $\text{mod}(C)$ returns the set of the variables modified in C and $\text{free}(R)$ the set of the variables occurring free in R. This rule asserts that all the assertions concerning disjoint memory locations to the ones mentioned in P and Q are conserved by the C command. This rule is the most important rule of separation logic, as it allows to do local reasoning about some piece of code, prove its correctness, and use it in any place where its memory is not shared with other code.

Note that the frame rule uses the operator $*$ and not the operator $\wedge$ in the frame. This is important, as the $\wedge$ operator does not guarantee the independence of the memory addresses. Consequently, the modification of some cell of the memory may break the R assertion.

For example, the following reasoning:

$$\frac{\{\mathtt{x} \mapsto -\}\,\mathtt{[x]}\ \mathtt{:=}\ 4\{\mathtt{x} \mapsto 4\}}{\{\mathtt{x} \mapsto - \wedge \mathtt{y} \mapsto 5\}\ \mathtt{[x]}\ \mathtt{:=}\ 4\ \{\mathtt{x} \mapsto 4 \wedge \mathtt{y} \mapsto 5\}}$$

is not sound whenever $\mathtt{x} = \mathtt{y}$. The correct postcondition is $\mathtt{x} \mapsto 4 \wedge \mathtt{y} \mapsto 4$. If we use the operator $*$ we know $\mathtt{x} = \mathtt{y}$ can not be true, as the two heaps should be disjoint.

Example (borrowed from [17] section 2.2.)

Assuming our programming language accepts procedure calls, which are equivalent to function calls but without having a return value (or discarding the return value), the following code frees a binary tree from memory:

```
procedure freeTree(t)
{
   if (t > 0) {
      l := [t + 1];
      r := [t + 2];
      freeTree(l);
      freeTree(r);
      dispose t;
      dispose t + 1;
      dispose t + 2;
   } else {
      skip;
   }
}
```

To specify the precondition and postcondition we are going to use the *isTree* predicate defined in the examples section of the assertion language. The precondition is:

$$isTree(t)$$

and the postcondition:

$$\mathrm{emp}$$

In order to help the proving, we are going to annotate the program with intermediate assertions:

```
1  procedure freeTree(t)
2  {
3     {isTree(t)}
4     if (t >= 0) {
5        {t ≥ 0 ∧ isTree(t)}
6        {∃v,l′,r′(t ↦ v,l′,r′ ∗ isTree(l′) ∗ isTree(r′))}
7        l := [t + 1];
8        {∃v,l′,r′(t ↦ v,l,r′ ∗ isTree(l) ∗ isTree(r′))}
9        r := [t + 2];
10       {∃v,l′,r′(t ↦ v,l,r ∗ isTree(l) ∗ isTree(r))}
11       {t ↦ −,l,r ∗ isTree(l) ∗ isTree(r))}
12       freeTree(l);
13       {t ↦ −,l,r ∗ emp ∗ isTree(r))}
14       {t ↦ −,l,r ∗ isTree(r))}
15       freeTree(r);
16       {t ↦ −,l,r ∗ emp)}
17       {t ↦ −,l,r}
18       dispose t + 2;
19       {t ↦ −,l}
20       dispose t + 1;
21       {t ↦ −}
22       dispose t;
23       {emp}
24    } else {
25       {t < 0 ∧ isTree(t)}
26       {emp}
27    }
28    {emp}
29 }
```

When there are two assertions without any instruction between them, it means we are using the consequence rule to transform the assertion. For example between the lines 5

and 6 the consequence rule is used as

$$t \geq 0 \wedge isTree(t) \;\Rightarrow\; \exists v,l,r(t \mapsto v,l,r * isTree(l) * isTree(r))$$

is true by definition of *isTree*. Some uses of the consequence rule are not represented as the transformations are quite intuitive.

Between 6 and 8 the lookup rule is used in combination to the consequence rule, the frame rule and the auxiliary variable elimination to read the value of the left hand side of the tree:

$$\frac{}{\{t+1 \mapsto l' \wedge l = l\}\ \texttt{l := [t + 1]}\ \{l = l' \wedge t+1 \mapsto l'\}} \quad \text{(Lookup rule)}$$

$$\frac{\{t+1 \mapsto l'\}\ \texttt{l := [t + 1]}\ \{l = l' \wedge t+1 \mapsto l'\}}{\begin{array}{c}\{t \mapsto v,l',r' * isTree(l') * isTree(r')\}\\ \texttt{l := [t + 1]}\\ \{t \mapsto v,l,r' * isTree(l) * isTree(r')\}\end{array}} \quad \text{(Frame and consequence rules)}$$

$$\frac{\begin{array}{c}\{t \mapsto v,l',r' * isTree(l') * isTree(r')\}\\ \texttt{l := [t + 1]}\\ \{t \mapsto v,l,r' * isTree(l) * isTree(r')\}\end{array}}{\begin{array}{c}\{\exists v,l',r'(t \mapsto v,l',r' * isTree(l') * isTree(r'))\}\\ \texttt{l := [t + 1]}\\ \{\exists v,l',r'(t \mapsto v,l,r' * isTree(l) * isTree(r'))\}\end{array}} \quad \text{(Auxiliary variable elimination)}$$

In a similar way, between the lines 8 and 10 the value of the right hand of the tree is loaded in `r`.

Between the lines 12-14, 15-17 the frame rule is used again. The reason to use the frame rule is that the signature of `freeTree` contains mentions a heap with a single tree in the precondition and an empty heap in the postcondition. The frame rule allow us to add new independent heap element to the assertions maintaining the correctness:

$$\frac{\{isTree(l)\}\texttt{freeTree(l)}\{\mathrm{emp}\}}{\{t \mapsto -,l,r * isTree(l) * isTree(r))\}\texttt{freeTree(l)}\{t \mapsto -,l,r * \mathrm{emp} * isTree(r))\}}$$

Finally, the deallocation rule is applied at the lines 18, 20 and 22. For the first two cases, using the frame rule is also necessary to adapt the assertions.

The rule of composition is used to join all the statements inside of the if.

After proving the correctness of the body of the if, we can finish using the rule of condition:

$$\frac{\{isTree(t) \wedge t \geq 0\}\ \texttt{...}\ \{\text{emp}\}\ \ ,\ \ \{isTree(t) \wedge t < 0\}\ \texttt{skip}\ \{\text{emp}\}}{\{isTree(t)\}\ \texttt{if (t >= 0) \{...\} else \{skip\}}\ \ \{\text{emp}\}}$$

2.3 Variables as Resource

One of the limitations of separation logic is that even if it allows to reason in a modular way about the dynamic memory (the heap), local variables continue to be global. Using the frame rule, we can prove the correctness of a small piece of code and use the frame rule to extend the correctness to a bigger program including it. The small piece of code is not obliged to know about the memory it does not touch. But, for the local variables, the treatment is different, and all the pieces of code must know about all the existing local variables in order to preserve their properties.

For example, once we had proved the correctness of the following pieces of code:

A:

```
{7 > 0}
x := 7;
{x > 0}
```

B:

```
{5 = 5}
y := 5;
{y = 5}
```

If we want to prove the correctness of the composition of the two, even if we know that the variables are independent, in Hoare Logic we have to prove the pieces once again, as the assertions have changed.

```
{7 > 0 ∧ 5 = 5}
x := 7;
{x > 0 ∧ 5 = 5}
y := 5;
{x > 0 ∧ y = 5}
```

As a partial solution to the problem, functions might be used, since each functions has its own local variables, which are independent even if the names are shared.

Instead of that, in this section we present a simplification[1] of Variables as Resource in Hoare Logic [4], which applies an approach similar to the used in Separation Logic to local variables. Thanks to that, it is possible to use the frame rule to reason locally about code using independent sets of local variables. The simplification we present takes the assertions and inference rules from the sequential part of LRG.

In this logic the state, assertions and inference rules are updated. The programming language is maintained unchanged, but for simplification purposes we are going to ignore the function calls.

2.3.1 State

In this logic a new elements is added to the pair of store and heap, forming the new state $\sigma = (s, i, h)$. The new element is the logical variable map (i), used to store the values of logical variables necessary for specification. Until now, the logical variables were stored in the store together with the local variables. In Variables as Resource, they must be stored independently to allow specifying which local variables a program is using without having to worry about the logical variables.

Logical variables are stored in a similar way as to loclal variables, but instead of being a map from variable names to integers is a map from logical variable names to integers: $i : LVar \to \mathbb{Z}$. The sets of valid local variable names and logical variable names should be disjoint.

In order to allow easier identification, in the following sections we are going to use lower case letters for the local variables and upper case letters for the logical ones.

As a side note, since i is never mentioned in the programming language specification, the variables in i are never changed by the program.

2.3.2 Assertion language

In the new assertion language, B continues to represent a boolean expression an E and F an integer expression, but now they are evaluated over the union of s and i. The new

[1] The simplification we present takes the assertions and inference rules from the sequential part of LRG.

notation for the evaluation is: $[\![B]\!]_{s,i}$ and $[\![E]\!]_{s,i}$. Note that for the evaluation to be valid, s and i must be disjoint ($\text{dom}(s) \cap \text{dom}(i) = \varnothing$) and all the free variables in the assertions must be defined in the union of s and i. In case

In Variables as Resource, only few elements are added to the assertions grammar [2] , but the semantics of some of most of the existing ones change.

$$\begin{aligned} p,q,r ::= & \ B \mid E \mapsto F \\ & \mid \text{false} \mid p \Rightarrow q \\ & \mid \forall X.p \mid \text{emp}_s \mid \text{emp}_h \\ & \mid p * q \mid p -\!\!* \ q \\ & \mid p -\!\!\circledast \ q \mid \text{Own}(x) \end{aligned}$$

Before redefining the semantics of the assertions we first overload the meaning of $\perp$ and $\uplus$:

- $(s,i,h) \perp (s',i',h')$ iff $s \perp s' \wedge i = i' \wedge h \perp h'$
- $(s,i,h) \uplus (s',i',h') = (s \uplus s', i, h \uplus h')$

The new semantics, where $\sigma = (s,i,h)$:

$$\begin{array}{ll} \sigma \models B & \text{iff } [\![B]\!]_{s,i} = true \\ \sigma \models \text{emp}_s & \text{iff } \text{dom}(s) = \varnothing \\ \sigma \models \text{emp}_h & \text{iff } \text{dom}(h) = \varnothing \\ \sigma \models \text{Own}(x) & \text{iff } \text{dom}(s) = \{x\} \\ \sigma \models E \mapsto F & \text{iff } \text{dom}(h) = \{[\![E]\!]_{s,i}\} \wedge h([\![E]\!]_{s,i}) = [\![F]\!]_{s,i} \\ \sigma \models \text{false} & \text{never} \\ \sigma \models p \Rightarrow q & \text{iff } \sigma \models p \ \Rightarrow \ \sigma \models q \\ \sigma \models \forall X.p & \text{iff } \forall v \in \mathbb{Z}((s, i[X \leftarrow v], h) \models p) \\ \sigma \models p * q & \text{iff } \exists \sigma_1, \sigma_2.\ \sigma_1 \perp \sigma_2 \wedge \sigma_1 \uplus \sigma_2 = \sigma \wedge \sigma_1 \models p \wedge \sigma_2 \models q \\ \sigma \models p -\!\!* \ q & \text{iff } \forall \sigma'(\sigma \perp \sigma' \wedge \sigma \models p \ \Rightarrow \ \sigma \uplus \sigma' \models q) \\ \sigma \models p -\!\!\circledast \ q & \text{iff } \exists \sigma'(\sigma \perp \sigma' \wedge \sigma \models p \ \Rightarrow \ \sigma \uplus \sigma' \models q) \end{array}$$

[2] In order to be more consistent with most of the publications about Rely-Guarantee reasoning, we begin using lower case p and q to represent the assertions, but the meaning does not change.

The principal differences are, the addition of emp_s and $\text{Own}(x)$ to reason about the elements in the local variables, and the redefinition of $p * q$, $p -\!\!* q$ and $p -\!\!\circledast q$ to take into account also the local variables can be split in disjoint groups. Additionally, the $\forall$ quantifier is now applied over logical variables only.

Note that in order $\sigma \models B$ to be true, $[\![B]\!]_{s,i}$ must be true, and for that B must be correctly defined. That is why $B = B$ in not a tautology in this system, as free variables in B might be undefined in the current state. Also, $\neg(B1 = B2)$ and $B1 \neq B2$ are not equivalent.

In addition to the changes to the assertion language, we define de following syntactic sugar:

$$O ::= \bullet \,|x, O$$
$$x_1, \ldots, x_n, \bullet \Vdash p \stackrel{\text{def}}{=} (\text{Own}(x_1) * \cdots * \text{Own}(x_n)) \wedge p$$
$$\text{emp} \stackrel{\text{def}}{=} \text{emp}_s \wedge \text{emp}_h$$

Examples

Some examples of the new assertions:

1. There is a reserved memory cell in the heap and the store is empty:

$$\exists L\,(L \mapsto -) \wedge \text{emp}_s$$

2. There is a x variable in the store and its value is equal to the X logical variable:

$$x = X$$

3. x is the only variable in the store:

$$\text{Own}(x) \quad \text{or} \quad x \Vdash \text{true}$$

4. x and y are in the store, and the value of the memory address x is y:

$$x \mapsto y$$

5. x and y are the only values in the store, and the value of the memory address x is y:

$$x, y \Vdash x \mapsto y$$

2.3.3 Inference rules

In Variables as Resource, some of the rules from Hoare Logic and Separation Logic, must be adapted.

$$\overline{\{p\}\ \texttt{skip}\ \{p\}} \qquad \text{(Empty statement axiom)}$$

This rule is maintained without change.

$$\overline{\{x, O \Vdash X = E \wedge \mathrm{emp}_h\}\ x := E\ \{x, O \Vdash x = X \wedge \mathrm{emp}_h\}} \qquad \text{(Assigment axiom)}$$

This rule was heavily modified. The new rule says that if we have a variable x in the store (and other variables O), an empty heap and a logical variable X with the value of the E expression, after the assignment x is equal to X.

$$\frac{\{p\}\ C_1\ \{p'\} \quad \{p'\}\ C_2\ \{q\}}{\{p\}\ C_1 ; C_2\ \{q\}} \qquad \text{(Rule of composition)}$$

This rule is maintained without change.

$$\frac{p \Rightarrow p' \quad \{p'\} C\ \{q'\} \quad q' \Rightarrow q}{\{p\}\ C\ \{q\}} \qquad \text{(Consequence rule)}$$

This rule is maintained without change.

$$\frac{p \Rightarrow B = B \quad \{p \wedge B\}\ C_1\ \{q\} \quad \{p \wedge \neg B\}\ C_2\ \{q\}}{\{p\}\ \texttt{if (} B \texttt{) \{ } C_1 \texttt{ \} else \{ } C_2 \texttt{\}} \{q\}} \qquad \text{(Rule of condition)}$$

This rule has a new condition, which is that the precondition p implies that B is well defined, in other words, that all the free variables in B are present in s or i. The remaining of the rule does not change.

$$\frac{p \Rightarrow B = B \quad \{p \wedge B\}\ C\ \{p\}}{\{p\}\ \texttt{while (} B \texttt{) \{ } C \texttt{ \}}\ \{p \wedge \neg B\}} \qquad \text{(While rule)}$$

As the previous one, this rule also has the new $p \Rightarrow B = B$ precondition.

$$\frac{(x, O \Vdash x = X \wedge \mathrm{emp}_h) \Rightarrow E_1 = E_1 \wedge \cdots \wedge E_k = E_k}{\begin{array}{c}\{x, O \Vdash x = X \wedge \mathrm{emp}_h\} \\ x := \mathtt{cons}(E_1, \ldots, E_n) \\ \{x, O \Vdash x = Y \wedge (Y \mapsto [X/x]E_1, \ldots, [X/x]E_k)\}\end{array}} \quad \text{(Allocation rule)}$$

This rule was completely changed. The new condition requires that the precondition implies that all the expressions are well defined. The rest of the rule says that if x is in the store and has a value X, and the heap is empty, after the execution, x has value Y and Y points to a continuous memory region with values $[X/x]E_1, \ldots, [X/x]E_k$.

$$\frac{}{\{x, O \Vdash x = X \wedge E \mapsto Y\}\ x := [E_1]\ \{x, O \Vdash x = Y \wedge [X/x]E \mapsto Y\}} \quad \text{(Lookup rule)}$$

This rule was heavily changed. The new rule says that if in the store there is a x variable with the value X and the heap is single memory cell with address E and value Y, after the operation the x value is Y and the memory cell address is now $[X/x]E$.

$$\frac{O \Vdash E_1 \mapsto - \Rightarrow E_2 = E_2}{\{O \Vdash E_1 \mapsto -\}\ [E_1] := E_2\ \{O \Vdash E_1 \mapsto E_2\}} \quad \text{(Update rule)}$$

This rule is quite similar to the original. There is a new condition, which requires the precondition of the statement implies that E_2 is correctly defined, and the execution of the execution of the statement now maintains the store elements.

$$\frac{}{\{O \Vdash E \mapsto -\}\ \mathtt{dispose}\ E\ \{O \Vdash \mathrm{emp}_h\}} \quad \text{(Deallocation rule)}$$

The deallocation rule remains almost unchanged, with the exceptions that now maintains the store elements.

$$\frac{\{p\}\ C\ \{q\}}{\{\exists X.p\}\ C\ \{\exists X.q\}} \quad \text{(Auxiliary Variable Elimination)}$$

This rule is maintained almost unchanged, with the only difference of the lack of the side condition. The side condition is no longer necessary as X is a logical variable and can not appear in C.

(Substitution rule)

This rule is no longer necessary.

$$\frac{\{p\}\ C\ \{q\}}{\{p * r\}\ C\ \{q * r\}} \qquad \text{(Frame rule)}$$

This rule is maintained, but the side condition is no longer necessary as $*$ ensures the store values modified in C are independent from the ones in r.

In addition to the previous rules, there are two new rules:

$$\frac{\{p\}\ C\ \{q\} \quad \{p'\}\ C\ \{q'\}}{\{p \wedge p'\}\ C\ \{q \wedge q'\}} \qquad \text{(Conjuntion rule)}$$

and

$$\frac{\{p\}\ C\ \{q\} \quad \{p'\}\ C\ \{q'\}}{\{p \wedge p'\}\ C\ \{q \wedge q'\}} \qquad \text{(Disjuntion rule)}$$

which quite intuitively allow the use of conjunction and disjunction of preconditions and postconditions, if the conditions hold independently.

Example

As an example we are going to prove modularly the program mentioned in the introduction:

```
{x,y ⊩ X = 7 ∧ Y = 5}
x := 7;
y := 5;
{x,y ⊩ x = X ∧ y = Y}
```

First we are going to prove the each statement independently:

A:

```
{x ⊩ X = 7 ∧ emp_h}
x := 7;
{x ⊩ x = X ∧ emp_h}
```

Can be proved directly using the assignment axiom.

B:

```
{y ⊩ Y = 7 ∧ emp_h}
y := 5;
{y ⊩ y = Y ∧ emp_h}
```

Can be also proved using the assignment axiom.

Once the two statements are proved, we can extend the proof to the fool program. First we apply the frame rule on both statements:

$$\frac{\{x \Vdash X = 7 \wedge \mathrm{emp}_h\}\ \texttt{x := 7;}\ \{x \Vdash x = X \wedge \mathrm{emp}_h\}}{\{(x \Vdash X = 7 \wedge \mathrm{emp}_h) * (y \Vdash Y = 5)\}\ \texttt{x := 7;}\ \{(x \Vdash x = X \wedge \mathrm{emp}_h) * (y \Vdash Y = 5)\}}\ \text{(Frame rule)}$$

$$\frac{\{y \Vdash Y = 5 \wedge \mathrm{emp}_h\}\ \texttt{y := 5;}\ \{y \Vdash y = Y \wedge \mathrm{emp}_h\}}{\{(y \Vdash Y = 5 \wedge \mathrm{emp}_h) * (x \Vdash x = X)\}\ \texttt{y := 5;}\ \{(y \Vdash y = Y \wedge \mathrm{emp}_h) * (x \Vdash x = X)\}}\ \text{(Frame rule)}$$

After that we can use the consequence rule to get the following judgements:

$$\{x, y \Vdash X = 7 \wedge Y = 5\}\ \texttt{x := 7;}\ \{x, y \Vdash x = X \wedge Y = 5\}$$
$$\{x, y \Vdash x = X \wedge Y = 5\}\ \texttt{x := 5;}\ \{x, y \Vdash x = X \wedge y = Y\}$$

Finally, we only have to apply the rule of composition to prove the full program:

```
{x,y ⊩ X = 7 ∧ Y = 5}
x := 7;
y := 5;
{x,y ⊩ x = X ∧ y = Y}
```

The postcondition might be a bit strange, as instead of having $x = 7$ and $y = 5$ we have $x = X$ and $y = Y$. But remember that X and Y are logical variables and their value do not change. That is why $x = 7$ and $x = X$, and $y = 5$ and $y = Y$ are equivalent.

If we compare this methodology with the one used in Hoare logic, it might seem that it is more complicated for the same result. Nevertheless, it is important to note that usually

A and B are going to be much bigger programs, where avoiding proving once again the code could reduce significantly the cost of the full proof.

2.4 Dafny

Dafny [14] is a program verification tool which includes a programing language and specification constructs. The Dafny programming language is primarily an imperative programing language, but it has support for additional paradigms as object orientation and functional programming. The object support is somehow limited, since inheritance is not supported at the time of writing. Nevertheless, generic classes and dynamic allocation are implemented. From the point of view of functional programming, the language has support for algebraic data types, recursive functions and predicates, and basic pattern matching. The specification constructs include pre- and postconditions, to specify functions, methods and predicates, framing constructs to allow local reasoning when working with dynamic memory and pointers, and termination metrics to prove *total correctness*. All the specification constructs are real entities of the programming language,included in the grammar, avoiding the need to use specifically crafted comments as a way to represent them, which is the case in other verifications tools like KeY [1] or frama-c [5].

At the compilation time, Dafny program are statically verified for *total correcness*, ensuring all the contracts are meet and that every execution does terminates. The verification process work by translating the Dafny code to an intermediate language Boogie [2] specifically designed to work as a back-end for other program verification tool. The translation must ensure that the correctness of the Boogie program implies the correctness of the original program. Boogie is used to generate first order verification conditions, which are given to a logic reasoning engine to be proved correct. In the case of Dafny, the used engine is the Z3 [6] satisfiability modulo theories (SMT) solver.

If the solver is capable of verifying the conditions in a specified time limit, the original Dafny code is once again translated, but this time to C# [7] code, which is a hight level general purpose programming language developed by Microsoft. This C# code can be compiled to the Common Intermediate Language (CIL), which can be executed using a runtime such as Microsoft's .Net Framework or Mono. The biggest advantage of using C# as a target is it allows Dafny code to interoperate with all the other languages of the CLI [8] environment. Thanks to that, it is possible to use Dafny to verify safety critical code in already existing projects, while the rest of the code is maintained in the original

language without modification.

In the following sections we proceed to explain the features of Dafny we used in the implementation.

2.4.1 Functions and predicates

Dafny programming language has support for functions and predicates (being the last ones simply syntactic sugar to functions returning a boolean value). The functions on Dafny do not accept code (statements), the body of the function must be an expression, which can be recursive.

For example, the `add` function returns the sum of two integers:

```
function add(a: int, b: int): int {
    a + b
}
```

and the `fact` function calculates the factorial:

```
function fact(n: int): int {
    if n <= 1 then 1
    else n*fact(n-1)
}
```

All functions in Dafny are pure and total by default. A function to be pure means that the same input parameters are always going to produce the same result and that the function does not have any side effect, in other words, the function does not depend on the global state neither it changes it. A function to be total means that the function is defined for all the possible inputs. For example, the division of integers is not total, as division by zero is not defined, even if zero is an integer. On the other hand, the addition of integers is total.

If we want a Dafny function not to be pure, i.e., the value of the function to depend in a global variable, we must specify it using the `reads` clause. This clause is especially useful when the function accepts pointers, as the value pointed by the pointer might change while the pointer is maintained. We present an example predicate, which checks if the first value from an array is bigger than 5. Arrays in Dafny are reference types, so they are accessed through pointers. For now ignore the requires clause.

```
predicate firstGr5(a: array<int>)
    requires a != null && a.Length > 0
    reads a
{
    a[0] > 5
}
```

On the other hand there is no way in Dafny for a function to modify global state.

If we want a function not to be total, we must declare a condition to limit which inputs are accepted using the `requires` keyword. For example, the `firstGr5` predicate requires the a pointer not to be null and the array to be non empty.

At compilation time, the Dafny runtime checks that the body of the function is well defined for all the acceptable inputs and also it does not accede any state outside the ones listed in the read. It also checks all the preconditions (`requires`) are meet in any point a function is called.

Additionally, if the function is recursive, Dafny also checks if any function call terminates. To do so, Dafny searches a decreasing expression. This decreasing expression is analogous to the one mentioned in the recursion extension for Hoare logic. The expression must depend on the parameters of the function, and must accept a well founded decreasing metrics, i.e, there is a minimum acceptable value for the expression. If Dafny is not able to find the expression automatically, we can provide one using the `decreases` keyword.

As an example, for the factorial function, a suitable decreasing expression is n:

```
function fact(n: int): int
    decreases n
{
    if n <= 0 then 1
    else n*fact(n-1)
}
```

the lower bound of n is 0 as for values less than zero no recursive call is made thanks to the condition of the if.

In some cases, in can be interesting to have functions that could not terminate. For those cases, the `inductive` keyword is added before `predicate` (at the time of writing this report inductive is only implemented for predicates and not functions returning other types

of values). As inductive predicates do not terminate, they acquire the semantics of the minor fix point of the predicate, and thanks to that their properties can be proved.

Dafny functions accept also an additional `ensures` clause. This clause can be used to specify a property of the returned value that the function satisfies. For example:

```
function fact(n: int): int
    ensures fact(n) >= n
    decreases n
{
    if n <= 0 then 1
    else n*fact(n-1)
}
```

The ensures is statically checked at compilation time to ensure it is true. In some cases, the property is to complex to prove it automatically, and a lemma should be used to prove it.

2.4.2 Lemmas

Lemmas are the basic construct of Dafny for proving things. Lemmas are ghost methods (i.e. their body is formed by a sequence of statements) but unlike the normal methods they are not compiled to executable code.

Each lemma has a `ensures` clause which is proved to be true. Additionally, the lemmas might also include a `requires` to specify some precondition. For example, we present the `FactGr` lemma, which ensures the factorial of n is always bigger or equal to n:

```
lemma FactGr(n: int)
    requires n >= 0
    ensures fact(n) >= n
```

(The requires is not necessary here, but we add it for illustrative purposes.)

If the lemma is declared without a body, it is leaved unproved, but it can be used from other points, allowing leaving properties to be proved lately. If a body is added to the lemma, the lemma is checked to be correct.

For simple properties, like the one used in `FactGr`, an empty body is enough for Dafny to prove the correctness. In other cases, helping Dafny is necessary. There are many approaches we can use.

The first one is to add assertions in the body. Assertions are expressions checked to be true at the point they are executed. Once the assertions are proved, they are added to the knowledge Dafny has about the proof in that point, helping him to prove more complex things. In some cases, it might be interesting to check if Dafny is able to prove something giving it some knowledge, but whitout proving this knowledge is true. To do so there is the `assume` keyword, which works exactly as the assert but which is not checked to be true.

Some examples of the assertion and assumes are:

```
assert x >= 5;
assume r < 7;
assume x + x + x >= x*x;
```

The second one is to add lemmas. When a lemma is called in a body, its precondition is checked to be true, and the lemmas ensure is introduced in the knowledge base. A lemma call is equivalent to adding a assert with the precondition and a assume with the ensure:

```
FactGr(5);
// equivales to
assert 5 > 0;
assume 5 >= fact(5);
```

The third one is to use `ifs` inside the body to separate the prove in two. If the two sides of the if are proved, the ensemble is also true. Additionally, using `if` allows to make proves inductively: one of the braces of the if is the base case, and the other one(s) are the inductive cases. Inside the inductive cases, we call the lemma itself but with a lesser parameter. As for the functions, Dafny checks the lemma always terminates searching a decreasing expression, and asking for one if necessary.

Combining the tree techniques mentioned before, it is possible to prove interesting properties. For example, we are going to prove the sum of the first n numbers multiplied by 2 is equal to $n(n+1)$:

```
function sum(n: int): int
    requires n >= 0
{
    if n == 0 then 0
    else n + sum(n-1)
}

lemma sumEq(n: int)
    requires n >= 1
    ensures sum(n)*2 == n*(n+1)
{
    if n == 1 {
    } else {
        sumEq(n-1);
        assert sum(n-1)*2 == (n-1)*n;
        assert sum(n-1)*2 + 2*n == (n-1)*n + 2*n;
        assert sum(n)*2 == (n-1+2)*n;
        assert sum(n)*2 == (n+1)*n;
        assert sum(n)*2 == n*(n+1);
    }
}
```

In the example we are helping Dafny more than necessary for illustration purposes. This example can be proved using an empty body.

Sometimes, having a long list of asserts one after another is quite tedious. For those cases the `calc` environment can be used.

```
lemma sumEq(n: int)
    requires n >= 1
    ensures sum(n)*2 == n*(n+1)
{
    if n == 1 {
    } else {
        sumEq(n-1);
        calc {
            sum(n-1)*2 == (n-1)*n;
            ==>
            sum(n-1)*2 + 2*n == (n-1)*n + 2*n;
            ==>
            sum(n)*2 == (n-1+2)*n;
            ==>
            sum(n)*2 == (n+1)*n;
            ==>
            sum(n)*2 == n*(n+1);
        }
    }
}
```

`calc` also accepts using == (equality) or < and > between the steps. Additionally, if some step is very difficult and requires additional information to be made, intermediate assertions, assumes and lemmas can be called:

```
calc {
    x == y;
    ==> {assert a == b;}
    x + a == x + b;
}
```

There is a useful method to prove lemmas, which is the prove by contradiction. It is usually done with a calculation beginning with a not followed by the precondition and ended in a false:

```
lemma less(a: int, b: int)
    requires a < b
    ensures a + 5 < b + 5
{
    calc {
        ! (a+5 < b+5) && (a < b);
        ==>
        ! (a < b) && (a < b);
        ==>
        false;
    }
}
```

2.4.3 Algebraic data types

Dafny has support for algebraic data types. Algebraic data types are a way to create new types typically used in functional programming. They take a similar role to the structs or objects in object oriented programs.

Algebraic data types are defined by the type name and a list of constructors:

```
datatype example = A | B | C(int) | D(example)
```

In the example A, B, C and D are the constructors and example is the type name. The constructors can take different arguments. A and B take no argument, and they are constant constructors. C takes an integer as an argument and D takes another example as an argument.

To create an instance of a example variable, we call one of the constructors.

```
    var a: example := A;
    var b: example := B;
    var c: example := C(5);
    var d: example := D(a);
```

(The last one takes a as it needs other example as parameter).

Almost any type of data representable by computer can be encoded using data types. For example, the booleans:

```
datatype boolean' = true | false
```

naturals (0 is represented by Z, 1 by C(Z), 2 by C(C(Z)), ...):

```
datatype nat' = Z | C(nat')
```

Polymorphic lists:

```
datatype list<T> = Nil | Cons(head: T, tail: list<T>)
```

head and tail are called destructor's, and are optional.

Being l a variable of type list there are several things we can do:

- l.Nil? is a predicate returning true if l was constructed with Nil.
- l.Cons? is a predicate returning true if l was constructed with Cons.
- If l.Cons? then l.head returns the variable of type T used to construct l.
- If l.Cons? then l.tail returns the list used to construct l.

Finally, instead of using destructors, it is possible to access the elements of a data type using match:

```
math l
case Nil => ...
case Cons(h, t) => ... // here h is the integer
                       // and t is the list
```

As a final note, algebraic data types are immutable. This means that the only way to change a value from a data type is to create a copy with the value changed.

3. CHAPTER

Proving the correctness of concurrent programs

Concurrency is a very powerful tool, it allows programs to made a better usage of the system resources, especially in multi core systems. For example, if a program is divided into two threads, one for the interactions with the user and the other to made the relevant connections to Internet, while the user is giving some input slowly, the second thread can take advantage of the unused processor time to exchange data with the network. Other example, if a computationally taxing application is being executed in a multi core system, and the program is divided into multiple threads, the processor will be able to execute calculations in all cores at the same time, speeding up the application by the number of cores.

There are multiple models for concurrency, being the most common and most efficient one the memory sharing model. In this model, the threads communicate between them using some memory locations which are accessible for all of them. The problem of this model is it makes formal program verification very difficult, as some memory locations might change at arbitrary times, causing the assertions about those memory locations to be violated.

If instead shared memory model, message based models (like the actor model) are used, it is easy to use separation logic to reason about concurrent programs, using the following inference rule:

$$\frac{\{P_1\}C_1\{Q_1\} \quad \{P_2\}C_2\{Q_2\}}{\{P_1 * P_2\}C_1\|C_2\{Q_1 * Q_2\}} \tag{3.1}$$

where $C_1\|C_2$ stands for concurrent execution of C_1 and C_2. The operator $*$ ensures that all the memory addresses from the preconditions $P1$ and $P2$ and postconditions $Q1$ and $Q2$

are independent, avoiding any possible interferences between the two threads.

In practice, real computers use shared memory model, and even if it is possible to implement alternative models at software level, in order to verify these implementations a new methodology must be used.

Rely-Guarantee reasoning is a method for verification of shared-memory concurrent programs. Rely-Guarantee can be combined with Separation logic and Variables as Resource to form Local Rely-Guarantee Reasoning (LRG) [9].

3.1 Local Rely-Guarantee reasoning

Rely-Guarantee reasoning is a tool to verify concurrent programs making use of shared-memory concurrency. Most of the state accessed by the threads is considered private and it is analyzed using standard techniques as Hoare logic or Separation logic. But for the shared data a new model needs to be established.

In concurrent programs threads might interfere between them changing the values of the shared memory in a nondeterministic way. Changes can happen, before, after or in the middle of any command (except if the command is atomic). All the threads must be aware of the possible interferences. For that, Rely-Guarantee reasoning introduces the two state assertions. This assertions relate two states, both specified using standard assertions: $p \ltimes q$. If the program is in a state satisfying p, we know that a transition to a state satisfying q might happen. Everything else, not listed in the interferences, is guaranteed to not happen.

The interferences a thread causes are the guarantee conditions of the thread G. The interferences a thread must be aware are the rely conditions of the thread R. The rely conditions of one thread are always going to be the combination of the guarantee conditions of the remaining threads.

In order to prove the correction of a thread having its rely conditions, the standard pre a postconditions used in the proof must be stablished. This means finding equivalent assertions which are not affected by the transitions specified in G. Since the assertions are not affected, the proofs maintain the correctness even in concurrent executions.

There are many variations of Rely-Guarantee reasoning. The most simple ones reason only about programs with a local state composed by a store and a global state with limited variables. There are more advanced ones, like RGSep [19], which combine separa-

tion logic with rely-guarantee reasoning. The one analyzed in this report is LRG: Local Rely-Guarantee reasoning. This one combines in addition to Separation logic an Rely-Guarantee reasoning, the Variables as Resource approach. Thanks to that and other improvements, in this logic it is possible to prove concurrent programs in a modular way, since unlike the previous one, it is not necessary that the shared state used in some sections of the code to be known by other parts of the program.

LRG uses the same state and basic assertions we have seen in the Variables as Resource, but the programming language is extended with parallel and atomic constructs, and a new type of assertions, the actions are introduced.

3.1.1 Programming language

In LRG, the concurrency is implemented at the language level, extending the programming language used in separation logic with two new statements.

Follows the new grammar:

```
C ::= ...
    | C '||' C
    | 'atomic' '(' B ')' '{' C '}'
```

The new statements are the parallel execution statement and the atomic statement. The first one says that the statement to the right and to the left of the operator $\|$ are going to be executed in a concurrent way. The small step semantics are the following:

$$\frac{\sigma, C_1 \rightsquigarrow \sigma', C_1'}{\sigma, (C_1 \| C_2) \rightsquigarrow \sigma', (C_1' \| C_2)} \qquad \frac{\sigma, C_2 \rightsquigarrow \sigma', C_2'}{\sigma, (C_1 \| C_2) \rightsquigarrow \sigma', (C_1 \| C_2')}$$

$$\frac{}{\sigma, (\mathtt{skip} \| \mathtt{skip}) \rightsquigarrow \sigma', \mathtt{skip}}$$

where $\sigma = (s, i, h)$.

The second one, the atomic statement, says that in the moment B is true, the C instruction is going to be executed atomically, i.e. no intermediate step from C are going to be visible externally and during its execution no other thread might modify the shared state. At the beginning of C B continues to be true, since the evaluation of B is done inside the atomic

region. It follows the small step semantic:

$$\frac{[\![B]\!]_\sigma = \text{true} \quad \sigma, C \rightsquigarrow^* \sigma', \texttt{skip}}{\sigma, \texttt{atomic}\ (B)\ \{\ C\ \} \rightsquigarrow \sigma', \texttt{skip}} \qquad \frac{[\![B]\!]_\sigma = \text{true} \quad \sigma, C \rightsquigarrow^* \textbf{abort}}{\sigma, \texttt{atomic}\ (B)\ \{\ C\ \} \rightsquigarrow \textbf{abort}}$$

where $\rightsquigarrow^*$ represent zero or more steps.

In order to ease the use of atomic, the syntactic sugar $\langle C \rangle$ is equivalent to $\texttt{atomic}(\text{true})\{\ C\ \}$.

Example

The following example program creates a dynamic variable x and a lock l, and spawns two threads. Each threads acquires the lock, adds one to x (memory pointed by x) and releases the lock.

```
x := cons(0);
l := 0;

{
    atomic(l == 0) {
        l := 1;
    }
    ⟨t1 := [x]⟩;
    t1 := t1 + 1;
    ⟨[x] := t1⟩;
    [x] := t1;

    ⟨l := 0⟩;
} || {
    atomic(l == 0) {
        l := 1;
    }

    ⟨t2 := [x]⟩;
    t2 := t2 + 1;
    ⟨[x] := t2⟩;
    [x] := t2;

    ⟨l := 0⟩;
}
```

3.1.2 Assertion language

The assertion language for LRG logic is far more complex than the one used for Hoare logic and Separation logic.

In Rely-Guarantee reasoning, an special kind of assertion, called action, must be specified. This actions are a relation between two standard assertions, and specify modifications to the shared state. If we can prove the execution of any number of actions maintain the validity of the standard assertions in a piece of code, a property called stability, we can prove the code works correctly even in a concurrent environment. Since all the shared state can only be modified in a way specified by the actions and the actions have no effect in the assertions, the code must be correct.

In order to use the actions correctly, the local and shared state of the threads must be differentiated. In the first approaches, the heap was divided in two disjoint partial functions, and an special notation and operations were used to allow moving data between them. In LRG instead of using that, invariant-fenced actions are used. In this approach, all the modification done by the actions must be limited by an invariant which is a standard assertion. What is mentioned in the invariant is part of the shared state, while the rest is considered private. The invariant can change within the program, allowing moving data between the states.

The grammar for the assertions in LRG is:

$$\begin{aligned} p,q,r,I &::= B \mid \mathrm{emp}_h \mid \mathrm{emp}_s \mid \ldots \\ a,R,G &::= p \ltimes q \mid [p] \mid \exists X.a \mid a \Rightarrow a \\ &\quad \mid a \wedge a \mid a \vee a \mid \ldots \end{aligned}$$

p, q, r and I are standard assertions equivalent to the ones from the Variables as Resource section, with the same semantic. a, R and G are actions.

$(\sigma,\sigma') \models a$ is a satisfaction judgement which says that the action a holds for the pair of states σ and σ', where the semantics are defined as follows (being $\sigma = (s,i,h)$ and

$\sigma' = (s', i', h')$:

$$
\begin{array}{lll}
(\sigma,\sigma') \models p \ltimes q & \text{iff} & i = i' \wedge \sigma \models p \wedge \sigma' \models q \\
(\sigma,\sigma') \models [p] & \text{iff} & \sigma = \sigma' \wedge \sigma \models p \\
(\sigma,\sigma') \models a * a' & \text{iff} & \exists \sigma_1,\sigma_2,\sigma_1',\sigma_2' \ (\sigma_1 \uplus \sigma_2 = \sigma \wedge \sigma_1' \uplus \sigma_2' = \sigma' \\
 & & \wedge(\sigma_1,\sigma_1') \models a \wedge (\sigma_2,\sigma_2') \models a') \\
(\sigma,\sigma') \models \exists X.a & \text{iff} & i = i' \wedge \exists n, i'' \ (i'' = i[X \leftarrow n] \\
 & & \wedge((s,i'',h),(s',i'',h')) \models a) \\
(\sigma,\sigma') \models a \Rightarrow a' & \text{iff} & (\sigma,\sigma') \models a \Rightarrow (\sigma,\sigma') \models a' \\
(\sigma,\sigma') \models a \wedge a' & \text{iff} & (\sigma,\sigma') \models a \wedge (\sigma,\sigma') \models a' \\
(\sigma,\sigma') \models a \vee a' & \text{iff} & (\sigma,\sigma') \models a \vee (\sigma,\sigma') \models a'
\end{array}
$$

In other words:

- $(\sigma,\sigma') \models p \ltimes q$ asserts p holds over the first initial state and q over the final state.
- $(\sigma,\sigma') \models [p]$ asserts that the state is maintained and satisfies p.
- $(\sigma,\sigma') \models a * a'$ asserts that both the initial and the final state can be divided in two independent states, one pair satisfying a and the other satisfying a'.
- $(\sigma,\sigma') \models \exists X.a$ asserts that exists some value for the variable X in i for which a in satisfied.
- The rest of them are standard logical connectives with their usual meaning.

Some syntactic sugar is also defined:

$$\text{Emp} \stackrel{\text{def}}{=} \text{emp} \ltimes \text{emp} \quad \text{True} \stackrel{\text{def}}{=} \text{true} \ltimes \text{true} \quad \text{Id} \stackrel{\text{def}}{=} [\text{true}]$$

Precise assertions

The precision is a property of the assertions. If an assertion is precise, it means that for any possible state there is one and only one sub-state which satisfies the assertion.

Definition 1 (Precise Assertions) *An assertion p is precise, i.e.,* precise(p) *holds, if and only if* $\forall s,i,h,s_1,s_2,h_1,h_2 \ (s_1 \subseteq s \wedge s_2 \subseteq s \wedge h_1 \subseteq h \wedge h_2 \subseteq h \wedge (s_1,i,h_1) \models p \wedge (s_2,i,h_2) \models p \Rightarrow s_1 = s_2 \wedge h_1 = h_2)$.

This property will be mentioned and used in the following sections.

Stability

As mentioned before, stability is the property allowing us to reason about concurrent programs. We can consider actions events might occur while executing some piece of code. Stability allows us to check if this events might influence the execution and results of the code.

Definition 2 (Stability) *We say p is stable with respect to the action a, i.e.,* $\mathrm{Sta}(p,a)$ *holds, if and only if* $\forall \sigma, \sigma'(\sigma \models p \wedge (\sigma, \sigma') \models a \Rightarrow \sigma' \models p)$.

Informally, stability means that the validity of p is conserved for any transition satisfying a.

In order to allow doing local reasoning about concurrent executions, the following property will be very interesting:

$$\mathrm{Sta}(p,a) \wedge \mathrm{Sta}(p',a') \Rightarrow \mathrm{Sta}(p * p', a * a')$$

Unfortunately, this property does not hold in the vast majority of cases, even if we restrict p to be precise. That is why invariant-fenced actions are introduced.

Invariant-Fenced Actions

Invariant-fenced actions are actions whose effects are limited by a precise invariant I. This means I restricts the variables and memory that might be accessed by the action, and also the changes that might occur.

Definition 3 (Fence) *An action a is fenced by invariant I, denoted $I \rhd a$, if and only if* $([I] \Rightarrow a) \wedge (a \Rightarrow (I \ltimes I)) \wedge \mathrm{precise}(I)$.

Informally, the $I \rhd a$ means that a holds over any identity transition satisfying I and that any transition satisfying a conserves the invariant I.

The reason why I can be used to divide the local and shared memory of the treads is the following: a conserves the invariant I and I is a separation logic assertion. Since in separation logic assertions everything not mentioned in the assertions does not exist or is

it maintained, a conserving I means that only memory addresses and variables referenced in I could change. That is why what is referenced in I is the shared state, and the rest local state.

Invariant-fenced actions allow using the frame rule for local reasoning thanks to the following property:

Lemma 1 $(\mathrm{Sta}(p,a) \wedge \mathrm{Sta}(p',a') \wedge (p \Rightarrow I) \wedge I \rhd a) \Rightarrow \mathrm{Sta}(p * p', a * a')$

Examples

Some simple actions:

1. Free the memory at address x:

$$(x \mapsto -) \ltimes (emp_h)$$

2. Maintain the value at memory address m:

$$[m \mapsto -]$$

3. Maintain m reserved but the value might change:

$$(m \mapsto -) \ltimes (m \mapsto -)$$

4. Increase the value of x variable:

$$(x = X) \ltimes \exists X'(X' > X \wedge x = X')$$

5. Add 5 or subtract 2 to y:

$$(y = Y) \ltimes (y = Y + 5 \wedge y = Y - 2)$$

Examples of pure assertions:

1. $x, y \Vdash x \mapsto y$

2. $x \Vdash x = X \wedge \mathrm{emp}_h$

3. $x \Vdash x \mapsto -$

4. $x, y \Vdash (x \mapsto y * y \mapsto x)$

An example of stability: Let

- $p \stackrel{\text{def}}{=} a_1 \mapsto v_1$
- $G \stackrel{\text{def}}{=} (a_2 \mapsto v2) \ltimes (a_2 \mapsto v_2 + 1)$

$\mathrm{Sta}(p, G)$ holds.

An example of a fence: Let

- $I \stackrel{\text{def}}{=} L \mapsto - \wedge \mathrm{emp}_s$
- $a \stackrel{\text{def}}{=} [L \mapsto - \wedge \mathrm{emp}_s] \vee \exists Y.(L \mapsto Y \ltimes L \mapsto Y + 1)$

$I \triangleright a$ holds.

3.1.3 Inference rules

In Local Rely-Guarantee reasoning the inference rules are divided into two groups: the sequential rules, used when R and G conditions are empty, and the inference rules for concurrency, used when interferences might happen. The rules of the first group are the ones reviewed previously in the Variables as Resource section, in the previous chapter.

In this section we will present only the inference rules for concurrency. Note that some rules of the concurrent group depend on the correctness of a sequential code section, to be proved with the sequential rules.

For the concurrent rules, the program behaviour is represented using the following type of expression, called the judgement for well-formed concurrent programs:

$$R; G; I \vdash \{p\}\ C\ \{q\}$$

where R is the rely condition, G is the guarantee condition, I is the invariant used to fence R and G, p is the precondition and q is the postcondition. If the expression is proved correct, the code C is guaranteed to satisfy the q postcondition for any suitable execution.

The inference rules for concurrency[1]:

$$\frac{\{p\}\ C\ \{q\}}{\mathrm{Emp};\mathrm{Emp};\mathrm{emp} \vdash \{p\}\ C\ \{q\}} \quad \text{(ENV)}$$

This rule says that if the rely and guarantee condition are empty the code correctness can be proved using just the sequential rules. Since the shared resource is empty this is quite intuitive.

$$\frac{R;G;I \vdash \{p\}\ C_1\ \{q\} \quad R;G;I \vdash \{q\}\ C_2\ \{r\}}{R;G;I \vdash \{p\}\ C_1;C_2\ \{r\}} \quad \text{(P-SEQ)}$$

This rule is analogous to the sequential rule of composition. The postcondition of the first statement and the precondition of the second one must match, and R, G and I are conserved.

$$\frac{\begin{array}{c} p \Rightarrow B = B \quad \{p \wedge B\}\ C\ \{q\} \quad \mathrm{Sta}(\{p,q\}, R*\mathrm{Id}) \\ p \ltimes q \Rightarrow G * \mathrm{True} \quad p \vee q \Rightarrow I * \mathrm{true} \quad I \triangleright \{R,G\} \end{array}}{R;G;I \vdash \{p\}\ \mathtt{atomic}(B)\{C\}\ \{q\}} \quad \text{(ATOMIC)}$$

where $\mathrm{Sta}(\{p,q\}), R*\mathrm{Id}) \stackrel{\mathrm{def}}{=} \mathrm{Sta}(p, R*\mathrm{Id}) \wedge \mathrm{Sta}(q, R*\mathrm{Id})$ and $I \triangleright \{R,G\} \stackrel{\mathrm{def}}{=} I \triangleright R \wedge I \triangleright G$. This rule is used to prove the correctness of atomic statements. The conditions are:

1. the precondition p implies B is well defined,
2. $\{p \wedge B\}\ C\ \{q\}$ can be proved using sequential rules,
3. p and q are stable under R and the identity transition,
4. the transition from p to q imply G for the shared state and true for the rest,
5. p or q imply the invariant for the shared state and true for the rest, and
6. both R and G are fended by the I invariant.

Conditions 1 and 2 are similar to the sequential conditions for the if. The 3rd condition ensures that the sequential proof can be extended to the concurrent version as p and q are maintained in the case of interference. The 4th ensures the code satisfies its guarantee conditions. The 5th ensures the invariant is conserved by the C statement and the 6th one is the fence to separate shared and local state.

[1] The names used for the rules correspond with the names used in [9].

$$\frac{p \Rightarrow (B = B) * I \quad R;G;I \vdash \{p \wedge B\}\ C\ \{p\}}{R;G;I \vdash \{p\}\ \texttt{while (} B \texttt{) \{ } C \texttt{ \}}\ \{p \wedge \neg B\}} \quad \text{(P-WHILE)}$$

This rule is analogous to the sequential rule, but requiring C to be held also for R, G and I, and p to imply the resources to evaluate B are disjoint with I, i.e., they are not part of the shared state.

$$\frac{p \Rightarrow (B = B) * I \quad R;G;I \vdash \{p \wedge B\}\ C_1\ \{q\} \quad R;G;I \vdash \{p \wedge \neg B\}\ C_2\ \{q\}}{R;G;I \vdash \{p\}\ \texttt{if (} B \texttt{) \{ } C_1 \texttt{ \} else \{ } C_2 \texttt{\}}\{q\}} \quad \text{(P-IF)}$$

As the previous P-WHILE rule, this one also is analogous to the sequential rule, and have the same additional requirements.

$$\frac{\begin{array}{c} R \wedge G_2;G_1;I \vdash \{p_1 * r\}\ C_1\ \{q_1 * r_1\} \\ R \wedge G_1;G_2;I \vdash \{p_2 * r\}\ C_2\ \{q_2 * r_2\} \\ r \vee r_1 \vee r_2 \Rightarrow I \quad I \rhd R \end{array}}{R;G_1 \vee G_2;I \vdash \{p_1 * p_2 * r\}\ C_1 \mid\mid C_2\ \{q_1 * q_2 * (r_1 \wedge r_2)\}} \quad \text{(PAR)}$$

This rule is used to verify parallel executions. The disjoint preconditions p_1 and p_2 are distributed between the threads C_1 and C_2 as the private state. The initial r and final r_1 and r_2 are the shared state between the threads. The conditions of the rule are that each thread is independently derivable using the inference rules for concurrency, that the disjuntion of the shared state implies I and that I fences R. Note that the rely conditions for the independent executions are formed adding the guarantee condition of the other thread to the initial R of the parent. Additionally, the guarantee of the parent thread is the disjunction of the rely of the child's.

$$\frac{R;G;I \vdash \{p\}\ C\ \{q\} \quad \mathrm{Sta}(r, R' * \mathrm{Id}) \quad I' \rhd \{R', G'\} \quad r \Rightarrow I' * \mathrm{true}}{R * R', G * G', I * I' \vdash \{p * r\}\ C\ \{q * r\}} \quad \text{(FRAME)}$$

The frame rule for the concurrent version is quite more complex that the sequential one. This rule allows local reasoning as it ensures the correctness of some concurrent code is not lost if a disjoint state is added. The conditions of the rule are:

1. $R;G;I \vdash \{p\}\ C\ \{q\}$ is concurrently derivable,
2. r (assertion over shared state) is stable under R and the identity transition,
3. I' fences both R' and G' and
4. r implies I' invariant for the shared state and true for the rest.

This rule is correct thanks to the Lemma 1. The first condition will ensure p and q are stable under R, and the conditions 2, 3 and 4 ensure the other prerequisites to ensure that the union of p and r and q and r is going to be stable under $R*R'$.

$$\frac{R*R';G*G';I*I' \vdash \{p\}\ C\ \{q\} \quad I \rhd \{R,G\}}{R,G,I \vdash \{p\}\ C\ \{q\}} \tag{HIDE}$$

If part of the shared state used in C (as C might create child threads sharing data) is not accessible from the outside this rule allows to leave this part unspecified. R', G' are the rely conditions over the shared state fenced by I'. When we delete I' from the invariant, implicitly we are moving data from the shared state to the local state, making it inaccessible from the outside and removing the need of R' and G'.

$$\frac{R;G;I \vdash \{p\}\ C\ \{q\} \quad x \notin \text{free}(R,G,I)}{\{\exists X.p\}\ C\ \{\exists X.q\}} \tag{P-EX}$$

where $\text{free}(R,G,I) = \text{free}(R) \cup \text{free}(G) \cup \text{free}(I)$. This rule is analogous to the auxiliary variable elimination rule for sequential programs, but adapted to add new restrictions requiring x in not free in R, G or I.

$$\frac{R;G;I \vdash \{p\}\ C\ \{q\} \quad R;G;I \vdash \{p'\}\ C\ \{q'\}}{R;G;I \vdash \{p \wedge p'\}\ C\ \{q \wedge q'\}} \tag{P-CONJ}$$

and

$$\frac{R;G;I \vdash \{p\}\ C\ \{q\} \quad R;G;I \vdash \{p'\}\ C\ \{q'\}}{R;G;I \vdash \{p \wedge p'\}\ C\ \{q \wedge q'\}} \tag{P-DISJ}$$

This two rules are equivalent to the conjunction and disjunction rule for sequential programs.

$$\frac{\begin{array}{c} p' \Rightarrow p \quad R' \Rightarrow R \quad G \Rightarrow G' \quad q \Rightarrow q' \\ R;G;I \vdash \{p\} C\ \{q\} \\ p' \vee q' \Rightarrow I' * true \quad I' \rhd \{R',G'\} \end{array}}{R';G';I' \vdash \{p'\}\ C\ \{q'\}} \tag{CSQ}$$

This rule is analogous to the consequence rule for sequential programs, but new requirements have been added to ensure the consistency. R' must imply R, G must imply G', and the new invariant I' should be implied by $p' \vee q'$ and must fence both R' and G'.

In addition to those rules, four useful inference rules can be derived from the previous ones:

$$\frac{\{p\}\ C\ \{q\} \quad \mathrm{Sta}(r, R*\mathrm{Id}) \quad I \rhd \{R, G\} \quad r \Rightarrow I * \mathrm{true}}{R; G; I \vdash \{p * r\}\ C\ \{q * r\}} \quad \text{(ENV-SHARE)}$$

This rule is derived form the ENV and FRAME rules, and it allows to use sequential reasoning to prove the correctness of C if C does not access the shared state.

$$\frac{R; G; I \vdash \{p\}\ C\ \{q\}}{R; G; I \vdash \{p * r\}\ C\ \{q * r\}} \quad \text{(FR-PRIVATE)}$$

This rule is derived from the rule FRAME when R' and G' are Emp and I' is emp. It allow to ignore some of the shared state when it is not used by the code.

$$\frac{R; G; I \vdash \{p\}\ C\ \{q\} \quad \mathrm{Sta}(r, R') \quad I' \rhd \{R', G'\} \quad r \Rightarrow I'}{R * R', G * G', I * I' \vdash \{p * r\}\ C\ \{q * r\}} \quad \text{(FR-SHARE)}$$

This rule is derived from the rule FRAME when r only contains shared state. It's use is similar to the one of the rule FRAME.

$$\frac{\begin{array}{c}(R \vee G_2) \wedge G_2'; G_1 * G_1'; I * I' \vdash \{p_1 * m * r\}\ C_1\ \{q_1 * m_1' * r_1'\} \\ (R \vee G_1) \wedge G_1'; G_2 * G_2'; I * I' \vdash \{p_2 * m * r\}\ C_2\ \{q_2 * m_2' * r_2'\} \\ I \rhd \{R, G_1, G_2\} \quad I' \rhd \{G_1', G_2'\} \quad r \vee r_1' \vee r_2' \Rightarrow I' \quad m \vee m_1' \vee m_2' \Rightarrow I'\end{array}}{R; G_1 \vee G_2; I \vdash \{p_1 * p_2 * m * r\}\ C_1 \parallel C_2\ \{q_1 * q_2 * (m_1' \wedge m_2') * (r_1 \wedge r_2)\}} \quad \text{(PAR-HIDE)}$$

This rule is a generalization of the rule PAR, obtained applying the rule PAR followed by the rule HIDE to hide the m resource. This rule is useful when a parent thread wants to create a shared resource for the child's but without sharing this resource with the full program.

Example

We present a very simple example program and verify it using LRG.

The program code is just

```
atomic(true) {
    x := x + 1
}
```

and it is run in a concurrent environment where more threads executing the same code might exists. The precondition and postcondition are:

$$p \stackrel{\text{def}}{=} x \Vdash x \geq Y \land \text{emp}_h$$
$$q \stackrel{\text{def}}{=} x \Vdash x > Y \land \text{emp}_h$$

So we begin the execution in a state where x is greater or equal than Y logical variable, and we end in a state where x can only be greater than Y. The specification is quite weak in order to be stable over the R condition. The R, G, and I we are going to use for the verification are:

$$R \stackrel{\text{def}}{=} (x \Vdash x \geq Y \land \text{emp}_h \ltimes x \Vdash x > Y \land \text{emp}_h) \lor \text{Id}$$
$$G \stackrel{\text{def}}{=} R$$
$$I \stackrel{\text{def}}{=} \text{Own}(x) \land \text{emp}_h$$

For this definitions,

- $I \rhd \{R, G\}$,
- $\text{Sta}(\{p, q\}, R * \text{Id})$,
- $p \ltimes q \Rightarrow G * \text{True}$, and
- $p \lor q \Rightarrow I * \text{true}$

hold. So in order to prove the correctness of the code using the atomic rule it only remains to prove the tautology $p \Rightarrow \text{true} = \text{true}$, and to prove the code sequentially.

To prove the code sequentially, we are going to use begin using the assignment axiom:

$$\frac{}{\{x \Vdash X = x + 1 \land \text{emp}_h\}\ \texttt{x := x + 1}\ \{x \Vdash x = X \land \text{emp}_h\}} \quad \text{(Assignment axiom)}$$

After that we are going to add independent state with the frame rule:

$$\frac{\{x \Vdash X = x + 1 \land \text{emp}_h\}\ \texttt{x := x + 1}\ \{x \Vdash x = X \land \text{emp}_h\}}{\begin{array}{c}\{(x \Vdash X = x + 1 \land \text{emp}_h) * (X > Y \land \text{emp}_s)\} \\ \texttt{x := x + 1} \\ \{(x \Vdash x = X \land \text{emp}_h) * (X > Y \land \text{emp}_s)\}\end{array}} \quad \text{(Frame rule)}$$

And apply the rule of consequence to arrive to the following triple:

$$\{x \Vdash X = x+1 \wedge \mathrm{emp}_h \wedge X > Y\}$$
$$\texttt{x := x + 1}$$
$$\{x \Vdash x = X \wedge \mathrm{emp}_h \wedge X > Y\}$$

We continue using the auxiliary variable elimination rule for X:

$$\frac{\begin{array}{c}\{x \Vdash X = x+1 \wedge \mathrm{emp}_h \wedge X > Y\}\\ \texttt{x := x + 1}\\ \{x \Vdash x = X \wedge \mathrm{emp}_h \wedge X > Y\}\end{array}}{\begin{array}{c}\{\exists X.x \Vdash X = x+1 \wedge \mathrm{emp}_h \wedge X > Y\}\\ \texttt{x := x + 1}\\ \{\exists X.x \Vdash x = X \wedge \mathrm{emp}_h \wedge X > Y\}\end{array}} \quad \text{(Auxiliary variable elimination)}$$

And finish using the consequence rule once again to get:

$$\{x \Vdash x \geq Y \wedge \mathrm{emp}_h\}\ \texttt{x := x + 1}\ \{x \Vdash x > Y \wedge \mathrm{emp}_h\}$$

or, which is the same:

$$\{p\}\ \texttt{x := x + 1}\ \{q\}$$

The full program can be proved using the rule ATOMIC:

$$\frac{\begin{array}{c}p \Rightarrow \mathrm{true} = \mathrm{true} \quad \{p \wedge \mathrm{true}\}\ C\ \{q\} \quad \mathrm{Sta}(\{p,q\}, R * \mathrm{Id})\\ p \ltimes q \Rightarrow G * \mathrm{True} \quad p \vee q \Rightarrow I * \mathrm{true} \quad I \rhd \{R, G\}\end{array}}{R;G;I \vdash \{p\}\ \texttt{atomic(true) \{x := x + 1;\}}\ \{q\}} \quad \text{(ATOMIC)}$$

4. CHAPTER

Implementing LRG in Dafny

4.1 Implementation

To implement LRG in Dafny we used the functional features of the language:

- algebraic data types,
- pure functions,
- predicates.

Dafny supports writing imperative code and proving it using integrated resources as lemmas, assertions, calculation... Nevertheless, the imperative language used in Dafny is not compatible with the one used in LRG as it does not support concurrent executions neither atomic operations. In addition, the memory model used in Dafny is based in frames instead of separation logic.

Since modifying Dafny in order to support the custom programming language used in LRG was out of the scope of this project, instead of that we implemented de language over Dafny using algebraic data types and functions.

For each element in the programing and the assertion language, we define the grammar using algebraic data types, and afterwards we give a meaning to each constructor using a function or a predicate.

4.1.1 Expressions

The integer expression grammar is defined using the following algebraic data type:

```
datatype Expr = Var(VName) | Int(int) | Add(Expr, Expr)
    | Substract(Expr, Expr)
```

`Var` and `Int` are the only non recursive constructors. The first one takes a `VName`, a variable name, as a parameter (`VName` is a synonim to `string` in our code). The second takes an integer as an argument. The rest of them are recursive constructors which take two expressions as an argument. `Add` is for addition, `Substract` for subtraction.

To give semantic meaning to the expressions we use the `ValExpr` function:

```
function ValExpr(expr: Expr, s: Store, i: LvMap): int
    requires DefinedExpr(expr, s, i)
{
    var si := union(s, i);
    match expr
    case Var(x) => si[x]
    case Int(n) => n
    case Add(e0, e1) => ValExpr(e0, s, i) + ValExpr(e1, s, i)
    case Substract(e0, e1) => ValExpr(e0, s, i) + ValExpr(e1, s, i)
}
```

This function takes the expression to be evaluated, the store and the logical variable map as a parameter. This function does in Dafny exactly the same as the $[\![E]\!]_{s,i}$ in the LRG logic, calculate the value of E expression over the union of s and i. Note that the function is not total, it has a precondition requiring all the free variables in `expr` are present in `s` or `i`.

Examples

Some expressions and their equivalent in Dafny:

- x: `Var("x")`
- 7: `Int(7)`
- $x+2$: `Add(Var("x"), Int(2))`

- $x-(2+y)$: `Substract(Var("x"), Add(Int(2), Var("y")))`

4.1.2 Binary expressions

The binary expressions are similar to the integer expressions. Their grammar is:

```
datatype BExpr = True | False
    | Not(BExpr) | And(BExpr, BExpr) | Or(BExpr, BExpr) |
    | Xor(BExpr, BExpr) | Equal(Expr, Expr)
    | Greater(Expr, Expr) | GreaterEqual(Expr, Expr)
    | Less(Expr, Expr) | LessEqual(Expr, Expr)
```

`True` and `False` are constants and because of that they do not take any parameter. `Nor`, `And`, `Or` and `Xor` represent the usual binary operators and take one or two binary expressions as parameters. `Equal`, `Greater`, `GreaterEqual`, `Less` and `LessEqual` on the other hand are relational operators which take integer expressions as an argument.

The semantics are given by `ValBExpr`:

```
function ValBExpr(bexpr: BExpr, s: Store, i: LvMap):bool
    requires DefinedBExpr(bexpr, s, i)
```

We omit the body as it is quite long an is not necessary to understand how `ValBExpr` works. The signature is equivalent to the one used by `ValExpr` but replacing the integer expression with a binary expression and the returned value is equivalent to the value of $[\![B]\!]_{s,i}$ evaluation in LRG.

Examples

Some binary expressions and their equivalent in Dafny:

- true: `True`
- true $\vee$ false: `Or(True, False)`
- $5 >= n$: `GreaterEqual(Int(5), Var("n"))`
- $y < 2 \wedge x = 2$:
  `And(Less(Var("y"), Int(2)), Equal(Var("x"), Int(2))`

4.1.3 Programming language

The programming language used in LRG logic is quite simple grammatically, as only one level of statements exist in addition to the expressions.

It follows the grammar:

```
datatype Stmt = Assign(dst: VName, val: Expr)
    | Load(dst: VName, addr: Expr)
    | Store(addr: Expr, val: Expr)
    | Skip
    | Cons(dst: VName, vals: seq<Expr>)
    | Dispose(addr: Expr)
    | Seq(c1: Stmt, c2: Stmt)
    | If(cond: BExpr, c1: Stmt, c2: Stmt)
    | While(cond: BExpr, c: Stmt)
    | Atomic(cond: BExpr, c: Stmt)
    | Par(c1: Stmt, c2: Stmt)
```

Each constructor is equivalent to one element of the LRG logic's programming language's grammar:

- `Assign` is equivalent to $x := E$
- `Load` is equivalent to $x := [E]$
- `Store` is equivalent to $[E] := E$
- `Skip` is equivalent to **skip**
- `Cons` is equivalent to $x := \textbf{cons}(E, \ldots, E)$
- `Dispose` is equivalent to $\textbf{dispose}(E)$
- `Seq` is equivalent to $C_1 \,;\, C_2$
- `If` is equivalent to $\textbf{if}\ (B)\ \{C\}\ \textbf{else}\ \{C\}$
- `While` is equivalent to $\textbf{while}\ (B)\ \{C\}$
- `Atomic` is equivalent to $\textbf{atomic}\ (B)\ \{C\}$
- `Par` is equivalent to $C_1 \,||\, C_2$

To give semantics to the programming language in Dafny, we can create a function which simulates the execution of a statement. The signature will be something in the line of:

```
inductive predicate StmtStep (stmt: Stmt, m0: Mem, m1: Mem)
```

where `Mem` is equivalent to the type of (s,i,h), `m0` represents the initial state and `m1` the final state. This predicate returns true if the statement `stmt` executed in the state `m0` yields the state `m1`. Since a while might cycle causing the predicate not to terminate and the final state never to be reached the termination of the predicate can not be proved. Consequently we make the predicate inductive, as Dafny does not require inductive predicates to terminate.

Nevertheless, implementing the semantics of the programming language is not necessary to use the inference rules proposed in LRG. Implementing the semantics is only necessary to prove the correctness and completeness of the inference rules, and for that reason, our implementation does not include the `StmtStep` predicate.

Examples

Some programs and their equivalence in Dafny:

LRG:

```
y := 5;
x := [y];
```

Dafny:

```
Seq(Assign("y", Int(5)), Load("x", Var("y")))
```

LRG:

```
y := 0;
while (y < 5) {
    y := y + 1;
}
```

Dafny:

```
Seq(
    Assign("y", Int(0)),
    While(Bool(Less(Var("y"), Int(5))),
        Assign("y", Add(Var("y"), Int(1)))
    )
)
```

4.1.4 Assertion language

As in the assertion language we have both standard assertions and actions, we need two separate data types to represent them.

Assertions

The assertion grammar is:

```
datatype Assertion = Bool(BExpr) | emp_h | emp_s | Own(VName)
    | Points(Expr, Expr) | Reserved(Expr)
    | Indep(Assertion, Assertion) | Wand(Assertion, Assertion)
    | And(Assertion, Assertion) | Or(Assertion, Assertion)
    | Neg(Assertion) | Exists(VName, Assertion)
    | Impl(Assertion, Assertion)
```

and the semantics are given by a predicate with the signature:

```
predicate SemanticAssertion(assrt: Assertion, mem: Mem)
```

This predicate is true if and only if $\sigma \models p$ holds where p is `assrt` and σ is `mem`. The equivalences between Dafny and LRG are:

- `Bool` is equivalent to B
- `emp_h` is equivalent to emp_h
- `emp_s` is equivalent to emp_s
- `Points` is equivalent to $E \mapsto E$

- `Reserved` is equivalent to $E \mapsto -$
- `Indep` is equivalent to $p * q$
- `Wand` is equivalent to $p \mathrel{-\!\!*} q$
- `And` is equivalent to $p \wedge q$
- `Or` is equivalent to $p \vee q$
- `Neg` is equivalent to $\neg p$
- `Exists` is equivalent to $\exists X.p$
- `Impl` is equivalent to $p \Rightarrow q$

Technically `Reserved` is only syntactic sugar in LRG, but is implemented inside the assertions in order to be easier to prove.

In addition to the data type definition there are some useful functions to help create complex assertions in a easier way:

```
function OwnFromSeq(sn: seq<VName>): Assertion
function AndFromSeq(sa: seq<Assertion>): Assertion
function OrFromSeq(sa: seq<Assertion>): Assertion
function IndepFromSeq(sa: seq<Assertion>): Assertion
function MultiplePoints(e: Expr, se: seq<Expr>): Assertion
```

The first one is equivalent to $x_1, \ldots, x_n, \bullet \Vdash \text{true}$. It takes a sequence of variables as argument, and it returns the equivalent assertion formed combining `Own` and `And`. The second one simply joins a bunch of assertions with a conjunction and the third one with a disjunction. The fourth one joins a sequence of assertions using the $*$ independence operator. Finally, the fifth one is equivalent to $E \mapsto E_1, \ldots, E_n$. The `e` parameter is the address E and `se` is the sequence of expressions $(E_1, \ldots, E_n)$.

Thanks to this assertions things like:

```
And(Bool(Equal(Var("a"), Int(5))),
    And(q,
        And(r, s)
    )
)
```

are transformed to

```
AndFromSeq([
    Bool(Equal(Var("a"), Int(5))),
    q,
    r,
    s
])
```

which are much easier to read and debug.

The constant emp is also implemented in Dafny, using a function without parameters:

```
function emp(): Assertion {
    Assertion.And(emp_s, emp_h)
}
```

Actions

The action grammar is:

```
datatype Action = Change(Assertion, Assertion)
    | Mantain(Assertion) | Indep(Action, Action)
    | Exists(VName, Action) | Impl(Action, Action)
    | And(Action, Action) | Or(Action, Action)
```

and the semantics are given by a predicate with the signature:

```
predicate SemanticAction(assrt: Action, m: Mem, m': Mem)
```

This predicate is true if and only if $\sigma, \sigma' \models p$ holds where p is `assert`, σ is `mem` and σ' is `mem'`. The equivalences between Dafny and LRG are:

- `Change` is equivalent to $p \ltimes q$
- `Maintain` is equivalent to $[p]$
- `Indep` is equivalent to $a_1 * a_2$

- `Exists` is equivalent to $\exists X.a$
- `Impl` is equivalent to $a_1 \Rightarrow a_2$
- `And` is equivalent to $a_1 \wedge a_2$
- `Or` is equivalent to $a_1 \vee a_2$

The actions Emp, True and Id are defined using functions without parameters is a similar way to emp assertion.

```
function Id(): Action {
    Mantain(Bool(True))
}
function TrueAct(): Action {
    Change(Bool(True), Bool(True))
}
function Emp(): Action {
    Change(emp(), emp())
}
```

Examples

We present some assertions and actions, and their equivalence in Dafny.

- $x > y$: `Bool(Greater(Var("x"), Var("y")))`
- $x \mapsto y, z$: `MultiplePoints("x", [Var("y"), Var("z")])`
- true $\Rightarrow 1 < 2$: `Impl(Bool(True), Less(Int(1), Int(2)))`
- true $\ltimes a$: `Change(Bool(True), a)`
- $[x \mapsto -]$: `Maintain(Reserved(x))`
- Id $\vee$ Emp: `Action.Or(Id(), Emp())`

4.1.5 Inference rules

Inference rules can be implemented in several different ways. One approach could be to create a lemma for each inference rule, where the conditions of the rule are encoded in the requirements and the consequence in the ensure clause.

An alternative approach is to create a predicate to check if a program is correct. The predicate is encoded in a way such that it is true if and only if it exists an inference rule chain beginning in the judgement for well formed sequential programs ($R;G;I \vdash \{p\}\ C\ \{q\}$) to check and ending in rules without any conditions.

In this implementation, we used the second approach.

Since in LRG logic there are two levels of inference rules, the sequential rules and the concurrent rules, each one using a different specification, two predicates are necessary.

We first explain the implementation of the sequential rules, as the concurrent rules depend on them.

Sequential rules

The sequential rules are implemented in `SequentiallyDerivable`:

```
inductive predicate SequentiallyDerivable(
        p: Assertion,
        C: Stmt,
        q: Assertion)
```

The parameters of the rule `p`, `C` and `q` correspond to the precondition, code and postcondition of $\{p\}\ C\ \{q\}$ program specification.

To check if it is derivable, the conditions of all the rules are joined with a disjunction. That way, if some of the rules is applicable the predicate will return true.

The conditions of the rules might require to prove other specifications of the form $\{p'\}\ C'\ \{q'\}$, which is implemented in Dafny using recursive calls. The predicate is inductive because the chain of rules needed to prove a program might be infinite and cycles can be created in the chain, causing the predicate not to terminate.

For some rules, the consequence of the rule is not of the form $\{p\}\ C\ \{q\}$. For those cases, additional conditions are needed to ensure that the consequence of the rule is equivalent to the received parameters. For example if the rule was:

$$\frac{p \Rightarrow q}{\{p\}\ \textbf{skip}\ \{q\}}$$

in addition to $p \Rightarrow q$ condition we should check also that $C = \textbf{skip}$.

In many rules, there are implications between assertions. To prove them, it is necessary to check that for all the states the implication is true: if the condition is $p \Rightarrow q$ we need to prove that $\forall \sigma. \sigma \models p \Rightarrow q$. As writing it in Dafny is quite tedious:

```
forall m :: SemanticAssertion(p, m) ==> SemanticAssertion(q, m)
```

specially when p and q are quite complex, the check is done inside the `AssertionImpl` function:

```
predicate AssertionImpl(p: Assertion, q: Assertion) {
    forall m :: SemanticAssertion(Assertion.Impl(p, q), m)
}
```

Concurrent rules

The sequential rules are implemented in `ConcurrentlyDerivable`:

```
inductive predicate ConcurrentlyDerivable(
        R: Action,
        G: Action,
        I: Assertion,
        p: Assertion,
        C: Stmt,
        q: Assertion)
```

The parameters of the rule `R`, `G`, `I`, `p`, `C` and `q` correspond to the rely conditions, guarantee condition, invariant, precondition, code and postcondition of $R; G; I \vdash \{p\}\ C\ \{q\}$ program specification.

The predicate is constructed in the same way as the one for sequential rules, using the disjunction to join all the conditions of the rules. The bigger difference is that in addition to implications between assertions, implications between actions, stability checks and fences might appear. To check each of the new conditions, there are three new predicate:

```
predicate ActionImpl(a1: Action, a2: Action)
predicate Sta(p: Assertion, a: Action)
predicate Fence(I: Assertion, a: Action)
```

4.2 Project structure

The implementation of the system in separated in four different files:

1. `util.dfy`
2. `programmin_language.dfy`
3. `assertions.dfy`
4. `inference_rules.dfy`

Each file includes a module with the same name of the file without the `dfy` extension. In order to use the resources of other modules, they must be imported at the beginning of the module.

util.dfy
In this file they are defined general usage functions like `domain` to get the domain of a `map` or `union` to join two maps.

programming_language.dfy
In this file both the expressions and the programming language are implemented. It contains also helper functions for them as `FvExpr` to get the free variables from a expression or `DefinedExpr` to check if it is well defined. It depends on `util` for some operations.

assertions.dfy
In this file both the assertions and actions the programming language are implemented. The contains also helper functions as `FvAction` to get the free variables in an action or `ActionImpl` to check if an action implies another. It depends on `util` for some operations and on `programming_language` for the expressions.

inference_rules.dfy
In this file the sequential and concurrent inference rules are implemented. `Sta` and `Fence` and some lemmas about them are also implemented here (the lemmas were taken directly from [9] and because of that they are not proved). Id depends on all the previous files.

4.3 Proving methodology

In order to prove a program using the Dafny implementation, a new `dfy` file should be created. First of all, in the new file we must import all the files from the implementation.

Which the files imported, the program to be proved and the assertions to be used should be translated to Dafny. Both the program and the assertions must be associated to some names. The preferable way would be to use constants but they do not allow to call functions inside them. As a consequence, the alternative approach is to use functions without parameters, which return always the same value. For example, we could implement a program and name it `simple` with the following code:

```
function simple(): Stmt {
    Assign("x", Int(5))
}
```

The only drawback of using functions is we have to remember to use the parentheses when referencing them.

Once the program and assertions are implemented, the must create a lemma with an empty requires clause. The ensures clause should be a call to `ConcurrentlyDerivable` with the program and its specification as parameters. If the program is not concurrent the call must be to `SequentiallyDerivable`.

After that it only remains to prove the lemma is correct. For any program containing more than one statement, using additional lemmas will be necessary to prove the intermediate steps are correct.

Example

We present the proof of a sequential skip statement as an illustration of the methodology (the imports are omitted):

```
function skip(): Stmt {
    Skip
}
function p(): Assertion {
    Bool(True)
```

```
lemma prove_skip()
    ensures SequentiallyDerivable(p(), skip(), p())
{
}
```

Limitations

One of the limitations of the methodology is the inability to accept arbitrary functions to be used for the specification purpose. For example, if we want to reason a value corresponds with the factorial of other number, with the current assertion language it is not possible to do so.

We tried to add the opportunity to call to arbitrary functions in the expressions and assertions, adding a new constructor Function to both of them. Unfortunately, due to limitations in how function variables are specified in Dafny the solution did not work.

The workaround for this is to add manually the required functions inside the expression and assertion definitions, creating a new constructor and specifying the semantics.

5. CHAPTER

Examples

In this section we present two different examples where we use our implementation to prove the correctness of a program.

The first one is a formalization in our tool of the example presented in the Variables as Resource section (subsection 2.3.3), and is a sequential program. The second one is a formalization of the example presented in LRG section (subsection 4.1.4). This one is a sequential program.

The full code of both proofs is included with the LRG implementation directory (see appendix A for more information).

5.1 Example of a sequential program

In this example we formalize and proof the following code:

$\{x, y \Vdash X = 7 \land Y = 5\}$
```
x := 7;
y := 5;
```
$\{x, y \Vdash x = X \land y = Y\}$

To do so, we will use the following intermediate assertions:

$p_0 = \{x, y \Vdash X = 7 \land Y = 5\}$
$\{(x \Vdash X = 7 \land \mathrm{emp}_h) * (y \Vdash Y = 5)\}$
$p_{\texttt{x assigment}} = \{x \Vdash X = 7 \land \mathrm{emp}_h\}$
```
x := 7;
```
$q_{\texttt{x assigment}} = \{x \Vdash x = X \land \mathrm{emp}_h\}$
$\{(x \Vdash x = X \land \mathrm{emp}_h) * (y \Vdash Y = 5)\}$
$p_1 = \{x, y \Vdash x = X \land Y = 5\}$
$\{(y \Vdash Y = 5 \land \mathrm{emp}_h) * (x \Vdash x = X)\}$
$p_{\texttt{y assigment}} = \{y \Vdash Y = 7 \land \mathrm{emp}_h\}$
```
y := 5;
```
$q_{\texttt{y assigment}} = \{y \Vdash y = Y \land \mathrm{emp}_h\}$
$\{(y \Vdash y = Y \land \mathrm{emp}_h) * (x \Vdash x = X)\}$
$p_2 = \{x, y \Vdash x = X \land y = Y\}$

First of all, we have defined most assertions and both statements in Dafny using constant function (we omit the bodies):

```
function p0(): Assertion
function p1(): Assertion
function p2(): Assertion
function p_x_assigment_p(): Assertion
function q_x_assigment(): Assertion
function p_y_assigment(): Assertion
function q_y_assigment(): Assertion
function x_mem(): Assertion
function y_mem(): Assertion
function x_command(): Stmt
function y_command(): Stmt
```

The assertions without name from the previous code segment are formed combining `x_mem` and `y_mem` with other assertions.

Once the necessary constants are defined, we can proceed with the proofs. We begin proving both assignments independently using the assignment axiom. To do so in Dafny, we formulate two lemmas, which are automatically proved:

```
lemma x_assignment_axiom()
    ensures SequentiallyDerivable(
        p_x_assigment(),
        x_command(),
        q_x_assigment()
    )
{
}

lemma y_assignment_axiom()
```

```
    ensures SequentiallyDerivable(
        p_y_assigment(),
        y_command(),
        q_y_assigment()
    )
{
}
```

With the assignments proved, we can use the frame rule to add the memory corresponding to y to the assertions of the first statement and the memory corresponding to x to the assertions of the second statement:

```
lemma x_frame_rule()
    ensures SequentiallyDerivable(
        Assertion.Indep(p_x_assigment(), y_mem()),
        x_command(),
        Assertion.Indep(q_x_assigment(), y_mem())
    )
{
    x_assignment_axiom();
}

lemma y_frame_rule()
    ensures SequentiallyDerivable(
        Assertion.Indep(p_y_assigment(), x_mem()),
        y_command(),
        Assertion.Indep(q_y_assigment(), x_mem())
    )
{
    y_assignment_axiom();
}
```

It must be noted that calling the previously proved lemmas inside the method body is necessary, since their `ensures` clauses correspond with the conditions of the frame rules.

After that, we continue applying the consequence rule to adapt the form of the assertions, and be able to apply the rule of composition. The consequence rule has the additional conditions $p \Rightarrow p'$ and $q' \Rightarrow q$. This consequences must be proved, but in order to simplify the example, we decided to formulate the additional conditions in the auxiliary lemmas `helper_x_implication` and `helper_y_implication` and call this lemmas in order to prove the consequence rule is correctly applied:

```
lemma x_consequence_rule()
    ensures SequentiallyDerivable(p0(), x_command(), p1())
```

```
{
    x_frame_rule();
    helper_x_implication();
}
lemma y_consequence_rule()
    ensures SequentiallyDerivable(p1(), y_command(), p2())
{
    y_frame_rule();
    helper_y_implication();
}
```

Finally, we apply the rule of composition, to join both statements and verify the full program.

```
lemma rule_of_composition()
    ensures SequentiallyDerivable(
        p0(),
        Seq(x_command(), y_command()),
        p2()
    )
{
    x_consequence_rule();
    y_consequence_rule();
}
```

In order the verification to be complete, the auxiliary lemmas must also be proved, but due to the time constraints and some difficulties with Dafny, we only have checked them manually.

5.2 Example of a concurrent program

In this example we formalize and proof of the following code:

```
{x ⊩ x ≥ Y ∧ emp_h}
atomic (true) {
    x := x + 1;
}
{x ⊩ x > Y ∧ emp_h}
```

To do so we will first prove the sequential code using the following intermediate assertions:

- $p \stackrel{\text{def}}{=} x \Vdash x \geq Y \land \text{emp}_h$
- $p_1 \stackrel{\text{def}}{=} \exists X (x \Vdash X = x + 1 \land \text{emp}_h \land X > Y)$
- $p_2 \stackrel{\text{def}}{=} x \Vdash X = x + 1 \land \text{emp}_h \land X > Y$
- $p_3 (\stackrel{\text{def}}{=} x \Vdash X = x + 1 \land \text{emp}_h) * (X > Y \land \text{emp}_s)$
- $p_4 \stackrel{\text{def}}{=} x \Vdash X = x + 1 \land \text{emp}_h$
- $q_4 \stackrel{\text{def}}{=} x \Vdash X = x + 1 \land \text{emp}_h$
- $q_3 (\stackrel{\text{def}}{=} x \Vdash x = X \land \text{emp}_h) * (X > Y \land \text{emp}_s)$
- $q_2 \stackrel{\text{def}}{=} x \Vdash x = X \land \text{emp}_h \land X > Y$
- $q_1 \stackrel{\text{def}}{=} \exists X (x \Vdash x = X \land \text{emp}_h \land X > Y)$
- $q \stackrel{\text{def}}{=} x \Vdash x > Y \land \text{emp}_h$

As in the previous example, all the assertions and the `x := x+1` statement are implemented using constant functions:

```
    function p(): Assertion
    function p1(): Assertion
    function p2(): Assertion
    function p3(): Assertion
    function p4(): Assertion
    function q4(): Assertion
    function q3(): Assertion
    function q2(): Assertion
    function q1(): Assertion
    function q(): Assertion
    function C(): Stmt
```

We begin the proof using the assignment axiom:

```
lemma assignment_axiom()
    ensures SequentiallyDerivable(p4(), C(), q4())
{
    assert OwnFromSeq(["x"]) == Own("x");
}
```

An assertion is included in the body to help Dafny with the proof.

Afterwards, we use the frame rule to add $X > Y \wedge \mathrm{emp}_s$ independent assertion to the precondition and postcondition:

```
lemma frame_rule()
    ensures SequentiallyDerivable(p3(), C(), q3())
{
    assignment_axiom();
}
```

and we follow with the rule of consequence (where `helper_CR_1` is an auxiliary lemma to proof the implications):

```
lemma helper_CR_1()
    ensures AssertionImpl(p2(), p3())
    ensures AssertionImpl(q3(), q2())

lemma consequence_rule_1()
    ensures SequentiallyDerivable(p2(), C(), q2())
{
    frame_rule();
    helper_CR_1();
}
```

The next step is to apply the auxiliary variable elimination rule to quantify X existentially:

```
lemma auxiliary_variable_elimination()
    ensures SequentiallyDerivable(p1(), C(), q1())
{
    consequence_rule_1();
}
```

and we finish the sequential proof with another application of the rule of consequence (where `help_CR_2` is an auxiliary lemma to proof the implications):

```
lemma helper_CR_2()
    ensures AssertionImpl(p(), p1())
    ensures AssertionImpl(q1(), q())

lemma consequence_rule_2()
    ensures SequentiallyDerivable(p(), C(), q())
{
```

```
    auxiliary_variable_elimination();
    helper_CR_2();
}
```

Having the sequential proof completed, it only remains one application of the rule ATOMIC. In order to do so we implement R, G and I and the statement for the full code in Dafny:

```
function R(): Action {
    Action.Or(Change(p(), q()), Id())
}

function G(): Action {
    R()
}

function I(): Assertion {
    Assertion.And(Own("x"), emp_h)
}

function C'(): Stmt {
    Atomic(True, C())
}
```

We also write the headers of many auxiliary lemmas:

```
lemma helper_condition()
    ensures forall m :: SemanticAssertion(p(), m) ==> DefinedBExpr(True, m.0,
    m.1)

lemma helper_Sta()
    ensures Sta(p(), Action.Indep(R(), Id()))
    ensures Sta(q(), Action.Indep(R(), Id()))

lemma helper_AndTrue()
    requires SequentiallyDerivable(p(), C(), q())
    ensures SequentiallyDerivable(Assertion.And(p(), Bool(True)), C(), q());

lemma helper_guarantee()
    ensures ActionImpl(Change(p(), q()), Action.Indep(G(), TrueAct()))

lemma helper_invariant()
    ensures AssertionImpl(Assertion.Or(p(), q()), Assertion.Indep(I(), Bool(
    True)))

lemma helper_fence()
    ensures Fence(I(), R())
    ensures Fence(I(), G())
```

- `helper_condition` is used to proof $p \Rightarrow \text{true} = \text{true}$
- `helper_Sta` is used to proof $\text{Sta}(\{p,q\}, R * \text{Id})$
- `helper_AndTrue` is used to proof $p \wedge true$ is equivalent to p when proving $\{p\}\ C\ \{q\}$
- `helper_guarantee` is used to proof $p \ltimes q \Rightarrow G * \text{True}$
- `helper_fence` is used to proof $I \triangleright \{R, G\}$

We finish the verification using the atomic rule to proof the full program:

```
lemma ATOM_rule()
    ensures ConcurrentlyDerivable(R(), G(), I(), p(), C'(), q())
{
    helper_condition();
    helper_guarantee();
    helper_invariant();
    helper_fence();
    helper_Sta();
    consequence_rule_2();
    helper_AndTrue();
}
```

As for the previous example, most of the auxiliary lemmas remain to be proved in order the verification to be complete.

6. CHAPTER

Conclusions and future work

In this chapter we present the conclusions of the project and we also propose some ideas for future works.

6.1 Conclusions

On the one hand, the learning process to understand LRG logic was more difficult than expected. The first publications about Hoare logic date from about 50 years ago, and many development has been done in tools and didactic methodologies to teach it. On the contrary, Separation logic, Variables as Resource and Rely-Guarantee reasoning are much newer, and the documentation and tools about them are more limited.

For Separation logic, the problem was not very significant, as in the last 15 years, some academics have begun to teach courses about it creating a reasonable amount of didactic material. Furthermore, an important verification tool, KeY, has integrated Separation logic inside his proving system, extending its use to more public.

For Rely-Guarantee logic, on the other hand, the literature is much more limited. Almost all the publications regarding it are scientific papers, which are addressed to people already experienced in the area. The tools using RG are also almost exclusively research tools with limited real world usage. As a consequence, understanding Rely-Guarantee reasoning was a real challenge, and has required quite more implication than expected at the beginning on the project.

Variables as Resource was also documented exclusively in research papers, but due to the lower complexity of the logic system, and the similarities with separation logic was much easier to understand.

Most of this methodologies (expect Rely-Guarantee reasoning, which was directly replaced by the improved LRG) are explained in this report, and that is why we think this document can be useful for other students wanting to begin working or researching about this technologies.

On the other hand, we discovered some limitations of Dafny implementing and, specially, using the LRG methodology.

First off all, Dafny is a very young tool. The project is in active development, and many changes happen from version to version. However, the documentation of the project is limited. In addition to a pair of tutorials and some research papers, the only available documentation is the Dafny reference manual, which is incomplete, and lacks important information about advanced features of the language can be used. In addition, since the Dafny language users base is so reduced, most of the typical question and answer sites as StackOverflow[1] contain very few solutions for problems. The lack of documentation has severally hampered the implementation of the LRG logic and, in some cases, we perhaps use inelegant workarounds to fix problems with the language.

On the other hand, there is a list of features they are available in other tools, they would have been quite helpful to our project. For example, Dafny does not support operator overloading, which could have been used to ease the writing of complex expressions, assertions and programs. Even better than the overloading, would be that Dafny supports something similar to the `Notation` declaration of Coq proof assistant, which can be used in a similar way but is more flexible. We consider this feature quite important to do semantics, since with the actual notation is very easy to introduce typos in the programs and quite difficult to detect them. Other interesting feature, available also in Coq, is the `simpl` command. In Coq, this command replaces a function with itch body after making the parameter substitution. While trying to prove lemmas in Dafny, we discovered that many steps inside a calculation where this kind of substitutions, where we ended copying and pasting the same come many times and doing the variable substitutions manually. Having a `simpl` like feature in Dafny would reduce the errors and the effort for this proves greatly. In addition, as the transformation is done by the machine, it will be no necessity to check the step in the Z3 SMT solver integrated in Dafny.

[1] https://stackoverflow.com/

In conclusion, at the time of writing this report, we think that perhaps the proof assistants continue to be better suited for this works, even if they require to do more manual work. We are satisfied with the investigation of LRG logic, in which we learned a lot and we produced a report useful for other people to learn. The formalization of the logic in Dafny was also completed successfully. However, we are a bit disappointed of the implementations phase because the created tool is not easy to use in practice.

6.2 Future work

One of the obvious propositions for future work is to improve the proving methodology of our tool, to enable easier proofs for lemmas.

Another proposal is to program a simple parser to translate programs and assertions written in the format proposed in LRG to Dafny variable declarations. This implementation should be quite straightforward since the grammars are quite simple comparing to the ones used in production programming languages and we only need to create data types instead of low level machine instructions. It will be also interesting to allow the parser both to generate real programs and the Dafny program declarations. Like that, after verifying the code in Dafny it will be possible to generate a verified executable (assuming the translator is correct).

An important thing to mention is that LRG does not use the weakest precondition approach used to verify programs using Hoare logic. The weakest preconditions eliminates the burden of manually searching intermediate assertions between each two instructions, and generates less verification conditions to be checked by the SMT solvers or to be proved manually. An interesting future work will be to find an equivalent method for LRG logic, and implement creating a much powerful tool.

Other limitation of LRG is it lacks the capability of proving termination. It is important to note that in concurrent environments termination depends on properties like deadlock freedom, starvation freedom, wait freedom... mentioned in most books of concurrent programming. As a consequence proving concurrent program termination is not so easy as finding a decreasing clause for every while and recursive function call, a new methodology is necessary. A methodology for this already exists, LiLi [15], but there is no computer based tool implementing it. While implementing LRG we also implemented a first version of LiLi in Dafny, but was neither tested nor fully completed. An interesting future

work will be to finish the LiLi implementations and to use it in conjunction with LRG to prove concurrent programs.

Appendices

A. APPENDIX

Project organization

The code of the project is organized using the following directory structure:

```
.
|-- LiLi
|   |-- assertions.dfy
|   |-- decreasing_functions.dfy
|   |-- inference_rules.dfy
|   |-- programming_language.dfy
|   |-- test_programming_language.dfy
|   |-- ticket_lock.dfy
|   `-- util.dfy
`-- LRG
    |-- assertions.dfy
    |-- concurrent_example_proof.dfy
    |-- concurrent_example_spec.dfy
    |-- inference_rules.dfy
    |-- programming_language.dfy
    |-- sequential_example.dfy
    `-- util.dfy
```

In the `LiLi` directory they are the files used to implement the LiLi logic mentioned in the conclusion. The implementation is neither completed nor tested, and the code is structured

in a similar way to LRG logic's implementation.

In the `LRG` directory they are both the files to implement LRG logic, and the source files for the examples explained in the fifth chapter. The content of `assertions.dfy`, `inference_rules.dfy`, `programming_language.dfy` and `utils.dfy` is explained in the 4.2 section.

The remaining files are:

- `sequential_example.dfy`: this file contains the sequential example presented in section 5.1.
- `concurrent_example_spec.dfy`: this file contains the definitions used in the concurrent example of section 5.2.
- `concurrent_example_spec.dfy`: this file contains the lemma used in the concurrent example of section 5.2.

The concurrent example is separated in two files in order to allow faster compilation speeds.

Bibliography

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The key tool. *Software & Systems Modeling*, 4(1):32–54, 2005.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111, pages 364–387. Springer, September 2006.

[3] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[4] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in separation logic. *Electron. Notes Theor. Comput. Sci.*, 155:247–276, May 2006.

[5] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[7] ECMA International. *Standard ECMA-334 - C# Language Specification*. 4 edition, June 2006.

[8] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. Geneva, Switzerland, 5 edition, December 2010.

[9] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 315–327, New York, NY, USA, 2009. ACM.

[10] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, GA, 2016. USENIX Association.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[12] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983.

[13] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[14] Rustan Leino. Well-founded functions and extreme predicates in dafny: A tutorial. In Boris Konev, Stephan Schulz, and Laurent Simon, editors, *IWIL-2015. 11th International Workshop on the Implementation of Logics*, volume 40 of *EPiC Series in Computing*, pages 52–66. EasyChair, 2016.

[15] Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 385–399, New York, NY, USA, 2016. ACM.

[16] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[17] Peter W. O'Hearn. A primer on separation logic (and automatic program verification and analysis). In Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, editors,

Software Safety and Security - Tools for Analysis and Verification, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 286–318. IOS Press, 2012.

[18] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[19] Viktor Vafeiadis and Matthew Parkinson. *A Marriage of Rely/Guarantee and Separation Logic*, pages 256–271. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.