Diogo Santiago Serra

Licenciado em Engenharia Informática

A proof system for lock-free concurrency

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : António Ravara, Professor Auxiliar, FCT/UNL

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Fevereiro, 2013

A proof system for lock-free concurrency

To my grandmother Dária, whose kindness I left unmatched.

Acknowledgements

Firstly I am grateful to my supervisor, António Ravara, for proposing this dissertation's topic as it perfectly fits one of my main areas of interest – formally reasoning about concurrent programs. I am equally grateful to António for his availability and unwavering support even when this dissertation was not going so well.

Secondly I am grateful to Luís Caires, who at a critical stage delimited the scope of this work and contributed with invaluable insights which allowed me to conclude it. I am also thankful for his motivation along this period, without which I probably would have quit.

At OutSystems, where I currently work, I am grateful to António Melo and David Nunes for being patient about my starting date and allowing me to take some days off so I could properly finish this document.

I am also thankful to my colleagues Paulo Ferreira, with whom I continuously discussed some of this thesis' details, and Filipe Rodrigues, who put up with my ramblings about its execution.

Finally I am grateful to my family for their support at this time when I am simultaneously working and finishing my graduation, which I am currently unable to reciprocate.

Abstract

Software has become widespread, being used and relied upon on nearly every domain. Furthermore, as this globalization of software took place and multi-core architectures became the norm, several programs are now expected to run on a given device at the same time in a timely fashion. Attending this need, concurrent and distributed systems are a well known way of dealing with performance and scalability of computation. Although several such systems exist in the devices and services we depend on, it is frequent for those systems to be exploited or go wrong. Because most complex programs are built in modules and lack a formal specification of what they do, it is hard to prevent the emerging system from failing or being exploited. Therefore, one of the most sought after results by software industry is a way of reasoning about programs that prevents undesired behavior. Formal methods contribute to a rigorous specification, analysis, and verification of programs, having proven to be quite effective in this regard. Program logics, in particular, are able to verify validity of user-specified formulas and are the solution we propose to tackle this issue. Regarding concurrent programs, locks are a mechanism that make reasoning easier by serializing access to shared resources, taming concurrency. Since lock-free programs offer a better way of taking advantage of concurrency, we are especially interested in them. In this context, the `LL/SC` pair of primitives have proven to be more expressive than their widely used `CAS` counterpart. The goal of our work is then to develop a proof system capable of proving correctness of lock-free programs based on `LL/SC` primitives. In this dissertation we present a new program logic, based on those of concurrent separation logic and `RGSep`, which establishes a solid theoretical basis for reasoning about such programs.

Keywords: program logics, lock-free programming, load-link/store-conditional

Resumo

A disseminação do software fez com que este se tornasse dependentemente utilizado em vários domínios aplicacionais. Dada a sua globalização e o estabelecimento das arquiteturas *multi-core*, tornou-se expetável que vários programas executem ao mesmo tempo num dado dispositivo. Visando dar resposta a esta necessidade, os programas concorrentes e distribuídos fornecem uma maneira de lidar com o desempenho e escalabilidade de tais sistemas. Embora estes programas existam nos dispositivos e serviços de que dependemos, é frequente que estes falhem ou sejam manipulados. Assim sendo, uma das maiores necessidades na informática é uma maneira de garantir a correta execução destes programas. Métodos formais contribuem para uma rigorosa especificação, análise e verificação de programas, tendo-se provado bastante eficazes nesta matéria. Em particular, lógicas para verificação de programas são capazes de verificar a validade de fórmulas especificadas pelo utilizador, constituindo a solução que propomos para este problema. Respeitante a programas concorrentes, as *locks* são um mecanismo que facilita o raciocínio ao serializar o acesso a recursos partilhados, controlando a concorrência. Uma vez que programas não bloqueantes tiram melhor partido da concorrência, estes são o alvo da nossa atenção. Neste contexto, as primitivas `LL/SC` provaram ser uma alternativa mais expressiva que a mais frequentemente utilizada `CAS`. O objetivo do nosso trabalho é então desenvolver um sistema de prova capaz de provar a correção de programas não bloqueantes baseados nas primitivas `LL/SC`. Nesta dissertação é apresentada uma nova lógica, baseada naquelas presentes na lógica de separação concorrente e `RGSep`, que estabelece uma base teórica sólida para raciocinar sobre tais programas.

Palavras-chave: lógicas para verificação, programação não bloqueante, *load-link/store-conditional*

Contents

List of Figures

Listings

1 Introduction

1.1 Context and motivation

Software has become widespread, being used and relied upon on nearly every domain. With current research and development augmenting the amount of problems it can solve, it is bound to become present in every aspect of our lives. Whether due to performance or organizational requirements, modern software systems are distributed and communication-centered. Many such systems depend on third-party services, such as Clouds, making them very modular and dynamic in nature. Devices that provide these services are themselves highly distributed internally due to the concurrent computation present in multi-core hardware, which is also a well established technology to improve performance.

In this environment, an important problem that affects both local and distributed systems is undesired behavior. Dynamic and modular components of a system must not compromise the behavior and reliability of a device or service. However, software is not a reliable artifact by itself as it provides no warranty on its functionality. It is meant to be executed, and often provides no insight as to what it will do – leading to a huge resource investment in understanding software systems and fixing their bugs, as well as compromising current systems with bugs and exploits which make them vulnerable to malicious attacks. Ultimately these problems result in protocol incompatibility, wasted resources and security breaches, which often lead to service unavailability, information leak and malicious exploitation.

Preventing the above problems is a challenging task that requires a solid methodological and infrastructural basis for developing distributed and concurrent software, which is beginning to be built but is still clearly missing from current core information technologies. Due to the complexity of such systems, abstractions that are able to precisely model them play a key role in languages and structures used in their development since assurances regarding a software system's behavior must be provided before it is deployed in a potentially dangerous environment. Assurances themselves must also be precisely specified, so that they can be validated by automatic or semi-automatic software tools – drastically reducing the chance of errors occurring.

Algorithms are the basis underlying computation, being expressed by a well-defined list of instructions in order to compute a function. Algorithms are at the heart of software, they define hardware manipulation such as hard-drive and memory access, as well as distributed computation such as distributed hash-tables and minimum spanning trees. A computer program, like an algorithm, is a sequence of instructions written to perform a specified task with a computer. Therefore, we can reason about a program's correctness in the same way we reason about an algorithm's correctness.

Historically, there are mainly three known ways of proving a program's correctness – type systems [Chu40], model checking [CES09] and program logics [Hoa69]. Type systems are used to verify the type safety property, which as stated by Milner ensures "well-typed programs cannot go wrong". Although being a very useful property in practice, we are interested in verifying a user specified property, such as "if at the beginning of program execution variable N has a positive value, then at the end of program execution variable R will hold the factorial of the initial value of N," which describes what the program is supposed to do. Therefore, since type checking is targeted at proving one fixed property for all programs, it is not suited to our goal. Model checking is based on analyzing the state-space of a system, in order to verify if a given user-specified property holds. Notice that this is a generic approach – as long as we can model the states of an arbitrary system, model checking searches the reachable states for any property violation. However, while a system usually has a finite number of states, the state-space of a program can be unbounded – a while loop that diverges can generate such a state-space. Therefore, if we want to have any hope of verifying programs, we need a more specific approach that takes into account the concrete constructs of the programming language one wants to verify, as well as the properties the user can reason about. Program logics provide that kind of syntax directed way of relating language constructs with properties that hold through their execution. While usually proven correct with respect to the effect program constructs have on the program state, one can only derive such a relation from results which are specific to the programming language to be verified. This approach, pioneered by Floyd and Hoare [Flo67, Hoa69], has recently been the target of a lot of attention by the research community in the area. As long as the underlying logic is sound and statements correctly express the desired behavior, we can be sure about software's correctness. Therefore, program logics is indeed the adequate formalism for what we are looking for. One important final remark is that these systems, unless otherwise stated, only check for partial correctness – as termination of a program is in general an undecidable problem, a separate termination proof is required to ensure total correctness.

1.2 Problem description

Until a few years ago, sequential programs were the norm in computing – a program was a list of instructions which were executed in the order they were written. Furthermore, since program execution was fixed and the effect each program instruction had was rather straightforward, informally reasoning about sequential programs was also the norm. However, in an environment where several programs are expected to execute at any given time, multi-core architectures were developed to improve performance. Given the multi-core architecture, a setting where several programs can be executed in parallel, the opportunity for a single program to execute several subtasks concurrently arose. While allowing for more efficient programs, multi-core architectures and concurrent programs were very specialized solutions, since such architectures were significantly more expensive than a standard processor and concurrent programming is difficult. Nowadays, most domestic computers have multi-core processors and concurrent programs are becoming the norm in computing.

Concurrent programming although able to obtain more efficient algorithms is quite harder to reason about, since the execution of two concurrent processes cannot be seen as the serialization of the first process followed by the second or vice-versa – one has to account for every possible interleaving of program instructions. Since there are exponentially many interleavings on the size of the program, it becomes hard to guarantee that any such interleaving will indeed hold an acceptable result. Therefore, mechanisms to limit the possible interleavings of a concurrent program were developed – atomic commands and locks were created. When having the capability to solely allow for interleavings which serialize the parts of the program that access shared resources, also called the critical regions of a program, it becomes easier to deal with concurrency – one only has to ensure critical regions keep shared resources consistent. However, mechanisms such as atomic constructs and locks see shared resource races as catastrophic and totally disallow it, serializing their access – shared resources then become a bottleneck, serializing each possible access even if no harm would arise from it.

To deal with serialization of access to resources in concurrent programs, several progress conditions were proposed. One progress condition which has become quite popular in the area is the nonblocking property – processes waiting to access a shared resource do not indefinitely hold because of another process accessing the same resource. Furthermore, nonblocking can be divided into three main progress conditions according to the tightness they impose on the overall process development – wait-freedom, lock-freedom and obstruction-freedom. Wait-freedom, the strongest sub-property, ensures progress of all processes while disallowing starvation of any single process – no process can be delayed forever because of the processor scheduler. Lock-freedom, while still enforcing progress of all processes, drops the starvation requirement – since fair schedulers are still not a realistic assumption and the cost of every other running process enforcing starvation-freedom is too high, starvation is usually not an addressed issue. Finally the weakest nonblocking sub-property, obstruction-freedom, enforces that any single process, while executed in isolation at any point, will finish in a finite number of steps.

In this context, we are especially interested in lock-free programs as they impose the most realistic assumptions on the program executing environment. Besides being a blocking mechanism, since any process trying to acquire a locked resource is unable to proceed, locks greatly reduce the chance for concurrency – as stated above, locks will serialize access to shared resources, even if some concurrent interference could be tolerated. Lock-free algorithms take an optimistic approach, and try to access shared resources in spite of concurrent interference, backing off in case of inconsistency. As explained in [HS08], this opportunity for concurrency often increases performance of concurrent algorithms. However, even in this narrower setting there are still choices to be made, since several distinct hardware primitives support lock-free programming. The most popular of these primitives is `CAS`, which stands for compare and swap, already having a formal proof system [GCPV09] to prove correctness of programs based on it. Unfortunately, `CAS` turns out to be unable to detect several important conflicting updates – as long as the initial value of the variable being tested is preserved at the end of an update, `CAS` will be unable to detect that update – ultimately resulting in a number of `CAS` tests equal to the size of the structure we want to keep consistent.

Following a concrete example covered in [HS08], Listings 1.1, 1.2 and 1.3 present a `Java` lock-free queue based on the `CAS` primitive. Listing 1.1 contains the `Node` class, used to store an element of the queue and its successor. Note that because the successor needs to be atomically compared and swapped in *enqueue* and *dequeue* operations, the `next` field has to be of type `AtomicReference`. Although not presented here, the `head` and `tail` fields are also of type `AtomicReference` since they too require `CAS` operations. Listing 1.2 details the *enqueue* method, which adds an element to the queue. The optimistic approach is encoded in the `while(true)` cycle, as it will attempt to execute as long as it is unsuccessful. Adding an element to the queue firstly loads the value of the *tail* into a local variable, and then checks if another value has been added since it firstly loaded that variable. Then, it finishes upon successfully changing the tail to the created node without interference, or restarts in face of concurrent modification. Because dequeuing – presented in Listing 1.3 – is very similar to enqueuing, its explanation will be skipped.

Listing 1.1: Node class

```java
class Node<T>{
  public T value;
  public AtomicReference<Node<T>> next;

  public Node(T value){
    this.value = value;
    next = new AtomicReference<Node<T>>(null);
  }
}
```

Listing 1.2: Enqueue method

```java
public void enq(T value){
  Node<T> node = new Node<T>(value), last, next;
  while(true){
    last = tail.get();
    next = last.next.get();
    if(last == tail.get()){
      if(next == null){
        if(last.next.compareAndSet(last, node)){
          tail.compareAndSet(last, node);
          return;
        }
      }
      else tail.compareAndSet(last, next);
    }
  }
}
```

Listing 1.3: Dequeue method

```java
public T deq() throws EmptyStackException{
  Node<T> first, last, next;
  while(true){
     first = head.get();
     last = tail.get();
     next = first.next.get();
     if(first == head.get()){
       if(first == last){
         if(next == null)
           throw new EmptyStackException();
         tail.compareAndSet(last, next);
       }
       else{
         T value = next.value;
         if(head.compareAndSet(first, next))
```

```
                **return** value;
            }
        }
    }
}
```

Although this algorithm actually works on `Java`, as explained in [HS08], that is not so trivial to check. Furthermore, this program only works correctly because of `Java`'s garbage collection policy – which is even harder to tell. Assuming a language with explicit memory management, like `C` and `C++`, it is possible for addresses that are known by some active process to be disposed of and recycled – often, that is the most efficient way of managing memory. However, in this scenario, the algorithm stops working correctly. Imagine that a program which starts with a queue consisting of nodes stored at a and b starts dequeuing the node at a and gets interrupted at line 15 of Listing 1.2, after freeing a and having placed a `CAS` call with addresses a and b to update the head. Then a program which dequeues a and b, and enqueues two nodes is executed – leaving the tail pointing to the last enqueued node. If address a is recycled during the first enqueue operation, the interrupted program will find that `head` still points to address a, and will change its successor to be b, overriding the second enqueue operation independently of what value was enqueued and leaving the queue inconsistent as `head` and `tail` are now unlinked. This is the result of a well-known issue called the ABA[1] problem – a primitive checks for change by solely comparing if an address holds a given value, and cannot distinguish between no change and changes that reestablish the initial value.

`LL/SC`, which stand for load-link and store-conditional, are a recent pair of primitives that allow us to solve this problem – load-link reads the value of a variable, and store-conditional writes a value into a variable that has remained unchanged since the last load-link, returning the success status of this write. Because `SC` checks if a variable loaded with `LL` has ever been updated, we hope it is now clear that the `LL/SC` pair of primitives are stronger than the `CAS` primitive – `CAS` ignores the updates as long as the initial value is replaced, while `SC` does not. We also believe that `LL/SC` more intuitively captures the optimistic setting than `CAS`, since there is an analogy between `LL` and a transaction start, as well as `SC` and a transaction commit or rollback.

Finally, we are ready to state the concrete problem to be tackled. Given the difficulty of reasoning about the correctness of concurrent programs – especially that of nonblocking programs – our goal is then to develop a formal proof system for lock-free concurrency based on the `LL/SC` pair of primitives. More specifically, we developed such a proof system for an imperative language with the usual sequential constructs – skip, variable read and write, conditional branching, looping and sequential composition – shared memory

[1] http://en.wikipedia.org/wiki/ABA_problem

read and write, parallel composition of programs, and the lock-free `LL/SC` primitives. Furthermore, our inference rules focus on proving the "local footprint" of a program, and for this proof to carry over into a wider environment – allowing for a local and modular way of reasoning about programs.

1.3 Proposed solution

Although no formalism yet exists for the specific kind of program we want to verify, as will be detailed in the following chapter, several such formalisms already exist for other settings – including the concurrent setting. Fortunately, the approach used to solve these other settings is also adequate for our scenario. However, although the approach is the same, this does not mean its application to the specific set of primitives and properties we want to have is trivial. The approach, pioneered by Floyd and Hoare [Flo67, Hoa69], consists of the following components:

Programs A way of modeling the program one wants to verify, as well as the precise meaning of its constructs.

Program syntax The program syntax is given by the grammar of a preferably minimal set of commands, which are able to capture the constructs found on the programming languages one wishes to verify. In our setting, we need syntactical constructs for skip, variable read and write, conditional branching, looping, sequential composition, shared memory read and write, parallel composition, and the lock-free `LL/SC` primitives;

Operational semantics The operational semantics is a formal interpretation of the commands present in the programming syntax, which precisely models the possible states of a program and the transition between program states resulting from the execution of a command. In our setting, we use structural operational semantics [Plo04] – i.e. syntax oriented and inductive – for each of the above program constructs.

Assertions A way of expressing the properties one wants reason about, as well as the precise meaning of each property.

Assertion syntax The statement syntax is given by a grammar of properties one can write about a program, which establishes what can be proven regarding programs. In our setting we have classical logic constructs, as well as properties to reason about shared memory – the empty memory constant and the points-to and separating conjunction operators, introduced by Reynolds [Rey02]. Furthermore, assertions will distinguish between what holds on two disjoint pieces of memory – the program's local resources and the shared environment's resources, as in [Vaf07];

Assertion semantics The statement semantics establishes when a program state satisfies a given property. In our setting, we have a mathematical logic predicate that decides whether an assertion holds at a certain program state.

Proof system A way of using assertions to reason about programs.

Validity property The validity property tells us how to use statements to reason about programs, and when a given specification is either true or false. In our setting, validity will ensure that the program does not fault (i.e. access unallocated memory), does not have data races (i.e. modify variables some other is using) and, if executed in any state that satisfies a given precondition, then it establishes a certain postcondition;

Inference rules Inference rules are a syntax directed way of proving a specification's correctness, which must be derivable from the validity property (i.e. sound). In our setting, besides having a rule for each program construct, we also have framing rules which allow us to extend a local proof into a wider non-interfering environment – accounting for locality and modularity.

Therefore, the solution detailed on this dissertation consists in developing all of the above – bringing forward an original proof system which takes into account `LL/SC` primitives. Furthermore, an informal argument on the inference rules' soundness and significant examples of their application are presented, in order to validate that our proof system is indeed correct and adequate for our environment.

1.4 Main contributions

With this dissertation, providing the previous solution, we bring forward the following contributions:

Precise account of `LL/SC` primitives With the development of an operational semantics which rigorously captures the effect `LL/SC` primitives have on the program state, one is better able to reason about their usage on programming languages;

Adaptation of existing formalisms By applying results from other works to a different scenario – lock-free programs based on `LL/SC` primitives – we shed some light on how properties of lock-based systems relate to an optimistic scenario;

New inference rules While some results are already present on previous works, given no current formalism accounts for `LL/SC` primitives we need new inference rules and an underlying framework to address them;

Soundness argument In order to ensure syntactic proofs are indeed correct regarding their semantic counterparts, we supply a structured argument on their soundness – also explaining why some limitations still exist. Furthermore, with this argument we lay out the basis for formal soundness proofs and improvements on the system's expressiveness.

1.5 Outline

In this chapter we have just explained the need for a new proof system which addresses `LL/SC` primitives and allows reasoning about lock-free programs. Then, based on existing formalisms for similar problems, a solution targeting this need as well as its contributions to the area were proposed.

Chapter 2 presents aforementioned formalisms – Hoare logic, separation logic, concurrent separation logic and `RGSep` – additionally detailing their relevance and concepts used in this dissertation. These works establish the basis on which our proof system is built on, and give the reader enough context to follow it.

Chapter 3 contains the bulk of our work, introducing and defining the proof system we have developed for proving correctness of lock-free programs based on `LL/SC` primitives. Additionally, an argument on the system's soundness and examples showcasing its usage are presented as well.

Chapter 4 gives an overview of this dissertation and, then with more context, goes through its contributions once again – detailing what we are able to achieve. Concluding, some features that fall out of the scope of this work as well as limitations of the current system are pointed out, so it can be improved in the future.

2 Related work

Although there did not exist a way of proving the correctness of lock-free programs based on the `LL/SC` pair of primitives, such tools were already present for other kinds of program as a result of previous research on the field of program logics. As mentioned before, several formalisms which tackle the problem of proving correctness of both sequential and concurrent programs with varying degrees of expressiveness already exist. In this chapter we will detail the relevant works on which our proof system was based on, and explain how they come into play in the proposed solution. Note that the logics covered by this chapter will be presented in an incremental fashion, which corresponds to how their actual development was done – each logic adds some more commands to the programming language, or allows for more expressiveness regarding the one it was based upon.

Furthermore, although formal logics for the specific programs we want to prove are lacking, there already exists a great amount of informal reasoning about lock-free programs – resulting in a solid basis of the domain being tackled. Therefore, this chapter will also present existing lock-free concepts as well as the kind of properties we wish to be able to prove.

2.1 Hoare logic

Introduced by Tony Hoare in 1969, Hoare logic [Hoa69] was one of the pioneering program logics and the first of its kind – its key contribution being the introduction of a set of logical rules for formally reasoning about the correctness of a computer program. Hoare logic spawned a body of work around it, resulting in several formalizations and extensions. This section is based on the work of Glynn Winskel [Win93], which presents both an operational and denotational semantics for programs, as well as the correctness of Hoare's rules. Note that in order to keep definitions of the works presented in this chapter consistent, the notation slightly differs from the one used in that book.

Hoare logic focuses on proving correctness of sequential programs which only manipulate variables. This kind of program was the norm in the time of its publication, and our target programming language includes them. Therefore, Hoare logic is relevant to our work since it deals with an important fragment of the target language.

The programming syntax, presented in Figure 2.1, is expressed using an extended Backus–Naur form grammar – from now onwards EBNF grammar – and includes the skip, assignment, sequential composition, conditional branching and loop constructs. Henceforth the term command will also be used to refer to a program, since both are used interchangeably in the literature being covered. Additionally, the grammar of boolean and arithmetic expressions, needed for conditional tests and the right-hand side of assignments, is also shown in Figure 2.1. Arithmetic expressions allow for addition, subtraction and multiplication of subexpressions, while boolean expressions allow for conjunction, disjunction and negation of subexpressions, as well as the usual equal to and lesser or equal to comparisons between arithmetic values. Although several other constructs such as `do-while` commands and greater than comparison may exist in both the programming language and its expressions, the syntax should be minimal while still allowing for any target program to be modeled using it, as minimality allows for easier reasoning and shorter correctness proofs. Moreover, such non-minimal commands and expressions can promptly be obtained as a composition of the presented ones.

$$
\begin{array}{lcll}
C & ::= & & \text{Commands} \\
 & | & \texttt{skip} & \text{(skip)} \\
 & | & V := E & \text{(assignment)} \\
 & | & C ; C & \text{(sequential composition)} \\
 & | & \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C & \text{(conditional)} \\
 & | & \texttt{while}\ B\ \texttt{do}\ C & \text{(while)} \\
 \\
V & ::= & x \mid y \mid z \mid \cdots & \text{Variables} \\
E & ::= & V \mid \mathbb{Z} \mid E + E \mid E - E \mid E \times E & \text{Arithmetic expressions} \\
B & ::= & \texttt{true} \mid \texttt{false} \mid E = E \mid E \leq E & \text{Boolean expressions} \\
 & | & \neg B \mid B \wedge B \mid B \vee B &
\end{array}
$$

Figure 2.1: Programming language syntax

At the core of Hoare logic is a formal account of what programs do. We call such formalism the semantics of a programming language, and it can be expressed in several ways – the most common ones being operational and denotational semantics. Denotational semantics assign meaning to programs recurring to some other well defined formalism, through an interpretation function which maps programs to elements belonging to the domain of the target formalism. The evaluation of expressions presented in Figure 2.3 is an example of a denotational semantics, mapping arithmetic and boolean expressions into integers and booleans. Operational semantics, instead of mapping programs into another domain, assigns meaning to programs themselves. More specifically the works mentioned in this chapter use structural operational semantics [Plo04], which assign meaning to programs through the behavior of their parts – allowing for a syntax directed (i.e. structural) and inductive definition. Figure 2.4 is an example of structural operational semantics, presenting a transition function to detail how a command evolves into another command.

$$\begin{array}{lcl} \mathit{Value} & \triangleq & \mathbb{Z} \\ \mathit{Store} & \triangleq & V \rightarrow \mathit{Value} \\ \mathit{Config} & \triangleq & C \times \mathit{Store} \end{array}$$

Figure 2.2: Program states and data

$\mathcal{E} : E \times \mathit{Store}$	$\rightarrow$	Value	Arithmetic expression
$\mathcal{E}[\![n]\!]_s$	$\triangleq$	n	(integer)
$\mathcal{E}[\![x]\!]_s$	$\triangleq$	$s(x)$	(variable)
$\mathcal{E}[\![e_1\ \mathit{op}\ e_2]\!]_s$	$\triangleq$	$\mathit{op}_{\mathcal{A}}(\mathcal{E}[\![e_1]\!]_s, \mathcal{E}[\![e_2]\!]_s)$	(operator)
$\mathcal{B} : B \times \mathit{Store}$	$\rightarrow$	$\{\top, \bot\}$	Boolean expression
$\mathcal{B}[\![\mathtt{true}]\!]_s$	$\triangleq$	$\top$	(true)
$\mathcal{B}[\![\mathtt{false}]\!]_s$	$\triangleq$	$\bot$	(false)
$\mathcal{B}[\![e_1 = e_2]\!]_s$	$\triangleq$	$(\mathcal{E}[\![e_1]\!]_s = \mathcal{E}[\![e_2]\!]_s)$	(equals)
$\mathcal{B}[\![e_1 \leq e_2]\!]_s$	$\triangleq$	$(\mathcal{E}[\![e_1]\!]_s \leq \mathcal{E}[\![e_2]\!]_s)$	(lesser or equal)
$\mathcal{B}[\![\neg b]\!]_s$	$\triangleq$	$\neg\mathcal{B}[\![b]\!]_s$	(negation)
$\mathcal{B}[\![b_1\ \mathit{op}\ b_2]\!]_s$	$\triangleq$	$\mathit{op}_{\mathcal{A}}(\mathcal{B}[\![b_1]\!]_s, \mathcal{B}[\![b_2]\!]_s)$	(operator)

Figure 2.3: Expression evaluation functions

In order to assign meaning to a program, one firstly has to model what a program state is. Notice that programs operate on integer values, and only manipulate variables which store values – this way values can be modeled as integers and variables as a total function, assigning a value to each variable. The modeling of program states is presented in Figure 2.2, and expresses just this. Since a program solely manipulates variables, a given program store uniquely identifies the correspondent program state and vice-versa. After

$$\rightsquigarrow\, : \mathit{Config} \to \mathit{Config} \qquad \text{Program transition}$$

$$\frac{\mathcal{E}[\![e]\!]_s = v}{(x := e, s) \rightsquigarrow (\texttt{skip}, s[x \mapsto v])} \qquad \text{(assign)}$$

$$\frac{}{(\texttt{skip}; c, s) \rightsquigarrow (c, s)} \qquad \text{(sequential}_1\text{)}$$

$$\frac{(c_1, s) \rightsquigarrow (c_1', s')}{(c_1; c_2, s) \rightsquigarrow (c_1'; c_2, s')} \qquad \text{(sequential}_2\text{)}$$

$$\frac{\mathcal{B}[\![b]\!]_s = \top}{(\texttt{if}\; b\; \texttt{then}\; c_1\; \texttt{else}\; c_2, s) \rightsquigarrow (c_1, s)} \qquad \text{(if–then–else}_T\text{)}$$

$$\frac{\mathcal{B}[\![b]\!]_s = \bot}{(\texttt{if}\; b\; \texttt{then}\; c_1\; \texttt{else}\; c_2, s) \rightsquigarrow (c_2, s)} \qquad \text{(if–then–else}_F\text{)}$$

$$\frac{\mathcal{B}[\![b]\!]_s = \bot}{(\texttt{while}\; b\; \texttt{do}\; c, s) \rightsquigarrow (\texttt{skip}, s)} \qquad \text{(while}_F\text{)}$$

$$\frac{\mathcal{B}[\![b]\!]_s = \top}{(\texttt{while}\; b\; \texttt{do}\; c, s) \rightsquigarrow (c; \texttt{while}\; b\; \texttt{do}\; c, s)} \qquad \text{(while}_T\text{)}$$

Figure 2.4: Operational semantic rules

defining what a program state is, we can then define what the evaluation of an expression at a given program state is – as presented in Figure 2.3. Note that in order to avoid having several similar definitions for binary operator evaluation, we use a single rule where op can range over any binary operator and $op_{\mathcal{A}}$ matches the corresponding algebraic or logic operator – for example, the binary syntactic + operator matches with the addition operator over integers. Finally, after defining what a program state is and having a way of evaluating expressions, we can assign meaning to commands. Configurations, also presented in Figure 2.2, contain a program and the initial state in which it is to be executed. The meaning of a program is then described as a transition function, which takes a program and a state, and models the execution of a single computational step of the program at that state – being thus called small-step semantics – resulting in a new state and the rest of the program to be executed. Note that unless the program diverges because of a `while` loop, iterating the transition function will result in a configuration where no transition exists, holding the final state of a program execution. Finally, $f[x \mapsto v]$ – used to define the semantics of assign – refers to the result of modifying function f so that x maps to v.

Beyond giving a precise account of program execution, program logics also allow us to reason about what programs actually do. The way one has to reason about a program in Hoare logic is through assertions – that is, a statement of what properties hold at a given program state. The allowed statements one can make about a program are presented in Figure 2.5, and include boolean expressions, conjunction and negation of sub-properties, as well as existential quantification – this last statement being the only addition to boolean expressions.

$$
\begin{array}{lcll}
P & ::= & & \text{Assertions} \\
 & | & B & \text{(boolean expression)} \\
 & | & P \wedge P & \text{(conjunction)} \\
 & | & \neg P & \text{(negation)} \\
 & | & \exists_V\, P & \text{(existential quantification)}
\end{array}
$$

Figure 2.5: Assertion language syntax

Just like we have a precise way of telling what a program does, we also have a rigorous way of deciding when an assertion holds – this is done through the assertion semantics, presented in Figure 2.6. Once again we recur to an evaluation function, which takes an assertion and a program state and maps them into a boolean – using the syntax and semantics of first-order logic – deciding whether the assertion holds in a given state or not.

$$
\begin{array}{lcll}
\mathcal{P} : P \times \mathit{Store} & \to & \{\top, \bot\} & \text{Assertion} \\
\mathcal{P}[\![b]\!]_s & \triangleq & \mathcal{B}[\![b]\!]_s) & \text{(boolean expression)} \\
\mathcal{P}[\![p_1 \wedge p_2]\!]_s & \triangleq & \mathcal{P}[\![p_1]\!]_s \wedge \mathcal{P}[\![p_2]\!]_s & \text{(conjunction)} \\
\mathcal{P}[\![\neg p]\!]_s & \triangleq & \neg \mathcal{P}[\![p]\!]_s & \text{(negation)} \\
\mathcal{P}[\![\exists_x\, p]\!]_s & \triangleq & \exists_{v \in \mathit{Value}}\, \mathcal{P}[\![p]\!]_{s[x \mapsto v]} & \text{(existential quantification)}
\end{array}
$$

Figure 2.6: Assertion semantics

Perhaps more importantly than being able to say something holds in a given program state, we can also reason about the full execution of a given program – since we are interested in proving properties of a whole algorithm implementation. To reason about a complete program, Hoare logic provides the user with Hoare triples consisting of a precondition, a program and a postcondition – where both pre and post conditions are assertions. The validity of a Hoare triple can then be expressed by the following definition:

Validity. We say a Hoare triple $\{p\}c\{q\}$ holds, and write it as $\vDash \{p\}c\{q\}$, when any execution of command c that starts in a state satisfying assertion p, and terminates, will result in a program state that satisfies assertion q. Formally, we can define it as follows:

$$\vDash \{p\}c\{q\} \triangleq \forall_{s,s' \in \mathit{Store}} \left(\mathcal{P}[\![p]\!]_s \wedge \left[(c,s) \rightsquigarrow^* (\mathtt{skip}, s') \right] \right) \Rightarrow \mathcal{P}[\![q]\!]_{s'}$$

Therefore, to reason about a program, one imposes a condition upon the initial state under which, when executed, the program will result in a final state satisfying a given property. Note that if a program diverges, this property will vacuously be satisfied – a program that does not terminate satisfies any Hoare triple. That is why the definition of validity only ensures partial correctness of programs – total correctness also requires programs to terminate, dismissing these troublesome situations. The presented works are only interested in partial correctness, as a separate termination proof can be provided in cases where total correctness is required.

Although the above suffices to reason about programs, we are interested in automating correctness proofs in order for usual programmers with no knowledge of the underlying logic to be able to prove correctness of their code. However, it is easy to see that checking if a Hoare triple $\{p\}c\{q\}$ holds is undecidable – to check that $\{\mathtt{true}\}c\{\mathtt{false}\}$ does not hold is to solve the halting problem. In order to deal with this problem, and to make proofs manageable, inference rules presented in Figure 2.7 provide a syntax-directed way of proving the validity of a Hoare triple. Note that since proving an arbitrary Hoare triple is undecidable, these rules cannot be complete – that is, able to provide a proof of every valid Hoare triple.

$$\frac{}{\vdash \{p\}\mathtt{skip}\{p\}} \quad \text{(skip)}$$

$$\frac{}{\vdash \{p[e/x]\}x := e\{p\}} \quad \text{(assign)}$$

$$\frac{\vdash \{p \wedge b\}c_1\{q\} \quad \vdash \{p \wedge \neg b\}c_2\{q\}}{\vdash \{p\}\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\{q\}} \quad \text{(if–then–else)}$$

$$\frac{\vdash \{p \wedge b\}c\{p\}}{\vdash \{p\}\mathtt{while}\ b\ \mathtt{do}\ c\{p \wedge \neg b\}} \quad \text{(while)}$$

$$\frac{p \Rightarrow p' \quad \vdash \{p'\}c\{q'\} \quad q' \Rightarrow q}{\vdash \{p\}c\{q\}} \quad \text{(consequence)}$$

$$\frac{\vdash \{p\}c_1\{q'\} \quad \vdash \{q'\}c_2\{q\}}{\vdash \{p\}c_1; c_2\{q\}} \quad \text{(sequential)}$$

Figure 2.7: Proof rules

Inference rules provide an axiomatic way of proving validity of Hoare triples, by relating commands with assertions that hold through their execution. For example, the assign rule tells us that if an assertion holds in a state prior to an assignment, then after the assignment we can substitute the expression appearing on the right-hand side of the assignment by the variable on its left-hand side. Although this informal explanation of a particular rule may seem convincing, one must take the care to ensure that whenever $\vdash \{p\}c\{q\}$ holds, then $\vDash \{p\}c\{q\}$ also holds – it is often the case that non-trivial side conditions are necessary, therefore a correctness proof is required to ensure that $\vdash \{p\}c\{q\}$ indeed implies $\vDash \{p\}c\{q\}$. Reciprocal implication would also ensure completeness of the rules, which as stated before is impossible. Although being important, as we feel it would not add much to this presentation, we forward the interested reader to the correctness proof present in the work of Glynn Winskel [Win93].

Concluding, Hoare logic provides developers with a way of proving correctness of their programs and logicians with an approach to develop similar proof-systems. Currently there exist several implementations based on this approach, such as `ESC/Java` [FLL$^+$02] for `Java` programs and `Spec#` [BLS05] for `C#` programs. As already stated, this logic deals with the sequential and variable manipulation fragment – present in most programming languages. However, current programming languages also have memory manipulation commands – such as memory allocation and pointer dereference – as well as parallel composition of programs. Therefore, although being a significant step towards our solution since it provides us with the means to develop a proof-system, it falls short on dealing with such programs.

2.2 Separation logic

Introduced by John Reynolds in 2002, separation logic [Rey02] is an extension of Hoare logic targeted at proving correctness of programs that manipulate pointer data structures. Memory manipulation is present in most programming languages, either directly as in `C` or indirectly as in `Java`, therefore it is desirable to be able to think about primitives that allow just this. As the programming language we aim at basing our proof-system upon also supports these primitives, separation logic is directly relevant to the solution we pretend to develop. Note that all the sections about program logics follow the same structure used to present Hoare logic, once again providing every element of the solution we propose for a fragment of the programs we wish to deal with – except for a soundness proof.

In order to deal with data manipulation primitives, we extend the command language of Figure 2.1 with four new constructs – allocating, loading, storing and disposing of memory addresses. The extended syntax is then presented in Figure 2.8, and complements that of Figure 2.1.

$$
\begin{array}{rcll}
C & ::= & \ldots & \text{Commands} \\
 & | & V := \texttt{alloc}(E, \ldots, E) & \text{(allocate)} \\
 & | & V := [E] & \text{(load)} \\
 & | & [E] := E & \text{(store)} \\
 & | & \texttt{disp}(E) & \text{(dispose)}
\end{array}
$$

Figure 2.8: Programming language syntax

The structural operational semantics in Figure 2.10 gives a precise account of what these new commands do. Note that because memory now has to be considered, a program state is no longer solely given by the assignment it makes of variables. As shown in Figure 2.9, several new components are needed to model a state – addresses, atoms and heaps. Since a computer's memory is bounded and programs use pointer arithmetic, addresses are modeled as a finite subset of the integers. Atoms are reserved values, which are not addresses – the most well known case being the `nil` value. We can then define a heap to be a mapping between addresses and values stored at those addresses – it is represented as a partial function, as addresses which are free to be taken do not belong to the domain of the heap. Program states are then defined as a pair consisting of a *store*, recording the values stored in each variable, and a *heap* that keeps track of the taken addresses as well as their values. Finally, configurations are either a pair consisting of a program to be executed and its initial state, or the `abort` *configuration* – representing a fault resulting from an illegal dereference.

$$
\begin{array}{lcl}
\mathit{Atom} \cup \mathit{Address} & \subseteq & \mathbb{Z} \\
\mathit{Atom} \cap \mathit{Address} & = & \varnothing \\
\texttt{nil} & \in & \mathit{Atom} \\
\mathit{Heap} & \triangleq & \mathit{Address} \rightharpoonup \mathit{Value} \\
\mathit{State} & \triangleq & \mathit{Store} \times \mathit{Heap} \\
\mathit{Config} & \triangleq & (C \times \mathit{State}) \cup \{\texttt{abort}\}
\end{array}
$$

Figure 2.9: Program states and data

After defining the domain on which a program will execute, the operational semantics in Figure 2.10 models the transition function between program configurations – once again using small-step semantics. The more cautious reader will notice that although the transition function presented here is an extension of the one presented in Figure 2.4, the configurations on which they operate are different. For the sake of minimality we will not present that modification here, but one can simply add a heap to the configuration of Figure 2.4 and alter the correspondent transition function to directly pass the initial heap onto the resulting one. Although we believe the rules to be pretty straightforward, we would like to point out that some program executions may now lead to an `abort` state – resulting from trying to operate on a memory address not present on the heap's domain. One last remark that may not be so obvious, is that since memory allocation is nondeterministic – an `alloc` operation can allocate any contiguous block of memory addresses – a program's execution is also no longer deterministic. Therefore, running the same program twice in an equal initial state may yield two different final configurations.

Although used extensively on the referenced works, we would like to make clear the definition of $\mathrm{dom}(f)$ and $f\rceil d$ – used on Figure 2.4. Expression $\mathrm{dom}(f)$ refers to the domain of function f, while $f\rceil d$ is the result of restricting function f to domain d.

$$\frac{\begin{array}{l}\mathcal{E}[\![e_1]\!]_s = v_1 \wedge \ldots \wedge \mathcal{E}[\![e_n]\!]_s = v_n \wedge \\ \quad \{a, \ldots, a+n-1\} \subseteq \mathit{Address} \setminus \mathrm{dom}(h)\end{array}}{\begin{array}{l}\langle x := \mathtt{alloc}(e_1, \ldots, e_n), (s, h)\rangle \rightsquigarrow \\ \quad \langle \mathtt{skip}, (s[x \mapsto a], h[a \mapsto v_1, \ldots, a+n-1 \mapsto v_n])\rangle\end{array}} \quad \text{(allocate)}$$

$$\frac{\mathcal{E}[\![e]\!]_s = a \wedge a \in \mathrm{dom}(h)}{\langle x := [e], (s, h)\rangle \rightsquigarrow \langle \mathtt{skip}, (s[x \mapsto h(a)], h)\rangle} \quad \text{(load)}$$

$$\frac{\mathcal{E}[\![e]\!]_s \notin \mathrm{dom}(h)}{\langle x := [e], (s, h)\rangle \rightsquigarrow \mathtt{abort}} \quad \text{(load}_A\text{)}$$

$$\frac{\mathcal{E}[\![e_1]\!]_s = a \wedge a \in \mathrm{dom}(h) \wedge \mathcal{E}[\![e_2]\!]_s = v}{\langle [e_1] := e_2, (s, h)\rangle \rightsquigarrow \langle \mathtt{skip}, (s, h[a \mapsto v])\rangle} \quad \text{(store)}$$

$$\frac{\mathcal{E}[\![e_1]\!]_s \notin \mathrm{dom}(h)}{\langle [e_1] := e_2, (s, h)\rangle \rightsquigarrow \mathtt{abort}} \quad \text{(store}_A\text{)}$$

$$\frac{\mathcal{E}[\![e]\!]_s = a \wedge a \in \mathrm{dom}(h)}{\langle \mathtt{disp}(e), (s, h)\rangle \rightsquigarrow \langle \mathtt{skip}, (s, h\rceil(\mathrm{dom}(h) \setminus \{a\}))\rangle} \quad \text{(dispose)}$$

$$\frac{\mathcal{E}[\![e]\!]_s \notin \mathrm{dom}(h)}{\langle \mathtt{disp}(e), (s, h)\rangle \rightsquigarrow \mathtt{abort}} \quad \text{(dispose}_A\text{)}$$

$$\frac{\langle c_1, (s, h)\rangle \rightsquigarrow \langle c_1', (s', h')\rangle}{\langle c_1; c_2, (s, h)\rangle \rightsquigarrow \langle c_1'; c_2, (s', h')\rangle} \quad \text{(sequential}_2\text{)}$$

$$\frac{\langle c_1, (s, h)\rangle \rightsquigarrow \mathtt{abort}}{\langle c_1; c_2, (s, h)\rangle \rightsquigarrow \mathtt{abort}} \quad \text{(sequential}_A\text{)}$$

Figure 2.10: Operational semantic rules

Besides being able to model the effect pointer manipulation primitives have on the program state, the goal of program logics is to reason about those operations as well. In order to do that, one has to extend the assertion language with constructs that also take the heap into account. Figure 2.11 introduces the novelties in the assertion language – the *empty* constant and the *separation* and *points to* operators.

$$
\begin{array}{rcll}
P & ::= & \ldots & \text{Assertions} \\
 & | & \texttt{emp} & \text{(empty)} \\
 & | & P * P & \text{(separation)} \\
 & | & E \mapsto E & \text{(points to)} \\
 & & & \\
e \mapsto - & \triangleq & \exists_x\, e \mapsto x \text{, where } x \notin \mathrm{fv}(e) & \text{(anonymous)} \\
e \mapsto e_1, \ldots, e_n & \triangleq & e \mapsto e_1 * \ldots * e + n - 1 \mapsto e_n & \text{(multi-valued)}
\end{array}
$$

Figure 2.11: Assertion language syntax

The meaning of these new elements in the assertion language is detailed on Figure 2.12. Formula `emp` states that the current program heap does not contain any address in its domain, and separation checks if the current heap can be split into two disjoint sub-heaps where one satisfies the first statement and the other satisfies the second. Finally, the *points to* operator ensures the heap consists only of a given memory cell, which holds an assigned value. Two other useful constructs are also presented – which can be built using previous ones – the *anonymous* and *multi-valued* versions of the *points to* operator. Note that, similarly to the operational semantics, the type of the evaluation function defined in Figure 2.12 differs from that of Figure 2.6 which it extends. Once again, it is easy to adapt the definition of Figure 2.6 to a scenario where the additional heap argument is ignored.

$$
\begin{array}{lcll}
\mathcal{P} : P \times \mathit{State} & \rightarrow & \{\top, \bot\} & \text{Assertion} \\
\mathcal{P}[\![\texttt{emp}]\!]_{(s,h)} & \triangleq & \mathrm{dom}(h) = \varnothing & \text{(empty)} \\
\mathcal{P}[\![p_1 * p_2]\!]_{(s,h)} & \triangleq & \exists_{h_1,h_2}\, h = h_1 \uplus h_2 \wedge \mathcal{P}[\![p_1]\!]_{(s,h_1)} \wedge \mathcal{P}[\![p_2]\!]_{(s,h_2)} & \text{(separation)} \\
\mathcal{P}[\![e_1 \mapsto e_2]\!]_{(s,h)} & \triangleq & \mathcal{E}[\![e_1]\!]_s = a \wedge \mathrm{dom}(h) = \{a\} \wedge h(a) = \mathcal{E}[\![e_2]\!]_s & \text{(points to)}
\end{array}
$$

Figure 2.12: Assertion semantics

After having defined both separation logic's programs and properties, we are again ready to reason about a program as a whole. Once more, we recur to the usual $\{p\}c\{q\}$ triples to reason about the correctness of algorithms. Because a program may now fault, the definition of validity has to be changed in order to deal with such troublesome executions. Intuitively, correctness of a program additionally guarantees that any execution starting in a "good" state will not fault.

Validity. We say a triple $\{p\}c\{q\}$ holds, and write it as $\vDash \{p\}c\{q\}$, when any execution of command c that starts in a state satisfying assertion p does not fault and, if it terminates, results in a program state that satisfies assertion q. Formally, we can define it as follows:

$$\begin{aligned}\vDash \{p\}c\{q\} \quad &\triangleq \quad \forall_{\sigma,\sigma' \in State}\, \mathcal{P}[\![p]\!]_\sigma \Rightarrow \\ &\big(\langle c, \sigma\rangle \not\rightsquigarrow^* \texttt{abort}\big) \wedge \big(\forall_{\sigma' \in State} \langle c, \sigma\rangle \rightsquigarrow^* \langle \texttt{skip}, \sigma'\rangle \Rightarrow \mathcal{P}[\![q]\!]_{\sigma'}\big)\end{aligned}$$

Undecidability of checking whether $\vDash \{p\}c\{q\}$ holds is an easy result, as separation logic is an extension of Hoare logic and the definition of validity presented above is equivalent to the one in Hoare logic when dealing with that specific fragment – as a program that does not access the heap never faults. Because the main goal of proof systems is to provide a syntax oriented way of proving correctness of programs, separation logic also provides inference rules for this purpose. These rules are presented in Figure 2.13, and we would like to highlight the frame rule. Being able to solely focus on the addresses a specific program uses is a desirable property – called local reasoning. Local reasoning is important as it limits the scope of what is to be proven as much as possible, while also allowing for modular proofs. At the heart of local reasoning is the *frame* rule, allowing for a local proof to be extended with an arbitrary statement about a non-interfering heap. Note that every other rule only talks about the relevant addresses – both in the pre and post conditions. Locality is then a property which we believe is key to keeping proofs manageable, and aimed at maintaining in our work. Adding one final remark about proof rules, we would like to explain the meaning of the *writes* function. While functions like *fv* and *dom* are well known – representing the free variables of a logical formula and the domain of a function – the meaning of the *writes* function is not standard, and informally represents the set of all left-hand sides of assignments which are variables. Although the *writes* function can easily be defined, it shall be omitted in this chapter for the sake of brevity.

$$\frac{}{\vdash \{\texttt{emp}\}x := \texttt{alloc}(e_1, \ldots, e_n)\{x \mapsto e_1, \ldots, e_n\}} \quad \text{(allocate)}$$

$$\frac{}{\vdash \{p[e'/x] \wedge e \mapsto e'\}x := [e]\{p \wedge e \mapsto e'\}} \quad \text{(load)}$$

$$\frac{}{\vdash \{e_1 \mapsto -\}[e_1] := e_2\{e_1 \mapsto e_2\}} \quad \text{(store)}$$

$$\frac{}{\vdash \{e \mapsto -\}\texttt{disp}(e)\{\texttt{emp}\}} \quad \text{(dispose)}$$

$$\frac{\vdash \{p\}c\{q\} \quad \text{fv}(f) \cap \text{writes}(c) = \varnothing}{\vdash \{p * f\}c\{q * f\}} \quad \text{(frame)}$$

Figure 2.13: Proof rules

Concluding, separation logic provides a formal account of programs that manipulate pointer data structures, as well as a way of proving their correctness. In order to do this, program states had to be augmented with a heap – keeping track of memory addresses. Accordingly, assertions and validity were also extended to this scenario. Perhaps more importantly, separation logic allows for local reasoning, enabling us to solely focus on the "footprint" of a program. As previously stated, locality is a solid step towards making program logics realistic in practice – therefore, we aimed at preserving it. Although covering the data manipulation primitives we pretend to have on our target programming language, separation logic only allows for sequential composition of programs – falling short on providing a mechanism to deal with concurrency.

2.3 Concurrent separation logic

Introduced by Peter O'Hearn and formalized by Stephen Brookes in 2007, concurrent separation logic [O'H07, Bro07] is an extension of separation logic which aims at proving correctness of concurrent programs that manipulate shared pointer data structures. As previously explained concurrency is an actual form of computing supported by most current programming languages, which we also aim at providing a means of reasoning about. Therefore, because concurrent separation logic allows for concurrent composition of programs, it is directly related to our work. Since the original definition by Brookes [Bro07] uses a trace semantics – which is significantly different from the structural operational semantics present in most works – in this section we will follow one of the latest research papers on the topic by Viktor Vafeiadis [Vaf11], with the goal of unifying forthcoming developments.

$$
\begin{array}{rcll}
C & ::= & \ldots & \text{Commands} \\
 & \mid & C \parallel C & \text{(parallel)} \\
 & \mid & \texttt{resource}\ R\ \texttt{in}\ C & \text{(resource)} \\
 & \mid & \texttt{with}\ R\ \texttt{when}\ B\ \texttt{do}\ C & \text{(with)} \\
 & \mid & \texttt{within}\ R\ \texttt{do}\ C & \text{(within)}
\end{array}
$$

Figure 2.14: Programming language syntax

The programming syntax, presented in Figure 2.14, adds three new constructs to the programming language presented in the previous section, and an auxiliary construct needed to define the operational semantics – parallel composition, resource creation, resource acquisition and resource ownership. Previously, using sequential composition only the left side of a command could execute, however – as can be seen in Figure 2.15 – parallel composition allows for any of the subprograms to execute, generating all possible interleavings. Therefore, despite only adding a few new constructs, program execution may result in exponentially larger number of program states. In order to have some control over these concurrent traces, resources are added to restrict those interleavings.

$$\frac{\langle c_1, \sigma\rangle \rightsquigarrow \langle c_1', \sigma'\rangle \wedge \text{locked}(c_1') \cap \text{locked}(c_2) = \varnothing}{\langle c_1 \parallel c_2, \sigma\rangle \rightsquigarrow \langle c_1' \parallel c_2, \sigma'\rangle} \quad (\text{parallel}_1)$$

$$\frac{\langle c_2, \sigma\rangle \rightsquigarrow \langle c_2', \sigma'\rangle \wedge \text{locked}(c_1) \cap \text{locked}(c_2') = \varnothing}{\langle c_1 \parallel c_2, \sigma\rangle \rightsquigarrow \langle c_1 \parallel c_2', \sigma'\rangle} \quad (\text{parallel}_2)$$

$$\frac{}{\langle \texttt{skip} \parallel \texttt{skip}, \sigma\rangle \rightsquigarrow \langle \texttt{skip}, \sigma\rangle} \quad (\text{parallel}_3)$$

$$\frac{\langle c_1, \sigma\rangle \rightsquigarrow \texttt{abort}}{\langle c_1 \parallel c_2, \sigma\rangle \rightsquigarrow \texttt{abort}} \quad (\text{parallel}_{A1})$$

$$\frac{\langle c_2, \sigma\rangle \rightsquigarrow \texttt{abort}}{\langle c_1 \parallel c_2, \sigma\rangle \rightsquigarrow \texttt{abort}} \quad (\text{parallel}_{A2})$$

$$\frac{\big[\text{accesses}(c_1, s) \cap \text{writes}(c_2, s)\big] \cup \big[\text{accesses}(c_2, s) \cap \text{writes}(c_1, s)\big] \neq \varnothing}{\langle c_1 \parallel c_2, (s, h)\rangle \rightsquigarrow \texttt{abort}} \quad (\text{parallel}_r)$$

$$\frac{\langle c, \sigma\rangle \rightsquigarrow \langle c', \sigma'\rangle}{\langle \texttt{resource}\ r\ \texttt{in}\ c, \sigma\rangle \rightsquigarrow \langle \texttt{resource}\ r\ \texttt{in}\ c', \sigma'\rangle} \quad (\text{resource}_1)$$

$$\frac{}{\langle \texttt{resource}\ r\ \texttt{in}\ \texttt{skip}, \sigma\rangle \rightsquigarrow \langle \texttt{skip}, \sigma\rangle} \quad (\text{resource}_2)$$

$$\frac{\langle c, \sigma\rangle \rightsquigarrow \texttt{abort}}{\langle \texttt{resource}\ r\ \texttt{in}\ c, \sigma\rangle \rightsquigarrow \texttt{abort}} \quad (\text{resource}_A)$$

$$\frac{\mathcal{B}[\![b]\!]_s = \top}{\langle \texttt{with}\ r\ \texttt{when}\ b\ \texttt{do}\ c, (s, h)\rangle \rightsquigarrow \langle \texttt{within}\ r\ \texttt{do}\ c, (s, h)\rangle} \quad (\text{with})$$

$$\frac{\langle c, \sigma\rangle \rightsquigarrow \langle c', \sigma'\rangle}{\langle \texttt{within}\ r\ \texttt{do}\ c, \sigma\rangle \rightsquigarrow \langle \texttt{within}\ r\ \texttt{do}\ c', \sigma'\rangle} \quad (\text{within}_1)$$

$$\frac{}{\langle \texttt{within}\ r\ \texttt{do}\ \texttt{skip}, \sigma\rangle \rightsquigarrow \langle \texttt{skip}, \sigma\rangle} \quad (\text{within}_2)$$

$$\frac{\langle c, \sigma\rangle \rightsquigarrow \texttt{abort}}{\langle \texttt{within}\ r\ \texttt{do}\ c, \sigma\rangle \rightsquigarrow \texttt{abort}} \quad (\text{within}_A)$$

Figure 2.15: Operational semantic rules

A resource is referenced by its name, and can either be owned by a program inside a `within` construct or available to be acquired through the `with` command when a given condition is met. Additionally, because we are interested in preventing programs which have data races – where one program would modify a variable or memory address some other is using, or vice-versa – a rule to handle such situations is added. The more cautious reader will note that the functions *locked*, *accesses* and *writes* are not defined in this document. Informally, they respectively refer to the set of resources a program owns, and the sets of variables and memory addresses a program uses and assigns – their definition being given in the work of Vafeiadis [Vaf11].

Note that since states have the same components, assertions remain unchanged – we already have a way of reasoning about stores and heaps. However, because several concurrent programs may now modify the same data structures, one should be able to express what holds on this shared state as well. Therefore, concurrent separation logic extends Hoare triples with a fourth argument – an invariant which describes what holds on the shared state. Validity of a quadruple is then defined as follows:

Validity. A quadruple $\Gamma \vDash \{p\}c\{q\}$ is true when any execution of command c starting in a state satisfying assertion p does not fault nor have data races, maintains shared resource invariant Γ and if it terminates results in a state which satisfies assertion q. Formally, it can be given by the following inductive definition:

$$\Gamma \vDash \{p\}c\{q\} \triangleq \forall_{n\in\mathbb{N}}\forall_{s\in \mathit{Store}}\forall_{h\in \mathit{Heap}}\, \mathcal{P}[\![p]\!]_{(s,h)} \Rightarrow \mathsf{safe}_n(c,s,h,\Gamma,q)$$

$$\begin{aligned}
&\mathsf{safe}_0(c,s,h,\Gamma,q) \triangleq \top\\
&\mathsf{safe}_{n+1}(c,s,h,\Gamma,q) \triangleq (c = \texttt{skip}) \Rightarrow \mathcal{P}[\![q]\!]_{(s,h)} \land\\
&\quad \forall_{h_F\in \mathit{Heap}}\, \langle c,(s,h\uplus h_F)\rangle \not\rightsquigarrow \texttt{abort} \land\\
&\quad \mathsf{accesses}(c,s) \subseteq \mathsf{dom}(h) \land\\
&\quad \forall_{c'\in C}\forall_{h_\Gamma,h_F,h'\in \mathit{Heap}}\forall_{s'\in \mathit{Store}}\, \mathcal{P}[\![\circledast_{r\in \mathrm{locked}(c')\setminus \mathrm{locked}(c)}\,\Gamma(r)]\!]_{(s,h_\Gamma)} \land\\
&\qquad \langle c,(s,h\uplus h_\Gamma\uplus h_F)\rangle \rightsquigarrow \langle c',(s',h')\rangle \Rightarrow\\
&\qquad\quad \exists_{h'',h'_\Gamma\in \mathit{Heap}}\, h' = h''\uplus h'_\Gamma\uplus h_F \land \mathcal{P}[\![\circledast_{r\in \mathrm{locked}(c)\setminus \mathrm{locked}(c')}\,\Gamma(r)]\!]_{(s',h'_\Gamma)} \land\\
&\qquad\qquad \mathsf{safe}_n(c',s',h'',\Gamma,q)
\end{aligned}$$

Although the above definition may seem quite complex, that is because it encompasses the frame property as well. On the inductive step it is mandatory that program c, when extended with a heap satisfying the shared invariant and a frame which remains unaltered during execution, does not fault and results in a state preserving the invariant and keeping the frame unchanged. Again, for the sake of brevity, we omit the definition of Γ and $\circledast$ and forward the more interested reader to the work of Vafeiadis [Vaf11]. Informally, Γ is a set of resource names and assertions describing what holds on each piece of the heap protected by those resources, and $\circledast$ stands for the separation of every assertion in a set.

Finally, syntax oriented inference rules are presented in Figure 2.16. Note that although extending previous inference rules, the ones presented in this section have one additional argument – the shared resource invariant. Like other similar cases, older rules can easily be adapted to deal with this extra argument – leaving invariant Γ unchanged. The `resource` rule states that a program which respects an invariant including property f associated to resource name r, when put inside a `resource` construct for r and extended with a heap satisfying f will finish with an additional heap – possibly different – satisfying f. Conversely, the `with` rule states that a program starting when a given condition is met and with an additional heap satisfying f, if when finished reestablishes f then it can be put inside a `with` construct for any given resource name associated to f. Besides that, we would like to point out the side condition of the parallel composition rule. In order to reason about concurrent programs, the use each thread makes of the shared state must be pairwise disjoint – indeed, a severe restriction.

$$\frac{\Gamma \vdash \{p_1\}c_1\{q_1\} \quad \Gamma \vdash \{p_2\}c_2\{q_2\}}{\Gamma \vdash \{p_1 * p_2\}c_1 \parallel c_2\{q_1 * q_2\}} \quad \text{(parallel)}$$

$$\begin{array}{ll}\text{if} & \mathrm{fv}(\Gamma, p_1, c_1, q_1) \cap \mathrm{writes}(c_2) = \varnothing \\ \text{and} & \mathrm{fv}(\Gamma, p_2, c_2, q_2) \cap \mathrm{writes}(c_1) = \varnothing\end{array}$$

$$\frac{\Gamma, r : f \vdash \{p\}c\{q\}}{\Gamma \vdash \{p * f\}\texttt{resource}\ r\ \texttt{in}\ c\{q * f\}} \quad \text{(resource)}$$

$$\frac{\Gamma \vdash \{(p * f) \wedge b\}c\{q * f\}}{\Gamma, r : f \vdash \{p\}\texttt{with}\ r\ \texttt{when}\ b\ \texttt{do}\ c\{q\}} \quad \text{(with)}$$

Figure 2.16: Proof rules

Concluding, concurrent separation logic allows us to reason about concurrent programs with shared pointer data structures. At its core is parallel composition of programs and resource creation and acquisition, sufficient to cover most lock-based systems. Furthermore, validity is now extended to account for invariants about the shared state. Although being a significant improvement towards dealing with concurrent programs, the parallel rule provided in concurrent separation logic solely lets us reason about programs without interference. Therefore, since most concurrent programs take advantage of interference, this program logic falls short on proving correctness of many existing algorithms.

2.4 Rely-guarantee reasoning

Introduced by Cliff Jones in 1981, rely-guarantee reasoning [CJ07] allows for proving correctness of "racy" programs. Because many concurrent programs take advantage of interference among themselves, this proof system greatly increases the amount of programs that can be proven using program logics. However, since rely-guarantee is based on the Vienna development method (VDM [Jon90]) – a different formal method for the development of computer-based systems – we will not detail its formal definitions, which are briefly explained in this section.

As it deals with concurrency, the constructs present in the programming language of rely-guarantee are almost equal to those presented in the previous section. However, rely-guarantee does not allow for heap manipulation – only operating on the store. Furthermore, properties are replaced by binary predicates which depend on the previous state $\overleftarrow{\sigma}$ before program execution and the current state σ resulting from it. Regarding operational semantics, one important modification is that expression evaluation stops being atomic – as a result, expressions like $x + x$ and $2 \times x$ may not yield the same result because of interference.

Rely and guarantee are additional arguments on the specification of a program, which encode the extent to which the environment can interfere with the local program, and what changes a program is allowed to perform on the shared state. In order to reason about interference, rely-guarantee introduces the notion of stability. Informally, stability asserts that a formula must resist interference from a "well behaved" environment – one respecting the rely condition. Using stability, it is now possible for properties to be preserved in a concurrent environment which is not disjoint from the local one – allowing for proof rules which do not forbid races. Therefore, the main contribution of rely-guarantee is the addition of new elements which allow inference rules to prove correctness of concurrent programs that interfere with each other.

Concluding, rely-guarantee reasoning introduces two new relations to a program specification and breaks atomic evaluation of expressions into several steps, allowing for more interleavings. Based on a new notion of stability, it is then possible for inference rules to address concurrent programs which interfere with each other. This is important because, as mentioned before, most concurrent programs take advantage of interference. However, since rely-guarantee does not allow for memory manipulation and lacks the presence of a frame rule – making local reasoning impossible – a way of addressing these requirements is still needed.

2.5 RGSep

Resulting from Viktor Vafeiadis' thesis in 2007, `RGSep` [Vaf07] aims at combining the interference reasoning present in rely-guarantee with shared pointer data structures present in concurrent separation logic. The end result is a formalism which subsumes the previous ones it extends, effectively combining the best of both works. Since lock-free programs are heavily based on cooperation – a targeted form of interference – we are especially interested in this aspect of the logic. Because we believe a full formalization of `RGSep` falls out of the scope of this document, we will only highlight its key definitions and redirect the interested reader to the aforementioned reference for a more detailed read on the topic.

One key difference RGSep has regarding concurrent separation logic is that the first allows us to reason about change in the shared state – while the latter can only express invariants. Since this is now possible, and rely-guarantee conditions are enough to restrict access to the shared state, RGSep no longer needs resources to reason about interference – one could simply add the separation of every resource invariant to the pre and post conditions and use the identity relationship on the rely and guarantee conditions of programs. Therefore, as can be seen in Figure 2.17, programming language constructs to manipulate resources are replaced by a single command to atomically execute a program.

$$\begin{array}{lcll} C & ::= & \ldots & \text{Commands} \\ & \mid & C \parallel C & \text{(parallel)} \\ & \mid & \texttt{atomic}\, C & \text{(atomic)} \end{array}$$

Figure 2.17: Programming language syntax

Additionally, the previous transition function is refined to $\overset{r}{\underset{\lambda}{\rightsquigarrow}}$, where λ can be either a program step p or an environment execution e, and r is a rely condition – a relation between initial and final heap which every environment transition must respect so reasoning about interference is possible. Furthermore, configurations are now a disjoint pair of states, representing the local and shared states. Program steps remain as presented before, ignoring the environment's state except on atomic execution of programs – which may now acquire the shared state for itself. Therefore, we focus on the following operational semantics' rule which models an environment action:

$$\frac{(s, s') \in r \wedge \mathrm{dom}(l) \cap \mathrm{dom}(s') = \varnothing}{\langle c, (l, s)\rangle \overset{r}{\underset{\mathrm{e}}{\rightsquigarrow}} \langle c, (l, s')\rangle} \quad \text{(environment)}$$

This rule states that if an environment's step is allowed by the rely condition, then it can be performed at any time during a program's execution – resulting in a new shared state. Note that although the rely condition is promptly used to disallow environment transitions which do not respect it, the guarantee – a relation between initial and final heap which the program itself must respect so other concurrent programs can use as a rely to reason about interference – will only be enforced when validity of a specification is checked.

Because we explicitly have a shared state component in program configurations, assertions can now make use of them. Figure 2.18 presents the assertion syntax, the only difference being the addition of boxed statements which talk about the shared state. Note that because it is not possible to nest boxed statements, the previous assertions from separation logic are referred as P_{sl} – as well as any other expression with that subscript.

$$
\begin{array}{rcll}
P & ::= & & \text{Assertions} \\
 & | & P_{sl} & \text{(local)} \\
 & | & \boxed{P_{sl}} & \text{(shared)} \\
 & | & P * P & \text{(separation)} \\
 & | & P \wedge P & \text{(conjunction)} \\
 & | & \neg P & \text{(negation)}
\end{array}
$$

Figure 2.18: Assertion language syntax

Accordingly, the assertion evaluation function also has to account for this new shared state. As can be seen in Figure 2.19, former separation logic assertions are checked against the local heap while environment statements are matched against the shared state. Note that these latter ones require the local state to be empty.

$$
\begin{array}{lcll}
\mathcal{P} : P \times \mathit{State} \times \mathit{State} & \rightarrow & \{\top, \bot\} & \text{Assertion} \\
\mathcal{P}[\![\, p_{sl} \,]\!]_{(l,s)} & \triangleq & \mathcal{P}_{sl}[\![p_{sl}]\!]_l & \text{(local)} \\
\mathcal{P}[\![\, \boxed{p_{sl}} \,]\!]_{(l,s)} & \triangleq & \mathrm{dom}(l) = \varnothing \wedge \mathcal{P}_{sl}[\![p_{sl}]\!]_s & \text{(shared)} \\
\mathcal{P}[\![p_1 * p_2]\!]_{(l,s)} & \triangleq & \exists_{l_1, l_2 \in \mathit{State}}\, l = l_1 \uplus l_2 \wedge \mathcal{P}[\![p_1]\!]_{(l_1,s)} \wedge \mathcal{P}[\![p_2]\!]_{(l_2,s)} & \text{(separation)} \\
\mathcal{P}[\![p_1 \wedge p_2]\!]_{(l,s)} & \triangleq & \mathcal{P}[\![p_1]\!]_{(l,s)} \wedge \mathcal{P}[\![p_2]\!]_{(l,s)} & \text{(conjunction)} \\
\mathcal{P}[\![\neg p]\!]_{(l,s)} & \triangleq & \neg \mathcal{P}[\![p]\!]_{(l,s)} & \text{(negation)}
\end{array}
$$

Figure 2.19: Assertion semantics

Validity. Because of the addition of rely and guarantee conditions in order to reason about correctness of programs, validity now has to account for them as well. Formula $\vDash c \text{ sat } (p, r, g, q)$ – where r and g are the rely and guarantee relations – holds whenever any execution of program c starting in a state satisfying assertion p and an environment respecting r will only make changes to the shared state that ensure g and, if it terminates, results in a program state that satisfies assertion q. Formally, it can be given by the following inductive definition:

$$\begin{aligned}
&\vdash c \text{ sat } (p, r, g, q) \triangleq \forall_{l,s \in \mathit{State}}\, \mathcal{P}[\![p]\!]_{(l,s)} \Rightarrow \\
&\qquad \forall_{l',s' \in \mathit{State}}\, \langle c, (l, s)\rangle \overset{r}{\rightsquigarrow}{}^{*} \langle \mathtt{skip}, (l', s')\rangle \Rightarrow \mathcal{P}[\![q]\!]_{(l',s')} \wedge \\
&\qquad \forall_{n \in \mathbb{N}}\, \langle c, (l, s), r\rangle \text{ guarantees}_n\, g
\end{aligned}$$

$$\begin{aligned}
&\langle c, (l, s), r\rangle \text{ guarantees}_0\, g \triangleq \top \\
&\langle c, (l, s), r\rangle \text{ guarantees}_{n+1}\, g \triangleq \exists_{\theta \in \mathit{Config}}\, \langle c, (l, s)\rangle \underset{\lambda}{\overset{r}{\rightsquigarrow}} \theta \Rightarrow \\
&\qquad \exists_{c' \in C} \exists_{l',s' \in \mathit{State}}\, \theta = \langle c', (l', s')\rangle \wedge \\
&\qquad\quad \langle c', (l', s')\rangle \text{ guarantees}_n\, g \wedge \\
&\qquad\quad (\lambda = p) \Rightarrow (s, s') \in g
\end{aligned}$$

Inference rules that have changed are presented in Figure 2.20. As intended, the parallel rule now allows composition of programs which may share part of the state – because separation only requires that local state be disjoint – although the whole construct is loosened to guarantee the changes made by both programs. Atomic execution of programs requires *precision* – an additional property on both pre and post conditions that ensures there are no two different heaps satisfying a given formula, which is useful for uniquely identifying sub-heaps – and the corresponding rule states that a program may use the shared state as long as the change to the resulting heap is allowed by the guarantee.

$$\frac{\vdash c_1 \text{ sat } (p_1, r, g_1, q_1) \qquad \vdash c_2 \text{ sat } (p_2, r, g_2, q_2)}{\vdash c_1 \parallel c_2 \text{ sat } (p_1 * p_2, r, g_1 \cup g_2, q_1 * q_2)} \quad \text{(parallel)}$$

$$\frac{p, q \text{ precise} \qquad \vdash c \text{ sat } (p * p', \varnothing, \varnothing, q * q') \qquad (p, q) \subseteq g}{\vdash \mathtt{atomic}\; c \text{ sat } (\boxed{p * f} * p', \varnothing, g, \boxed{q * f} * q')} \quad \text{(atomic)}$$

Figure 2.20: Proof rules

Concluding, RGSep combines the expressiveness of rely-guarantee reasoning with concurrent separation logic, resulting in a formalism which subsumes the two – its main advantage over rely-guarantee being a set of efficient rules which allow for local reasoning. Although being the best known way of reasoning about lock-based concurrent systems, as our goal is to prove correctness of lock-free programs, this proof system does not address the problem we're trying to tackle. The main contribution of RGSep to our work is then the adaptation of rely-guarantee reasoning to local rules, and the introduction of assertions which are capable of expressing what holds on the shared state.

3

Proof system for lock-free concurrency

In the previous chapter, several formalisms with different levels of expressiveness for reasoning about programs were presented. Hoare-logic pioneered program logics, separation logic extended it to deal with memory addresses, concurrent separation logic introduced parallel composition and shared resources, rely-guarantee reasoning allowed for dealing with interference and `RGSep` combined these last two to model change and relax side conditions while allowing for local proofs.

Similarly, we define a program logic with the goal of proving correctness of lock-free programs based on `LL/SC` primitives. As we hope the reader agrees, the end result is a proof system which clearly captures the essence of `LL/SC` through the usage of already existing and well known concepts.

3.1 Programs

Firstly, we present the syntax and operational semantics of the programming language we consider herein. Given we are interested in imperative programs which manipulate shared pointer data structures, constructs present in separation logic will be maintained. Although also being interested in concurrent programs, since both concurrent separation logic and `RGSep` use lock mechanisms to restrict access to the shared state, only parallel composition is preserved to deal with concurrency. Finally, as the goal is to model `LL/SC` primitives, new constructs are added to the language.

3.1.1 Program Syntax

Again, to specify which programs are part of our target programming language, an EBNF grammar is presented in Figure 3.1. Commands generated by C_b are the building blocks of our language, and include the usual store and heap manipulation constructs as well as two new ones to deal with concurrency – the `LL/SC` primitives. C_s encompasses the usual imperative sequential commands such as choice, looping and sequential composition. In order to allow for concurrency, parallel composition of previous commands is created with C_c .

$$
\begin{array}{lcll}
C_b & ::= & & \text{Basic Commands} \\
 & | & \texttt{skip} & \text{(skip)} \\
 & | & V := E & \text{(assignment)} \\
 & | & V := \texttt{alloc}(E, \ldots, E) & \text{(allocate)} \\
 & | & V := [V] & \text{(load)} \\
 & | & V := \texttt{LL}(V) & \text{(load-link)} \\
 & | & [V] := V & \text{(store)} \\
 & | & V := \texttt{SC}(V, V) & \text{(store-conditional)} \\
 & | & \texttt{disp}(V) & \text{(dispose)} \\
 & & & \\
C_s & ::= & & \text{Sequential Commands} \\
 & | & C_b & \text{(basic command)} \\
 & | & \texttt{if}\ B\ \texttt{then}\ C_s\ \texttt{else}\ C_s & \text{(conditional)} \\
 & | & \texttt{while}\ B\ \texttt{do}\ C_s & \text{(while)} \\
 & | & C_s ; C_s & \text{(sequential composition)} \\
 & & & \\
C_c & ::= & & \text{Concurrent Commands} \\
 & | & C_s & \text{(sequential command)} \\
 & | & C_c \parallel C_s & \text{(parallel composition)} \\
 & & & \\
V & ::= & x \,|\, y \,|\, z \,|\, \cdots & \text{Variables} \\
E & ::= & V \,|\, \mathbb{Z} \,|\, E + E \,|\, E - E \,|\, E \times E & \text{Arithmetic expressions} \\
B & ::= & \texttt{true} \,|\, \texttt{false} \,|\, E = E \,|\, E \leq E & \text{Boolean expressions} \\
 & | & \neg B \,|\, B \wedge B \,|\, B \vee B &
\end{array}
$$

Figure 3.1: Programming language syntax

We would like to point out, as the more cautious reader probably has already noticed, that this grammar slightly differs from the ones presented previously. One of the key principles needed to keep this kind of work manageable is minimality – without it, defining all of the following would be significantly more complex with little added benefit. Since we envisioned variables to be the required resources to access the shared state, most expressions have been replaced by variables to help unify forthcoming developments. Although it might take longer to encode programs, because arbitrary expressions are allowed on the right-hand side of variable assignment and memory address allocation, this does not limit the expressiveness of the language. Additionally, as will be detailed ahead, only top-level parallel composition is allowed – briefly, since truly local information is needed to model LL/SC, it makes definitions that manipulate this new component much simpler.

On a final note, with the program syntax presented in Figure 3.1 the developer has a way to encode his programs in the proof system being described. The way to model LL/SC primitives is pretty straightforward, as a new construct is added for each of these primitives.

3.1.2 Operational Semantics

Similarly to previous works, a structural operational semantics is used to model program execution – presented in Figures 3.3 and 3.4. Figure 3.2 introduces the additional components needed to define this operational semantics – threaded commands that represent programs running at a thread with a given identifier, and configurations to be manipulated by the transition function in order to model program execution.

In order to execute LL/SC, since some concurrent programs might have an updated view of an address' content while others might not, a way of knowing which memory addresses have not changed since a program last load-linked them is needed. Although assertion semantics in RGSep uses an additional parameter in order to separate shared memory from local memory, because lock-free programs do away with this separation of resources and allow for truly concurrent accesses, that is not enough to solve this problem. Because threads are a well known concept in concurrent programming, used to separate concurrent programs but not system resources, we find them to be a good analogy to our scenario. Addressing the requirement presented above, a component that maps threads into a set of memory locations which remained unchanged since the last load-link is added to program configurations. Although there are several equally good ways to encode that information, we opted for this one as it fits forthcoming definitions better – on validity, we are interested in specifying which memory addresses a thread can start and end with an updated view of.

$$\begin{array}{llll}
C_t & ::= & & \text{Threaded Command} \\
 & | & C_c@\mathit{ThreadId} & \text{(Single thread)} \\
 & | & C_t \parallel C_c@\mathit{ThreadId} & \text{(Multiple threads)}
\end{array}$$

$$\mathit{Config} \triangleq C_t \times \left[\mathit{ThreadId} \rightharpoonup \mathfrak{P}\left(\mathit{Address}\right)\right] \times \left(\mathit{Store} \times \mathit{Heap}\right) \cup \{\texttt{abort}\}$$

Figure 3.2: Program states and data

$$\rightsquigarrow: \mathit{Config} \times \mathit{Config} \qquad \text{Program transition}$$

$$\frac{s(y) = a \land a \in \mathrm{dom}(h) \land l(i) = l_i}{\begin{array}{l}\langle [x := \mathtt{LL}(y)]@i, l, (s, h)\rangle \rightsquigarrow \\ \qquad \langle \mathtt{skip}@i, l[i \mapsto l_i \cup \{a\}], (s[x \mapsto h(a)], h)\rangle\end{array}} \qquad \text{(load-link)}$$

$$\frac{s(y) \notin \mathrm{dom}(h)}{\langle [x := \mathtt{LL}(y)]@i, l, (s, h)\rangle \rightsquigarrow \mathtt{abort}} \qquad \text{(load-link}_A\text{)}$$

$$\frac{s(z) = a \land a \in \mathrm{dom}(h) \land a \in l(i)}{\begin{array}{l}\langle [x := \mathtt{SC}(y, z)]@i, l, (s, h)\rangle \rightsquigarrow \\ \qquad \langle \mathtt{skip}@i, l \downarrow a, (s[x \mapsto 1], h[a \mapsto s(y)])\rangle\end{array}} \qquad \text{(store-conditional}_T\text{)}$$

$$\frac{s(z) = a \land a \in \mathrm{dom}(h) \land a \notin l(i)}{\langle [x := \mathtt{SC}(y, z)]@i, l, (s, h)\rangle \rightsquigarrow \langle \mathtt{skip}@i, l, (s[x \mapsto 0], h)\rangle} \qquad \text{(store-conditional}_F\text{)}$$

$$\frac{s(z) \notin \mathrm{dom}(h)}{\langle [x := \mathtt{SC}(y, z)]@i, l, (s, h)\rangle \rightsquigarrow \mathtt{abort}} \qquad \text{(store-conditional}_A\text{)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![e_1]\!]_s = v_1 \land \ldots \land \mathcal{E}[\![e_n]\!]_s = v_n \land \\ \{a, \ldots, a + n - 1\} \subseteq \text{Address} \setminus \mathrm{dom}(h)\end{array}}{\begin{array}{l}\langle [x := \mathtt{alloc}(e_1, \ldots, e_n)]@i, l, (s, h)\rangle \rightsquigarrow \\ \qquad \langle \mathtt{skip}@i, l, (s[x \mapsto a], h[a \mapsto v_1, \ldots, a + n - 1 \mapsto v_n])\rangle\end{array}} \qquad \text{(allocate)}$$

$$\frac{}{\langle (x := e)@i, l, (s, h)\rangle \rightsquigarrow \langle \mathtt{skip}@i, l, (s[x \mapsto \mathcal{E}[\![e]\!]_s], h)\rangle} \qquad \text{(assign)}$$

$$\frac{s(y) = a \land a \in \mathrm{dom}(h)}{\langle (x := [y])@i, l, (s, h)\rangle \rightsquigarrow \langle \mathtt{skip}@i, l, (s[x \mapsto h(a)], h)\rangle} \qquad \text{(load)}$$

$$\frac{s(y) \notin \mathrm{dom}(h)}{\langle (x := [y])@i, l, (s, h)\rangle \rightsquigarrow \mathtt{abort}} \qquad \text{(load}_A\text{)}$$

$$\frac{s(x) = a \land a \in \mathrm{dom}(h)}{\langle ([x] := y)@i, l, (s, h)\rangle \rightsquigarrow \langle \mathtt{skip}@i, l \downarrow a, (s, h[a \mapsto s(y)])\rangle} \qquad \text{(store)}$$

$$\frac{s(x) \notin \mathrm{dom}(h)}{\langle ([x] := y)@i, l, (s, h)\rangle \rightsquigarrow \mathtt{abort}} \qquad \text{(store}_A\text{)}$$

$$\frac{s(x) = a \land a \in \mathrm{dom}(h)}{\langle \mathtt{disp}(x)@i, l, (s, h)\rangle \rightsquigarrow \langle \mathtt{skip}@i, l \downarrow a, (s, h\rceil(\mathrm{dom}(h) \setminus \{a\}))\rangle} \qquad \text{(free)}$$

Figure 3.3: Operational semantics

$$\frac{s(x) \notin \mathrm{dom}(h)}{\langle \mathtt{disp}(x)@i, l, (s,h)\rangle \rightsquigarrow \mathtt{abort}} \quad (\mathrm{free}_A)$$

$$\frac{\mathcal{B}[\![b]\!]_s = \top}{\langle (\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2)@i, l, (s,h)\rangle \rightsquigarrow \langle c_1@i, l, (s,h)\rangle} \quad (\text{if–then–else}_T)$$

$$\frac{\mathcal{B}[\![b]\!]_s = \bot}{\langle (\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2)@i, l, (s,h)\rangle \rightsquigarrow \langle c_2@i, l, (s,h)\rangle} \quad (\text{if–then–else}_F)$$

$$\frac{}{\begin{array}{l}\langle (\mathtt{while}\ b\ \mathtt{do}\ c)@i, l, \sigma\rangle \rightsquigarrow \\ \qquad \langle (\mathtt{if}\ b\ \mathtt{then}\ c; \mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{else}\ \mathtt{skip})@i, l, \sigma\rangle\end{array}} \quad (\mathrm{while})$$

$$\frac{\langle c_1@i, l, \sigma\rangle \rightsquigarrow \langle c_1'@i, l', \sigma'\rangle}{\langle (c_1; c_2)@i, l, \sigma\rangle \rightsquigarrow \langle (c_1'; c_2)@i, l', \sigma'\rangle} \quad (\mathrm{sequential}_1)$$

$$\frac{}{\langle (\mathtt{skip}; c_2)@i, l, \sigma\rangle \rightsquigarrow \langle c_2@i, l, \sigma\rangle} \quad (\mathrm{sequential}_2)$$

$$\frac{\langle c_1@i, l, \sigma\rangle \rightsquigarrow \mathtt{abort}}{\langle (c_1; c_2)@i, l, \sigma\rangle \rightsquigarrow \mathtt{abort}} \quad (\mathrm{sequential}_A)$$

$$\frac{i' \in \mathit{ThreadId} \setminus \mathrm{dom}(l)}{\langle (c_1 \parallel c_2)@i, l, \sigma\rangle \rightsquigarrow \langle c_1@i \parallel c_2@i', l[i' \mapsto \varnothing], \sigma\rangle} \quad (\mathrm{fork})$$

$$\frac{\langle t_1, l, \sigma\rangle \rightsquigarrow \langle t_1', l', \sigma'\rangle}{\langle t_1 \parallel t_2, l, \sigma\rangle \rightsquigarrow \langle t_1' \parallel t_2, l', \sigma'\rangle} \quad (\mathrm{parallel}_1)$$

$$\frac{\langle t_2, l, \sigma\rangle \rightsquigarrow \langle t_2', l', \sigma'\rangle}{\langle t_1 \parallel t_2, l, \sigma\rangle \rightsquigarrow \langle t_1 \parallel t_2', l', \sigma'\rangle} \quad (\mathrm{parallel}_2)$$

$$\frac{\langle t_1, l, \sigma\rangle \rightsquigarrow \mathtt{abort}}{\langle t_1 \parallel t_2, l, \sigma\rangle \rightsquigarrow \mathtt{abort}} \quad (\mathrm{parallel}_{1A})$$

$$\frac{\langle t_2, l, \sigma\rangle \rightsquigarrow \mathtt{abort}}{\langle t_1 \parallel t_2, l, \sigma\rangle \rightsquigarrow \mathtt{abort}} \quad (\mathrm{parallel}_{2A})$$

$$\frac{\begin{array}{c}\mathrm{assigns}(t_1) \cap \big[\mathrm{assigns}(t_2) \cup \mathrm{reads}(t_2)\big] \neq \varnothing \lor \\ \mathrm{assigns}(t_2) \cap \big[\mathrm{assigns}(t_1) \cup \mathrm{reads}(t_1)\big] \neq \varnothing\end{array}}{\langle t_1 \parallel t_2, l, \sigma\rangle \rightsquigarrow \mathtt{abort}} \quad (\text{parallel race})$$

$$\frac{}{\langle \mathtt{skip}@i \parallel \mathtt{skip}@i', l, \sigma\rangle \rightsquigarrow \langle \mathtt{skip}@i, l\rceil(\mathrm{dom}(l) \setminus \{i'\}), \sigma\rangle} \quad (\mathrm{join})$$

Figure 3.4: Operational semantics (cont.)

Giving a precise account of program execution, Figures 3.3 and 3.4 detail the operational semantics' transition function. Since only two new base constructs were added to the language, it is expectable for most rules to be similar to their concurrent separation logic counterparts – besides accounting for the additional components added to configurations, that is indeed so. The main differences regarding usual constructs are that commands which change the heap must now invalidate any thread's vision of modified addresses, and parallel composition must create a new thread with its own set of up-to-date heap locations and merge threads when they finish executing. Additionally, since the goal is to have a way of reasoning about heap data races with the `LL`/`SC` primitives, a program no longer aborts when a thread writes into a location which some other is accessing – this kind of situation does not pose a problem to program execution and should only be handled when reasoning about it.

Targeting `LL` and `SC`, five new transitions are present. The first transition rule for `LL` command (*load-link*) states that a thread which load-links a given variable into some other will assign the store value of the latter to the memory content of the location given by the former – adding that location into the set of addresses which the thread has an up-to-date view of. The first two transitions for `SC` (*store-conditional*$_T$ and *store-conditional*$_F$) state that a thread which conditionally stores y into memory address z and saves the result in x will succeed and set the content of memory address given by z to the store value of y – removing that location from every thread's up-to-date set of addresses – as well as assigning the store value of x to 1, or fail and solely assign the store value of x to 0 if the thread does not have an up-to-date view of location z. Remaining rules (*load-link*$_A$ and *store-conditional*$_A$) simply state that either a `LL` or `SC` involving a memory address which is not allocated will make the program abort.

As before, the definition of some auxiliary operators is in order. While $f[x \mapsto v]$, $\mathrm{dom}(f)$ and $\rceil$ were presented on the previous chapter, a few new ones are needed here. Function $\mathrm{cdom}(f)$ refers to the codomain of function f, and $l \downarrow a$ to the result of removing location a from every set of up-to-date memory addresses each thread has – kept in l. Although the first is well-known – and as such we will skip its definition – the second is a custom operator which fits our particular needs and can be defined as follows:

$$\begin{array}{rcl} \downarrow & : & \Big([\mathit{ThreadId} \rightharpoonup \mathfrak{P}\,(\mathit{Address})] \times \mathit{Address}\Big) \to [\mathit{ThreadId} \rightharpoonup \mathfrak{P}\,(\mathit{Address})] \\ l \downarrow a & \triangleq & \{t \mapsto (l_t \setminus \{a\}) \mid t \mapsto l_t \in l\} \end{array}$$

Because we still care about store data races, since no mechanism exists in our programming language to reason about them, some syntax-directed operators are also needed. Such is the case of $\mathrm{assigns}(c_t)$ and $\mathrm{reads}(c_t)$ functions – detailed in Figure 3.5 – which informally refer to the set of variables a thread at a given execution step can respectively assign or read. Note that these differ from $\mathrm{writes}(c)$ and $\mathrm{fv}(c)$ firstly because their type is different, but more importantly because the latter account for the execution of a whole program instead of a single step.

$$
\begin{array}{lcl}
\text{assigns} : C_t \to V & & \\
\text{assigns}(\texttt{skip}@i) & \triangleq & \varnothing \\
\text{assigns}((x := e)@i) & \triangleq & \{x\} \\
\text{assigns}([x := \texttt{alloc}(e_1, \ldots, e_n)]@i) & \triangleq & \{x\} \\
\text{assigns}((x := [y])@i) & \triangleq & \{x\} \\
\text{assigns}([x := \texttt{LL}(y)]@i) & \triangleq & \{x\} \\
\text{assigns}(([x] := e)@i) & \triangleq & \varnothing \\
\text{assigns}([x := \texttt{SC}(y, z)]@i) & \triangleq & \{x\} \\
\text{assigns}(\texttt{disp}(x)@i) & \triangleq & \varnothing \\
\text{assigns}((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)@i) & \triangleq & \varnothing \\
\text{assigns}((\texttt{while}\ b\ \texttt{do}\ c)@i) & \triangleq & \varnothing \\
\text{assigns}((c_1; c_2)@i) & \triangleq & \text{assigns}(c_1) \\
\text{assigns}((c_1 \parallel c_2)@i) & \triangleq & \varnothing \\
\text{assigns}(t_1 \parallel t_2) & \triangleq & \text{assigns}(t_1) \cup \text{assigns}(t_2)
\end{array}
$$

$$
\begin{array}{lcl}
\text{reads} : C_t \to V & & \\
\text{reads}(\texttt{skip}@i) & \triangleq & \varnothing \\
\text{reads}((x := e)@i) & \triangleq & \text{fv}(e) \\
\text{reads}([x := \texttt{alloc}(e_1, \ldots, e_n)]@i) & \triangleq & \text{fv}(e_1) \cup \ldots \cup \text{fv}(e_n) \\
\text{reads}((x := [y])@i) & \triangleq & \{y\} \\
\text{reads}([x := \texttt{LL}(y)]@i) & \triangleq & \{y\} \\
\text{reads}(([x] := e)@i) & \triangleq & \{x\} \cup \text{fv}(e) \\
\text{reads}([x := \texttt{SC}(y, z)]@i) & \triangleq & \{y, z\} \\
\text{reads}(\texttt{disp}(x)@i) & \triangleq & \{x\} \\
\text{reads}((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)@i) & \triangleq & \text{fv}(b) \\
\text{reads}((\texttt{while}\ b\ \texttt{do}\ c)@i) & \triangleq & \text{fv}(b) \\
\text{reads}((c_1; c_2)@i) & \triangleq & \text{reads}(c_1) \\
\text{reads}((c_1 \parallel c_2)@i) & \triangleq & \varnothing \\
\text{reads}(t_1 \parallel t_2) & \triangleq & \text{reads}(t_1) \cup \text{reads}(t_2)
\end{array}
$$

Figure 3.5: Assigns and reads functions

Equipped with the above operational semantics, it is now possible to have a precise account of what programs do. Concretely, given an initial configuration we can compute the states a command might lead to – by iterating the transition function. Although this was already possible for the subset of our language that did not deal with `LL`/`SC`, the new components described in this section were needed in order to deal with programs which use such primitives.

3.2 Assertions

After having a formal way of describing program execution, the next step in defining a proof system is to create a way of reasoning about those executions. Assertions are one of the usual components when reasoning about programs, stating a property about program states which might – or might not – hold on a particular state.

Since our proof systems deals with programs which manipulate store and heap data, it should be no surprise that the following developments are based on those of separation logic. However, because of the optimistic nature of lock-free primitives `LL/SC` and since we do not have a separate kind of resources those commands manipulate, neither concurrent separation logic nor `RGSep` are directly applicable here.

3.2.1 Assertion Syntax

Figure 3.6 presents the grammar of assertions, needed in order to write properties. Firstly a production rule for separation logic assertions is presented, meaning any separation logic property is available for reasoning about programs in our proof system. Following it comes a rule for generating the new properties we support, the main addition being shared assertions – written within a box and preceded by a superscript variable. Informally, shared assertions state that if a memory address given by a variable stays up-to-date then a given property about its value holds. Note that because it is only possible to create shared assertions with separation logic properties, it is not possible to nest them – which is intended, due to the meaning of such assertions.

$$
\begin{array}{rcll}
P_{sl} & ::= & & \text{Separation Logic Assertions} \\
 & | & B & \text{(boolean expression)} \\
 & | & P_{sl} \wedge P_{sl} & \text{(conjunction)} \\
 & | & \neg P_{sl} & \text{(negation)} \\
 & | & \mathtt{emp} & \text{(empty)} \\
 & | & P_{sl} * P_{sl} & \text{(local separation)} \\
 & | & E_1 \mapsto E_2 & \text{(points to)} \\
 & | & \exists_V \, P_{sl} & \text{(existential quantification)} \\
 \\
e \mapsto - & \triangleq & \exists_x \, e \mapsto x \,, \text{where } x \notin \mathrm{fv}(e) & \text{(anonymous)} \\
e \mapsto e_1, \ldots, e_n & \triangleq & e \mapsto e_1 * \ldots * e + n - 1 \mapsto e_n & \text{(multi-valued)} \\
 \\
P & ::= & & \text{Assertions} \\
 & | & P_{sl} & \text{(local assertion)} \\
 & | & {}^{V}\boxed{P_{sl}} & \text{(shared assertion)} \\
 & | & P * P & \text{(separation)} \\
 & | & \exists_V \, P & \text{(existential quantification)}
\end{array}
$$

Figure 3.6: Assertion language syntax

Before moving on to assertion semantics, one last syntactic definition is needed in order to reason about programs. Going back to concurrent separation logic, there was a component in assertions which dealt with the shared part of the heap – the resource context Γ. In a similar fashion, we will use resource contexts to manage interference and have a way of reasoning about shared data. Our resource contexts are then given by the following definition:

$$
\begin{aligned}
&\mathit{Resource} \subset \mathfrak{P}(P_{sl}) \\
&\{\} \in \mathit{Resource} \\
&\Gamma \in \mathit{Resource} \Rightarrow \Gamma \uplus \{\exists_v\, x \mapsto v * p\} \in \mathit{Resource} \\
&\\
&\mathrm{res} : \mathit{Resource} \to V \\
&\mathrm{res}(\{\}) \triangleq \varnothing \\
&\mathrm{res}(\Gamma \uplus \{\exists_v\, x \mapsto v * p\}) \triangleq \mathrm{res}(\Gamma) \cup \{x\}
\end{aligned}
$$

Firstly, note that resource contexts are sets of assertions – each one stating what holds on a part of the heap "controlled" by a given variable. Secondly, because each member of a resource context is an assertion, what holds on the shared state is clear and needs no additional definitions – we only need to specify which part of the state those properties refer to.

Given the above assertion syntax, it is now possible to write properties about what holds on a given state of program execution. Assertions from separation logic are kept in order to have a way of reasoning about both store and heap data, and resource contexts from concurrent separation logic are adapted to our scenario where variables control the access to shared resources. Additionally, boxed assertions were added to express the optimistic nature of `LL/SC` lock-free primitives – stating that if a given memory address has not been changed, then a property holds.

3.2.2 Validity

Having a way of writing properties, we now focus on their meaning and reasoning about execution of whole programs – instead of a single program state. Figure 3.7, extending those of Figures 2.6 and 2.12 here renamed to $\mathcal{P}_{sl}$, presents the semantics of assertions, as well as of resource contexts. Before focusing on concrete properties, let us first go through the arguments needed to use the $\mathcal{P}$ predicate – which defines when a given assertion holds. Firstly a set of up-to-date shared memory locations is needed in order to check whether properties associated to those addresses should still be true. Secondly, a function which maps variables into the heap partition they control is provided in order to know where to match shared assertions against. Lastly, as usual, both store and local heap are used to reason about data exclusively manipulated by a thread – although the store is actually shared, disjoint usage will be required in order to prevent races.

$$
\begin{array}{lll}
\mathcal{P} : P \times \mathfrak{P}\,(\mathit{Address}) \times (V \rightharpoonup \mathit{Heap}) \times \mathit{State} \rightarrow \{\top, \bot\} & & \\
\mathcal{P}[\![p_{sl}]\!]_{(\delta,h_\Gamma,s,h_l)} & \triangleq & \mathcal{P}_{sl}[\![p_{sl}]\!]_{(s,h_l)} \\
\mathcal{P}[\![{}^{x}\boxed{p_{sl}}]\!]_{(\delta,h_\Gamma,s,h_l)} & \triangleq & h_l = \varnothing \wedge h_\Gamma = h'_\Gamma \uplus \{x \mapsto h_x\} \wedge \Big(s(x) \in \delta \Rightarrow \mathcal{P}_{sl}[\![p_{sl}]\!]_{(s,h_x)}\Big) \\
\mathcal{P}[\![p_1 * p_2]\!]_{(\delta,h_\Gamma,s,h_l)} & \triangleq & h_l = h_{l_1} \uplus h_{l_2} \wedge h_\Gamma = h_{\Gamma_1} \uplus h_{\Gamma_2} \wedge \mathcal{P}[\![p_1]\!]_{(\delta,h_{\Gamma_1},s,h_{l_2})} \wedge \mathcal{P}[\![p_2]\!]_{(\delta,h_{\Gamma_2},s,h_{l_2})} \\
\mathcal{P}[\![\exists_x\, p]\!]_{(\delta,h_\Gamma,s,h_l)} & \triangleq & \exists_v\, \mathcal{P}[\![p]\!]_{(\delta,h_\Gamma,s[x\mapsto v],h_l)}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{R} : \mathit{Resource} \times (V \rightharpoonup \mathit{Heap}) \times \mathit{Store} \rightarrow \{\top, \bot\} & & \\
\mathcal{R}[\![\{\}]\!]_{(h,s)} & \triangleq & h = \varnothing \\
\mathcal{R}[\![\Gamma \uplus \{\exists_v\, x \mapsto v * p\}]\!]_{(h,s)} & \triangleq & h = h_\Gamma \uplus \{x \mapsto h_x\} \wedge \mathcal{R}[\![\Gamma]\!]_{(h_\Gamma,s)} \wedge \mathcal{P}_{SL}[\![\exists_v x \mapsto v * p]\!]_{(s,h_x)}
\end{array}
$$

Figure 3.7: Assertion semantics

Moving into assertion evaluation, as in `RGSep`, separation logic properties are handled as they were originally defined and matched against the store and local heap. In a similar fashion top level existential quantification also behaves as previously, verifying if any store value which makes the property true exists for the given variable. Shared assertions force the local heap to be empty – so that the heap associated to properties that are local cannot be extended by this new construct which only deals with shared resources – check if a heap associated to the superscript variable exists and, if the memory address given by that variable is up-to-date, verify whether the property holds on that heap. Separation, redefined to handle boxed assertions, now also splits shared heaps in addition to the local one – preserving compositionality.

Before going into resource contexts, one detail the more careful reader will have noticed is that the grammar presented above is ambiguous – both separation and existential quantification of local statements can be generated by either P or P_{sl}. However, ambiguity is not a problem here because $\mathcal{P}$ and $\mathcal{P}_{sl}$ agree on those statements. On evaluating separation, since shared heaps are not accessed, we can ignore the first split which is made – resulting on the original definition. As for existential quantification, since the predicate is matched against a store with the variable associated to an existentially quantified value and $\mathcal{P}_{sl}$ evaluated on that store and local heap, the result is also the same.

Resource contexts, also covered by Figure 3.7, are sets of properties with a particular structure used to state what holds on the shared part of the heap. Unlike shared assertions, which may not hold if an address has been changed by some other thread, resource contexts are invariants which state what always holds on the shared state and all threads must abide to. Additionally, since we enforce an empty heap when evaluating the empty context, the whole shared heap must be accounted for – unlike shared assertions, which allow us to skip specifying what holds on parts of the shared state we are not interested in at the moment and are given by the resource context. Finally, the inductive step simply checks that a property holds on the shared heap partition controlled by a given variable.

Previous developments concerned themselves with single program states, or arbitrary resource contexts. Since our goal is to reason about execution of whole programs, such definitions – by themselves – still do not provide us with enough means to do it. In order to put all these pieces together Figure 3.8 introduces validity – a way of checking whether the program meets a given specification.

$\Gamma \vDash \delta_1\{p\}c\{q\}\delta_2$ if and only if for all $l, i, l_i, s, h = h_s \uplus h_l$ and $h_\Gamma \in \text{partitions}(h_s)$
 whenever $\mathcal{R}[\![\Gamma]\!]_{(h_\Gamma,s)}, \mathcal{P}[\![p]\!]_{(l_i,h_\Gamma,s,h_l)}$ and $l_i \subseteq \text{eval}_\mathcal{E}(\delta_1, s)$ then
  not $\langle c@i, l[i \mapsto l_i], (s,h)\rangle \rightsquigarrow^* \texttt{abort}$ and
  if $\langle c@i, l[i \mapsto l_i], (s,h)\rangle \rightsquigarrow^* \langle \texttt{skip}@i, l', (s',h')\rangle$ then there exists h'_s, h'_l and h'_Γ
   such that $h' = h'_s \uplus h'_l, h'_\Gamma \in \text{partitions}(h'_s)$ and
    $\mathcal{R}[\![\Gamma]\!]_{(h'_\Gamma,s')}, \mathcal{P}[\![q]\!]_{(l'(i),h'_\Gamma,s',h'_l)}$ and $l'(i) \subseteq \text{eval}_\mathcal{E}(\delta_2, s')$

$\text{partitions} : \mathit{Heap} \to \mathfrak{P}(V \rightharpoonup \mathit{Heap})$
$\text{partitions}(h) \triangleq \{h_\Gamma \in V \rightharpoonup \mathit{Heap} \mid \text{disjoint}(\text{cdom}(h_\Gamma)) \wedge h = \bigcup \text{cdom}(h_\Gamma)\}$

$\text{disjoint} : \mathfrak{P}(\mathit{Heap}) \to \{\top, \bot\}$
$\text{disjoint}(h) \triangleq \forall_{s_1 \in h} \forall_{s_2 \in h}\, s_1 \neq s_2 \Rightarrow \text{dom}(s_1) \cap \text{dom}(s_2) = \varnothing$

$\text{eval}_\mathcal{E} : \left[\mathfrak{P}(E) \times \mathit{Store}\right] \to \mathfrak{P}(\mathit{Value})$
$\text{eval}_\mathcal{E}(\delta, s) \triangleq \{\mathcal{E}[\![e]\!]_s \mid e \in \delta\}$

Figure 3.8: Validity

Again, let us start by going through the arguments of predicate $\vDash$ – used to denote validity of a specification. A specification then has a resource context Γ stating what always holds on the shared part of the heap, an initial set of variables δ_1 that the program has previous load-linked and a precondition assertion p stating what holds at the beginning of program execution, a program c to be verified, a postcondition assertion q that the program will establish upon completing and a final set δ_2 of variables load-linked by the end of program execution.

Stepping into the definition itself – inspired by that of Vafeiadis [Vaf11] – it states that if the program starts with a set of up-to-date addresses contained in the set of previously load-linked variables, with a given store and heap which can be split in such a way that part of the heap satisfies the resource context and shared assertions while another satisfies the local properties of the precondition, then the program does not fault and ends in a configuration whose heap can also be split as previously – only now targeting the postcondition – and whose set of up-to-date addresses are contained in the set of final load-linked variables.

Concluding, the goal of being able to reason about lock-free programs based on LL/SC is fulfilled with the definition of validity. Note how both properties and resource contexts grasp the essence of aforementioned primitives – shared statements capture the optimistic nature of lock-free algorithms, allowing us to reason about threads based on the premise that a given address has not changed, while resource invariants establish what always holds on a heap partition and all threads must preserve.

3.3 Inference Rules

Despite having a semantic way of reasoning about the programs we are interested in, proving correctness of specifications exclusively using first-order logic would be extremely time costly. In order to keep proofs manageable, similarly to other related works our inference rules provide a syntax directed way of constructing them.

Again, although most of the following developments are based on those of concurrent separation logic and RGSep since we share a great part of the programming language and assertion syntax, our rules additionally have to address shared assertions used to reason about optimistic concurrency introduced by LL/SC primitives.

3.3.1 Rules

Inference rules, presented in Figures 3.9 and 3.10, define predicate $\vdash$ which denotes syntactic validity of a specification. Like semantic validity, it takes a resource context Γ, a set of previously load-linked variables δ, a precondition assertion p, a program c to be verified, a postcondition assertion q and a final set of load-linked variables δ'.

Although we will not detail every single rule, since most of them are a straightforward adaptation of rules presented before to handle our resource contexts and sets of load-linked variables, it should be noted that all rules disallow assignment into variables which are used in the resource context – such unconditional assignments could result on data races. Another noteworthy property all rules for basic commands except for `skip` have is that they use minimal heaps and sets of initial and final load-linked variables – it makes for simpler rules, which can be adapted to any valid heap and set of load-linked variables through composition of frame rules. This last property is usually called locality and is especially useful because it allows us to solely focus on the heap portion a program has impact on – the so-called memory footprint – and then extend it to any larger heap.

$$\frac{}{\Gamma \vdash \delta\{p\}\, \texttt{skip}\, \{p\}\delta} \quad \text{(skip)}$$

$$\frac{}{\Gamma \vdash \delta\{p[e/x]\}\, x := e\, \{p\}\delta} \quad \text{(assign)}$$
$$\text{if } x \notin \mathrm{fv}(\Gamma)$$

$$\frac{\Gamma \uplus \{\exists_v\, x \mapsto v * f\} \vdash \delta\{p\}\, c\, \{q\}\delta'}{\Gamma \vdash \delta\{p * \exists_v\, x \mapsto v * f\}\, c\, \{q * \exists_v\, x \mapsto v * f\}\delta'} \quad \text{(shared resource)}$$
$$\text{if } x \notin \delta'$$

$$\frac{}{\Gamma \vdash \{\}\{p[e/x] \wedge y \mapsto e\}x := [y]\{p \wedge y \mapsto e\}\{\}} \quad \text{(local load)}$$
$$\text{if } x \neq y \text{ and } x \notin \mathrm{fv}(\Gamma, e)$$

$$\frac{}{\Gamma \vdash \{z\}\{^{z}\boxed{p[e/x] \wedge y \mapsto e}\}x := [y]\{^{z}\boxed{p \wedge y \mapsto e}\}\{z\}} \quad \text{(shared load)}$$
$$\text{if } x \neq y \text{ and } x \notin \mathrm{fv}(\Gamma, e)$$

$$\frac{}{\Gamma \uplus \{\exists_v\, y \mapsto v * f\} \vdash \{\}\{\texttt{emp}\}\, x := \texttt{LL}(y)\, \{^{y}\boxed{f[x/v]}\}\{y\}} \quad \text{(load-link)}$$
$$\text{if } x \neq y \text{ and } x \notin \mathrm{fv}(\Gamma, f)$$

$$\frac{}{\Gamma \vdash \{\}\{x \mapsto -\}[x] := y\{x \mapsto y\}\{\}} \quad \text{(store)}$$

$$\frac{p' * q' \Rightarrow f[y/v]}{\begin{array}{l}\Gamma \uplus \{\exists_v\, z \mapsto v * f\} \vdash \\ \quad \{z\}\{p * p' * {}^{z}\boxed{q * q'}\}\, x := \texttt{SC}(y, z)\, \{(x = 1 \wedge p * q) \vee (x = 0 \wedge p * p')\}\{\}\end{array}} \quad \text{(store-conditional)}$$
$$\text{if } x \neq z \text{ and } x \notin \mathrm{fv}(\Gamma, f)$$

$$\frac{}{\Gamma \vdash \{\}\{\texttt{emp}\}x := \texttt{alloc}(e_1, \ldots, e_n)\{x \mapsto e_1, \ldots, e_n\}\{\}} \quad \text{(allocate)}$$
$$\text{if } x \notin \mathrm{fv}(\Gamma)$$

$$\frac{}{\Gamma \vdash \{\}\{\exists_v\, x \mapsto v\}\texttt{disp}(x)\{\texttt{emp}\}\{\}} \quad \text{(dispose)}$$

$$\frac{\Gamma \vdash \{x\}\{^{x}\boxed{p}\}\, c\, \{^{x}\boxed{q}\}\{x\}}{\Gamma \vdash \{x\}\{^{x}\boxed{p * f}\}\, c\, \{^{x}\boxed{q * f}\}\{x\}} \quad \text{(shared expand)}$$
$$\text{if } \mathrm{fv}(f) \cap \mathrm{writes}(c) = \varnothing$$

$$\frac{\Gamma \vdash \delta\{p\}\, c\, \{q\}\delta'}{\Gamma \vdash \big(\delta \cup \{x\}\big)\, \{p * {}^{x}\boxed{f}\}\, c\, \{q * {}^{x}\boxed{f}\}\, \big(\delta' \cup \{x\}\big)} \quad \text{(shared frame)}$$
$$\text{if } x \in \mathrm{res}(\Gamma) \setminus \big[\delta \cup \mathrm{fv}(c)\big] \text{ and } \mathrm{fv}(f) \cap \mathrm{writes}(c) = \varnothing$$

Figure 3.9: Proof rules

$$\frac{\Gamma \vdash \delta\{p\}\, c\, \{q\}\delta'}{\Gamma \vdash \delta\{p * f\}\, c\, \{q * f\}\delta'} \quad \text{if } \mathrm{fv}(f) \cap \mathrm{writes}(c) = \varnothing \qquad \text{(local frame)}$$

$$\frac{\Gamma \vdash \delta\{p\}\, c\, \{q\}\delta'}{\Gamma \uplus \Gamma' \vdash \delta\{p\}\, c\, \{q\}\delta'} \quad \text{if } \mathrm{fv}(\Gamma') \cap \mathrm{writes}(c) = \varnothing \qquad \text{(resource frame)}$$

$$\frac{\Gamma \vdash \{\}\{p_1\}\, c_1\, \{q_1\}\{\} \quad \Gamma \vdash \{\}\{p_2\}\, c_2\, \{q_2\}\{\}}{\Gamma \vdash \{\}\{p_1 * p_2\}\, c_1 \parallel c_2\, \{q_1 * q_2\}\{\}} \qquad \text{(parallel)}$$

$$\text{if } \mathrm{fv}(p_1, q_1, c_1) \cap \mathrm{writes}(c_2) = \mathrm{fv}(p_2, q_2, c_2) \cap \mathrm{writes}(c_1) = \varnothing \text{ and } \mathrm{frees}(c_1) = \mathrm{frees}(c_2) = \varnothing$$

$$\frac{\Gamma \vdash \delta\{p \wedge b\}\, c_1\, \{q\}\delta' \quad \Gamma \vdash \delta\{p \wedge \neg b\} c_2 \{q\}\delta'}{\Gamma \vdash \delta\{p\}\, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\, \{q\}\delta'} \qquad \text{(if–then–else)}$$

$$\frac{\Gamma \vdash \delta\{p \wedge b\}\, c\, \{p\}\delta}{\Gamma \vdash \delta\{p\}\, \texttt{while}\ b\ \texttt{do}\ c\, \{p \wedge \neg b\}\delta} \qquad \text{(while)}$$

$$\frac{p \Rightarrow p' \quad \Gamma \vdash \delta\{p'\}\, c\, \{q'\}\delta' \quad q' \Rightarrow q}{\Gamma \vdash \delta\{p\}\, c\, \{q\}\delta'} \qquad \text{(consequence)}$$

$$\frac{\Gamma \vdash \delta\{p\}\, c_1\, \{q'\}\delta'' \quad \Gamma \vdash \delta''\{q'\}\, c_2\, \{q\}\delta'}{\Gamma \vdash \delta\{p\}\, c_1; c_2\, \{q\}\delta'} \qquad \text{(sequential)}$$

$$\frac{\Gamma \vdash \delta\{p\}\, c\, \{q\}\delta'}{\Gamma \vdash \delta\{\exists_v\, p\}\, c\, \{\exists_v\, q\}\delta'} \quad \text{if } v \notin \mathrm{fv}(c) \qquad \text{(existential)}$$

Figure 3.10: Proof rules (cont.)

Detailing inference rules, let us start by the ones which alter resource context Γ. In order to share a local piece of heap, the *shared resource* rule states that if the new invariant is initially established on the local portion of the heap and all sub-programs maintain and have no local property that depends on it by the end of program execution, then both the postcondition and invariant still hold locally at that point – note that if there still are local properties based on an outdated load-link, this might not be the case. Although most proofs done in a given resource context hold on any larger one, the *resource frame* rule gives a precise account of when this can happen and easily allows for it – as long as the program does not modify any of the variables mentioned on the new resource context, it can be added to the existing one.

Briefly going over rules existing in previous formalisms, *skip* stays the same – now relaying the set of load-linked variables as well – while *assign* is extended with a new side condition that prevents modifying variables used in the resource context – such assignments could result on a data race. Regarding heap commands, *local load* has an additional side condition preventing shared data races as well, while *store* remains as before – note that since both these rules only target the local heap, they use empty sets of initial and final load-linked variables. Similarly, *allocate* cannot save the location into one of the resource's variables and *dispose* stays unchanged – both also using empty sets of load-linked variables. Addressing other previous commands, *if-then-else*, *while*, *sequential* composition, logical *consequence*, *existential* quantification and *local frame* remain the same – now accounting for the additional sets of load-linked variables. Although the rule for parallel composition is very similar to that of concurrent separation logic and `RGSep`, an additional side condition preventing programs from freeing memory is needed in order for the whole proof system to be sound – in the following subsection, this apparently too strong of a restriction will be justified.

Finally, we describe rules which make use of the new elements in our proof system's specifications – shared assertions and sets of load-linked variables. Starting with *load-link*, its rule requires a resource context which contains a memory heap managed by the right-hand side address, as well as an empty heap and set of initially load-linked variables. Then it states the given address has been load-linked and program execution ends with a shared assertion for the resource invariant, which holds as long as that address stays up-to-date – where the existentially quantified value the invariant describes has been replaced by the left-hand side of the assignment, so it can be locally manipulated by the program. In order to allow for several programs to read from a shared address, which does not pose a heap data race, the *shared load* rule allows for a program to reason about shared assertions as though they were local. This is safe to do as long as the address is not freed, because conditional reasoning done in the context of a load-linked variable is discarded if the corresponding store-conditional fails. Moving into *store-conditional*, its rule requires a resource context which has a mapping for the target address. Then it states that if both shared and local heap can be split in such a way that putting together a partition from each one results in a heap satisfying the invariant, either the address has not been changed and store-conditional succeeds – transferring part of the shared heap into the local state and vice-versa – or it fails, keeping both heaps intact. Note that since an update will have occurred either way, the shared assertion is always discarded.

Regarding compositionality, *shared frame* states that an additional shared assertion can be added to both pre and postcondition as long as the program has previously load-linked its control variable, which the resource context must have a mapping for, and does not mention it. Lastly, *shared expand* provides for a frame rule targeting shared assertions, stating that if a given assertion held as long as its control variable remained up-to-date, and the program did not change it nor any of its variables, then that assertion still holds.

Figure 3.11 presents auxiliary syntactic definitions needed in side conditions. Function *writes* computes the set of variables a program may modify throughout its execution,

$$\begin{array}{lcl}
\text{writes}(\texttt{skip}) & \triangleq & \varnothing \\
\text{writes}(x := e) & \triangleq & \{x\} \\
\text{writes}(x := \texttt{alloc}(e_1, \ldots, e_n)) & \triangleq & \{x\} \\
\text{writes}(x := [y]) & \triangleq & \{x\} \\
\text{writes}(x := \texttt{LL}(y)) & \triangleq & \{x\} \\
\text{writes}([x] := e) & \triangleq & \varnothing \\
\text{writes}(x := \texttt{SC}(y, z)) & \triangleq & \{x\} \\
\text{writes}(\texttt{disp}(x)) & \triangleq & \varnothing \\
\text{writes}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) & \triangleq & \text{writes}(c_1) \cup \text{writes}(c_2) \\
\text{writes}(\texttt{while}\ b\ \texttt{do}\ c) & \triangleq & \text{writes}(c) \\
\text{writes}(c_1; c_2) & \triangleq & \text{writes}(c_1) \cup \text{writes}(c_2) \\
\text{writes}(c_1 \parallel c_2) & \triangleq & \text{writes}(c_1) \cup \text{writes}(c_2) \\
 & & \\
\text{frees}(\texttt{skip}) & \triangleq & \varnothing \\
\text{frees}(x := e) & \triangleq & \varnothing \\
\text{frees}(x := \texttt{alloc}(e_1, \ldots, e_n)) & \triangleq & \varnothing \\
\text{frees}(x := [y]) & \triangleq & \varnothing \\
\text{frees}(x := \texttt{LL}(y)) & \triangleq & \varnothing \\
\text{frees}([x] := e) & \triangleq & \varnothing \\
\text{frees}(x := \texttt{SC}(y, z)) & \triangleq & \varnothing \\
\text{frees}(\texttt{disp}(x)) & \triangleq & \{x\} \\
\text{frees}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) & \triangleq & \text{frees}(c_1) \cup \text{frees}(c_2) \\
\text{frees}(\texttt{while}\ b\ \texttt{do}\ c) & \triangleq & \text{frees}(c) \\
\text{frees}(c_1; c_2) & \triangleq & \text{frees}(c_1) \cup \text{frees}(c_2) \\
\text{frees}(c_1 \parallel c_2) & \triangleq & \text{frees}(c_1) \cup \text{frees}(c_2)
\end{array}$$

Figure 3.11: Writes and frees functions

while *frees* stands for the one a program may deallocate. Note that these functions differ from the ones used in operational semantics, because those only compute the set of variables regarding a computational step and not the whole program execution.

Using the above inference rules, it is now possible to syntactically prove that a program meets a given specification. Furthermore, these rules allow us to solely focus on the local changes done by subprograms and obtain a proof for the whole program by composing them.

Concluding, we hope the reader agrees these rules capture the optimistic nature of `LL/SC` primitives through the usage of shared assertions which are created upon load-linking a given address and either kept in case the matching `SC` succeeds or discarded should it fail. Former constructs from concurrent separation logic blend well with these new shared assertions, which were inspired by those of `RGSep`, hopefully resulting in an easily understandable proof system to people familiar with such formalisms. Although sets of load-linked variables might seem like an unpolished loose end, the decision of including them on specifications was a deliberate one. Those sets enforce programs to be well-formed regarding load-link and store-conditional and allow for simpler rules by avoiding more complex side-conditions, since there is at most one open load-link for each control variable, so we decided to keep them.

Although inference rules were built incrementally through an iterative process of refining them to allow proving programs with increasingly harder interference to reason about, and are able to prove most of the examples present in concurrent separation logic, it should be noted that some restrictions were inherited from the same logic. Namely, because resources must be disjoint, with this system it is currently not possible to reason about programs which need two load-linked variables in order to ensure that an address accessible by either variable remains consistent – such as a node in a doubly linked list, which can be accessed by either head or tail. Additionally, as was briefly mentioned on the explanation of *shared load* and can be seen in the side condition for *parallel* composition, due to aliasing of store variables and shared memory reads, parallel programs are not allowed to deallocate memory – properties proven with these rules would still hold without that side condition, but the program could now abort and violate validity.

3.3.2 Soundness

Given inference rules present a syntactic directed alternative of proving validity of specifications, care must be taken to ensure they indeed agree with semantic validity – as seen before, side conditions are necessary to deal with aliasing, data races or some other property required for that agreement to hold. Therefore, although no formal proof of soundness is provided in this document, a brief argument explaining why rules are sound will be given for each one.

As previously, completeness – that is, being able to construct a proof of every semantically valid specification using inference rules – is not possible due to undecidability of checking whether some statements are valid. Recall that to prove a judgment like $\Gamma \vDash \delta\{\texttt{true}\}\, c\, \{\texttt{false}\}\delta'$ is to solve the halting problem.

skip Let the program start in an arbitrary resource context Γ and state satisfying assertion p, having previously load-linked variables δ, then `skip` will not change the set of load-linked variables δ nor local assertions occurring in p. Regarding shared assertions, then either their control address remained up-to-date and the assertion still holds, or they were modified – being outdated regarding the thread's local view – in which case the shared assertion vacuously holds. Finally, because `skip` does not change any variables nor access the heap, it does not raise data races or abort either.

assign Like `skip`, *assign* does not change the set of load-linked variables δ, simply relaying it. Although the heap is not accessed either, therefore never aborting, care must be taken since the store's mapping for variable x may now change. Because x cannot be a shared variable occurring in resource context Γ, and parallel composition does not allow for concurrent programs to change each other's local variables, then data races are not a problem either. Regarding local and shared assertions, given they did not mention variable x, it is safe to replace an arbitrary expression which will be assigned to the store value x maps into by x itself.

shared resource Unlike previous rules, this one has a premise which we can use to obtain additional information. Knowing that the specification is valid when a piece of local heap is moved into resource context Γ, then it is certain program c does not alter variable x and preserves shared invariant f controlled by that variable. Now, given that program c does not modify x nor invalidate f, it is safe to relay them from the precondition into the postcondition. Again, since program c did not have data races nor abort on the same store and heap, it is safe to say it will not do so. One important last remark, since shared and local heaps are always disjoint, is that a shared assertion cannot exist for address x because a local one already exists for that same address – hence the side condition stating there cannot be an open load-link for variable x.

local load Similarly to *assign*, since *local load* modifies the store, x is not allowed to be a shared variable occurring in resource context Γ – preventing data-races. Given no variables were load-linked, and none are, sets of load-linked variables remain empty. Since the rule only deals with local assertions, given no addresses are load-linked, we need not worry about shared assertions. Regarding memory faults, since the precondition states address y must be present, program execution does not abort. Finally, like *assign*, it is safe to replace an arbitrary expression which will be assigned to the store value x maps into by x itself – note that because variable x is modified while y and expression e remain unchanged in both pre and postcondition, x must differ from y and cannot occur in e.

shared load This rule complements *local load*, in the sense that it only deals with a shared assertion – not mentioning the local heap. Again, to guarantee data races do not take place, variable x cannot occur in resource context Γ. Since base rules deal with the smallest heap possible, *shared load* starts with a single load-linked variable z and shared assertion stating what holds on address y, and behaves like *local load* inside that assertion. However, two additional conditions must be met because address y is inside a shared assertion. Firstly the assertion must hold if address z remains up-to-date, and secondly program execution cannot abort. The former is easy to check because z has not been load-linked, so the thread's local view of that address remains unchanged. However, since concurrent programs can access shared heap location y, the latter is not so trivial. Therefore, since the system is currently unable to stop a program from deallocating memory some other is still accessing or check if a given shared address has been deallocated, to ensure program execution does not abort parallel composition of programs has the harsh side-condition stating that neither subprogram can deallocate memory. Note that this is true even regarding local heap, since memory is transfered from and to the shared state with no tracing information. Concluding, given that our inference rules do not allow memory location y to be removed, program execution does not abort.

load-link Let the program start with an empty set of load-linked variables and a resource context which states that invariant f always holds at address y. Again, to guarantee data races do not take place, variable x cannot occur in resource context Γ or expression f and must differ from y. Since y has been load-linked, it will then be the sole variable load-linked upon termination. Because address y belongs to the shared heap, and programs can only deallocate local memory, *load-link* does not abort. Finally, since invariant f at address y always holds and refers to the same heap as long as y is not updated, the post condition's shared assertion states exactly what holds by the end of program execution – where variable v, representing the currently present value at address y which was now read into x, is replaced by this latter variable.

store Let the program start with a local heap in which address x is allocated, and having no currently load-linked variables. Then, since it modifies no variables, *store* will not raise data races and, given local address x is allocated, will not abort. Finally, because address y has been updated to the value stored in y and is not shared, it will contain that value by the end of program execution.

store-conditional Let the program start with a single load-linked variable z, a resource context which states invariant f always holds at address y, and both a local assertion which separates p from p' as well as a shared assertion for variable z which separates q from q'. Again, to guarantee data races do not take place, variable x cannot occur in resource context Γ or expression f and must differ from z. Similarly to *load-link*, because address y belongs to the shared heap and programs can only deallocate local memory, *store-conditional* does not abort. Moving into assertions, having these four separate heaps at our disposal and knowing that putting together the two satisfying p' and q' we obtain a heap satisfying invariant f where variable y fills the role of value v stored at address z, then either the thread's view of that address is outdated and store-conditional fails, or it is up-to-date and store conditional successfully transfers local heap satisfying p' into the shared state and shared heap satisfying q into the local state, resulting in a shared heap controlled by address z satisfying invariant f and a local heap which can be split such a way that one piece satisfies p while the other satisfies q. Finally, since it is either outdated or has now been updated, address z is removed from the set of potentially up-to-date load-linked variables as well as its shared assertion.

allocate Let the program start with an empty heap and set of potentially up-to-date load-linked variables. Firstly, in order to avoid data races, the allocated memory address cannot be stored in a shared variable occurring in resource context Γ. Since *allocate* does not access existing heap locations, it does not abort nor change sets of load-linked addresses. Finally, the new location is stored in variable x which now points to a contiguous memory block with all given expressions, as stated in the postcondition.

dispose Since it does not modify any of the store's variables, *dispose* does not raise data races. Because the precondition states address x is local and must be allocated, program execution does not change the set of load-linked addresses nor abort either. Finally, as the only existing address was now deallocated, *dispose* will end with an empty heap as stated in the postcondition.

shared expand Let the program start with a single assertion for its only load-linked variable x stating that shared heap controlled by address x can be split in two pieces such that one satisfies assertion p while the other satisfies f. Then either address x is outdated regarding the thread's view, vacuously satisfying the postcondition, or it is up-to-date. Given address x is up-to-date, we know the precondition's assertion still currently holds at that address. Additionally, we know the program does not modify any variable occurring in f, as required by the side condition, nor access any of its heap addresses, once the rule's premise enforces all of its memory operations to be within the heap given by p. Then, since program c transforms a heap satisfying p into one satisfying q and it is safe to relay f, the postcondition holds. Finally, because the program remained the same and did not abort nor have data races, maintaining the initial and final set of load-linked variables, all these properties are directly inherited by the rule's premise.

shared frame As in the previous rule, given its premise *shared frame* does not abort nor have data races and does not access variables or addresses that might alter assertion f. Then, having previously load-linked x, either that address is outdated, in which case the postcondition's additional shared assertion vacuously holds, or it is up-to-date and the postcondition's additional shared assertion is relayed from the precondition. Finally, in order for specifications to be well-formed, variable x must control some shared heap present in resource context Γ so it can belong to the set of load-linked addresses, and cannot be present in that set nor accessed by program c so the new shared assertion is not a duplicate and remains valid.

local frame Similarly, given its premise, *local frame* does not abort nor have data races and does not access variables or addresses that might alter assertion f. Finally, since it is local and remains unaltered by program execution, assertion f as well as sets of load-linked variables can be safely relayed from the precondition into the postcondition.

resource frame Given that program c already meets the premise's specification, its shared resources are completely contained in resource context Γ. Therefore, *resource frame* is sound since it only adds an assertion which the program does not have impact on into the resource context.

parallel Let the program start with a local heap which can be split into two pieces where one satisfies assertion p_1 and another satisfies p_2. From the premises, we know giving local heaps satisfying p_1 and p_2 to programs c_1 and c_2 will transform them into heaps satisfying q_1 and q_2. Then, because both programs manipulate disjoint local heaps and do not modify each other's variables, *parallel* composition of programs c_1 and c_2 starting with local heaps satisfying p_1 and p_2 will result in disjoint heaps satisfying q_1 and q_2, as stated in the postcondition. As both programs manipulate disjoint local heaps and do not modify each other's stores, no data-races exist on the local state. Since they only access shared data described by resource context Γ, and given the premises are valid do not have data races on that resource context, no data-races exist on the shared state either. Finally, because programs do not fault with the given resource context and local heap, and no shared addresses are freed as enforced by the side condition, *parallel* composition does not abort.

if–then–else Let the program start with resource context Γ, initial set of load-linked addresses δ and local state satisfying assertion p. Then, given either branch does not have data races nor abort and finishes in a state satisfying assertion q, having load-linked variables $delta'$, the *if-then-else* construct also does so.

while Let the program start with resource context Γ, initial set of load-linked addresses δ and local state satisfying assertion p. Then, given that executing the body of *while* does not raise data races nor abort and both assertion p as well as set of load-linked variables δ are reestablished, its whole execution does not have data races nor abort either and will preserve those two components. Additionally, since validity only holds if the cycle stops, assertion b will not hold – as stated in the postcondition.

consequence Given the rule's premise, we know program c starting with resource context Γ, initial set of load-linked addresses δ and local state satisfying assertion p' will not raise data races nor abort and end with a set of variables δ', having established assertion q'. Then, as in previous formalisms it is safe to weaken precondition p to match p' as well as to extract a weaker postcondition q from q', established by the program.

sequential Let program c_1 start with assertion p and previously load-linked addresses δ, then as stated in the premise it will establish assertion q' and have load-linked addresses δ''. Similarly, program c_2 will start with assertion q' and load-linked addresses δ'' and end with postcondition q as well as load-linked addresses δ'. Since both premises use invariant Γ and guarantee programs do not have data races nor abort, *sequential* composition can safely start with assertion p and set of load-linked addresses δ, relay the final components from c_1 into the initial ones of c_2, and end with postcondition q and set of load-linked addresses δ'.

existential Let program c start in resource context Γ, with assertion p where variable v has been existentially quantified, having previously load-linked addresses δ. Then either variable v does not occur in assertion p, or it does occur in p but we know that assertion holds for some value mapped into by v – either way, precondition p holds. Given that program c starting with assertion p and load-linked variables δ does not alter the value of v nor have data races or abort, and terminates in a state satisfying q having load-linked variables δ', then either v does not occur in q or it was unaltered and is safe to existentially quantify – the same mapping used in the initial store will also be applicable. Either way, postcondition q holds.

Concluding, although being informal, the arguments presented in this section lay out a solid foundation for developing a soundness proof. Perhaps more importantly, they explain why side conditions such as disallowing concurrent memory deallocation are needed in order for syntactic and semantic validity to match.

3.4 Examples

Although also being important for showcasing the usage of inference rules, examples played a major role in building our proof system. As briefly mentioned before, several iterations were needed to raise the expressiveness of specifications and inference rules before examples presented in this section could be proven correct.

Therefore, the goal of this section is twofold. Firstly, specifications and inference rules are exemplified, allowing the reader to fully grasp how they work in practice. Secondly, through the presentation of examples with increasing interference requirements, changes that needed to be made in order for those examples to be proven correct are described – guiding the reader through the development process.

3.4.1 Counter

Starting out, we begin with a simple example presented in Figure 3.12 which increments a shared variable. In this example, program execution is put inside a resource context which states the address given by variable *counter* always contains a positive value. Then, we want to prove that concurrently incrementing address *counter* does not result in it ending with a negative value.

Going into detail, let the program start with an empty set of previously load-linked variables and having no local memory allocated. Since the resource context remains unaltered, and in most cases a command's postcondition and final set of load-linked variables are the precondition and initial set of load-linked variables of the command following it, only one of those pairs is shown between a sequential composition. However, on situations where one or more steps need to be applied for pairs to match corresponding

```
$\{\exists_v\, counter \mapsto v \wedge v \geq 0\} \vdash$
        $\{\}\{\texttt{emp}\}$
            $\{\}\{\texttt{emp}\}$
            while $res_1 \neq 1$ do
                $\{\}\{\texttt{emp}\}$
                $val_1 := \texttt{LL}(counter);$
                $\{counter\}\{{}^{counter}\boxed{val_1 \geq 0)}\}$
                $\{counter\}\{{}^{counter}\boxed{val_1 + 1 \geq 0)}\}$
                $val_1 := val_1 + 1;$
                $\{counter\}\{{}^{counter}\boxed{val_1 \geq 0)}\}$
                $res_1 := \texttt{SC}(val_1, counter)$
                $\{\}\{(res_1 = 0 \wedge \texttt{emp}) \vee (res_1 = 1 \wedge \texttt{emp})\}$
                $\{\}\{\texttt{emp}\}$
            $\{\}\{\texttt{emp}\}$
        $\|$
            $\{\}\{\texttt{emp}\}$
            while $res_2 \neq 1$ do
                $\{\}\{\texttt{emp}\}$
                $val_2 := \texttt{LL}(counter);$
                $\{counter\}\{{}^{counter}\boxed{val_2 \geq 0)}\}$
                $\{counter\}\{{}^{counter}\boxed{val_2 + 1 \geq 0)}\}$
                $val_2 := val_2 + 1;$
                $\{counter\}\{{}^{counter}\boxed{val_2 \geq 0)}\}$
                $res_2 := \texttt{SC}(val_2, counter)$
                $\{\}\{(res_2 = 0 \wedge \texttt{emp}) \vee (res_2 = 1 \wedge \texttt{emp})\}$
                $\{\}\{\texttt{emp}\}$
            $\{\}\{\texttt{emp}\}$
        $\{\}\{\texttt{emp}\}$
```

$$\frac{\{\exists_v\, counter \mapsto v \wedge v \geq 0\} \vdash \{\}\{\texttt{emp}\}\, \text{increment}(val_1) \parallel \text{increment}(val_2)\, \{\texttt{emp}\}\{\}}{\varnothing \vdash \{\}\{\exists_v\, counter \mapsto v \wedge v \geq 0\}\, \text{increment}(val_1) \parallel \text{increment}(val_2)\, \{\exists_v\, counter \mapsto v \wedge v \geq 0\}\{\}}$$

Figure 3.12: Counter example

rules, both assertions and sets of load-linked addresses are shown in order to improve readability.

Each program optimistically tries to increase shared address *counter*, retrying until successfully doing so. Firstly, address *counter* is load-linked into local variable *val*, which as ensured by the invariant will hold a positive value. Then this local variable is increased by one, remaining positive, and conditionally stored into address *counter* – which is allowed by the resource context, since it again meets that address' invariant. Finally, the success status is saved into local variable *res* and tested in the loop's condition.

Given these programs do not access each other's variables nor unsafely modify shared addresses, and do not deallocate memory, the rule for *parallel* composition states they will indeed keep shared address *counter* positive.

Although being quite simple, this example illustrates several patterns used in most `LL/SC` based programs. Note that because they are optimistic, programs usually loop until successfully writing into the target address. Additionally, to avoid data races, shared addresses are load-linked into local variables that are safe from interference and used for constructing the new value which will be conditionally stored.

Regarding expressiveness, this example corresponds to the first iteration of the proof system, where only reasoning about a single memory address is required and no distinction needs to be made on store-conditional's result.

3.4.2 List

Lists of data are a broadly used and well-understood data structure, also being extensively used as examples in works where dynamic memory allocation is addressed. Furthermore, since they require dealing with interference on more than one address, lists constitute the remainder of our examples and discussions.

$$\mathrm{list}(i) \triangleq (i = \texttt{nil} \wedge \texttt{emp}) \vee (i \neq \texttt{nil} \wedge \exists_j\, i \mapsto j * \mathrm{list}(j))$$

As can be seen above a list is either the `nil` value, in which case the heap must be empty, or a value corresponding to the location of an existing memory address pointing to the rest of the list.

```
$\{\exists_v\, head \mapsto v * \texttt{list}(v)\} \vdash$
    $\{\}\{\texttt{emp}\}$
    $node := \texttt{alloc}(\texttt{nil});$
    $\{\}\{\exists_v\, node \mapsto v\}$
    $res := 0;$
    $\{\}\{(res = 0 \wedge \exists_v\, node \mapsto v) \vee (res = 1 \wedge \texttt{emp})\}$
    $\texttt{while}\ res \neq 1\ \texttt{do}$
        $\{\}\{\exists_v\, node \mapsto v\}$
        $first := \texttt{LL}(head);$
        $\{head\}\{\exists_v\, node \mapsto v * {}^{head}\boxed{\texttt{list}(first)}\}$
        $[node] := first$
        $\{head\}\{node \mapsto first * {}^{head}\boxed{\texttt{list}(first)}\}$
        $res := \texttt{SC}(node, head)$
        $\{\}\{(res = 0 \wedge node \mapsto first) \vee (res = 1 \wedge \texttt{emp})\}$
        $\{\}\{(res = 0 \wedge \exists_v\, node \mapsto v) \vee (res = 1 \wedge \texttt{emp})\}$
    $\{\}\{\texttt{emp}\}$
```

Figure 3.13: Push example

Following previous conventions, Figure 3.13 presents a program which adds a memory address to a given list. The specification states program execution is done in a resource context which ensures address *head* contains a list, starts with an empty heap and no load-linked variables, finishing that way as well.

Firstly a new memory address is allocated, and its location stored in local variable *node*. As before the program loops until successfully inserting the new node, which is done by load-linking shared variable *head* containing the list's first location into local variable *first*, making the allocated node point to that location and finally attempting to store it into the shared state. Note that, as previously stated, for the sake of readability some proof steps are omitted. In order to detail those proofs, and show compositionality of rules, some missing steps are displayed on Figure 3.15.

One new requirement needed to prove correctness of list insertion is for invariants to be more broad than a single address – now the shared invariant is an arbitrary separate heap holding a list. Additionally, store-conditional's result must be taken into account. Depending on whether store-conditional is successful, either the newly allocated node was transfered into the shared heap or stayed local – needed for knowing it is still allocated inside the `while` body, and is not locally available upon finishing.

Another operation which is frequently done on lists is element removal. Figure 3.14 details a program which removes the first element from a given list. Again, program execution is done on a resource context ensuring address *head* contains a list, begins with no load-linked variables and an empty heap, ending that way too.

As previously, the program loops until successfully removing an element from the list or maintaining it empty. The loop's invariant states that either last store-conditional operation was unsuccessful, in which case the local heap must be empty, or it was successful. In this last case either the list did not contain any elements, maintaining the local heap empty, or it had at least one element which was transfered from the shared heap into the local one.

Inside the `while` loop, shared list stored at address *head* is load-linked into local variable *first*. Then, depending on whether it is empty, the program either tries to store the list itself or its second node. In this scenario, it is clear why memory cannot be concurrently deallocated – if another program had removed the first node in between, program execution would abort upon trying to obtain its second one. Finally, after concluding the loop, either no node was removed and there is nothing to do, or the now local first element is deallocated – in either case resulting on an empty heap and no load-linked variables.

Although it might look like this example requires as much expressiveness as element insertion, there are two additional requirements for the above proof to be possible. Firstly, in order to obtain the second node, concurrent reading of shared addresses is required – resulting in the need for *shared load* rule. Secondly, because memory can be transfered

```
$\{\exists_v\, head \mapsto v * \mathtt{list}(v)\} \vdash$

$\{\varnothing\}\{\mathtt{emp}\}$
$res := 0;$
$\{\}\{\mathtt{emp} \wedge res = 0\}$
$\{\}\{(res = 0 \wedge \mathtt{emp}) \vee (res = 1 \wedge (first = \mathtt{nil} \Rightarrow \mathtt{emp}) \wedge (first \neq \mathtt{nil} \Rightarrow \exists_v first \mapsto v))\}$
while $res \neq 1$ do
    $\{\}\{\mathtt{emp}\}$
    $first := \mathtt{LL}(head);$
    $\{head\}\{^{head}\boxed{\mathtt{list}(first)}\}$
    if $first \neq \mathtt{nil}$ then
      $\{head\}\{^{head}\boxed{\mathtt{list}(first)} \wedge first \neq \mathtt{nil}\}$
      $\{head\}\{^{head}\boxed{\exists_j first \mapsto j * \mathrm{list}(j)}\}$
      $second := [first]$
      $\{head\}\{^{head}\boxed{first \mapsto second * \mathrm{list}(second)}\}$
      $\{head\}\{^{head}\boxed{((first = \mathtt{nil} \Rightarrow \mathtt{emp}) \wedge (first \neq \mathtt{nil} \Rightarrow \exists_v first \mapsto v)) * \mathrm{list}(second)}\}$
     else
      $\{head\}\{^{head}\boxed{\mathtt{list}(first)} \wedge first = \mathtt{nil}\}$
      $second := first$
      $\{head\}\{^{head}\boxed{\mathtt{list}(second)} \wedge first = \mathtt{nil}\}$
      $\{head\}\{^{head}\boxed{((first = \mathtt{nil} \Rightarrow \mathtt{emp}) \wedge (first \neq \mathtt{nil} \Rightarrow \exists_v first \mapsto v)) * \mathrm{list}(second)}\}$
    ;
    $\{head\}\{^{head}\boxed{((first = \mathtt{nil} \Rightarrow \mathtt{emp}) \wedge (first \neq \mathtt{nil} \Rightarrow \exists_v first \mapsto v)) * \mathrm{list}(second)}\}$
    $res := \mathtt{SC}(second, head)$
    $\{\}\{(res = 0 \Rightarrow \mathtt{emp}) \wedge (res = 1 \Rightarrow ((first = \mathtt{nil} \Rightarrow \mathtt{emp}) \wedge (first \neq \mathtt{nil} \Rightarrow \exists_v first \mapsto v)))\}$
;
$\{\}\{(first = \mathtt{nil} \Rightarrow \mathtt{emp}) \wedge (first \neq \mathtt{nil} \Rightarrow \exists_v first \mapsto v)\}$
if $first \neq \mathtt{nil}$ then
    $\{\}\{\exists_v first \mapsto v\}$
    $\mathtt{disp}(first)$
    $\{\}\{\mathtt{emp}\}$
 else
    $\{\}\{\mathtt{emp}\}$
    skip
    $\{\}\{\mathtt{emp}\}$
$\{\}\{\mathtt{emp}\}$
```

Figure 3.14: Pop example

from and to the shared state, care must be taken so that programs cannot deallocate addresses others are concurrently reading – hence the harsh side condition on *parallel* rule.

As before, with the goal of simplifying proofs, some of the example's intermediate steps are omitted. However, to ensure we did not oversimplify and are indeed constructing proofs allowed by rules – as well as showcase their locality and compositionality – Figure 3.16 presents several of those steps.

Figure 3.15 presents steps necessary to ensure variable *first* can be read from a shared assertion into local memory address *node*, by composing rules *store* for making that address point to variable *first*, and *shared frame* for separating it with the previously existing assertion stating *first* is a shared list obtained from *head*.

Lastly Figure 3.16 details steps required to concurrently read shared address *first*, containing the list's next element, into variable *second*. Since a shared address is being read, the proof starts with *shared load*. Then *consequence* is used to remove tautology $j = j$ from the precondition, followed by *shared expand* to concatenate the list stored in variable j. Finishing the proof *existential* is used to existentially quantify variable j, which is not accessed by program execution, followed by *consequence* to push it inside the shared assertion.

Reviewing what was presented up until now, we've seen our rules are capable of expressing invariants that state what holds on arbitrarily large memory heaps, of distinguishing between successful and failed store-conditional operations, and dealing with programs that concurrently read shared locations. Additionally they do this in a local, modular and syntax-directed way which allows for easily composing proofs.

$$\begin{array}{c}
\hline
\{\exists_v\, head \mapsto v * \texttt{list}(v)\} \vdash \{\}\{\exists_v\, node \mapsto v\}\, [node] := first\, \{node \mapsto first\}\{\} \\
\hline
\{\exists_v\, head \mapsto v * \texttt{list}(v)\} \vdash \{head\}\{\exists_v\, node \mapsto v *\, {}^{head}\boxed{\texttt{list}(first)}\}\, [node] := first\, \{node \mapsto first *\, {}^{head}\boxed{\texttt{list}(first)}\}\{head\}
\end{array}$$

Figure 3.15: Detailed push derivation example

$$\begin{array}{c}
\hline
\{\exists_v\, head \mapsto v * \texttt{list}(v)\} \vdash \{head\}\{{}^{head}\boxed{j = j \wedge first \mapsto j}\}\, second := [first]\, \{{}^{head}\boxed{j = second \wedge first \mapsto j}\}\{head\} \\
\hline
\{\exists_v\, head \mapsto v * \texttt{list}(v)\} \vdash \{head\}\{{}^{head}\boxed{first \mapsto j}\}\, second := [first]\, \{{}^{head}\boxed{j = second \wedge first \mapsto j}\}\{head\} \\
\hline
\{\exists_v\, head \mapsto v * \texttt{list}(v)\} \vdash \{head\}\{{}^{head}\boxed{first \mapsto j * \mathrm{list}(j)}\}\, second := [first]\, \{{}^{head}\boxed{j = second \wedge first \mapsto j * \mathrm{list}(j)}\}\{head\} \\
\hline
\{\exists_v\, head \mapsto v * \texttt{list}(v)\} \vdash \{head\}\{\exists_j\, {}^{head}\boxed{first \mapsto j * \mathrm{list}(j)}\}\, second := [first]\, \{\exists_j\, {}^{head}\boxed{j = second \wedge first \mapsto j * \mathrm{list}(j)}\}\{head\} \\
\hline
\{\exists_v\, head \mapsto v * \texttt{list}(v)\} \vdash \{head\}\{{}^{head}\boxed{\exists_j\, first \mapsto j * \mathrm{list}(j)}\}\, second := [first]\, \{{}^{head}\boxed{first \mapsto second * \mathrm{list}(second)}\}\{head\}
\end{array}$$

Figure 3.16: Detailed pop derivation example

4 Conclusion

4.1 Summary

In this dissertation we have tackled the definition of a proof system for proving correctness of programs based on `LL/SC` primitives. As we have seen, such a system is valuable because otherwise programs' correctness depend on informal arguments which are hard to understand and might be wrong.

Firstly, we start out by looking at existing formalisms and pointing out their core components as well as why they are unable to address this problem. Then we conclude that although being unable to completely solve it, their components are only a small step away and already give us all the necessary tools for doing so.

Detailing our proof system we begin by its program syntax, which merely allows writing programs one wants to verify since the goal is to use current languages. After being able to write programs, we present an operational semantics describing their execution. Given that truly local information is needed for the first time – for knowing whether a thread has an up-to-date view of an address – a new component is added to program configurations.

Then assertions are introduced, in order to reason about program execution. Assertions stating what holds on the shared state are added to the syntax, and resources from concurrent separation are adapted to our scenario where addresses now control access to shared data structures. Validity, which describes assertion semantics, is modified to account for these additional components present in program configurations and assertions. Furthermore, it enforces programs to be well-behaved and allows reasoning about initial and final sets of load-linked variables.

After having a way to evaluate specifications, inference rules provide an easier syntax-directed alternative of proving validity other than using first-order logic. Additionally, these rules clearly capture the optimistic nature of `LL/SC` primitives and allow for local and modular proofs – enabling one to solely focus on composing proofs of variables and addresses modified by subprograms.

Since they provide an alternative to semantic validity, care must be taken so programs proven correct using inference rules indeed are so. Although we do not provide a formal soundness proof, an informal argument is provided – given it is an important topic which helps better understand rules and their side conditions. Despite not being formal, this argument is structured and covers all components of validity – leaving small room for error and being close to a formal proof.

Lastly we present a few examples, showing how the proof system works in practice. These examples have different expressiveness requirements and lead the system's development process, effectively guiding the user through it. Although some limitations still exist, the reasoning needed to prove these examples correct is already nontrivial.

Concluding, our work then brings forward a logic for reasoning about lock-free programs based on `LL/SC`. More concretely a formal semantics that models their execution, a language and corresponding semantics for stating properties about program execution, as well as a syntax-directed way of constructing correctness proofs of those specifications. Afterwards, despite not being formal, an argument explaining the system's soundness is presented. Finally, some examples showing its usage are presented.

Although it only scratched the surface and did not concern itself with termination, fairness, deadlock-freedom, or other well-known issues which can arise in lock-free programs, this proof system shows how existing formalisms can be adapted to deal with such programs while preserving desirable properties like locality and modularity.

If nothing else, it presents an example of how the arduous and error-prone process of reasoning about lock-free programs can be accomplished formally, in a structured and modular way.

4.2 Future Work

Generalize variables to expressions Because only dealing with variable names on memory addresses eases reasoning about aliasing, given they are unique, most of our proof system is based on them. However, since using arbitrary expressions as a way to specify memory addresses sometimes make programs more concise, this generalization would accelerate encoding programs into our proof system as well as simplify proofs.

Allow concurrent memory deallocation As memory can be exchanged between local and shared state, with no information about which locations might still get accessed concurrently, our proof system disallows concurrent memory deallocation. Due to this side condition, we are currently unable to prove many realistic lock-free programs. Given memory can only be transfered using *store-conditional*, one could add another component which kept track of those locations and only disallow deallocating them.

Allow reasoning about change on the shared state Currently, as in concurrent separation logic, we are only able to reason about shared state invariants. However, one would frequently also like to state the effect a program has on that state as well – e.g. verify that incrementing a shared counter makes it have a greater value than before. In a similar fashion to `RGSep` interference could be modeled using additional rely-guarantee components, allowing for invariants to be replaced by assertions that are preserved by the environment – which are capable of reasoning about change in spite of interference.

Refine resource granularity Another limitation regarding the shared state is that programs which require two or more of its addresses to keep a data structure consistent, such as a doubly linked list, cannot currently be proven. Since this limitation also exists on concurrent separation logic, and `RGSep` only reasons about programs which lock all resources, at the time we have no starting point on how to tackle it.

Formalize the soundness proof Although an informal argument on the system's soundness is provided, which we tried our best to follow a standard approach and cover every detail, a more rigorous proof aimed at formalizing this argument (even if done on paper) would present a great improvement. Additionally, if it was verified by tools such as `Isabelle`[1] or `Coq`[2], such a proof would finally put a nail in the coffin regarding this issue.

Develop an implementation Given it is aimed at proving correctness of real programs, to ease its usage and integration on the development process this theoretical proof system could be implemented as a tool to check correctness of programs written in a concrete programing language – such as `ESC/Java` [FLL+02] for `Java` programs or `Spec#` [BLS05] for `C#` programs.

[1]`http://www.cl.cam.ac.uk/research/hvg/Isabelle/`
[2]`http://coq.inria.fr/`

Bibliography

[BLS05] M. Barnett, K. Leino, and W. Schulte. The spec# programming system: An overview. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, 2005.

[Bro07] S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007.

[CES09] E.M. Clarke, E.A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009.

[Chu40] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[CJ07] J.W. Coleman and C.B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007.

[FLL$^+$02] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, pages 234–245. ACM, 2002.

[Flo67] R.W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19:19–31, 1967.

[GCPV09] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, volume 44, pages 16–28. ACM, 2009.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[HS08] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, February 2008.

[Jon90] C.B. Jones. *Systematic software development using VDM*, volume 2. Prentice Hall, 1990.

[O'H07] P.W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.

[Plo04] G. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61(0):3–15, 2004.

[Rey02] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74. IEEE Computer Society, 2002.

[Vaf07] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, PhD thesis, University of Cambridge, UK, 2007.

[Vaf11] V. Vafeiadis. Concurrent separation logic and operational semantics. *Electronic Notes in Theoretical Computer Science*, 276:335–351, 2011.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, February 1993.