Technical Report

UCAM-CL-TR-726
ISSN 1476-2986

Number 726

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Modular fine-grained concurrency verification

Viktor Vafeiadis

July 2008

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted July 2007 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Selwyn College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Traditionally, concurrent data structures are protected by a single mutual exclusion lock so that only one thread may access the data structure at any time. This coarse-grained approach makes it relatively easy to reason about correctness, but it severely limits parallelism. More advanced algorithms instead perform synchronisation at a finer grain. They employ sophisticated synchronisation schemes (both blocking and non-blocking) and are usually written in low-level languages such as C.

This dissertation addresses the formal verification of such algorithms. It proposes techniques that are modular (and hence scalable), easy for programmers to use, and yet powerful enough to verify complex algorithms. In doing so, it makes two theoretical and two practical contributions to reasoning about fine-grained concurrency.

First, building on rely/guarantee reasoning and separation logic, it develops a new logic, RGSep, that subsumes these two logics and enables simple, modular proofs of fine-grained concurrent algorithms that use complex dynamically allocated data structures and may explicitly deallocate memory. RGSep allows for ownership-based reasoning and ownership transfer between threads, while maintaining the expressiveness of binary relations to describe inter-thread interference.

Second, it describes techniques for proving linearisability, the standard correctness condition for fine-grained concurrent algorithms. The main proof technique is to introduce auxiliary single-assignment variables to capture the linearisation point and to inline the abstract effect of the program at that point as auxiliary code.

Third, it demonstrates this approach by proving linearisability of a collection of concurrent list and stack algorithms, as well as providing the first correctness proofs of the RDCSS and MCAS implementations of Harris et al.

Finally, it describes a prototype safety checker, SmallfootRG, for fine-grained concurrent programs that is based on RGSep. SmallfootRG proves simple safety properties for a number of list and stack algorithms and verifies the absence of memory leaks.

Acknowledgements

I am grateful to my supervisors, Alan Mycroft and Matthew Parkinson, for their invaluable guidance and assistance during this research. I am equally grateful to Tony Hoare, who together with Maurice Herlihy hosted me as an intern at Microsoft Research, and who made me interested in formal reasoning.

I would like to thank the Computer Laboratory, where most of this research was undertaken, the Cambridge programming research group (CPRG), and especially Anton Lokhmotov, with whom I shared an office for the last three years.

I would like to thank the Cambridge Gates Trust for granting me a scholarship and my college, Selwyn, for providing me accomodation and a friendly environment during my studies.

I would also like to thank the following people for stimulating discussions and for insightful comments in various aspects of my work: Nick Benton, Josh Berdine, Gavin Bierman, Cristiano Calcagno, Luca Cardelli, Joey Coleman, Byron Cook, Mike Gordon, Alexey Gotsman, Tim Harris, Peter O'Hearn, Maurice Herlihy, Tony Hoare, Cliff Jones, Simon Peyton Jones, Andy Pitts, John Reynolds, William Scherer, Marc Shapiro, Peter Sewell, Hongseok Yang, and everybody that I have forgotten to mention, as well as the anonymous referees who were exposed to early drafts of parts of my work.

Finally, I would like to thank my parents, George and Chryssi, my brother, Theodore, my family and my friends.

Contents

Chapter 1

Introduction

Parallelism has always been a challenging domain for processor architecture, programming, and formal methods. Traditionally, a data structure (such as a tree or a hashtable) is made concurrent by protecting it with a single mutual exclusion lock so that only one thread may access the data structure at any time. This approach makes it relatively easy to reason about correctness, but it severely limits parallelism, negating some of the benefits of modern multicore and multiprocessor systems.

Instead, there is a growing trend trying to perform synchronisation between threads at a finer grain, so that multiple threads can update different parts of the data structure at the same time. In order to achieve this, algorithms use sophisticated locking schemes (such as hand-over-hand locking), and non-blocking designs involving compare-and-swap (CAS) instructions and helping. There are already many such algorithms within the research community and they are getting adopted quite widely in the form of concurrent libraries such as `java.util.concurrent`.[1]

As a paradigm, fine-grained concurrency is rather complicated and error-prone. Since efficiency is a prime concern, the programmers use low-level languages (such as C) and avoid established abstractions, such as garbage collection, because of their runtime costs. Synchronisation between competing threads takes place at a finer grain than the invariants that are supposed to be preserved, thereby breaking abstraction, a well-known good programming practice. Consequently, modular reasoning about fine-grained concurrency is extremely difficult.

Being able to reason about such programs modularly is crucial. Since most fine-grained concurrent programs are part of standard libraries, we would like to prove their correctness once and for all, and not need to repeat the proof each time they are used. Most existing proof techniques (e.g. invariants [4], reduction [16, 28, 76], ownership [48, 49, 50], concurrent separation logic [58, 11, 8, 65]) are too simple to deal with fine-grain

[1] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html`

concurrency. More advanced techniques (e.g. temporal logic [67, 54], simulation [25, 18]) are too general and complicated, and usually not modular.

Rely/guarantee [51] is the probably the most suitable of the existing techniques. It is compositional in the sense that we can compose proofs of a program's components to get a proof for the entire program, but it is *not* modular. We cannot give a specification and a proof that is reusable in every valid usage context. This is because the specification of a component must know which variables the other components use, just to say that the component does not interfere with those variables. Since most fine-grained concurrency is in library code, modularity is needed rather than just compositionality.

This dissertation describes a framework that makes reasoning about fine-grained concurrent algorithms tractable. It is based on a combination of rely-guarantee reasoning, separation logic, linearisability, and auxiliary variables.

A key observation is that binary relations are necessary for describing the interface between threads of a concurrent system. Invariants are useful, but they are too weak and inexpressive in practice. More complex descriptions such as finite-state automata and temporal properties are usually unnecessary. This observation is not new: at least Jones and Lamport have made it before, but it seems to have been neglected.

A second observation is that reasoning about concurrent programs is very similar to reasoning about modular sequential programs. A sequential program with modules is essentially a coarse-grained concurrent program with one lock per module. Verification of these two classes of programs raises almost the same issues. The practical difference is that the difficult issues appear much earlier in concurrent programs than they do in modular sequential programs. As a result, most of the findings described in this dissertation apply equally to modular sequential programs.

Finally, saying that some state is private to one thread/component is different than saying that other threads/components cannot write to that state. The former entails the latter, but the converse implication does not hold. For example, a component can deallocate its private state without consulting the other threads. If, however, other components can see (but not update) this state, then we cannot deallocate it without consulting them because another component may be accessing it concurrently. Again, this observation is hardly original, but it has been ignored by many verification methods including rely/guarantee.

1.1 Contributions and dissertation outline

The main contribution of this dissertation is the development of a new logic, RGSep, that subsumes rely/guarantee and separation logic. RGSep enables simple, modular proofs of fine-grained concurrent algorithms that use complex dynamically allocated data struc-

tures and may explicitly deallocate memory. It permits reasoning based on ownership and ownership transfer between threads, while maintaining the expressiveness of binary relations to describe inter-thread interference.

Here is the outline of the dissertation:

- Chapter 2 explains some of the terminology in concurrency and outlines existing verification techniques. It defines common notation, and gives a detailed, uniform introduction to rely/guarantee and to the various versions of separation logic.

- Chapter 3 describes the core of RGSep, proves its soundness, and presents a proof of a fine-grained concurrent list algorithm that explicitly disposes (frees) nodes when they are no longer accessible.

- Chapter 4 addresses peripheral issues related to RGSep, such as when stability needs to be checked, and presents some extensions for further modularity.

- Chapter 5 discusses linearisability, the standard correctness requirement for fine-grained concurrent algorithms, and describes techniques for proving that an algorithm is linearisable.

  The verification techniques are demonstrated by linearisability proofs of a collection of concurrent list and stack algorithms, as well as the RDCSS and MCAS implementations of Harris et al. [34], which are the core components of their software transactional memory (STM) implementation.

- Chapter 6 describes SmallfootRG, a prototype safety checker for fine-grained concurrent programs that is based on RGSep. The tool proves simple safety properties for a number of list and stack algorithms and verifies the absence of memory leaks.

Chapter 2

Technical background

This chapter summarises the terminology and notation used throughout the dissertation and provides a uniform introduction to rely/guarantee and separation logic, the program logics that this work is based on.

The chapter starts with §2.1 giving an overview of shared memory concurrency and introducing some of the terminology associated with it. Next, §2.2 describes a simple parallel programming language, GPPL, syntactic conventions, proof terminology and notation. The following sections (§2.3 and §2.4) constitute the main part of this chapter, as they describe rely/guarantee and separation logic respectively. Finally, §2.5 discusses related proof methods and §2.6 concludes by comparing the various approaches.

2.1 Shared memory concurrency

A concurrent system consists of a number of *processes*, which execute mostly independently of one another, but occasionally *interact* with each other. Depending on the circumstances, inter-process interaction may or may not be desirable. Useful interaction is also known as *communication*, whereas undesirable interaction is called *interference*.

There are two main forms of communication between processes: shared memory and channels. In a channel-based system processes interact by sending values to channels and receiving values from channels. In a shared memory system processes interact by reading from and writing to a shared memory location. This dissertation focuses on the latter model.

A thread is a process in a shared-memory system. Some systems have a fixed number of threads; other systems permit new threads to be created at run-time. Some languages restrict the creation of threads to a nested way: a thread is *forked* into subthreads; when all subthreads terminate, they are *joined* together in one thread. Other languages do not force a nested thread structure. Instead, a new thread can be *spawned* at any time; the

newly created thread executes in parallel and vanishes when it terminates. Often each thread has an identifier, a unique number that distinguishes it from other threads. Some algorithms assume that threads have identifiers, Even if an algorithm does not mention thread identifiers, introducing thread identifiers may assist in its verification (for example, see §3.5).

The *scheduler* decides which process is to execute on which processor. Schedulers were traditionally distinguished into preemptive and non-preemptive depending on whether they can stop (preempt) an executing process in order to select another process to run, or whether each process must voluntarily yield control to the scheduler every so often. Now, most schedulers are preemptive.

Typically, schedulers provide few guarantees that a thread that is ready to execute will be selected for execution. A *weakly fair* scheduler ensures that at any time, if a thread is ready to execute for an infinitely long time sequence, eventually it will be allowed to execute. A *strongly fair* scheduler ensures that from any time onwards, if a thread is able to execute infinitely often, then it will eventually get executed. Weak fairness is easy to realise using a round-robin scheduler, but guaranteeing strong fairness is difficult and impractical. Fairness guarantees that eventually a thread will run, and hence receive unbounded time (modulo Zeno), but does not specify how often or for how long the thread will run.

2.1.1 Synchronisation

In a shared-memory system, some threads are *synchronised*, if they agree on the order that some events will happen. In order to reach this agreement, the threads communicate with each other using the available primitive operations provided by the hardware. For instance, the hardware could provide mutual exclusion locks (mutexes), atomic memory reads or writes, CAS (compare and swap), or memory barriers.

Blocking synchronisation Blocking synchronisation refers to a programming style using mutual exclusion locks (mutexes) to arrange inter-thread synchronisation. Mutual exclusion is a simple protocol where each shared resource has a flag keeping track of whether it is being used or not.

When a thread wants to use a shared resource, it atomically checks that it is not in use and updates the resource's flag to denote that it is now being used. If the resource was in use, then the thread waits (blocks) until that resource becomes available; otherwise, it goes on to use that resource. When it has finished working with the resource, it updates the resource's flag to say that it is now free.

Because mutual exclusion negates parallelism, programmers try to lock only the relevant parts of a shared data structure and use more permissive locking schemes. Hence,

algorithms employ techniques such as inherited locking and lock coupling, and use more sophisticated schemes such as MRSW (multiple readers, single writer) locks.

There is a great number of problems associated with mutexes: deadlock, livelock, starvation, priority inversion, convoy effect. As Harris et al. [35] put it, "locks do not compose." Nevertheless, their use is ubiquitous.

Non-blocking synchronisation Instead of using locks, more advanced algorithms employ other primitive operations such as atomic memory reads and writes, or CAS. Using these operations does not make an algorithm 'non-blocking'; avoiding locks does not make an algorithm 'lock-free.' The terms 'non-blocking' and 'lock-free' have a technical meaning relating to the progress an algorithm makes towards completion. In fact, mutexes can be encoded even with a minimal set of non-blocking primitives.

A synchronisation technique is *non-blocking* if it somehow achieves *progress* even if some threads of the system are descheduled or fail. It is *wait-free* [42] if it ensures that all running threads make progress even when other threads incur arbitrary delay. It is *lock-free* if it ensures that whenever at least one thread is running then some thread makes progress. It is *obstruction-free* [43] if it guarantees progress for any thread that eventually executes in isolation.

Herlihy [42] reduced the existence of wait-free algorithms to a consensus problem and showed that concurrency primitives such as test-and-set were too weak to implement wait-free algorithms, whereas other primitives —notably CAS— are universal. This means that any sequential algorithm can be turned into a wait-free algorithm by using only CAS for synchronisation.

Compare and swap Among the more complex operations, the most common one is *compare and swap*. CAS takes three arguments: a memory address, an expected value and a new value. It atomically reads the memory address and if it contains the expected value, it updates it with the new value; otherwise, it does nothing:

```
bool CAS(value_t *addr, value_t exp, value_t new) {
  atomic {
    if (*addr == exp) { *addr = new; return true; }
    else { return false; }
  }
}
```

Various more elaborate versions of CAS have been proposed, such as DCAS (double CAS) [31], DWCAS (double-width CAS, i.e. 64bit CAS on 32bit architectures), MCAS (multiple CAS, i.e. N-way CAS), and kCSS (k-compare single-swap), but are not so widely available. In any case, they can be constructed from single CAS.

Race conditions A *race condition* occurs when two threads try to access the same shared location at the same time and at least one of the accesses is a write. This is problematic if reading and writing that memory location is not atomic. If a thread reads some memory location while it is been updated by another thread, it may read a corrupted value containing parts of the old value and parts of the new value. Similarly, if two updates to the same location happen at the same time, then the final value of the location might be a mixture of the two values written.

When, however, the relevant memory accesses are atomic, race conditions are effectively a synchronisation technique and can be exploited to build synchronisation primitives such as mutexes. For example, Peterson's algorithm [66] implements mutual exclusion between two threads using atomic reads and writes. With Lamport's bakery algorithm [53], even a single bit atomic read/write suffices to implement mutual exclusion.

Software Transactional Memory Transactional Memory is a programming abstraction introduced by Herlihy [44] for hardware, and then by Shavit and Touitou [70] for software. Following the implementation of Harris and Fraser [33], STM has become quite popular for its simple and effective interface: The programmer writes an atomic block, and for an observer outside of the block the blocks' memory operations appear to have executed atomically.

STM may be implemented by a two-phase locking protocol, or more often by optimistic schemes. In the latter case there are severe restrictions in what actions the program is allowed to perform within atomic blocks. In particular, as the implementation may have to roll-back any statement within the transaction, externally observable effects such as I/O are banned.

STM implementations are usually quite complex and it is quite likely that they contain subtle bugs. Indeed, most STM implementations do not behave as expected if the memory is not statically partitioned to transactional and non-transactional [36]. This dissertation tackles the verification of the sort of algorithms used in STM implementations (for example, MCAS in §5.3.4).

Memory consistency Traditionally, concurrency can be given an interleaving semantics, and it is assumed that aligned single word reads and writes are executed atomically by the hardware. This model, also known as strong consistency, ensures that each thread observes shared memory operations happening in the same order.

Unfortunately, most modern processors do not conform to this model because their caches can cause memory operations to be reordered, and hence different threads can witness shared operations happen in different orders. Processors supporting weaker consistency models have special 'memory barrier' instructions to flush the cache and thereby recover an interleaving semantics.

This dissertation assumes that parallel composition of threads has an interleaving semantics. Although not entirely realistic, the use of interleaving semantics is almost universal in concurrency verification.

2.2 Proof terminology & notation

Programming language In order to provide a uniform presentation to rely/guarantee and the various versions of separation logic, we consider the following minimal imperative programming language, GPPL (standing for Generic Parallel Programming Language). Let C stand for commands, c for basic commands (e.g. assignments), B for boolean expressions, and E for normal integer expressions. Commands, C, are given by the following grammar:

$$
\begin{array}{rll}
C ::= & \textbf{skip} & \text{Empty command} \\
 & c & \text{Basic command} \\
 & C_1; C_2 & \text{Sequential composition} \\
 & C_1 + C_2 & \text{Non-deterministic choice} \\
 & C^* & \text{Looping} \\
 & \langle C \rangle & \text{Atomic command} \\
 & C_1 \| C_2 & \text{Parallel composition} \\
 & & \\
c ::= & \mathsf{assume}(B) & \text{Assume condition} \\
 & \mathtt{x} := E & \text{Variable assignment} \\
 & \ldots &
\end{array}
$$

GPPL is parametric with respect to the set of basic commands and expressions. In Section 2.4, we will see a particular set of basic commands, boolean and integer expressions. The basic command $\mathsf{assume}(B)$ checks whether B holds: if B is true, it reduces to **skip**, otherwise it diverges (loops forever). Since this dissertation discusses only partial correctness, $\mathsf{assume}(B)$ is a convenient way to encode conditionals and while loops:

$$
\begin{aligned}
\textbf{if}(B)\ C_1\ \textbf{else}\ C_2 &\stackrel{\text{def}}{=} (\mathsf{assume}(B); C_1) + (\mathsf{assume}(\neg B); C_2) \\
\textbf{while}(B)\ C &\stackrel{\text{def}}{=} (\mathsf{assume}(B); C)^*; \mathsf{assume}(\neg B)
\end{aligned}
$$

Similarly, we can encode (conditional) critical regions as follows:

$$
\begin{aligned}
\textbf{atomic}\ C &\stackrel{\text{def}}{=} \langle C \rangle \\
\textbf{atomic}(B)\ C &\stackrel{\text{def}}{=} \langle \mathsf{assume}(B); C \rangle
\end{aligned}
$$

$$\frac{}{(\mathbf{skip}; C_2), \sigma \to C_2, \sigma} \text{ (Seq1)} \qquad \frac{}{C^*, \sigma \to (\mathbf{skip} + (C; C^*)), \sigma} \text{ (Loop)}$$

$$\frac{C_1, \sigma \to C_1', \sigma'}{(C_1; C_2), \sigma \to (C_1'; C_2), \sigma'} \text{ (Seq2)} \qquad \frac{C, \sigma \to^* \mathbf{skip}, \sigma'}{\langle C \rangle, \sigma \to \mathbf{skip}, \sigma'} \text{ (Atom)}$$

$$\frac{}{(C_1 + C_2), \sigma \to C_1, \sigma} \text{ (Cho1)} \qquad \frac{C_1, \sigma \to C_1', \sigma'}{(C_1 \| C_2), \sigma \to (C_1' \| C_2), \sigma'} \text{ (Par1)}$$

$$\frac{}{(C_1 + C_2), \sigma \to C_2, \sigma} \text{ (Cho2)} \qquad \frac{C_2, \sigma \to C_2', \sigma'}{(C_1 \| C_2), \sigma \to (C_1 \| C_2'), \sigma'} \text{ (Par2)}$$

$$\frac{B(\sigma)}{\mathsf{assume}(B), \sigma \to \mathbf{skip}, \sigma} \text{ (Assume)} \qquad \frac{}{(\mathbf{skip} \| \mathbf{skip}), \sigma \to \mathbf{skip}, \sigma} \text{ (Par3)}$$

Figure 2.1: Small-step operational semantics of GPPL.

Figure 2.1 contains the small-step operational semantics of GPPL. Since we treat composition as interleaving, the semantics are pretty straightforward. Configurations of the system are just pairs (C, σ) of a command and a state; and we have transitions from one configuration to another.

According to Atom, atomic commands execute all the commands in its body, C, in one transition. In the premise, $\to^*$ stands for zero or more $\to$ transitions. There is an issue as to what happens when the body C of atomic command does not terminate. According to the semantics of Figure 2.1, no transition happens at all. This cannot be implemented, because one would effectively need to solve the halting problem. So, more realistically, one should add a rule saying that if C diverges then $\langle C \rangle$ may diverge. In the context of this dissertation, the body of atomic commands will always be a short instruction, such as a single memory read or write or a CAS, which always terminates.

The other rules are pretty straightforward. We use the rule Par3 instead of the rule $(C \| \mathbf{skip}), \sigma \to C, \sigma$ because it simplifies the statements of the lemmas in §3.3.

Finally, instances of GPPL will have rules for each primitive command, c. These primitive commands, c, need not execute atomically. As a convention, if the correctness of an algorithm depends on some primitive command's atomic execution, then this command will be enclosed in angle brackets, $\langle c \rangle$. This way, the algorithms make explicit any atomicity requirements they have.

Variables First, we must distinguish between *logical* variables and *program* variables. Logical variables are used in assertions, have a constant value, and may be quantified over. Program variables appear in programs, and their values can be changed with assignments.

An *auxiliary variable* [62] is a program variable that does not exist in the program itself, but is introduced in order to prove the program's correctness. Auxiliary variables do not affect the control-flow or the data-flow of the outputs, but play an important role

in reasoning: they allow one to abstract over the program counters of the other threads, and are used to embed the specification of an algorithm in its implementation. Since they do not physically get executed, they can be grouped with the previous or the next atomic instruction into one atomic block.

The simplest form of auxiliary variable is a *history* variable: a variable introduced to record some information about the past program state that is not preserved in the current state. There is also the dual concept of a *prophecy* variable that Abadi and Lamport [1] introduced to capture a finite amount of knowledge about the future execution of the program.

Auxiliary variables are also known as dummy variables or ghost variables, but the last term is ambiguous. A ghost variable is also a logical variable used in the precondition and postcondition of a Hoare triple in order to relate the initial and the final values of some program variables. For clarity, it is better to avoid this term altogether. The collection of all auxiliary variables is known as *auxiliary state*, whereas *auxiliary code* stands for the introduced assignment statements to the auxiliary variables.

Relations The rely/guarantee specifications use binary relations on states in order to specify how the state may change by (part of) a program. Here is a summary of the relational notation.

Predicates P of a single state σ describe a set of system states, whereas binary relations describe a set of actions (i.e. transitions) of the system. These are two-state predicates that relate the state σ just after the action to the state just before the action, which is denoted as $\overleftarrow{\sigma}$. Similarly, let $\overleftarrow{x}$ and x denote the value of the program variable x before and after the action respectively.

Given a single-state predicate P, we can straightforwardly define a corresponding two-state predicate, which requires P to hold in the new state σ, but places no constraint on the old state $\overleftarrow{\sigma}$. We denote this relation by simply overloading P. Similarly, we shall write $\overleftarrow{P}$ for the two-state predicate that is formed by requiring P to hold in the old state $\overleftarrow{\sigma}$ and which places no requirement on the new state σ.

$$\begin{aligned} P(\overleftarrow{\sigma}, \sigma) &\stackrel{\text{def}}{=} P(\sigma) \\ \overleftarrow{P}(\overleftarrow{\sigma}, \sigma) &\stackrel{\text{def}}{=} P(\overleftarrow{\sigma}) \end{aligned}$$

Relational notation abbreviates operations on predicates of two states. So, for example $P \wedge Q$ is just shorthand for $\lambda(\overleftarrow{\sigma}, \sigma).\ P(\overleftarrow{\sigma}, \sigma) \wedge Q(\overleftarrow{\sigma}, \sigma)$. Relational composition of predicates describes exactly the intended behaviour of the sequential composition of sequential programs.

$$(P; Q)(\overleftarrow{\sigma}, \sigma) \stackrel{\text{def}}{=} \exists \tau.\ P(\overleftarrow{\sigma}, \tau) \wedge Q(\tau, \sigma)$$

The program that makes no change to the state is described exactly by the identity relation,

$$\mathrm{ID}(\overleftarrow{\sigma}, \sigma) \stackrel{\text{def}}{=} (\overleftarrow{\sigma} = \sigma).$$

Finally, the familiar notation R^* (reflexive and transitive closure) represents any finite number of iterations of the program described by R. It is defined by:

$$R^* \stackrel{\text{def}}{=} \mathrm{ID} \vee R \vee (R; R) \vee (R; R; R) \vee \cdots$$

2.3 Rely/guarantee reasoning

Rely/guarantee is a compositional verification method for shared memory concurrency introduced by Jones [51]. Jones's insight was to describe interference between threads using binary relations. In fact, Jones also had relational postconditions because procedure specifications typically relate the state after the call to the state before the call.

Other researchers [72, 77, 68], in line with traditional Hoare logic, used postconditions of a single state. With single-state postconditions, we can still specify such programs, but we need to introduce a (ghost) logical variable that ties together the precondition and the postcondition. Usually, the proof rules with single-state postconditions are simpler, but the assertions may be messier, because of the need to introduce (ghost) logical variables.

Whether the postcondition should be a single-state predicate or a binary relation is orthogonal to the essence of rely-guarantee method, which is describing interference, but nevertheless important. In this section, following Jones [51] we shall use relational postconditions. In the combination with separation logic, for simplicity, we shall fall back to postconditions of a single state.

There is a wide class of related verification methods (e.g. [56, 15, 2, 40, 41, 23]), which are collectively known as assume-guarantee. These methods differ in their application domain and interference specifications.

Owicki-Gries

The Rely/Guarantee method can be seen as a compositional version of the Owicki-Gries method [62]. In her PhD, Owicki [61] came up with the first tractable proof method for concurrent programs. A standard sequential proof is performed for each thread; the parallel rule requires that each thread does not 'interfere' with the proofs of the other threads.

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\} \qquad (\dagger)}{\{P_1 \wedge P_2\}\ C_1 \| C_2\ \{Q_1 \wedge Q_2\}} \qquad \text{(Owicki-Gries)}$$

where (†) is the side-condition requiring that C_1 does not interfere with the proof of C_2 and vice versa. This means that every intermediate assertion between atomic actions in the proof outline of C_2 must be preserved by all atomic actions of C_1 and vice versa. Clearly, this is a heavy requirement and the method is not compositional.

Specifications

Rely/guarantee reasoning [51] is a compositional method based on the Owicki-Gries method. The specifications consist of four components (P, R, G, Q).

- The predicates P and Q are the *pre-condition* and *post-condition*. They describe the behaviour of the thread as a whole, from the time it starts to the time it terminates (if it does). The pre-condition P, a single-state predicate, describes an assumption about the initial state that must hold for the program to make sense. The post-condition Q is a two-state predicate relating the initial state (just before the program starts execution) to the final state (immediately after the program terminates). The post-condition describes the overall effect of the program to the state.

- R and G summarise the properties of the individual atomic actions invoked by the environment (in the case of R) and the thread itself (in the case of G). They are two-state predicates, relating the state $\overleftarrow{\sigma}$ before each individual atomic action to σ, the one immediately after that action. The *rely condition* R models all atomic actions of the environment, describing the interference the program can tolerate from its environment. Conversely, the *guarantee condition* G models the atomic actions of the program, and hence it describes the interference that it imposes on the other threads of the system.

There is a well-formedness condition on rely/guarantee specifications: the precondition and the postcondition must be stable under the rely condition, which means that they are resistant to interference from the environment. Coleman and Jones [17] have stability as an implicit side-condition at every proof rule. This is, however, unnecessary. Here, following Prensa [68], we will check stability only at the atomic block rule. (There are futher possibilities as to where stability is checked: these will be presented in Section 4.1.)

Definition 1 (Stability)**.** *A binary relation Q is* stable under *a binary relation R if and only if $(R; Q) \Rightarrow Q$ and $(Q; R) \Rightarrow Q$.*

The definition says that doing an environment step before or after Q should not make Q invalid. Hence, by induction, if Q is stable, then doing any number of environment transitions before and after Q should not invalidate Q. For single state predicates, these checks can be simplified, and we get the following lemma.

Lemma 2. *A single state predicate P is* stable under *a binary relation R if and only if* $(P(\overleftarrow{\sigma}) \wedge R(\overleftarrow{\sigma}, \sigma)) \Rightarrow P(\sigma)$.

When two threads are composed in parallel, the proof rules require that the guarantee condition of the one thread implies the rely condition of the other thread and vice versa. This ensures that the component proofs do not interfere with each other.

Proof rules

We turn to the rely/guarantee proof rules for GPPL, the simple programming language introduced in §2.2. Let $C\ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, Q)$ stand for the judgement that the command C meets the specification (P, R, G, Q).

The first rule allows us to weaken a specification. A stronger specification is possibly more desirable but more difficult to meet. A specification is weakened by weakening its obligations (the postcondition and the guarantee condition) and strengthened by weakening its assumptions (the precondition and the rely condition). When developing a program from its specification, it is always valid to replace the specification by a stronger one.

$$\frac{\begin{array}{c} C\ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, Q) \\ P' \Rightarrow P \qquad R' \Rightarrow R \qquad G \Rightarrow G' \qquad Q \Rightarrow Q' \end{array}}{C\ \mathsf{sat}_{\mathsf{RG}}\ (P', R', G', Q')} \qquad \text{(RG-Weaken)}$$

The following rule exploits the relational nature of the postcondition and allows us to strengthen it. In the postcondition, we can can always assume that the precondition held at the starting state, and that the program's effect was just some arbitrary interleaving of the program and environment actions.

$$\frac{C\ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, Q)}{C\ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, Q \wedge \overleftarrow{P} \wedge (G \vee R)^*)} \qquad \text{(RG-AdjustPost)}$$

Then, we have a proof rules for each type of command, C. The rules for **skip**, sequential composition, non-deterministic choice and looping are straightforward. In the sequential composition rule, note that the total effect, $Q_1; Q_2$, is just the relational composition of the two postconditions.

$$\frac{}{\mathbf{skip}\ \mathsf{sat}_{\mathsf{RG}}\ (\mathit{true}, R, G, \mathsf{ID})} \qquad \text{(RG-Skip)}$$

$$\frac{\begin{array}{l} C_1\ \mathsf{sat}_{\mathsf{RG}}\ (P_1, R, G, (Q_1 \wedge P_2)) \\ C_2\ \mathsf{sat}_{\mathsf{RG}}\ (P_2, R, G, Q_2) \end{array}}{(C_1; C_2)\ \mathsf{sat}_{\mathsf{RG}}\ (P_1, R, G, (Q_1; Q_2))} \qquad \text{(RG-Seq)}$$

$$\frac{C_1 \ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, Q) \qquad C_2 \ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, Q)}{(C_1 + C_2) \ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, Q)} \tag{RG-Choice}$$

$$\frac{C \ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, (Q \wedge P))}{C^* \ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, Q^*)} \tag{RG-Loop}$$

The rules for atomic blocks and parallel composition are more interesting. The atomic rule checks that the specification is well formed, namely that P and Q are stable under interference from R, and ensures that the atomic action satisfies the guarantee condition G and the postcondition Q. Because $\langle C \rangle$ is executed atomically, we do not need to consider any environment interference within the atomic block. That is why we check C with the identity rely condition.

$$\frac{(P; R) \Rightarrow P \qquad (R; Q) \Rightarrow Q \qquad (Q; R) \Rightarrow Q \qquad C \ \mathsf{sat}_{\mathsf{RG}}\ (P, \mathrm{ID}, \mathsf{True}, (Q \wedge G))}{\langle C \rangle \ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, Q)} \tag{RG-Atomic}$$

When composing two threads in parallel, we require that each thread is immune to interference by all the other threads. So, the thread C_1 can get interfered by the thread C_2 or by environment of the parallel composition. Hence, its rely condition must account for both possibilities, which is represented as $R \vee G_2$. Conversely, C_2's rely condition is $R \vee G_1$. Initially, the preconditions of both threads must hold; at the end, if both threads terminate, then both postconditions will hold. This is because both threads will have established their postcondition, and as each postcondition is stable under interference, so both will hold for the entire composition. Finally, the total guarantee is $G_1 \vee G_2$, because each atomic action belongs either to the first thread or the second.

$$\frac{C_1 \ \mathsf{sat}_{\mathsf{RG}}\ (P, (R \vee G_2), G_1, Q_1) \qquad C_2 \ \mathsf{sat}_{\mathsf{RG}}\ (P, (R \vee G_1), G_2, Q_2)}{(C_1 \| C_2) \ \mathsf{sat}_{\mathsf{RG}}\ (P, R, (G_1 \vee G_2), (Q_1 \wedge Q_2))} \tag{RG-Par}$$

Soundness and completeness

In line with the rest of the dissertation this section presented rely/guarantee proof rules for partial correctness. There is an alternative rule for loops that proves environment-independent termination. If the proof of the termination of a thread depends on the the termination of its environment, we quickly run into circular reasoning, which is generally unsound. Abadi and Lamport [2] gave a condition under which such circular reasoning is sound, and showed that all safety proofs trivially satisfy this condition.

Prensa [68] formalised a version of rely/guarantee rules (with a single-state postcondition) in Isabelle/HOL and proved their soundness and relative completeness. More recently, Coleman and Jones [17] presented a structural proof of soundness for the rules with relational postconditions.

The rely/guarantee rules are intentionally incomplete: they model interference as a relation, ignoring the environment's control flow. Hence, they cannot directly prove properties that depend on the environment's control flow. Nevertheless, we can introduce auxiliary variables to encode the implicit control flow constraints, and use these auxiliary variables in the proof. Modulo introducing auxiliary variables, rely/guarantee is complete. The various completeness proofs [68] introduce an auxiliary variable that records the entire execution history. Of course, introducing such an auxiliary variable has a global effect on the program to be verified. Therefore, the completeness result does not guarantee that a modular proof can be found for every program.

2.4 Separation logic

Separation logic [69, 47] is a program logic with a built-in notion of a resource, and is based on the logic of bunched implications (BI) [59]. Its main application so far has been reasoning about pointer programs that keep track of the memory they use and explicitly deallocate unused memory.

As separation logic is a recent development, there are various versions of the logic with complementary features, but there is not yet a standard uniform presentation of all these. The survey paper by Reynolds [69] is probably the best introduction to separation logic, but does not describe some of the more recent developments (e.g. permissions, and 'variables as resource') that are mentioned below.

Below we will consider an abstract version of separation logic influenced by Calcagno, O'Hearn, and Yang [13]. By instantiating this abstract separation logic, we can derive the various existing versions of separation logic.

2.4.1 Abstract separation logic

Resources are elements of a cancellative, commutative, partial monoid $(M, \odot, \mathsf{u})$, where the operator $\odot$ represents the addition of two resources. Adding two resources is a partial operation because some models forbid having two copies of the same resource; hence, $m \odot m$ might be undefined. Clearly enough, addition is commutative and associative and has an identity element: the empty resource u. It is also cancellative, because we can subtract a resource from a larger resource that contains it.

These properties are expressed in the following definition of a resource algebra.

Definition 3. *A resource algebra $(M, \odot, \mathsf{u})$ consists of a set M equipped with a partial binary operator $\odot : M \times M \rightharpoonup M$ and a distinguished element $\mathsf{u} \in M$, such that for all $m_1, m_2, m_3 \in M$ the following properties hold:*

$$
\begin{array}{ll}
m \odot \mathsf{u} = \mathsf{u} \odot m = m & \textit{(identity element)} \\
(m_1 \odot m_2) \odot m_3 = m_1 \odot (m_2 \odot m_3) & \textit{(associativity)} \\
m_1 \odot m_2 = m_2 \odot m_1 & \textit{(commutativity)} \\
(\textit{defined}(m_1 \odot m_2) \wedge m_1 \odot m_2 = m_1 \odot m_3) \implies m_2 = m_3 & \textit{(cancellation)}
\end{array}
$$

The restriction on having one unit element u can be relaxed to a set $\mathsf{U} \subseteq M$ of unit elements such that for all $m \in M$, there exists $u \in \mathsf{U}$ such that $m \odot u = u \odot m = m$ and for all $u' \in \mathsf{U} \setminus \{u\}$, $m \odot u'$ is undefined.

The structure of the commutative monoid induces a partial order, $\sqsubseteq$, defined as follows,

$$m_1 \sqsubseteq m_2 \overset{\text{def}}{\iff} \exists m'.\ m_2 = m_1 \odot m'.$$

Informally, m_1 is smaller than m_2 if it contains fewer resources. From the definition, u is the smallest element of the set, that is $\forall m.\ \mathsf{u} \sqsubseteq m$.

In order that reasoning about resources makes sense, the semantics of programs must respect resources. Separation logic requires that every command C obeys the following two locality conditions:

Definition 4 (Locality)**.**

- *If $(C, s_1 \odot s) \rightarrow^* \textit{fault}$, then $(C, s_1) \rightarrow^* \textit{fault}$.*
- *If $(C, s_1 \odot s) \rightarrow^* (\mathbf{skip}, s_2)$, then either there exists s_2' such that $(C, s_1) \rightarrow^* (\mathbf{skip}, s_2')$ and $s_2 = s \odot s_2'$, or $C, s_1 \rightarrow^* \textit{fault}$.*

The first property is equivalent to safety monotonicity, which says that a non-faulting program on a small state s_1 also does not fault on a larger state $s_1 \odot s$. The second is the frame property of commands: when a command that runs successfully with state s_1 is executed with a larger state $s_1 \odot s$, it does not depend on the additional state s and it does not modify s.

It is sufficient that all primitive commands, c, have the locality property. Then, by construction, all larger programs will have the locality property.

Assertions Separation logic assertions are given by the following grammar,

$$
\begin{array}{rl}
P, Q ::= & \textit{true} \mid \textit{false} \mid \textit{PrimP} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid P \Leftrightarrow Q \mid \neg P \\
& \mid \textit{emp} \mid P * Q \mid P \mathrel{-\!\!*} Q \mid P \mathrel{-\!\!\circledast} Q \\
& \mid \exists x.\ P \mid \forall x.\ P
\end{array}
$$

$$
\begin{array}{lcl}
m, i \models_{\mathsf{SL}} \mathit{true} & \iff & \mathit{always} \\
m, i \models_{\mathsf{SL}} \mathit{false} & \iff & \mathit{never} \\
m, i \models_{\mathsf{SL}} P \wedge Q & \iff & (m, i \models_{\mathsf{SL}} P) \wedge (m, i \models_{\mathsf{SL}} Q) \\
m, i \models_{\mathsf{SL}} P \vee Q & \iff & (m, i \models_{\mathsf{SL}} P) \vee (m, i \models_{\mathsf{SL}} Q) \\
m, i \models_{\mathsf{SL}} P \Rightarrow Q & \iff & (m, i \models_{\mathsf{SL}} P) \implies (m, i \models_{\mathsf{SL}} Q) \\
m, i \models_{\mathsf{SL}} P \Leftrightarrow Q & \iff & (m, i \models_{\mathsf{SL}} P) \iff (m, i \models_{\mathsf{SL}} Q) \\
m, i \models_{\mathsf{SL}} \neg P & \iff & \neg (m, i \models_{\mathsf{SL}} P) \\
m, i \models_{\mathsf{SL}} \mathit{emp} & \iff & m = \mathsf{u} \\
m, i \models_{\mathsf{SL}} P * Q & \iff & \exists m_1 m_2.\ (m_1 \odot m_2 = m) \wedge (m_1, i \models_{\mathsf{SL}} P) \wedge (m_2, i \models_{\mathsf{SL}} Q) \\
m, i \models_{\mathsf{SL}} P \mathrel{-\!\!*} Q & \iff & \forall m_1 m_2.\ ((m_1 \odot m = m_2) \wedge (m_1, i \models_{\mathsf{SL}} P)) \implies (m_2, i \models_{\mathsf{SL}} Q) \\
m, i \models_{\mathsf{SL}} P \mathrel{-\!\!\circledast} Q & \iff & \exists m_1 m_2.\ (m_1 \odot m = m_2) \wedge (m_1, i \models_{\mathsf{SL}} P) \wedge (m_2, i \models_{\mathsf{SL}} Q) \\
m, i \models_{\mathsf{SL}} \exists x.\ P & \iff & \exists v \in \mathsf{Val}.\ (m, (i \uplus \{x \mapsto v\}) \models_{\mathsf{SL}} P) \\
m, i \models_{\mathsf{SL}} \forall x.\ P & \iff & \forall v \in \mathsf{Val}.\ (m, (i \uplus \{x \mapsto v\}) \models_{\mathsf{SL}} P)
\end{array}
$$

Figure 2.2: Semantics of separation logic assertions

The first line lists the connectives of classical propositional logic, while the second line contains the new assertion forms pertinent to BI [59]: empty state (*emp*), separating conjunction ($*$), magic wand ($-\!\!*$), and septraction ($-\!\!\circledast$). Finally, the assertions include first order quantification.

The Kripke semantics of the logic are given in Figure 2.2. A model consists of an element of a resource algebra ($m \in M$) and an interpretation for the logical variables ($i : \mathsf{LogVar} \rightharpoonup \mathsf{Val}$). The well-known operators $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, and $\neg$ have their standard, classical meaning.

- *emp* asserts that the resource is empty; namely $m = \mathsf{u}$, where u is the identity element of M.

- $P * Q$ states that the resource can be divided into two disjoint parts m_1 and m_2 such that m_1 satisfies P and m_2 satisfies Q. This is also known as multiplicative conjunction. There is also an iterated version of separating conjunction, which will be denoted as $\circledast_{1 \leq i \leq n} P_i \stackrel{\text{def}}{=} P_1 * \ldots * P_n$.

- $P \mathrel{-\!\!*} Q$ asserts that for all disjoint resource extensions satisfying P, the combination of the resource and the extension satisfies Q.

- $P \mathrel{-\!\!\circledast} Q$ is satisfied by the difference between two resources, the bigger one satisfying Q and the smaller one satisfying P. Septraction can be defined in terms of magic wand, $P \mathrel{-\!\!\circledast} Q \iff \neg(P \mathrel{-\!\!*} \neg Q)$. This operator does not generally appear in papers discussing separation logic, but will be used extensively in this dissertation.[1]

[1] Similar operators have been previously used in context logic [78] and ambient logic. In context logic, septraction corresponds to $\blacktriangleleft$, whereas in ambient logic it corresponds to fusion ($\propto$). The name 'septraction' is due to Matthew Parkinson.

There are a few classes of assertions have useful additional properties. *Pure* assertions do not specify the resource (m), but only the interpretation of logical variables (i). *Intuitionistic* assertions specify a lower bound on the resource: if a resource m satisfies an intuitionistic assertion P, then any bigger resource m' satisfies P. *Exact* assertions specify exactly one resource. Given a *precise* assertion P, for any resource m, there is at most one sub-resource of m that satisfies P.

Definition 5 (Pure assertions)**.** *An assertion P is* pure *if and only if* $\forall i m_1 m_2.\ (m_1, i \models_{\mathsf{SL}} P) \iff (m_2, i \models_{\mathsf{SL}} P)$.

Definition 6 (Intuitionistic assertions)**.** *An assertion P is* intuitionistic *if and only if* $\forall i m m'.\ m \sqsubseteq m' \wedge (m, i \models_{\mathsf{SL}} P) \implies (m', i \models_{\mathsf{SL}} P)$.

Definition 7 (Exact assertions)**.** *An assertion P is* (strictly) exact *if and only if* $\forall i m_1 m_2.\ (m_1, i \models_{\mathsf{SL}} P) \wedge (m_2, i \models_{\mathsf{SL}} P) \implies m_1 = m_2$.

Definition 8 (Precise assertions)**.** *An assertion P is* precise *if and only if* $\forall i m m_1 m_2.\ m_1 \sqsubseteq m \wedge m_2 \sqsubseteq m \wedge (m_1, i \models_{\mathsf{SL}} P) \wedge (m_2, i \models_{\mathsf{SL}} P) \implies m_1 = m_2$.

Precise assertions are very important for concurrent separation logic (see §2.4.3).

Proof rules

In separation logic, programs are specified by Hoare triples, $\{P\}\ C\ \{Q\}$, which have a fault-avoidance partial correctness interpretation. This means that if the precondition P holds for the initial state, then the command C executes properly without faults (such as accessing unallocated memory). Moreover, if C terminates, the postcondition Q holds for the final state.

The novelty of separation logic is its frame rule:

$$\frac{\vdash_{\mathsf{SL}} \{P\}\ C\ \{Q\}}{\vdash_{\mathsf{SL}} \{P * R\}\ C\ \{Q * R\}} \qquad \text{(SL-Frame)}$$

This rule says that if a command C safely executes in an initial state satisfying P and produces a final state satisfying Q, then it also executes safely if additional resource R is present. The command C, if it terminates, will not use the additional resource R and, hence, R will still exist at the end. The soundness of the frame rule relies on using separating conjunction ($*$) to express disjointness of resources. Proving its soundness, however, is not at all obvious, because the additional state R may restrict the possible executions of C. For instance, if C allocates some new resource, then the existence of R restricts the resource allocator not to return a resource already present in R.

The following proof rules are the standard Floyd-Hoare rules for the empty program, sequential composition, non-deterministic choice, and loops.

$$\frac{}{\vdash_{\mathsf{SL}} \{P\}\ \mathbf{skip}\ \{P\}} \qquad \text{(SL-Skip)}$$

$$\frac{\vdash_{\mathsf{SL}} \{P\}\ C_1\ \{Q\} \qquad \vdash_{\mathsf{SL}} \{Q\}\ C_2\ \{R\}}{\vdash_{\mathsf{SL}} \{P\}\ C_1; C_2\ \{R\}} \qquad \text{(SL-Seq)}$$

$$\frac{\vdash_{\mathsf{SL}} \{P\}\ C_1\ \{Q\} \qquad \vdash_{\mathsf{SL}} \{P\}\ C_2\ \{Q\}}{\{P\}\ C_1 + C_2\ \{Q\}} \qquad \text{(SL-Choice)}$$

$$\frac{\vdash_{\mathsf{SL}} \{P\}\ C\ \{P\}}{\vdash_{\mathsf{SL}} \{P\}\ C^*\ \{P\}} \qquad \text{(SL-Loop)}$$

It is always valid to prove a stronger specification that the one required. To weaken a specification, we either strengthen the precondition or weaken the postcondition.

$$\frac{P' \Rightarrow P \qquad \vdash_{\mathsf{SL}} \{P\}\ C\ \{Q\} \qquad Q \Rightarrow Q'}{\vdash_{\mathsf{SL}} \{P'\}\ C\ \{Q'\}} \qquad \text{(SL-Conseq)}$$

Finally, here are the rules for disjoint concurrency. These rules state that if two threads require disjoint resources to execute (cf. the meaning of separating conjunction), then they can execute safely in parallel. If and when both terminate, each thread will have own some resource, which will be disjoint from the other thread's resource. Hence, at the postcondition, we can use separating conjunction to combine the two postconditions.

$$\frac{\vdash_{\mathsf{SL}} \{P\}\ C\ \{Q\}}{\vdash_{\mathsf{SL}} \{P\}\ \langle C \rangle\ \{Q\}} \qquad \text{(SL-DisjAtomic)}$$

$$\frac{\vdash_{\mathsf{SL}} \{P_1\}\ C_1\ \{Q_1\} \qquad \vdash_{\mathsf{SL}} \{P_2\}\ C_2\ \{Q_2\}}{\vdash_{\mathsf{SL}} \{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}} \qquad \text{(SL-Par)}$$

If one accepts that the programs $C \| \mathbf{skip}$ and C are equivalent, then the frame rule can be derived from Par and Skip. Alternatively, if we are given the frame rule, we can derive Skip from the simpler axiom $\{emp\}\ \mathbf{skip}\ \{emp\}$ by applying the frame rule.

2.4.2 Instances of abstract separation logic

The definitions of GPPL and abstract separation logic were parametric with respect to primitive commands c and primitive assertions $PrimP$ respectively. Below we will consider concrete instantiations for these primitive commands and assertions.

Let (heap-reading) expressions E be given by the grammar,

$$E ::= \mathtt{x} \mid x \mid n \mid [E] \mid E + E \mid E - E \mid \ldots$$

These consist of program variables, logical variables, constants, memory dereferences, and arithmetic operations. The syntax $[E]$ indicates that we are reading the value stored at memory location E. In the examples, when E points to some data structure with fields, we shall use $E.\mathit{field}$ as shorthand for $[E + \mathsf{offset\_of}(\mathit{field})]$. Having expressions with memory dereferences in assertions is somewhat problematic. Hence, we define *pure* expressions, e, to be those expressions that do not dereference memory; that is,

$$e ::= \mathtt{x} \mid x \mid n \mid e + e \mid e - e \mid \dots$$

Boolean expressions, B, that appear in programs are

$$B ::= B \wedge B \mid B \vee B \mid \neg B \mid E = E \mid E < E \mid E \leq E \mid \dots$$

Boolean expressions do not, in general, belong to the grammar of assertions because they can contain expressions with memory dereferences. Instead we can define two mappings from boolean expressions into assertions: (i) $\mathsf{def}(B)$ is the weakest assertion guaranteeing that the evaluation of B is defined (enough resources are available). (ii) $\mathsf{assn}(B)$ is the assertion equivalent to B.

Finally, primitive commands, ranged over by c, are given by the following grammar:

$c ::=$	$\mathsf{assume}(B)$	Assume boolean condition
	$\mathtt{x} := e$	Variable assignment
	$\mathtt{x} := [e]$	Memory read
	$[e] := e$	Memory write
	$\mathtt{x} := \mathtt{new}()$	Allocate memory cell
	$\mathtt{dispose}(e)$	Deallocate memory cell

Heaps First, consider the standard stack and heap model. This is the initial and the most widely used model for separation logic, but it is somewhat flawed in its treatment of stack variables (for details, see discussion at the end of this section).

Assume we have two fixed countably infinite sets: $\mathsf{VarName}$ consisting of variable names, and Loc consisting of locations (addresses in the heap). Let Val stand for the set of values, and define the resource algebra $(M, \odot, \mathsf{U})$ to be

$$\begin{aligned} M &= (\mathsf{VarName} \rightharpoonup \mathsf{Val}) \times (\mathsf{Loc} \rightharpoonup \mathsf{Val}) \\ \odot &= \begin{cases} \lambda(s_1, h_1)(s_2, h_2).\ (s_1.h_1 \uplus h_2) & \text{if} s_1 = s_2, \\ \mathit{undefined} & \text{if} s_1 \neq s_2 \end{cases} \\ \mathsf{U} &= \{(s, \emptyset) \mid s \in \mathsf{VarName} \rightharpoonup \mathsf{Val}\} \end{aligned}$$

The first component of M is known as the stack, and the second as the heap. Primitive

assertions in this model are:

$$PrimP ::= e_1 \mapsto e_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 < e_2 \mid e_1 \leq e_2 \mid \cdots$$

The first assertion says that the heap consists of exactly one memory cell with address e_1 and contents e_2. The other assertions are equalities and inequalities between pure expressions.

In terms of the Kripke model, their semantics is:

$$\begin{array}{lcl}
(s,h), i \models_{\mathsf{SL}} e_1 \mapsto e_2 & \stackrel{\text{def}}{\iff} & dom(h) = \{[\![e_1]\!]\, s\, i\} \ \wedge\ h([\![e_1]\!]\, s\, i) = [\![e_2]\!]\, s\, i \\
(s,h), i \models_{\mathsf{SL}} e_1 = e_2 & \stackrel{\text{def}}{\iff} & [\![e_1]\!]\, s\, i = [\![e_2]\!]\, s\, i \\
(s,h), i \models_{\mathsf{SL}} e_1 \neq e_2 & \stackrel{\text{def}}{\iff} & [\![e_1]\!]\, s\, i \neq [\![e_2]\!]\, s\, i
\end{array}$$

where $[\![e]\!]\, s\, i$ evaluates the pure expression e in the stack s and the interpretation i.

It is customary –but not necessary– to take Loc to be the set of positive integers. This way we can reason about programs that involve pointer arithmetic. Hence, we can define the following shorthand notation for describing multiple adjacent cells.

$$e \mapsto (e_1, e_2, \ldots, e_n) \stackrel{\text{def}}{=} e \mapsto e_1 * (e+1) \mapsto e_2 * \ldots * (e+n-1) \mapsto e_n$$

For clarity, sometimes we use field notation. Assume we have a finite set of field names and a mapping from field names to offsets in the structure they describe and that objects are allocated at aligned memory addresses. We shall use the following shorthand notation.

$$\begin{array}{rcl}
e_1 \mapsto \{.\mathit{field}{=}e_2\} & \stackrel{\text{def}}{=} & (e_1 {+} \mathsf{offset_of}(\mathit{field})) \mapsto e_2 \wedge (e \ \mathbf{mod}\ \mathsf{obj_size} = 0) \\
e \mapsto \{.\mathit{field}_1{=}e_1, \ldots, .\mathit{field}_n{=}e_n\} & \stackrel{\text{def}}{=} & e \mapsto \{.\mathit{field}_1{=}e_1\} * \ldots * e \mapsto \{.\mathit{field}_n{=}e_n\}
\end{array}$$

Finally, we write an underscore (_) in the place of an expression whose value we do not care about. Formally, this is existential quantification. For example, $e \mapsto \_$ stands for $\exists x.\, e \mapsto x$ where $x \notin \mathsf{fv}(e)$.

We turn to the axioms for the stack & heap model of separation logic. They are known as the small axioms, because they deal with the smallest heap affected by command. If there is more heap present, the frame rule says that it remains unaffected.

- Variable assignment are treated by the standard Hoare axiom, where $Q[e/\mathtt{x}]$ substitutes e for all occurrences of $\mathtt{x}$ in Q.

$$\{Q[e/\mathtt{x}]\}\ \mathtt{x} := e\ \{Q\}$$

- To write to a heap cell that cell must exist in the heap:

$$\{e \mapsto \_\}\ [e] := e'\ \{e \mapsto e'\}$$

- Similarly, to read a cell $[e]$, separation logic requires the thread owns the cell; its contents are copied into variable `x`; the cell's contents are unchanged; afterwards, the thread still owns it. (The logical variable y is used to handle the case when `x` occurs in e.)

$$\{e = y \wedge e \mapsto z\}\ \mathtt{x} := [e]\ \{y \mapsto z \wedge \mathtt{x} = z\}$$

- $\mathtt{cons}(e_1, \ldots, e_n)$ allocates a new block of n heap cells. The heap is initially empty; at the end, it contains the new block of cells.

$$\{emp\}\ \mathtt{x} := \mathtt{cons}(e_1, \ldots, e_n)\ \{\mathtt{x} \mapsto (e_1, \ldots, e_n)\}$$

- $\mathtt{dispose}(e)$ deallocates a heap cell. The heap initially contains the cell being disposed; after disposal it is no longer contained in the heap.

$$\{e \mapsto \_\}\ \mathtt{dispose}(e)\ \{emp\}$$

Permissions Permissions arose as an extension to the standard heap model, to enable read-sharing between parallel threads. Boyland [10] defined an early model. Then, Bornat et al. [8] provided a general model for permissions and two instances of that model, and Parkinson [63] gave a generic instance of that model, which overcame some shortcomings of the earlier instances of the model. Here, I will present an abstract model of permissions that encompasses all the previous models.

A permission algebra $(K, \oplus, \top)$ is a resource algebra with a top element but without its unit element. Intuitively, $\oplus$ adds two disjoint permissions, $\top$ represents full permission, and there is no 'zero' permission. In the definition below, $K_\bot$ stands for $K \uplus \{\bot\}$ and implies that $\bot \notin K$.

Definition 9. *A permission algebra $(K, \oplus, \top)$ consists of a set K, a binary operation $\oplus : (K_\bot \times K_\bot) \to K_\bot$, and a distinguished element $\top \in K$ such that $(K_\bot, \oplus, \bot)$ is a resource algebra, and for all $k \in K_\bot$, $k \sqsubseteq \top$.*

Examples of permissions algebras are:

1. Fractional permissions: Numbers in $(0, 1]$ with addition as $\oplus$ and 1 as $\top$. All other numbers in $(0, 1)$ are partial permissions.

2. Counting permissions: Integers with 0 being full permission, and

$$k_1 \oplus k_2 = \begin{cases} \textit{undefined} & \text{if } k_1 \geq 0 \wedge k_2 \geq 0 \\ \textit{undefined} & \text{if } (k_1 \geq 0 \vee k_2 \geq 0) \wedge k_1 + k_2 < 0 \\ k_1 + k_2 & \text{otherwise} \end{cases}$$

In this model, -1 is a read permission, whereas positive numbers, $+k$, indicate that k read permissions have been removed.

3. Non-empty subsets of a countable set A with disjoint union as $\oplus$ and A as $\top$.

To model a heap with permissions, it is best to extend $\oplus$ to act on permission-value pairs and on functions of type $\mathsf{Loc} \rightharpoonup (\mathsf{Perm} \times \mathsf{Val})$.

$$(k_1, v_1) \oplus (k_2, v_2) \stackrel{\text{def}}{=} \begin{cases} (k_1 \oplus k_2, v_1) & \text{if } v_1 = v_2 \text{ and } \mathit{defined}(k_1 \oplus k_2) \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

The operator $\oplus$ on functions $(h_1 \oplus h_2)$ is defined if and only if $h_1(a) \oplus h_2(a)$ is defined for all $a \in (\mathit{dom}(h_1) \cap \mathit{dom}(h_2))$. If $h_1 \oplus h_2$ is defined, it has domain $\mathit{dom}(h_1) \cup \mathit{dom}(h_2)$ with the following values:

$$(h_1 \oplus h_2)(a) \stackrel{\text{def}}{=} \begin{cases} h_1(a) \oplus h_2(a) & \text{if } a \in (\mathit{dom}(h_1) \cap \mathit{dom}(h_2)) \\ h_1(a) & \text{if } a \in (\mathit{dom}(h_1) \setminus \mathit{dom}(h_2)) \\ h_2(a) & \text{if } a \in (\mathit{dom}(h_2) \setminus \mathit{dom}(h_1)) \end{cases}$$

As expected, adding two heaps is defined whenever for each location their overlap, both heaps store the same value and permissions that can be added together. The result is a heap whose permissions for the location in the overlap is just the sum of the individual heaps.

Now, we can define a resource algebra $(M, \odot, \mathsf{U})$ as follows:

$$\begin{aligned} M &= (\mathsf{VarName} \rightharpoonup \mathsf{Val}) \times (\mathsf{Loc} \rightharpoonup (\mathsf{Perm} \times \mathsf{Val})) \\ \odot &= \begin{cases} \lambda(s_1, h_1)(s_2, h_2).\ (s_1.h_1 \oplus h_2) & \text{if } s_1 = s_2, \\ \mathit{undefined} & \text{if } s_1 \neq s_2 \end{cases} \\ \mathsf{U} &= \{(s, \emptyset) \mid s \in \mathsf{VarName} \rightharpoonup \mathsf{Val}\} \end{aligned}$$

Primitive assertions in the permissions model are:

$$\mathit{PrimP} ::= e_1 \stackrel{k}{\mapsto} e_2 \mid e_1 = e_2 \mid e_1 < e_2$$

The first assertion says that the heap consists of exactly one memory cell with address e_1, accounting figure k, and contents e_2. Usually k is a constant, but in some cases, it might be a pure expression. In terms of the Kripke model, the first assertion means:

$$(s, h), i \models_{\mathsf{SL}} e_1 \stackrel{k}{\mapsto} e_2 \iff^{\text{def}} \mathit{dom}(h) = \{[\![e_1]\!]\ s\} \wedge h(e_1) = ([\![k]\!]\ s\ i, [\![e_2]\!]\ s\ i)$$

Aside: Variables as resource Unfortunately, the two models mentioned above are somewhat flawed in their treatment of the stack. Because the $\odot$ operator divides only the heap and not the stack, the primitive commands, c, do not satisfy the locality property (Def. 4). Hence, the frame rule (SL-Frame) is unsound and requires a side-condition for soundness: "The frame R must not contain any variables modified by the command C." The parallel composition rule (SL-Par) also requires a side-condition, but it is trickier. It suffices, however, that all stack variables are local, and they are never shared between threads or their specifications.

The "variables as resource" model [9, 64] avoids these side-conditions by treating program variables in a similar way to the heap. Separating conjunction ($*$) splits variable ownership as well as heap ownership. By default each assertion owns some portion of the variables it mentions and there are special assertions that specify the variable ownership exactly.

For simplicity, we will not use the "variables as resource" model. Instead, whenever a global variable `x` is shared between two threads, we will treat it as a heap cell stored at the fixed address &`x`. This way, we avoid any side-conditions on SL-Par.

2.4.3 Concurrent separation logic

The proof rules of §2.4.1 did not permit sharing of resources among threads. Hence, we cannot reason about parallel programs involving inter-thread communication.

Concurrent separation logic [58] overcomes this limitation by introducing resource invariants. The proof rules now have the form $J \vdash_{\mathsf{SL}} \{P\}\ C\ \{Q\}$, where J is a *precise* separation logic assertion representing an invariant that is true about the program separately from the precondition and postcondition. The intention is that J holds at all times during execution of the program except when a thread is inside an atomic block.

One can view $J \vdash_{\mathsf{SL}} \{P\}\ C\ \{Q\}$ as a simple rely/guarantee statement. The command C relies on J holding each time it enters an atomic command. In response, it guarantees that it will not access the resource J only within atomic commands and that it will make J hold each time it exits an atomic command.

Concurrent separation logic has the following rule for atomic commands, which grants threads temporary access to the invariant, J, within an atomic command.

$$\frac{\mathit{emp} \vdash_{\mathsf{SL}} \{P * J\}\ C\ \{Q * J\} \qquad J \text{ is precise}}{J \vdash_{\mathsf{SL}} \{P\}\ \langle C \rangle\ \{Q\}} \qquad \text{(SL-Atomic)}$$

The following rule allows us to take some local state, R, and treat it as shared state for the duration of the command C.

$$\frac{J * R \vdash_{\mathsf{SL}} \{P\}\ C\ \{Q\} \qquad R \text{ is precise}}{J \vdash_{\mathsf{SL}} \{P * R\}\ C\ \{Q * R\}} \qquad \text{(SL-ResFrame)}$$

Ownership transfer

The SL-Atomic rule combines the local state with the resource invariant on entry to the atomic block, and takes them apart on exit. The state put back in the invariant does not need to be the same as the state that was initially taken out. If the domains of these two states differ, we have a transfer of ownership. For example, consider the following invariant:

$$J \stackrel{\text{def}}{=} (x \mapsto 0 * \mathsf{list}(y)) \vee (x \mapsto 1)$$

where $\mathsf{list}(y)$ is some predicate describing a linked list starting at address y. In this example, x is essentially a mutual exclusion lock. When it is unlocked, the resource contains the shared list $\mathsf{list}(y)$. When it is locked, the thread that acquired the lock owns the list. Writing to $[x]$ moves the ownership of $\mathsf{list}(y)$ from the writer to the resource or vice versa:

$$J \vdash_{\mathsf{SL}} \{emp\}\ \mathbf{atomic}([x] = 0)\ \{[x] := 1\}\ \{\mathsf{list}(y)\}$$
$$J \vdash_{\mathsf{SL}} \{\mathsf{list}(y)\}\ \mathbf{atomic}\ \{[x] := 0\}\ \{emp\}$$

Soundness

There are various proofs of soundness for concurrent separation logic. Brookes [11, 12] gave a trace semantics for a simple programming language and proved the soundness of concurrent separation logic for the stack and heap model, for the permissions model, and for the variables-as-resource model. Hayman [37] presented an alternative semantics based on Petri nets and proved the soundness of the logic in that model. Recently, Calcagno et al. [13] proved the soundness of a more abstract version of concurrent separation logic.

2.5 Other proof methods

Invariants

Invariants [4] are the simplest way to reason about a concurrent system. A single-state assertion is invariant throughout a program if it holds initially and it is preserved by all atomic actions of the program. To prove a property P about a state of the program, one simply establishes a suitable program invariant J that entails the property P. Invariant-based reasoning is modular, because if J is invariant throughout C_1 and C_2, then it is invariant throughout $C_1 \| C_2$.

The problem is that any non-trivial property P depends on control flow, and hence the invariant J must encode this control flow. This can be achieved by using auxiliary variables or control predicates, but both methods are unsatisfactory because the invariant's size grows rapidly.

Resource invariants

Concurrent separation logic has the notion of a resource invariant, an invariant that is associated with a mutual exclusion lock. Resource invariants, however, are not pertinent to separation logic: they are a more general concept.

Resource invariants encode a common protocol whereby the lock owns some state. When the lock is not currently acquired, the resource invariant holds. When a thread acquires a lock, it knows that the resource invariant held at the moment of the acquisition. Thereafter, it may invalidate the resource invariant provided that it restores the resource invariant when it releases the lock.

Of course, resource invariants are just a special case of normal invariants, as the following encoding demonstrates:

$$Invariant = \ldots \wedge (lockNotAcquired \Rightarrow ResourceInvariant)$$

Therefore, they suffer from the same problems as normal invariants do. For moderately complex algorithms, a lot of auxiliary state is needed to express control-flow information and their size grows rapidly. Resource invariants are often used together with a syntactic restriction or ownership-based system that ensures that threads preserve the resource invariant when the lock is not acquired. Such restrictions exploit the modular structure of some programs removing many trivial implications, but do not address the inherent limitations of using invariants to describe shared state.

Ownership

In ownership-based systems, each object is 'owned' by another object or by a thread, and only the object's owner is allowed to access it. The owner of an object need not be fixed; it can change over time. For example, we can let mutual exclusion locks own some resources. When a thread locks a mutex, it also acquires the ownership of the resources the mutex protects; it returns the ownership back to the mutex when it releases the lock. One can see Separation Logic as an instance of ownership.

Spec# [5] is a object-oriented programming language that supports ownership-based verification. Each object has an auxiliary field storing its current owner. Assertions are written in classical first order logic, but may refer only to objects owned by the current component. Hence, local reasoning is possible. Spec# also enables the user to specify object invariants and has an auxiliary boolean field per object recording whether the object invariant holds for that object. The built-in operations `pack` and `unpack` set that field and check the invariant.

Jacobs et al. [49, 50] extended this methodology to handle pessimistic coarse-grained concurrency. Shared state is owned by a lock. When a lock is acquired, the ownership

of the protected state is transferred to the thread that acquired the lock. There is also a simple mechanism for preventing deadlock based on statically-assigned lock levels.

Ownership properties can also be enforced by a type system based on the calculus of capabilities [21]. Grossman [32] extended Cyclone with type-safe multi-threading and locking based on capabilities. Similar work was done by Microsoft's Vault project [22, 26].

2.6 A comparison

Global versus local reasoning In Hoare logic [46], assertions describe properties of the *whole* memory, and hence specifications, such as $\{P\}\ C\ \{Q\}$, describe a change of the whole memory. This is inherently *global reasoning*. Anything not explicitly preserved in the specification could be changed, for example $\{\mathtt{x} = 4\}\ \mathtt{y:=5}\ \{\mathtt{x} = 4\}$. Here `y` is allowed to change, even though it is not mentioned in the specification. The same is true for most of the other traditional methods: temporal logics, Owicki-Gries, rely/guarantee.

The situation is different in separation logic and in the other ownership-based approaches. Assertions describe properties of *part* of the memory, and hence specifications describe changes to *part* of the memory. The rest of the memory is guaranteed to be unchanged. This is the essence of *local reasoning*; specifications describe only the memory used by a command. With local reasoning, we can reason about independent modules independently.

In Spec#, the splitting of the state is determined by auxiliary "owner" fields. In Separation Logic, the assertions themselves describe their footprint and separating conjunction ensures that two assertions describe separate parts of the state. This makes separation logic theoretically nicer, but potentially harder to mechanise.

Relations versus invariants Binary relations are much better in describing concurrent systems and interfaces between components of a sequential system than invariants. For example, consider a variable x whose value may increase but never decrease, namely $x \geq \overleftarrow{x}$. If we read x twice, we know that the second time we will have read a larger or equal value to the one read the first time. Proving this with invariants requires auxiliary state. Introduce an auxiliary variable y, initially containing $-\infty$, and the first time we read x assign the value read to y. Then we can use the invariant $x \geq y$ to complete the proof.

Invariants are just a special case of relations; they may still be useful in simpler examples. The power of Jones's rely/guarantee and Lamport's TLA is that they use relations where their competitors used invariants.

In Chapter 3, we will bring together the local reasoning and binary relations, thereby marrying the modularity of separation logic and the expressiveness of Rely/Guarantee.

Chapter 3

Combining rely/guarantee and separation logic

This chapter describes RGSep, a new logic that marries rely/guarantee reasoning and separation logic. It subsumes the two proof systems that it is based on. Any rely/guarantee or concurrent separation logic proof can be encoded as an RGSep proof; moreover, some proofs are much easier in RGSep than in either separation logic or rely/guarantee alone.

This chapter presents the basic elements of RGSep and proves its soundness based on an instrumented operational semantics. As an example, Section 3.5 proves the safety of a concurrent linked list algorithm that has fine-grained locking and deallocates memory explicitly. The various adaptations and extensions to RGSep will be discussed in the next chapter.

3.1 The combined logic

3.1.1 Local and shared state assertions

The total state, σ, of the system consists of two components: the local state l, and the shared state s. Abstractly, l and s are elements of a resource algebra $(M, \odot, \mathsf{u})$ (see Def. 3 in §2.4) such that $l \odot s$ is defined. More concretely, one can think of each component state as a partial finite function from locations to values (cf. the heap model in §2.4). In this model, $l \odot s$ is defined if and only if the domains of the two states are disjoint; then, the total state is simply the union of the two disjoint states. The resource algebra model is, however, more general and permits other instantiations such as permissions.

We could easily specify a state using two assertions, one describing the local state and another describing the shared state. This approach, however, has some drawbacks:

specifications are longer, meta-level quantification is needed to relate the values in the local and the shared parts of the state, and extending this setting to a domain with multiple disjoint regions of shared state is clumsy.

Instead, we consider a unified assertion language that describes both the local and the shared state. Here is the syntax of our assertions:

$p, q, r ::=$	P	Local assertion
	$\boxed{P}$	Shared assertion
	$p * q$	Separating conjunction
	$p \wedge q$	Normal conjunction
	$p \vee q$	Disjunction
	$\forall x.\ p$	Universal quantification
	$\exists x.\ p$	Existential quantification

where P stands for any separation logic assertion (defined in §2.4). We will often call $\boxed{P}$ assertions as boxed assertions. Note, however, that boxes are not modalities in the usual sense as they cannot be nested.

Formally, the semantics of assertions are given in terms of a Kripke structure (l, s, i) where $l, s \in M$, $l \odot s$ is defined, and $i : \mathsf{LogVar} \rightharpoonup \mathsf{Val}$ is a mapping from logical variables to values.

$$
\begin{array}{lcl}
l, s, i \models_{\mathsf{RGSep}} P & \stackrel{\text{def}}{\iff} & l, i \models_{\mathsf{SL}} P \\
l, s, i \models_{\mathsf{RGSep}} \boxed{P} & \stackrel{\text{def}}{\iff} & (l = \mathsf{u}) \wedge (s, i \models_{\mathsf{SL}} P) \\
l, s, i \models_{\mathsf{RGSep}} p_1 * p_2 & \stackrel{\text{def}}{\iff} & \exists l_1, l_2.\ (l = l_1 \odot l_2) \wedge (l_1, s, i \models_{\mathsf{RGSep}} p_1) \wedge (l_2, s, i \models_{\mathsf{RGSep}} p_2) \\
l, s, i \models_{\mathsf{RGSep}} p_1 \wedge p_2 & \stackrel{\text{def}}{\iff} & (l, s, i \models_{\mathsf{RGSep}} p_1) \wedge (l, s, i \models_{\mathsf{RGSep}} p_2) \\
l, s, i \models_{\mathsf{RGSep}} p_1 \vee p_2 & \stackrel{\text{def}}{\iff} & (l, s, i \models_{\mathsf{RGSep}} p_1) \vee (l, s, i \models_{\mathsf{RGSep}} p_2) \\
l, s, i \models_{\mathsf{RGSep}} \forall x.\ p & \stackrel{\text{def}}{\iff} & \forall v.\ (l, s, [i \mid x \mapsto v] \models_{\mathsf{RGSep}} p) \\
l, s, i \models_{\mathsf{RGSep}} \exists x.\ p & \stackrel{\text{def}}{\iff} & \exists v.\ (l, s, [i \mid x \mapsto v] \models_{\mathsf{RGSep}} p)
\end{array}
$$

Note that the definition of $*$ splits the local state, but not the shared state. We say that $*$ is multiplicative over the local state, but additive over the shared state. In particular, $\boxed{P} * \boxed{Q} \iff \boxed{P \wedge Q}$. The semantics of shared assertions, $\boxed{P}$, could alternatively be presented without $l{=}\mathsf{u}$. This results in an equally expressive logic, but the definition above leads to shorter assertions in practice.

RGSep formulas include the separation logic formulas and overload the definition of some separation logic operators ($*$, $\wedge$, $\vee$, $\exists$ and $\forall$) to act on RGSep assertions. This overloading is intentional and justified by the following Lemma (writing $\mathsf{Local}P$ for the first RGSep assertion kind):

Lemma 10 (Properties of local assertions)**.**

$$\begin{aligned}
\mathsf{Local}(\mathit{false}) &\iff \mathit{false} \\
(\mathsf{Local}(P) * \mathsf{Local}(Q)) &\iff \mathsf{Local}(P * Q) \\
(\mathsf{Local}(P) \wedge \mathsf{Local}(Q)) &\iff \mathsf{Local}(P \wedge Q) \\
(\mathsf{Local}(P) \vee \mathsf{Local}(Q)) &\iff \mathsf{Local}(P \vee Q) \\
(\exists x.\ \mathsf{Local}(P)) &\iff \mathsf{Local}(\exists x.\ P) \\
(\forall x.\ \mathsf{Local}(P)) &\iff \mathsf{Local}(\forall x.\ P)
\end{aligned}$$

These follow directly from the semantic definitions. Because of this lemma, we can reduce the notational overhead by making the Local implicit. This should not cause any confusion, because according to Lemma 10, the RGSep operators and the separation logic operators coincide for local assertions.

Finally, the grammar disallows top-level negations so that boxed assertions only appear in positive positions. This assists in defining stability (see §3.1.3). It not a severe restriction because (a) top-level negations do not arise in practice, and (b) if we give the usual semantics to negation then, $\neg\mathsf{Local}(P) \iff \mathsf{Local}(\neg P)$ and $\neg\boxed{P} \iff \boxed{\neg P} \vee \neg \mathit{emp}$.

3.1.2 Describing interference

The strength of rely/guarantee is the relational description of interference between parallel processes. Instead of using relations directly, RGSep describes interference in terms of actions $P \rightsquigarrow Q$ that describe the changes performed to the shared state. These resemble Morgan's *specification statements* [57], and P and Q will typically be linked with some existentially quantified logical variables. (We do not need to mention separately the set of modified shared locations, because these are all included in P.) The meaning of an action $P \rightsquigarrow Q$ is that it replaces the part of the shared state that satisfies P prior to the action with a part satisfying Q without changing the rest of the shared state. For example, consider the following action:

$$\mathtt{x} \mapsto M \ \rightsquigarrow\ \mathtt{x} \mapsto N \wedge N \geq M \qquad (\mathsf{Increment})$$

It specifies that the value in the heap cell `x` may be changed, but its value is never decremented. The logical variables M and N are existentially bound with scope ranging over both the precondition and the postcondition. In this action, the heap footprints of the precondition and of the postcondition both consist of the location `x`. The footprints of the precondition and the postcondition, however, need not be the same. When they are different, this indicates a transfer of ownership between the shared state and the local state of a thread. For instance, consider a simple lock with two operations: Acquire which

$$\frac{P \text{ exact}}{(P \rightsquigarrow P) \subseteq G} \quad \text{(G-Exact)} \qquad \frac{(P \rightsquigarrow Q) \in G}{(P \rightsquigarrow Q) \subseteq G} \quad \text{(G-Axiom)}$$

$$\frac{(P \rightsquigarrow Q) \subseteq G}{(P[e/x] \rightsquigarrow Q[e/x]) \subseteq G} \quad \text{(G-Sub)} \qquad \frac{(P \rightsquigarrow Q) \subseteq G \quad \models_{\mathsf{SL}} P' \Rightarrow P \quad \models_{\mathsf{SL}} Q' \Rightarrow Q}{(P' \rightsquigarrow Q') \subseteq G} \quad \text{(G-Cons)}$$

Figure 3.1: Rules and axioms for an action allowed by a guarantee.

changes the lock bit from 0 to 1, and removes the protected object, $list(y)$, from the shared state; and Release which changes the lock bit from 1 to 0, and puts the protected object back into the shared state. We can represent these two operations formally as

$$(\mathtt{x} \mapsto 0) * list(\mathtt{y}) \rightsquigarrow \mathtt{x} \mapsto 1 \qquad \text{(Acquire)}$$

$$\mathtt{x} \mapsto 1 \rightsquigarrow (\mathtt{x} \mapsto 0) * list(\mathtt{y}) \qquad \text{(Release)}$$

An action $P \rightsquigarrow Q$ represents the modification of some shared state satisfying P to some state satisfying Q. Its semantics is the following relation:

$$[\![P \rightsquigarrow Q]\!] \stackrel{\text{def}}{=} \{(s_1 \odot s_0, s_2 \odot s_0) \mid \exists i.\ (s_1, i \models_{\mathsf{SL}} P) \wedge (s_2, i \models_{\mathsf{SL}} Q)\}$$

It relates some initial shared state s_1 satisfying the precondition P to a final state s_2 satisfying the postcondition. In addition, there may be some disjoint shared state s_0 which is not affected by the action. In the spirit of separation logic, we want the action specification as 'small' as possible, describing s_1 and s_2 but not s_0, and use the frame rule to perform the same update on a larger state. The existential quantification over the interpretation, i, allows P and Q to have shared logical variables, such as M and N in Increment.

RGSep represents the rely and guarantee conditions as sets of actions. The relational semantics of a set of actions is the reflexive and transitive closure of the union of the semantics of each action in the set. This allows each action to run any number of times in any interleaved order with respect to the other actions.

$$[\![P_1 \rightsquigarrow Q_1, \ldots, P_n \rightsquigarrow Q_n]\!] = \left(\bigcup_{i=1}^{n} [\![P_i \rightsquigarrow Q_i]\!]\right)^*$$

As a sanity condition, we can require that the assertions P and Q appearing in an action $P \rightsquigarrow Q$ are *precise*. This sanity condition is not strictly necessary, but its use is justified by the side-condition of the atomic rule in §3.1.4. In practice, P and Q are not only precise, but often also *exact* (see Def. 7 in §2.4).

A specification, $P_1 \rightsquigarrow Q_1$ is allowed by a guarantee G if its effect is contained in G.

Definition 11. $(P \rightsquigarrow Q) \subseteq G \quad \stackrel{\text{def}}{\iff} \quad [\![P \rightsquigarrow Q]\!] \subseteq [\![G]\!]$.

Figure 3.1 provides derived rules to be used in proofs. G-Cons is similar to the rule of consequence, but the second implication is reversed, $Q \Rightarrow Q'$.

Contexts in actions The specifications of some examples become clearer if we extend actions $P \rightsquigarrow Q$ with a context F with the following semantics:

$$[\![P \rightsquigarrow Q \text{ provided } F]\!] \stackrel{\text{def}}{=} [\![P \rightsquigarrow Q]\!] \cap [\![P * F \rightsquigarrow Q * F]\!]$$

This definition ensures that the context F is not changed by the action. If F is an *exact* assertion, then $[\![P \rightsquigarrow Q \text{ provided } F]\!] = [\![P * F \rightsquigarrow Q * F]\!]$. Every extended action in this dissertation will actually have the context F be an exact assertion or an assertion that can trivially be made exact by dropping an existential quantification.

3.1.3 Stability of assertions

Rely/guarantee reasoning requires that every pre- and post-condition in a proof is stable under environment interference. A separation logic assertion S is stable under interference of a relation R if and only if whenever S holds initially and we perform an update satisfying R, then the resulting state still satisfies S.

Definition 12 (Stability)**.** $\mathsf{sem\_stable}(S, R)$ *if and only if for all* s, s' *and* i *such that* $s, i \models_{\mathsf{SL}} S$ *and* $(s, s') \in R$, *then* $s', i \models_{\mathsf{SL}} S$.

This is the same definition as in §2.3, but makes the interpretation i of the logical variables explicit. By representing the interference R as a set of actions, we can reduce stability to a simple syntactic check.

Lemma 13 (Checking stability)**.**

- $\mathsf{sem\_stable}(S, [\![P \rightsquigarrow Q]\!])$ *if and only if* $\models_{\mathsf{SL}} ((P \mathrel{-\!\!\circledast} S) * Q) \Rightarrow S$.
- $\mathsf{sem\_stable}(P, (R_1 \cup R_2)^*)$ *if and only if* $\mathsf{sem\_stable}(S, R_1)$ *and* $\mathsf{sem\_stable}(S, R_2)$.

Informally, the first property says that if from a state that satisfies S, we remove the part of the state satisfying P, and replace it with some state satisfying Q, then this should imply that S holds again. In the case when the action cannot run, because there is no sub-state of S satisfying P, then $P \mathrel{-\!\!\circledast} S$ is *false* and the implication holds trivially. An assertion S is stable under interference of a set of actions R if and only if it is stable under interference by every action in R.

RGSep forbids interference on the local state, but permits interference on the shared state. Hence, given an RGSep assertion, only the parts of it that describe the shared state may be affected by interference. We shall say that an RGSep assertion is (syntactically) stable under R, if all the boxed assertions it contains are stable under R.

Definition 14. *Let p* stable under *R be defined by induction on p as follows*

- *P* stable under *R always holds.*
- *$\boxed{P}$* stable under *R if and only if $(P; R) \Rightarrow P$.*
- *For $op ::= * \mid \wedge \mid \vee$, let $(p_1\ op\ p_2)$* stable under *R if and only if p_1* stable under *R and p_2* stable under *R.*
- *For $Q ::= \forall \mid \exists$, let $(Qx.\ p)$* stable under *R if and only if p* stable under *R.*

If an assertion is syntactically stable, then it is also semantically stable in the following sense:

Lemma 15. *If p* stable under *R, $l, s, i \models_{\mathsf{RGSep}} p$ and $(s, s') \in R$, then $l, s', i \models_{\mathsf{RGSep}} p$.*

The converse is not true. For example, consider a relation R that writes an arbitrary value to x without changing the rest of the heap. Then, $\boxed{\exists n.\ x \mapsto n}$ is stable under R, whereas the assertions $\boxed{\exists n.\ x \mapsto n \wedge n \leq 0}$ and $\boxed{\exists n.\ x \mapsto n \wedge n > 0}$ are not. Therefore, $\boxed{\exists n.\ x \mapsto n \wedge n \leq 0} \vee \boxed{\exists n.\ x \mapsto n \wedge n > 0}$ is not syntactically stable although it is semantically equivalent to $\boxed{\exists n.\ x \mapsto n}$.

3.1.4 Specifications and proof rules

Specifications of a command C are quadruples (p, R, G, q), where

- The precondition p describes the set of initial states in which C might be executed (both its local and shared parts).
- The rely R is a relation (i.e. a set of actions) describing the interference caused by the environment.
- The guarantee G is a relation describing the changes to the shared state, caused by the program.
- The postcondition q describes the possible resulting local and shared states, should the execution of C terminate.

The judgement $\vdash C$ **sat** (p, R, G, q) says that any execution of C from an initial state satisfying p and under environment interference R (*i*) does not fault (e.g. accesses unallocated memory), (*ii*) causes interference at most G, and, (*iii*) if it terminates, its final state satisfies q.

First, we have the familiar specification weakening rule:

$$\frac{\begin{array}{rr} & R \subseteq R' \quad p \Rightarrow p' \\ \vdash C\ \mathbf{sat}\ (p', R', G', q') & G' \subseteq G \quad q' \Rightarrow q \end{array}}{\vdash C\ \mathbf{sat}\ (p, R, G, q)} \qquad (\textsc{Weaken})$$

From separation logic, RGSep inherits the frame rule: If a program runs safely with initial state p, it can also run with additional state r lying around. Since the program runs safely without r, it cannot access the additional state; hence, r is still true at the end. Since the frame, r, may also specify the shared state, the FRAME rule checks that r is stable under interference from both the program and its environment. Otherwise, they may invalidate r during their execution. In the simple case when r does not mention the shared state, the stability check is trivially satisfied.

$$\frac{\vdash C \textbf{ sat } (p, R, G, q) \qquad r \text{ stable under } (R \cup G)}{\vdash C \textbf{ sat } (p * r, R, G, q * r)} \quad (\textsc{Frame})$$

We also have the other standard structural rules (CONJ, DISJ, EX, ALL).

$$\frac{\vdash C \textbf{ sat } (p_1, R, G, q) \quad \vdash C \textbf{ sat } (p_2, R, G, q)}{\vdash C \textbf{ sat } (p_1 \vee p_2, R, G, q)} \quad (\textsc{Disj}) \qquad \frac{x \notin \mathsf{fv}(q, C, R, G) \quad \vdash C \textbf{ sat } (p, R, G, q)}{\vdash C \textbf{ sat } (\exists x.\ p, R, G, q)} \quad (\textsc{Ex})$$

$$\frac{\vdash C \textbf{ sat } (p, R, G, q_1) \quad \vdash C \textbf{ sat } (p, R, G, q_2)}{\vdash C \textbf{ sat } (p, R, G, q_1 \wedge q_2)} \quad (\textsc{Conj}) \qquad \frac{x \notin \mathsf{fv}(p, C, R, G) \quad \vdash C \textbf{ sat } (p, R, G, q)}{\vdash C \textbf{ sat } (p, R, G, \forall x.\ q)} \quad (\textsc{All})$$

Then, there is a proof rule for each construct in the language. The rules for the empty program, sequential composition, non-deterministic choice, and loops are completely standard. Similarly to FRAME, SKIP checks that the precondition, p, is stable under the rely, R. Because the empty program does not change the state, p is trivially stable under interference from the program itself.

$$\frac{p \text{ stable under } R}{\vdash \textbf{skip sat } (p, R, G, p)} \quad (\textsc{Skip}) \qquad \frac{\vdash C \textbf{ sat } (p, R, G, p)}{\vdash C^* \textbf{ sat } (p, R, G, p)} \quad (\textsc{Loop})$$

$$\frac{\vdash C_1 \textbf{ sat } (p, R, G, r) \quad \vdash C_2 \textbf{ sat } (r, R, G, q)}{\vdash (C_1; C_2) \textbf{ sat } (p, R, G, q)} \quad (\textsc{Seq}) \qquad \frac{\vdash C_1 \textbf{ sat } (p, R, G, q) \quad \vdash C_2 \textbf{ sat } (p, R, G, q)}{\vdash (C_1 + C_2) \textbf{ sat } (p, R, G, q)} \quad (\textsc{Choice})$$

For primitive commands, c, that do not access the shared state, we adopt the separation logic rules. We have the following rule scheme:

$$\frac{\vdash_{\mathsf{SL}} \{P\}\ c\ \{Q\}}{\vdash c \textbf{ sat } (P, R, G, Q)} \quad (\textsc{Prim})$$

The parallel composition rule of RGSep is very similar to the parallel composition rule of rely/guarantee. Its crucial difference is that the precondition and postcondition of the

composition are the separating conjunction ($*$) of the preconditions and postconditions of the individual threads. In essence, this is the normal conjunction of the shared state assertions, and the separating conjunction of the local state assertions (cf. the semantics of $*$ in §3.1.1).

$$\frac{\begin{array}{c}\vdash C_1 \textbf{ sat } (p_1, R \cup G_2, G_1, q_1) \\ \vdash C_2 \textbf{ sat } (p_2, R \cup G_1, G_2, q_2)\end{array}}{\vdash (C_1 \| C_2) \textbf{ sat } (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2)} \quad \text{(Par)}$$

As the interference experienced by thread C_1 can come from C_2 or from the environment of the parallel composition, we have to ensure that both interferences ($R \cup G_2$) are allowed. Similarly C_2 must be able to tolerate interference from C_1 and from the environment, R.

The most complex rule is that of atomic commands, $\langle C \rangle$. Instead of tackling the general case directly, it is easier if we have two rules. The first rule checks that the atomic block meets its specification in an empty environment, and then checks that the precondition and the postcondition are stable with respect to the actual environment, R. This reduces the problem from an arbitrary rely condition to an empty rely condition.

$$\frac{\begin{array}{c}\vdash \langle C \rangle \textbf{ sat } (p, \emptyset, G, q) \\ p \text{ stable under } R \quad q \text{ stable under } R\end{array}}{\vdash \langle C \rangle \textbf{ sat } (p, R, G, q)} \quad \text{(AtomR)}$$

The second rule is somewhat trickier. Here is a first attempt:

$$\frac{\vdash C \textbf{ sat } (P * P', \emptyset, \emptyset, Q * Q') \qquad (P \rightsquigarrow Q) \subseteq G}{\vdash \langle C \rangle \textbf{ sat } (\boxed{P} * P', \emptyset, G, \boxed{Q} * Q')}$$

Within an atomic block, we can access the shared state $\boxed{P}$, but we must check that changing the shared state from P from Q is allowed by the guarantee G. This rule is sound, but too weak in practice. It requires that the critical region changes the *entire* shared state from P to Q and that the guarantee condition allows such a change. We can extend the rule by allowing the region to change only *part* of the shared state P into Q, leaving the rest of the shared state (F) unchanged, and checking that the guarantee permits the small change $P \rightsquigarrow Q$.

$$\frac{P, Q \text{ precise} \qquad \vdash C \textbf{ sat } (P * P', \emptyset, \emptyset, Q * Q') \qquad (P \rightsquigarrow Q) \subseteq G}{\vdash \langle C \rangle \textbf{ sat } (\boxed{P * F} * P', \emptyset, G, \boxed{Q * F} * Q')} \quad \text{(Atom)}$$

Precision For soundness, the rule requires that P and Q are *precise*[1] assertions and that *all* branches of the proof use the same P and Q for the same atomic region. Without this requirement, the logic admits the following erroneous derivation:

[1]Precise assertions were defined in §2.4.1 (Def. 8).

$$\dfrac{\dfrac{P{=}Q'{=}x{\mapsto}1 \text{ and } P'{=}F{=}Q{=}emp}{\langle\mathbf{skip}\rangle \mathbf{\ sat\ } (\boxed{x{\mapsto}1}, \emptyset, G, \boxed{emp} * x{\mapsto}1)}\ \textsc{Atom} \quad \dfrac{F{=}x{\mapsto}1 \text{ and } P{=}Q{=}P'{=}Q'{=}emp}{\langle\mathbf{skip}\rangle \mathbf{\ sat\ } (\boxed{x{\mapsto}1}, \emptyset, G, \boxed{x{\mapsto}1})}\ \textsc{Atom}}{\dfrac{\langle\mathbf{skip}\rangle \mathbf{\ sat\ } (\boxed{x \mapsto 1}, \emptyset, G, (\boxed{emp} * x \mapsto 1) \wedge \boxed{x \mapsto 1})}{\langle\mathbf{skip}\rangle \mathbf{\ sat\ } (\boxed{x \mapsto 1}, \emptyset, G, \mathit{false})}\ \textsc{Weaken}}\ \textsc{Conj}$$

Semantically, this specification is satisfied only if $\langle\mathbf{skip}\rangle$ never terminates! Precision is a technical requirement inherited from concurrent separation logic. It ensures that the splitting of the resultant state into local and shared portions is unambiguous.

The CONCUR 2007 paper [75] has a different stability requirement. It requires that the entire shared postcondition ($Q * F$ in this rule) is precise. Ensuring that P and Q are precise is typically trivial because P and Q describe the small update performed by C, whereas $Q * F$ may be a much larger and complex assertion.

Atomic rule in the CONCUR'07 paper Instead of the two rules AtomR and Atom, the CONCUR'07 paper [75] has the following proof rule:

$$\dfrac{\begin{array}{ccccc}(P' \rightsquigarrow Q') \subseteq G & \vdash C \mathbf{\ sat\ } (P' * P'', \emptyset, \emptyset, Q' * Q'') & \boxed{\exists \overline{y}.P} \text{ stable under } R \\ FV(P'') \cap \{\overline{y}\} = \emptyset & \models_{\mathsf{SL}} P \Rightarrow P' * F \qquad \models_{\mathsf{SL}} Q' * F \Rightarrow Q & \boxed{Q} \text{ stable under } R\end{array}}{\vdash \langle C\rangle \mathbf{\ sat\ } (\boxed{\exists \overline{y}.\ P} * P'', R, G, \exists \overline{y}.\ \boxed{Q} * Q'')}\ (\textsc{Atomic})$$

This rule is derivable by applying the rules AtomR, Conseq, Ex, and Atom. In the other direction, Atom is derivable from this complex rule, but AtomR is not.

$$\dfrac{\text{(Stab)} \quad \dfrac{\text{(Imp)} \quad \dfrac{\dfrac{\vdash C \mathbf{\ sat\ } (P' * P'', \emptyset, G, Q' * Q'') \qquad (P' \rightsquigarrow Q') \subseteq G}{\vdash \langle C\rangle \mathbf{\ sat\ } (\boxed{P' * F} * P'', \emptyset, G, \boxed{Q' * F} * Q'')}\ \textsc{Atom}}{\vdash \langle C\rangle \mathbf{\ sat\ } (\exists \overline{y}.\ \boxed{P} * P'', \emptyset, G, \boxed{Q} * Q'')}\ \textsc{Ex}}{\vdash \langle C\rangle \mathbf{\ sat\ } (\boxed{\exists \overline{y}.\ P} * P'', \emptyset, G, \exists \overline{y}.\ \boxed{Q} * Q'')}\ \textsc{Conseq}}{\vdash \langle C\rangle \mathbf{\ sat\ } (\boxed{\exists \overline{y}.\ P} * P'', R, G, \exists \overline{y}.\ \boxed{Q} * Q'')}\ \textsc{AtomR}$$

In the derivation, (Stab) stands for $\boxed{\exists \overline{y}.P}$ stable under R and $\boxed{Q}$ stable under R, and (Imp) stands for $FV(P'') \cap \{\overline{y}\} = \emptyset$, $\models_{\mathsf{SL}} P \Rightarrow P' * F$, and $\models_{\mathsf{SL}} Q' * F \Rightarrow Q$.

3.2 Operational semantics

The exposition below follows the abstract semantics for separation logic presented in §2.4. Let $(M, \odot, \mathsf{u})$ be a resource algebra and let l and s range over elements of M. These stand for the local state, the shared state, and local state of other threads respectively.

RGSep explicitly deals with the separation between a thread's own state and the shared state. The semantics keep track of this separation: states are structured and consist of these two components.

$$\frac{\mathsf{oracle}_{P\rightsquigarrow Q}(s,l_2)=(l',s') \quad (C,(l\odot s,\mathsf{u}))\xrightarrow{\emptyset}{}^{*}(\mathbf{skip},(l_2,\mathsf{u}))}{(\langle C\rangle_{P\rightsquigarrow Q},(l,s))\xrightarrow[\mathbf{p}]{R}(\mathbf{skip},(l',s'))} \qquad \frac{\mathsf{oracle}_{P\rightsquigarrow Q}(s,l_2)=\mathit{undef} \quad (C,(l\odot s,\mathsf{u}))\xrightarrow{\emptyset}{}^{*}(\mathbf{skip},(l_2,\mathsf{u}))}{(\langle C\rangle_{P\rightsquigarrow Q},(l,s))\xrightarrow[\mathbf{p}]{R}\mathit{fault}} \qquad \frac{(C,(l\odot s,\mathsf{u}))\xrightarrow{\emptyset}{}^{*}\mathit{fault}}{(\langle C\rangle_{P\rightsquigarrow Q},(l,s))\xrightarrow[\mathbf{p}]{R}\mathit{fault}}$$

$$\frac{c(l,l') \quad (l',s)\in\mathsf{States}}{(c,(l,s))\xrightarrow[\mathbf{p}]{R}(\mathbf{skip},(l',s))} \qquad \frac{(\neg\exists l'.\ c(l,l'))}{(c,(l,s))\xrightarrow[\mathbf{p}]{R}\mathit{fault}} \qquad \frac{R(s,s') \quad (l,s')\in\mathsf{States}}{(C,(l,s))\xrightarrow[\mathbf{e}]{R}(C,(l,s'))}$$

$$\frac{}{(\mathbf{skip};C,\sigma)\xrightarrow[\mathbf{p}]{R}(C,\sigma)} \qquad \frac{(C,\sigma)\xrightarrow[\mathbf{p}]{R}(C_1,\sigma')}{(C;C',\sigma)\xrightarrow[\mathbf{p}]{R}(C_1;C',\sigma')} \qquad \frac{}{(C_1+C_2,\sigma)\xrightarrow[\mathbf{p}]{R}(C_1,\sigma)}$$

$$\frac{}{(C_1+C_2,\sigma)\xrightarrow[\mathbf{p}]{R}(C_2,\sigma)} \qquad \frac{}{(C^*,\sigma)\xrightarrow[\mathbf{p}]{R}(\mathbf{skip}+(C;C^*),\sigma)} \qquad \frac{}{(\mathbf{skip}\|\mathbf{skip},\sigma)\xrightarrow[\mathbf{p}]{R}(\mathbf{skip},\sigma)}$$

$$\frac{(C_1,\sigma)\xrightarrow[\mathbf{p}]{R}(C_1',\sigma')}{(C_1\|C_2,\sigma)\xrightarrow[\mathbf{p}]{R}(C_1'\|C_2,\sigma')} \qquad \frac{(C_2,\sigma)\xrightarrow[\mathbf{p}]{R}(C_2',\sigma')}{(C_1\|C_2,\sigma)\xrightarrow[\mathbf{p}]{R}(C_1\|C_2',\sigma')} \qquad \frac{(C,\sigma)\xrightarrow[\mathbf{p}]{R}\mathit{fault}}{(C;C_2,\sigma)\xrightarrow[\mathbf{p}]{R}\mathit{fault}}$$

$$\frac{(C,\sigma)\xrightarrow[\mathbf{p}]{R}\mathit{fault}}{(C\|C_2,\sigma)\xrightarrow[\mathbf{p}]{R}\mathit{fault}} \qquad \frac{(C,\sigma)\xrightarrow[\mathbf{p}]{R}\mathit{fault}}{(C_2\|C,\sigma)\xrightarrow[\mathbf{p}]{R}\mathit{fault}}$$

Figure 3.2: Operational semantics: Config_1 reduces to Config_2 $\quad \mathit{Config}_1 \xrightarrow[R]{\lambda} \mathit{Config}_2$

Definition 16. $\mathsf{States} \stackrel{\text{def}}{=} \{(l,s) \mid l \in M \wedge s \in M \wedge (l \odot s)\ \mathit{is\ defined}\}$

Let σ range over these structured states, and overload the operator $\odot$ to act on structured states, σ, as follows:

Definition 17. *$(l_1,s_1)\odot(l_2,s_2)$ is defined as $(l_1\odot l_2,s_1)$ if $s_1=s_2$; otherwise it is undefined.*

Figure 3.2 contains a semantics of GPPL that keeps track of the splitting of the state σ into its two components: l and s. If we ignore the splitting of the state, we get back the standard semantics of §2.2. Configurations are either *fault* or a pair of a command and a structured state, (C,σ). A reduction step, $\mathit{Config}_1 \xrightarrow[\lambda]{R} \mathit{Config}_2$, goes from Config_1 to Config_2 with possible environment interference R and a label λ. The label indicates whether this is a program action, $\mathbf{p}$, or an environment action, $\mathbf{e}$. Finally, $\xrightarrow{R}{}^{*}$ is the reflexive and transitive closure of $\xrightarrow[\lambda]{R}$; it stands for zero or more $\xrightarrow[\lambda]{R}$ reductions.

In the semantics, atomic commands, $\langle C\rangle$, are annotated with *precise* assertions P and Q to specify how to split the state between shared and local when exiting from the block. In concurrent separation logic, the resource invariant decides the splitting between local and shared state. Instead, RGSep decides the splitting using these P and Q as an oracle.

The function $\mathsf{oracle}_{P \rightsquigarrow Q} : M \times M \rightharpoonup M \times M$ which determines the splitting is defined as:

$$\mathsf{oracle}_{P \rightsquigarrow Q}(s, l) = \begin{cases} (l', s_0 \odot s_2), & \text{if there exists } s_1 \text{ such that } s = s_1 \odot s_0 \text{ and } s_1 \models_{\mathsf{SL}} P \\ & \quad \text{and } l = l' \odot s_0 \odot s_2 \text{ and } s_2 \models_{\mathsf{SL}} Q \\ \textit{undefined}, & \text{otherwise} \end{cases}$$

As P and Q are precise, s_1, s_0, s_2 and l' are uniquely determined, and hence the function $\mathsf{oracle}_{P \rightsquigarrow Q}$ is well defined.

Consider the semantics of atomic blocks (first three rules in Figure 3.2). All three rules combine the local state with the shared state ($l \odot s'$) and execute the command C under no interference ($\emptyset$). If executing the command C successfully returns the local state l_2, the first rule uses the oracle to determine how to split the resulting local state into a new shared and local state, (l', s'). The other two rules handle the cases where the program fails in the evaluating the body, or the oracle fails to find a splitting of l_2.

An alternative approach in formalising the semantics of atomic blocks would be to combine only the part of the shared state that satisfies P with the local state inside the atomic block. This second approach is closer to the proof rules, but relies much more heavily on the annotation $P \rightsquigarrow Q$.

The next three rules govern primitive commands, c, and environment transitions. Primitive commands are represented as subsets of $M \times M$ such that the locality properties of §2.4.1 hold. The primitive command c executes correctly, if it runs correctly by accessing only the local state. Otherwise, c fails. Its execution does not affect the shared and environment states. An environment transition can happen anytime and affects only the shared state and the environment state, provided that the shared-state change is described by the rely relation, R; the local state is unchanged.

The remaining rules deal with the standard language constructs: sequential composition, non-deterministic choice, loops, and parallel composition. Note that the semantics has the reduction $(\mathbf{skip} \| \mathbf{skip}, \sigma) \xrightarrow[\mathbf{p}]{R} (\mathbf{skip}, \sigma)$ instead of the reduction $(\mathbf{skip} \| C, \sigma) \xrightarrow[\mathbf{p}]{R} (C, \sigma)$ and its symmetric version. This simplifies stating Lemma 23 in §3.3.

3.3 Soundness

A program C, executing in an initial state σ and an environment satisfying R, guarantees G if and only if it does not fail and all its shared-state changes satisfy G. Formally, this is defined by induction on the length of the trace n as follows:

Definition 18 (Guarantee)**.** $(C, \sigma, R)\,\mathbf{guarantees}\,G \overset{\text{def}}{\iff} \forall n.\ (C, \sigma, R)\,\mathbf{guarantees}_n\,G$, *where* $(C, (l, s), R)\,\mathbf{guarantees}_0\,G$ *holds always; and* $(C, (l, s), R)\,\mathbf{guarantees}_{n+1}\,G$ *holds if and only if whenever* $(C, (l, s)) \xrightarrow[\lambda]{R}$ *Config, then there exist* C', l', s' *such that*

1. *Config* $= (C', (l', s'))$;

2. $(C', (l', s'), R)$ **guarantees**$_n$ G; *and*

3. *if* $\lambda = \mathbf{p}$, *then* $(s, s') \in G$.[2]

A program C satisfies the specification (p, R, G, q) if and only if all executions that satisfy its assumptions about the initial state and the environment interference also satisfy its guarantee and its postcondition.

Definition 19. $\models C$ **sat** (p, R, G, q) *if and only if whenever* $l, s, i \models_{\mathsf{RGSep}} p$, *then*

1. $(C, (l, s), R)$ **guarantees** G; *and*

2. *if* $(C, (l, s)) \xrightarrow{R}{}^{*} (\mathbf{skip}, (l', s'))$, *then* $l', s', i \models_{\mathsf{RGSep}} q$.

The reduction rules satisfy the usual locality properties of separation logic and permit the rely condition to be weakened.

Lemma 20 (Locality)**.**

1. *If* $(C, (l_1 \odot l', s_1)) \xrightarrow{R}{}^{*} (C', (l_2, s_2))$, *then either* $(C, (l_1, s_1)) \xrightarrow{R}{}^{*}$ *fault or there exists an* l_2' *such that* $(C, (l_1, s_1)) \xrightarrow{R}{}^{*} (C', (l_2', s_2))$ *and* $\sigma_2' \odot \sigma' = \sigma_2$.

2. *If* $(C, (l_1 \odot l', s_1)) \xrightarrow{R}{}^{*}$ *fault*, *then* $(C, (l_1, s_1)) \xrightarrow{R}{}^{*}$ *fault*.

3. *If* $(C, \sigma) \xrightarrow{R}{}^{*} (C', \sigma')$ *and* $R \subseteq R'$, *then* $(C, \sigma) \xrightarrow{R'}{}^{*} (C', \sigma')$

The next lemma follows directly from Definition 18 and from Lemma 20.

Lemma 21. *If* (C, σ, R) **guarantees** G, *then*

- $(C, \sigma \odot \sigma', R)$ **guarantees** G;

- *If* $R' \subseteq R$, *then* (C, σ, R') **guarantees** G; *and*

- *If* $G \subseteq G'$, *then* (C, σ, R) **guarantees** G'.

The soundness of the logic follows by a structural induction on the command C. Using the following lemmas, we decompose the proof into simpler parts. First, since the rely condition is reflexive and transitive, executing an atomic command introduces the following cases:

Lemma 22. *If* C *is an atomic command* $\langle C \rangle$ *or a primitive command* c, *then*

[2]The proof rules of §3.1, the operational semantics of §3.2 and the definitions and lemmas of this section have been formalised in Isabelle/HOL. The proofs of the lemmas and theorems in these three sections have been mechanically checked.

- $(C,\sigma) \xrightarrow{R}{}^{*} (C,\sigma') \iff (C,\sigma) \xrightarrow[\mathbf{e}]{R} (C,\sigma')$
- $(C,\sigma) \xrightarrow{R}{}^{*} \mathit{fault} \iff \exists\sigma''.\ (C,\sigma) \xrightarrow[\mathbf{e}]{R} (C,\sigma'') \xrightarrow[\mathbf{p}]{R} \mathit{fault}$
- $(C,\sigma) \xrightarrow{R}{}^{*} (\mathbf{skip},\sigma') \iff \exists\sigma''\sigma'''.\ (C,\sigma) \xrightarrow[\mathbf{e}]{R} (C,\sigma'') \xrightarrow[\mathbf{p}]{R} (\mathbf{skip},\sigma''') \xrightarrow[\mathbf{e}]{R} (\mathbf{skip},\sigma')$

In a parallel composition, $C_1 \| C_2$, if we have the guarantee of the two commands, then (1) we have the guarantee of their parallel composition; and (2) if the composition can make a reduction, then the two commands can also make that reduction given an extended rely condition.

Lemma 23. *If* $(C_1,\sigma_1,(R \cup G_2))$ **guarantees** G_1 *and* $(C_2,\sigma_2,(R \cup G_1))$ **guarantees** G_2 *and* $\sigma_1 \odot \sigma_2 = \sigma$*, then*

- $(C_1 \| C_2, \sigma, R)$ **guarantees** $G_1 \cup G_2$*;*
- *if* $(C_1 \| C_2,\sigma) \xrightarrow{R}{}^{*} (C_1' \| C_2',\sigma')$*, then there exist* σ_1'*,* σ_2' *such that* $(C_1,\sigma_1) \xrightarrow{R\cup G_2}{}^{*} (C_1',\sigma_1')$*,* $(C_2,\sigma_2) \xrightarrow{R\cup G_1}{}^{*} (C_2',\sigma_2')$*, and* $\sigma_1' \odot \sigma_2' = \sigma'$*.*

This lemma relies on Lemma 20 and on the definition of **guarantees**. The proof is relatively straighforward and is included below for completeness.

Proof of Lemma 23. For the first part, we do an induction on n to prove that for all n, C_1, C_2, l_1, l_2, and s, if $(C_1,(l_1,s),R \cup G_2)$ **guarantees** G_1 and $(C_2,(l_2,s),R \cup G_1)$ **guarantees** G_2, then $(C_1 \| C_2,(l_1 \odot l_2,s),R)$ **guarantees**$_n$ $G_1 \cup G_2$

The base case is trivial. For the $n+1$ case, consider the possible reductions of $C_1 \| C_2$.

- Environment transition: $(C_1 \| C_2,(l_1 \odot l_2,s)) \xrightarrow[\mathbf{e}]{R} (C_1 \| C_2,(l_1 \odot l_2,s'))$ and $R(s,s')$. Hence, $(C_1,(l_1,s)) \xrightarrow[\mathbf{e}]{R} (C_1,(l_1,s'))$ and $(C_2,(l_2,s)) \xrightarrow[\mathbf{e}]{R} (C_2,(l_2,s'))$. The conclusion follows from the induction hypothesis.
- Case $(\mathbf{skip} \| \mathbf{skip},\sigma) \xrightarrow[\mathbf{p}]{R} (\mathbf{skip},\sigma)$. Trivial: follows from the induction hypothesis.
- C_1 makes a program transition: $(C_1,(l_1 \odot l_2,s)) \xrightarrow[\mathbf{p}]{R} (C_1',(l',s'))$. Hence from locality (Lemma 20), $(C_1,(l_1 \odot l_2,s)) \xrightarrow[\mathbf{p}]{R} \mathit{fault}$ or there exists l_1' such that $l' = l_1' \odot l_2$ and $(C_1,(l_1,s)) \xrightarrow[\mathbf{p}]{R} (C_1',(l_1',s'))$.

  As $(C_1,(l_1,s),R \cup G_2)$ **guarantees** G_1, the first case is ruled out. Moreover, we get that $(s,s') \in G_1$ and $(C_1',(l_1',s'),R \cup G_2)$ **guarantees** G_1. Therefore, $(C_2,(l_2,s')) \xrightarrow[\mathbf{e}]{R} (C_2,(l_2,s'))$ and $(C_2,(l_2,s'),R \cup G_1)$ **guarantees** G_2.

  From the induction hypothesis, $(C_1' \| C_2,(l_1' \odot l_2,s'))$ **guarantees**$_n$ $G_1 \cup G_2$. Also, $(s,s') \in G_1 \subseteq G_1 \cup G_2$. Therefore $(C_1 \| C_2,(l_1' \odot l_2,s'))$ **guarantees**$_{n+1}$ $G_1 \cup G_2$, as required.

- C_2 makes a program transition. Symmetric.

- C_1 or C_2 fail. These cases cannot arise because $(C_1, (l_1, s), R \cup G_2)$ **guarantees** G_1 and $(C_2, (l_2, s), R \cup G_1)$ **guarantees** G_2.

For the second part, we do an induction on n to prove that for all n, C_1, C_2, l_1, l_2, and s, if $(C_1, (l_1, s), R \cup G_2)$ **guarantees** G_1, and $(C_2, (l_2, s), R \cup G_1)$ **guarantees** G_2, and $(C_1 \| C_2, (l_1 \odot l_2, s)) \xrightarrow{R}{}^{n} (C_1' \| C_2', (l', s'))$, then there exist l_1', l_2' such that $l_1' \odot l_2' = l'$ and $(C_1, (l_1, s)) \xrightarrow{R \cup G_2}{}^{n} (C_1', (l_1', s'))$ and $(C_2, (l_2, s)) \xrightarrow{R \cup G_1}{}^{n} (C_2', (l_2', s'))$.

The base case is trivial. For the $n+1$ case, consider the possible transitions of $C_1 \| C_2$ that could lead to $C_1' \| C_2'$. There are three cases to consider.

- Environment transition: $(C_1 \| C_2, (l_1 \odot l_2, s)) \xrightarrow[\mathbf{e}]{R} (C_1 \| C_2, (l_1 \odot l_2, s''))$ and $R(s, s'')$.

  Hence, $(C_1, (l_1, s)) \xrightarrow[\mathbf{e}]{R} (C_1, (l_1, s''))$ and $(C_2, (l_2, s)) \xrightarrow[\mathbf{e}]{R} (C_2, (l_2, s''))$. The conclusion then follows from the induction hypothesis.

- C_1 makes a program transition: $(C_1, (l_1 \odot l_2, s)) \xrightarrow[\mathbf{p}]{R} (C_1'', (l'', s''))$.

  From Lemma 20 and $(C_1, (l_1, s), R \cup G_2)$ **guarantees** G_1, there exists l_1'' such that $(C_1, (l_1, s)) \xrightarrow[\mathbf{p}]{R} (C_1'', (l_1'', s''))$, $G_1(s, s'')$, and $(C_1'', (l_1'', s''), R \cup G_2)$ **guarantees** G_1.

  Therefore, $(C_2, (l_2, s'')) \xrightarrow[\mathbf{e}]{R} (C_2, (l_2, s''))$ and $(C_2, (l_2, s''), R \cup G_1)$ **guarantees** G_2.

  From the induction hypothesis, there exist l_1', l_2' such that $(C_1'', (l_1'', s'')) \xrightarrow{R \cup G_2}{}^{n} (C_1', (l_1', s'))$, $(C_2, (l_2'', s'')) \xrightarrow{R \cup G_1}{}^{n} (C_2', (l_2', s'))$, and $l_1' \odot l_2' = l'$.

  Thus, $(C_1, (l_1, s)) \xrightarrow{R \cup G_2}{}^{n+1} (C_1', (l_1', s'))$ and $(C_2, (l_2, s)) \xrightarrow{R \cup G_1}{}^{n+1} (C_2', (l_2', s'))$, as required.

- C_2 makes a program transition. Symmetric. □

The following two lemmas govern sequential composition:

Lemma 24. *If* (C_1, σ, R) **guarantees** G *and for all* σ' *such that* $(C_1, \sigma) \xrightarrow{R}{}^{*} (\mathbf{skip}, \sigma')$, (C_2, σ', R) **guarantees** G, *then* $(C_1; C_2, \sigma, R)$ **guarantees** G.

Lemma 25. *If* $(C_1; C_2, \sigma) \xrightarrow{R}{}^{*} (\mathbf{skip}, \sigma'')$, *then there exists* σ' *such that* $(C_1, \sigma) \xrightarrow{R}{}^{*} (\mathbf{skip}, \sigma')$ *and* $(C_2, \sigma') \xrightarrow{R}{}^{*} (\mathbf{skip}, \sigma'')$.

A simple way to prove the frame rule is to derive it from the rules for parallel composition and the empty command. To do so, we need C and $C \| \mathbf{skip}$ to be observationally equivalent. The following Lemma makes that explicit.

Lemma 26. $(C, \sigma) \xrightarrow{R}{}^{*} (C', \sigma') \iff (C \| \mathbf{skip}, \sigma) \xrightarrow{R}{}^{*} (C' \| \mathbf{skip}, \sigma')$.

The lemmas presented so far depend only on the operational semantics of GPPL, and are independent of the proof rules. The following lemma, however, is about the proof rules. It says that for every specification (p, R, G, q) that a program meets, there is a stronger stable specification (p', R, G, q') that it also satisfies.

Lemma 27. *If* $\vdash C$ **sat** (p, R, G, q)*, then there exist* p' *and* q' *such that* $\models_{\mathsf{RGSep}} p \Rightarrow p'$*,* p' stable under R*,* $\models_{\mathsf{RGSep}} q' \Rightarrow q$*,* q' stable under R*, and* $\vdash C$ **sat** (p', R, G, q')*.*

Proof. By induction on the proof rules.

- Weaken: Applying the induction hypothesis to the premise of the rule, there exist stable p'' and q'' such that $p' \Rightarrow p''$, $q'' \Rightarrow q'$, and $\vdash C$ **sat** (p'', R, G, q''). Since $p \Rightarrow p'$ and $q' \Rightarrow q$, these p'' and q'' satisfy the required assertion.

- Frame: From the induction hypothesis, there exist stable p' and q' such that $p \Rightarrow p'$, $q' \Rightarrow q$, and $\vdash C$ **sat** (p', R, G, q'). As r is stable under R, $p' * r$ and $q' * r$ meet our requirements.

- Skip, Atom, AtomR, Prim: The preconditions and the postconditions of these rules are already stable.

- Loop: From the induction hypothesis, p is stable under R. Hence, take $p' = q' = p$.

- Seq: From the induction hypothesis on C_1, there exist stable p' and q' such that $p \Rightarrow p'$, $q' \Rightarrow q$, and $\vdash C_1$ **sat** (p', R, G, q'). Hence, applying Weaken, we get $\vdash C_1$ **sat** (p', R, G, q). Similarly, from the induction hypothesis for C_2 and Weaken, there exists a stable r' such that $r' \Rightarrow r$ and $\vdash C_2$ **sat** (q, R, G, r'). Hence, from Seq, $\vdash (C_1; C_2)$ **sat** (p', R, G, r') as required.

- Choice: From the induction hypotheses, there exist stable p_1, q_1, p_2, q_2 such that $p \Rightarrow p_1$, $q_1 \Rightarrow q$, $\vdash C_1$ **sat** (p_1, R, G, q_1), $p \Rightarrow p_2$, $q_2 \Rightarrow q$, $\vdash C_2$ **sat** (p_2, R, G, q_2). From Weaken, $\vdash C_1$ **sat** $(p_1 \wedge p_2, R, G, q_1 \vee q_2)$ and $\vdash C_2$ **sat** $(p_1 \wedge p_2, R, G, q_1 \vee q_2)$. From Choice, $\vdash (C_1 + C_2)$ **sat** $(p_1 \wedge p_2, R, G, q_1 \vee q_2)$, as required.

- Par: From the induction hypotheses, there exist stable p_1', q_1', p_2', q_2' such that $p_1 \Rightarrow p_1'$, $q_1' \Rightarrow q_1$, $\vdash C_1$ **sat** $(p_1, R \cup G_2, G_1, q_1)$, $p_2 \Rightarrow p_2'$, $q_2' \Rightarrow q_2$, $\vdash C_2$ **sat** $(p_2, R \cup G_1, G_2, q_2)$. Hence, take $p' = (p_1' * p_2')$, and $q' = (q_1' * q_2')$.

- Disj, Conj, Ex, All: Follows from induction hypotheses, by applying Weaken and the relevant rule, because if p and q are stable then $p \wedge q$, $p \vee q$, $\exists x.\ p$, and $\forall x.\ p$ are stable. □

At last, here is the proof of soundness of RGSep.

Theorem 28 (Soundness)**.** *If* $\vdash C$ **sat** (p, R, G, q)*, then* $\models C$ **sat** (p, R, G, q)*.*

Proof. The proof is by induction on the proof rules, by proving the soundness of each rule separately.

- Atom:

$$\frac{P, Q \text{ precise} \qquad \vdash C \textbf{ sat } (P * P', \emptyset, \emptyset, Q * Q') \qquad (P \rightsquigarrow Q) \subseteq G}{\vdash \langle C \rangle_{P \rightsquigarrow Q} \textbf{ sat } (\boxed{P * F} * P', \emptyset, G, \boxed{Q * F} * Q')} \text{ (Atom)}$$

It suffices to consider three possible reduction sequences (Lemma 22). Moreover, since $R = \emptyset$, we can ignore the environment actions. Therefore, we can assume that $l, s, i \models \boxed{P * F} * P'$ and $(\langle C \rangle_{P \rightsquigarrow Q}, (l, s)) \xrightarrow[\mathbf{p}]{\emptyset} \mathit{Config}$ and prove that there exist l', s' such that $\mathit{Config} = (\textbf{skip}, (l', s'))$ and $(s, s') \in G$ and $l', s', i \models \boxed{Q * F} * Q'$.

1.	$[\![P \rightsquigarrow Q]\!] \subseteq G$	assumption
2.	$\models C$ **sat** $(P * P', \emptyset, \emptyset, Q * Q')$	assumption
3.	$l, s, i \models \boxed{P * F} * P'$	assumption
4.	$s, i \models_{\mathsf{SL}} (P * F)$ and $l, i \models_{\mathsf{SL}} P'$	from 3
5.	$s = s_1 \odot s_0$ and $s_1, i \models_{\mathsf{SL}} P$ and $s_0, i \models_{\mathsf{SL}} F$	from 4, elim-$\exists$
6.	$l \odot s_1, i \models_{\mathsf{SL}} P * P'$	from 4, 5
7.	$(C, (l \odot s_1, \emptyset), \emptyset)$ **guarantees** $\emptyset$	from 6
8.	$(C, (l \odot s, \emptyset), \emptyset)$ **guarantees** $\emptyset$	from 5, 7, Lem 21

If we consider the reductions of $C, (l \odot s, \emptyset), \emptyset$ in the environment $\emptyset$, we have three possibilities:

- $(C, (l \odot s, \emptyset))$ does not terminate.
- $(C, (l \odot s, \emptyset)) \xrightarrow{\emptyset}{}^* \mathit{fault}$
- $(C, (l \odot s, \emptyset)) \xrightarrow{\emptyset}{}^* (\textbf{skip}, (l_2, \emptyset))$.

In the first case, $\langle C \rangle$ does not reduce; hence there is nothing to prove. The second case is ruled out because of (10). Hence, we are left we the third case:

9.	$(C, (l \odot s, \emptyset)) \xrightarrow{\emptyset}{}^* (\textbf{skip}, (l_2, \emptyset))$	
10.	$(C, (l \odot s_1, \emptyset)) \xrightarrow{\emptyset}{}^* (\textbf{skip}, (l_2', \emptyset)) \wedge l_2 = l_2' \odot s_0$	from 8, 9, Lem 20, elim-$\exists$
11.	$l_2', i \models_{\mathsf{SL}} Q * Q'$	from 2, 9
12.	$l_2' = s_2 \odot l''$ and $s_2, i \models_{\mathsf{SL}} Q$ and $l'', i \models Q'$	from 11, elim-$\exists$
13.	$l_2 = l'' \odot s_0 \odot s_2$	from 10, 12
14.	$\mathsf{oracle}_{P \rightsquigarrow Q}(s, l_2) = s_0 \odot s_2$	from 5, 12, 13

We now proceed by case analysis on the possible reduction rules. The two faulting rules do not apply, because from 10 we know the body cannot fail, and from 16 we know $\mathsf{oracle}_{P' \rightsquigarrow Q'}(s, l_2)$ is defined. Hence, the only applicable rule is:

$$\frac{\mathsf{oracle}_{P \rightsquigarrow Q}(s, l_2) = (l', s') \qquad (C, (l \odot s, \mathsf{u})) \xrightarrow{\emptyset}{}^{*} (\mathbf{skip}, (l_2, \mathsf{u}))}{(\langle C \rangle_{P \rightsquigarrow Q}, (l, s)) \xrightarrow[\mathbf{p}]{R} (\mathbf{skip}, (l', s'))}$$

Therefore, we can further assume that $l' \odot s' = l_2$ and $s' = s_0 \odot s_2$.

15.	$l' \odot (s_0 \odot s_2) = l_2$	assumption
16.	$l' = l''$	from 13, 15, $\odot$ cancellation
17.	$l', (s_0 \odot s_2), i \models_{\mathsf{RGSep}} \boxed{Q * F} * Q''$	from 5, 12, 16
18.	$(s_1 \odot s_0, s_2 \odot s_0) \in [\![P' \rightsquigarrow Q']\!]$	from 5, 12
19.	$(s_1 \odot s_0, s_2 \odot s_0) \in G$	from 18, 1

We have shown properties 17 and 19, as required.

- AtomR: It suffices to consider three possible reduction sequences (Lemma 22). Moreover, as p and q are stable under R, we can ignore the environment actions. Hence, we can assume $l, s, i \models p$ and $(\langle C \rangle_{P' \rightsquigarrow Q'}, (l, s)) \xrightarrow[\mathbf{p}]{R} \mathit{Config}$ and prove that there exist l', s' s.t. $\mathit{Config} = (\mathbf{skip}, (l', s'))$ and $(s, s') \in G$ and $l', s', i \models q$, which follows directly from $\models \langle C \rangle$ **sat** $(p, \emptyset, G, q)$.

- Prim: Again, it suffices to consider three possible reduction sequences (Lemma 22). Assume $\sigma, i \models_{\mathsf{RGSep}} P$, and prove (i) (c, σ, R) **guarantees** G and (ii) if $(c, \sigma) \xrightarrow{R}{}^{*} (\mathbf{skip}, \sigma')$, then $\sigma', i \models_{\mathsf{RGSep}} Q$.

  To prove (i), assume $(c, \sigma) \xrightarrow[\mathbf{e}]{R} (c, \sigma') \xrightarrow[\mathbf{p}]{R} \mathit{Config}$ and prove exists σ'' such that $\mathit{Config} = (\mathbf{skip}, \sigma'')$ and $(\sigma', \sigma'') \in G$. Let $(l, s) = \sigma$ and $(l', s') = \sigma'$. By the first reduction, we know $l = l'$, and as P only depends on local state, then $\sigma' \models P$. Therefore, *Config* is not a fault, and hence $\sigma'' = (l'', s')$; so $(\sigma', \sigma'') \in G$.

  To prove (ii), assume $(c, \sigma) \xrightarrow[\mathbf{e}]{R} (c, \sigma') \xrightarrow[\mathbf{p}]{R} (\mathbf{skip}, \sigma'') \xrightarrow[\mathbf{e}]{R} (\mathbf{skip}, \sigma''')$. By construction, we know $\sigma'', i \models_{\mathsf{SL}} Q$. Let $(l'', s'', e'') = \sigma''$ and $(l''', s''', e''') = \sigma'''$, hence $l'' = l'''$. Therefore, $\sigma''', i \models_{\mathsf{RGSep}} Q$ as required.

- Skip: Trivial.

- Disj, Conj, Ex, All: Trivial.

- Par: Follows from Lemma 23.

- Seq: Follows from Lemmas 24 and 25.

- Weaken: Follows from Lemmas 20 and 21.

- Loop: The proof of this rule also assumes that both $\vdash C$ **sat** (p, R, G, p) and $\models C$ **sat** (p, R, G, p). By induction on n, first prove that $\forall n.\ \models C^n$ **sat** (p, R, G, p).

Base case: $C^0 \stackrel{\text{def}}{=} \textbf{skip}$. Since $\vdash C\ \textbf{sat}\ (p, R, G, p)$, according to Lemma 27, p is stable under R. Hence, as SKIP is sound, $\vDash C^0\ \textbf{sat}\ (p, R, G, p)$.

Inductive step: $C^{n+1} \stackrel{\text{def}}{=} (C; C^n)$ Applying the SEQ rule,

$$\frac{\vDash C\ \textbf{sat}\ (p, R, G, p) \qquad \dfrac{\text{ind. hypothesis}}{\vDash C^n\ \textbf{sat}\ (p, R, G, p)}}{\vDash C^{n+1}\ \textbf{sat}\ (p, R, G, p)}\ \textsc{Seq}$$

Hence, for all n, $\vDash C^n\ \textbf{sat}\ (p, R, G, p)$. Finally, by the CHOICE rule, the same follows for $C^* = \bigoplus_n C^n$.

- FRAME: As C and $C \| \textbf{skip}$ are equivalent with respect to the operational semantics (cf. Lemma 26), we can derive the frame rule from the parallel rule:

$$\dfrac{\dfrac{\vDash C\ \textbf{sat}\ (p, R, G, q) \qquad \dfrac{r \text{ stable under } R \cup G}{\vDash \textbf{skip}\ \textbf{sat}\ (r, R \cup G, \emptyset, r)}\ \textsc{Skip}}{\vDash (C \| \textbf{skip})\ \textbf{sat}\ (p * r, R, G, q * r)}\ \textsc{Par}}{\vDash C\ \textbf{sat}\ (p * r, R, G, q * r)}$$

□

3.4 Encodings of SL and RG

Separation logic (without resource invariants) and rely/guarantee are trivial special cases of RGSep. This is best illustrated by the parallel composition rule:

$$\frac{\begin{array}{c}\vdash C_1\ \textbf{sat}\ (p_1, R \cup G_2, G_1, q_1) \\ \vdash C_2\ \textbf{sat}\ (p_2, R \cup G_1, G_2, q_2)\end{array}}{\vdash C_1 \| C_2\ \textbf{sat}\ (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2)} \qquad (\textsc{Par})$$

- When all the state is shared, we get the standard rely/guarantee rule. In this case, as the local state is empty, we get $p_1 * p_2 \iff p_1 \wedge p_2$ and $q_1 * q_2 \iff q_1 \wedge q_2$.

  Formally, one can encode $\vdash C\ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, Q)$ as $\vdash C\ \textbf{sat}\ (\boxed{P}, R, G, \boxed{Q})$.

- When all the state is local, we get the separation logic rules. (Since there is no shared state, we do not need to describe its evolution: R and G are simply the identity relation.)

  Formally, one can encode $\vdash_{\mathsf{SL}} \{P\}\ C\ \{Q\}$ as $\vdash C\ \textbf{sat}\ (P, \emptyset, \emptyset, Q)$.

Deriving the proof rules of rely/guarantee and separation logic from their encodings in RGSep is straightforward, and omitted.

Resource invariants Encoding full concurrent separation logic [58] in RGSep is similarly quite easy. The resource invariant J is simply threaded through the assertions, and the rely and guarantee conditions assert that it remains unaffected by interference. Formally, encode $J \vdash_{\mathsf{SL}} \{P\}\ C\ \{Q\}$ as $\vdash C\ \mathbf{sat}\ (P * \boxed{J},\ R,\ R,\ Q * \boxed{J})$, where $R = \{J \rightsquigarrow J\}$.

Again the proof is straightforward, but for demonstration let us derive the concurrent separation logic rule for atomic blocks (SL-Atomic).

$$\dfrac{\dfrac{\models C\ \mathbf{sat}\ (J * P, \emptyset, \{J \rightsquigarrow J\}, J * Q) \qquad J \text{ precise} \qquad \dfrac{}{(J \rightsquigarrow J) \subseteq \{J \rightsquigarrow J\}}\ \text{G-Axiom}}{\models \langle C \rangle\ \mathbf{sat}\ (\boxed{J * emp} * P, \emptyset, \{J \rightsquigarrow J\}, \boxed{J * emp} * Q)}\ \textsc{Atom}}{\models \langle C \rangle\ \mathbf{sat}\ (\boxed{J} * P, \emptyset, \{J \rightsquigarrow J\}, \boxed{J} * Q)}$$

Now, apply rule AtomR, where (†) is the previous proof tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\models_{\mathsf{SL}} (J \mathbin{-\!\circledast} J) * J \Rightarrow J}}{\boxed{J}\ \mathsf{stable\ under}\ \{J \rightsquigarrow J\}}}{\boxed{J} * P\ \mathsf{stable\ under}\ \{J \rightsquigarrow J\}} \qquad \dfrac{\dfrac{\dfrac{}{\models_{\mathsf{SL}} (J \mathbin{-\!\circledast} J) * J \Rightarrow J}}{\boxed{J}\ \mathsf{stable\ under}\ \{J \rightsquigarrow J\}}}{\boxed{J} * Q\ \mathsf{stable\ under}\ \{J \rightsquigarrow J\}} \qquad (\dagger)}{\vdash \langle C \rangle\ \mathbf{sat}\ (\boxed{J} * P, \{J \rightsquigarrow J\}, \{J \rightsquigarrow J\}, \boxed{J} * Q)}\ \textsc{AtomR}$$

Hence, we have derived the concurrent separation logic rule for atomic commands:

$$\dfrac{\vdash_{\mathsf{SL}} \{P * J\}\ C\ \{Q * J\} \qquad J \text{ is precise}}{J \vdash_{\mathsf{SL}} \{P\}\ \langle C \rangle\ \{Q\}} \qquad \text{(SL-Atomic)}$$

3.5 Example: lock-coupling list

This section demonstrates a fine-grained concurrent linked list implementation of a mutable set data structure. Instead of having a single lock for the entire list, there is one lock per list node.

Locks Here is the source code for locking and unlocking a node:

```
lock(p)   { atomic(p.lock = 0) { p.lock := tid; } }
unlock(p) { atomic { p.lock := 0; } }
```

The simplest implementation for `lock` is to use a conditional critical region, which is just syntactic sugar for an atomic block whose body starts with an assume command (see §2.2). These locks store the identifier of the thread that acquired the lock: `tid` represents the thread identifier of the current thread. Storing thread identifiers is not necessary for the algorithm's correctness, but it facilitates the proof. Similarly, `unlock` uses an `atomic` block to indicate that the write to `p.lock` must be done indivisibly.

The following three predicates represent a node in the list: (1) $N_s(x, v, y)$ represents a node at location x with contents v and tail pointer y and with the lock status set to s;

```
locate(e) {
  local p, c;
  p := Head;
  lock(p);
  c := p.next;
  while (c.value < e) {
    lock(c);
    unlock(p);
    p := c;
    c := p.next;
    lock(c);
  }
  return(p, c);
}
```

```
remove(e) {
  local x, y, z;
  (x, y) := locate(e);
  if (y.value = e) {
    lock(y);
    z := y.next;
    x.next := z;
    unlock(x);
    dispose(y);
  } else {
    unlock(x);
  }
}
```

```
add(e) {
  local x, y, z;
  (x, z) := locate(e);
  if (z.value ≠ e) {
    y := new Node();
    y.lock := 0;
    y.value := e;
    y.next := z;
    x.next := y;
  }
  unlock(x);
}
```

Figure 3.3: Source code for lock coupling list operations.

(2) $U(x, v, y)$ represents an unlocked node at location x with contents v and tail pointer y; and (3) $L_t(x, v, y)$ represents a node locked with thread identifier t. We will write $N_{\_}(x, v, y)$ for a node that may or may not be locked.

$$\begin{array}{lcl} N_s(x,v,y) & \stackrel{\text{def}}{=} & x \mapsto \{.\text{lock}{=}s; .\text{value}{=}v; .\text{next}{=}y\} \\ U(x,v,y) & \stackrel{\text{def}}{=} & N_0(x,v,y) \\ L_t(x,v,y) & \stackrel{\text{def}}{=} & N_t(x,v,y) \wedge t > 0 \end{array}$$

The thread identifier parameter in the locked node is required to specify that a node can only be unlocked by the thread that locked it.

$$t \in T \wedge U(x,v,n) \rightsquigarrow L_t(x,v,n) \qquad \text{(Lock)}$$

$$t \in T \wedge L_t(x,v,n) \rightsquigarrow U(x,v,n) \qquad \text{(Unlock)}$$

The Lock and Unlock actions are parameterised with a set of thread identifiers, T. This allows us to use the actions to represent both relies and guarantees. In particular, we take a thread with identifier `tid` to have the guarantee with $T = \{\texttt{tid}\}$, and the rely to use the complement on this set.

From the Atomic rule, we can derive the following rules for the lock primitives. (Arguably, these specifications are not as simple as one might hope. Chapter 4 describes variant proof rules that enable much simpler specifications for `lock` and `unlock`.)

$$\frac{\begin{array}{cc} P \text{ stable under } R & Q \text{ stable under } R \\ P \Rightarrow N_{\_}(\text{p}, v, n) * F & L_{\text{tid}}(\text{p}, v, n) * F \Rightarrow Q \end{array}}{\vdash \texttt{lock(p)} \textbf{ sat } (\boxed{P}, R, G, \boxed{Q})} \qquad \text{(Lock)}$$

$$\frac{P \text{ stable under } R \qquad Q \text{ stable under } R \qquad P \Rightarrow L_{\text{tid}}(\mathrm{p}, v, n) * F \qquad U(\mathrm{p}, v, n) * F \Rightarrow Q}{\vdash \texttt{unlock(p)} \textbf{ sat } (\boxed{P}, R, G, \boxed{Q})} \quad \text{(UNLOCK)}$$

The Algorithm Now we build a fine-grained concurrent linked list implementation of a set using the lock mechanism we have defined. The list has operations `add`, which adds an element to the set, and `remove`, which removes an element from the set. Traversing the list uses *lock coupling*: the lock on one node is not released until the next node is locked. Somewhat like a person climbing a rope "hand-over-hand," you always have at least one hand on the rope.

Figure 3.3 contains the source code. An element is added to the set by inserting it in the appropriate position while holding the lock of its previous node. It is removed by redirecting the previous node's pointer while both the previous and the current node are locked. This ensures that deletions and insertions can happen concurrently in the same list. The algorithm makes two assumptions about the list: (1) it is sorted; and (2) the first and last elements have sentinel values $-\infty$ and $+\infty$ respectively. This allows us to avoid checking for the end of the list.

First, consider the action of adding a node to the list. Here is an action that ignores the sorted nature of the list:

$$t \in T \wedge L_t(x, u, n) \rightsquigarrow L_t(x, u, m) \wedge U(m, v, n)$$

To add an element to the list, we must have locked the previous node, and then we can swing the tail pointer to the added node. The added node must have the same tail as previous node before the update.

To ensure that the sorted order of the list is preserved, the actual action must be specified with respect to the next node as well. We ensure the value we add is between the previous and next values.

$$\begin{aligned}(t \in T) \wedge (u < v < w) \wedge (L_t(x, u, n) * N_s(n, w, y)) \\ \rightsquigarrow L_t(x, u, m) * U(m, v, n) * N_s(n, w, y)\end{aligned} \quad \text{(Insert)}$$

The final permitted action is to remove an element from the list. The action specifies that to remove node n from the list, both n and the previous node (x) must be locked. The tail of the previous node is then updated to the removed node's tail, m.

$$(t \in T) \wedge (v < \infty) \wedge (L_t(x, u, n) * L_t(n, v, m)) \rightsquigarrow L_t(x, u, m) \quad \text{(Remove)}$$

We define $\mathcal{I}(T)$ as the four actions given above: Lock, Unlock, Insert and Remove. These are depicted in Figure 3.4. $\mathcal{I}(\{t\})$ allows the thread t: (i) to lock an unlocked node, (ii)

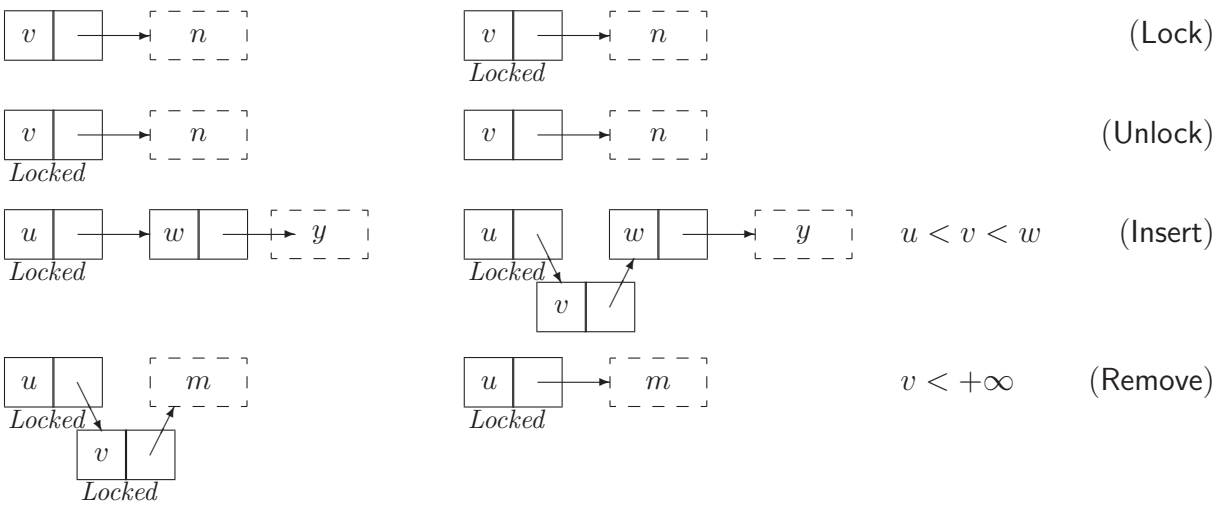

Figure 3.4: Pictorial representation of the actions

to unlock a node that it had locked, (*iii*) to insert a node in the list immediately after a node that it had locked, and (*iv*) if two adjacent nodes in the list are locked by t, to remove the second node from the list by swinging a pointer past it. For a thread with thread identifier t, take $R = \mathcal{I}(\overline{\{t\}})$ and $G = \mathcal{I}(\{t\})$.

We can use separation to describe the structure of the shared list. The following predicate, $ls(x, A, y)$, describes a list segment starting at location x with the final tail value of y, and with contents A. We write ϵ for the empty sequence and $\cdot$ for the concatenation of two sequences.

$$\begin{aligned} ls(x, A, y) \stackrel{\text{def}}{=}\ & (x = y \wedge A = \epsilon \wedge emp) \\ & \vee\ (\exists v z B.\ x \neq y \wedge A = v{\cdot}B * N\_(x, v, z) * ls(z, B, y)) \end{aligned}$$

Because of separation logic, we do not need any reachability predicates. Instead, the 'list segment' predicate is simply a recursively defined predicate. The definition above ensures that the list does not contain any cycles.

The algorithm works on sorted lists with the first and last values being $-\infty$ and $+\infty$ respectively. We define $s(A)$ to represent this restriction on a logical list A.

$$sorted(A) \stackrel{\text{def}}{=} \begin{cases} true & \textbf{if } (A = \epsilon) \vee (A = a{\cdot}\epsilon) \\ (a < b) \wedge sorted(b.B) & \textbf{if } (A = a{\cdot}b{\cdot}B) \end{cases}$$

$$s(A) \stackrel{\text{def}}{=} (\exists B.\ A = -\infty{\cdot}B{\cdot}{+}\infty) \wedge sorted(A) \wedge emp$$

Figures 3.5 and 3.6 contain the proof outlines of `locate`, `add`, and `remove`. The outline presents the intermediate assertions in the proof. Further, we must prove that every shared state assertion is stable under the rely. These proofs involve reasoning

$$
\begin{array}{l}
\texttt{locate(e)}\ \{ \\
\quad \texttt{local p, c, t;} \\
\quad \left\{ \boxed{\exists A.\ ls(\texttt{Head}, A, \mathrm{nil}) * s(A)} \wedge -\infty < \texttt{e} \right\} \\
\quad \texttt{p} := \texttt{Head}; \\
\quad \left\{ \boxed{\begin{array}{l} \exists Z B.\ ls(\texttt{Head}, \epsilon, \texttt{p}) * N(\texttt{p}, -\infty, Z) \\ * ls(Z, B, \mathrm{nil}) * s(-\infty \cdot B) \end{array}} \wedge -\infty < \texttt{e} \right\} \\
\quad \texttt{lock(p)}; \\
\quad \left\{ \exists Z.\ \boxed{\begin{array}{l} \exists B.\ ls(\texttt{Head}, \epsilon, \texttt{p}) * L(\texttt{p}, -\infty, Z) \\ * ls(Z, B, \mathrm{nil}) * s(-\infty \cdot B) \end{array}} \wedge -\infty < \texttt{e} \right\} \\
\quad \langle \texttt{c} := \texttt{p.next}; \rangle \\
\quad \left\{ \boxed{\begin{array}{l} \exists B.\ ls(\texttt{Head}, \epsilon, \texttt{p}) * L(\texttt{p}, -\infty, \texttt{c}) \\ * ls(\texttt{c}, B, \mathrm{nil}) * s(-\infty \cdot B) \end{array}} \wedge -\infty < \texttt{e} \right\} \\
\quad \langle \texttt{t} := \texttt{c.value}; \rangle \\
\quad \left\{ \exists u.\ \boxed{\begin{array}{l} \exists A B Z.\ ls(\texttt{Head}, A, \texttt{p}) * L(\texttt{p}, u, \texttt{c}) \\ * N(\texttt{c}, \texttt{t}, Z) * ls(\texttt{c}, B, \mathrm{nil}) * s(A \cdot u \cdot \texttt{t} \cdot B) \end{array}} \wedge u < \texttt{e} \right\} \\
\quad \texttt{while}\ (\texttt{t} < \texttt{e})\ \{ \\
\qquad \left\{ \exists u.\ \boxed{\begin{array}{l} \exists A B Z.\ ls(\texttt{Head}, A, \texttt{p}) * L(\texttt{p}, u, \texttt{c}) \\ * N(\texttt{c}, \texttt{t}, Z) * ls(\texttt{c}, B, \mathrm{nil}) * s(A \cdot u \cdot \texttt{t} \cdot B) \end{array}} \wedge u < \texttt{e} \wedge \texttt{t} < \texttt{e} \right\} \\
\qquad \texttt{lock(c)}; \\
\qquad \left\{ \exists u Z.\ \boxed{\begin{array}{l} \exists A B.\ ls(\texttt{Head}, A, \texttt{p}) * L(\texttt{p}, u, \texttt{c}) \\ * L(\texttt{c}, \texttt{t}, Z) * ls(Z, B, \mathrm{nil}) * s(A \cdot u \cdot \texttt{t} \cdot B) \end{array}} \wedge \texttt{t} < \texttt{e} \right\} \\
\qquad \texttt{unlock(p)}; \\
\qquad \left\{ \exists Z.\ \boxed{\begin{array}{l} \exists A B.\ ls(\texttt{Head}, A, \texttt{c}) * L(\texttt{c}, \texttt{t}, Z) \\ * ls(Z, B, \mathrm{nil}) * s(A \cdot \texttt{t} \cdot B) \end{array}} \wedge \texttt{t} < \texttt{e} \right\} \\
\qquad \texttt{p} := \texttt{c}; \\
\qquad \left\{ \exists u Z.\ \boxed{\begin{array}{l} \exists A B.\ ls(\texttt{Head}, A, \texttt{p}) * L(\texttt{p}, u, Z) \\ * ls(Z, B, \mathrm{nil}) * s(A \cdot u \cdot B) \end{array}} \wedge u < \texttt{e} \right\} \\
\qquad \langle \texttt{c} := \texttt{p.next}; \rangle \\
\qquad \left\{ \exists u.\ \boxed{\begin{array}{l} \exists A B.\ ls(\texttt{Head}, A, \texttt{p}) * L(\texttt{p}, u, \texttt{c}) \\ * ls(\texttt{c}, B, \mathrm{nil}) * s(A \cdot u \cdot B) \end{array}} \wedge u < \texttt{e} \right\} \\
\qquad \langle \texttt{t} := \texttt{c.value}; \rangle \\
\qquad \left\{ \exists u.\ \boxed{\begin{array}{l} \exists A B Z.\ ls(\texttt{Head}, A, \texttt{p}) * L(\texttt{p}, u, \texttt{c}) \\ * N(\texttt{c}, \texttt{t}, Z) * ls(Z, B, \mathrm{nil}) * s(A \cdot u \cdot \texttt{t} \cdot B) \end{array}} \wedge u < \texttt{e} \right\} \\
\quad \} \\
\quad \left\{ \exists u v.\ \boxed{\begin{array}{l} \exists A B Z.\ ls(\texttt{Head}, A, \texttt{p}) * L(\texttt{p}, u, \texttt{c}) \\ * N(\texttt{c}, v, Z) * ls(Z, B, \mathrm{nil}) * s(A \cdot u \cdot v \cdot B) \end{array}} \wedge u < \texttt{e} \wedge \texttt{e} \leq v \right\} \\
\quad \texttt{return}\ (\texttt{p}, \texttt{c}); \\
\}
\end{array}
$$

Figure 3.5: Outline verification of `locate`.

$\texttt{add}(\texttt{e})$ { $\texttt{local}\ \texttt{x},\texttt{y},\texttt{z},\texttt{t};$
$\{\boxed{\exists A.\ ls(\texttt{Head}, A, \texttt{nil}) * s(A)} \wedge -\infty < \texttt{e}\}$
$(\texttt{x},\texttt{z}) := \texttt{locate}(\texttt{e});$
$\left\{\exists uv.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{z}) * N(\texttt{z}, v, Z)\\ * ls(Z, B, \texttt{nil}) * s(A{\cdot}u{\cdot}v{\cdot}B)\end{array}} \wedge u < \texttt{e} \wedge \texttt{e} \leq v\right\}$
$\langle \texttt{t} = \texttt{z.value}; \rangle$ $\texttt{if}(\texttt{t} \neq \texttt{e})$ {
$\left\{\exists uv.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{z}) * N(\texttt{z}, v, Z)\\ * ls(Z, B, \texttt{nil}) * s(A{\cdot}u{\cdot}v{\cdot}B)\end{array}} \wedge u < \texttt{e} \wedge \texttt{e} < v\right\}$
$\texttt{y} = \texttt{cons}(0, \texttt{e}, \texttt{z});$
$\left\{\exists uv.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{z}) * N(\texttt{z}, v, Z)\\ * ls(Z, B, \texttt{nil}) * s(A{\cdot}u{\cdot}v{\cdot}B)\end{array}} * U(\texttt{y}, \texttt{e}, \texttt{z}) \wedge u < \texttt{e} \wedge \texttt{e} < v\right\}$
$\langle \texttt{x.next} = \texttt{y}; \rangle$
$\{\exists uv.\ \boxed{\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{y}) * N(\texttt{y}, \texttt{e}, Z) * ls(Z, B, \texttt{nil}) * s(A{\cdot}u{\cdot}\texttt{e}{\cdot}B)}\}$
}
$\texttt{unlock}(\texttt{x});$
$\{\exists v.\ \boxed{\exists A.\ ls(\texttt{Head}, A, \texttt{nil}) * s(A)}\}$
}

$\texttt{remove}(\texttt{e})$ { $\texttt{local}\ \texttt{x},\texttt{y},\texttt{z},\texttt{t};$
$\{\boxed{\exists A.\ ls(\texttt{Head}, A, \texttt{nil}) * s(A)} \wedge -\infty < \texttt{e} \wedge \texttt{e} < +\infty\}$
$(\texttt{x},\texttt{y}) = \texttt{locate}(\texttt{e});$
$\left\{\exists uv.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{y}) * N(\texttt{y}, v, Z)\\ * ls(Z, B, \texttt{nil}) * s(A{\cdot}u{\cdot}v{\cdot}B)\end{array}} \wedge u < \texttt{e} \wedge \texttt{e} \leq v \wedge \texttt{e} < +\infty\right\}$
$\langle \texttt{t} = \texttt{y.value}; \rangle$ $\texttt{if}(\texttt{t} = \texttt{e})$ {
$\left\{\exists u.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{y}) * N(\texttt{y}, \texttt{e}, Z)\\ * ls(Z, B, \texttt{nil}) * s(A{\cdot}u{\cdot}\texttt{e}{\cdot}B)\end{array}} \wedge \texttt{e} < +\infty\right\}$
$\texttt{lock}(\texttt{y});$
$\left\{\exists u.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{y}) * L(\texttt{y}, \texttt{e}, Z)\\ * ls(Z, B, \texttt{nil}) * s(A{\cdot}u{\cdot}\texttt{e}{\cdot}B)\end{array}} \wedge \texttt{e} < +\infty\right\}$
$\langle \texttt{z} := \texttt{y.next}; \rangle$
$\left\{\exists u.\ \boxed{\begin{array}{l}\exists AB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{y}) * L(\texttt{y}, \texttt{e}, \texttt{z})\\ * ls(\texttt{z}, B, \texttt{nil}) * s(A{\cdot}u{\cdot}\texttt{e}{\cdot}B)\end{array}} \wedge \texttt{e} < +\infty\right\}$
$\langle \texttt{x.next} := \texttt{z}; \rangle$
$\{\exists u.\ \boxed{\exists AB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{z}) * ls(\texttt{z}, B, \texttt{nil}) * s(A{\cdot}u{\cdot}B)} * L(\texttt{y}, \texttt{e}, \texttt{z})\}$
$\texttt{unlock}(\texttt{x});$
$\{\boxed{\exists A.\ ls(\texttt{Head}, A, \texttt{nil}) * s(A)} * L(\texttt{y}, \texttt{e}, \texttt{z})\}$
$\texttt{dispose}(\texttt{y});$
} $\texttt{else}$ { $\{\exists u.\ \boxed{\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{y}) * ls(\texttt{y}, B, \texttt{nil}) * s(A{\cdot}u{\cdot}B)}\}$
$\texttt{unlock}(\texttt{x});$ }
$\{\boxed{\exists A.\ ls(\texttt{Head}, A, \texttt{nil}) * s(A)}\}$
}

Figure 3.6: Outline verification of `add` and `remove`.

about the septraction operator ($-\circledast$); they are long, but straightforward. Since Chapter 6 describes how they can be automated, it is unnecessary to present them here. For an example of such a proof, please see [74].

Theorem 29. *The lock coupling algorithm is safe, and maintains the sorted nature of the list.*

3.6 Related work (SAGL)

Concurrently with this work, Feng, Ferreira and Shao [27] proposed a different combination of rely/guarantee and separation logic, SAGL. Both RGSep and SAGL partition memory into shared and private parts, but there are some notable differences:

Assertion language SAGL has different assertions for the local state and for the shared state. As it has multiple regions of shared state, it has a different assertion for each region. On the contrary, RGSep has a unified assertion language that describes both the local and the shared state. Hence, in RGSep, we can write an assertion that links the local and the shared state. We can also do the same in SAGL, but we need quantification at the meta-level.

Rely/guarantee representation In SAGL, the rely and guarantee conditions are relations and stability checks are semantic implications. RGSep, however, provides convenient syntax for writing down these relations, and reduces the semantic implication into a simple logic implication (see Lemma 13, §3.1.3).

Separation logic inclusion SAGL is presented as a logic for assembly code, and was not intended to be applied at different abstraction levels. In this presentation, it does not have separation logic as a proper subsystem, as it lacks the standard version of the frame rule [69]. This means that it cannot prove the usual separation logic specification of procedures such as `copy_tree` [60]. It should be possible to extend SAGL to include the frame rule for procedures, but such extension is by no means obvious.

In contrast, RGSep subsumes separation logic [69], as well as the single-resource variant of concurrent separation logic [58]. Hence, the same proofs there (for a single resource) go through directly in RGSep (for procedures see Chapter 4). Of course, the real interest in these logics is the treatment of additional examples, such as lock coupling, that neither separation logic nor rely/guarantee can handle tractably.

Atomicity assumptions In SAGL, every primitive command is assumed to be atomic. RGSep instead requires one to specify what is atomic; everything else is considered non-atomic. In RGSep, non-atomic commands cannot update shared state, so we only need

stability checks when there is an atomic command. On the other hand, SAGL must check stability after every single command.

With this all being said, there are remarkable similarities between RGSep and SAGL. That they were arrived at independently is perhaps encouraging as to the naturalness of the basic ideas.

Chapter 4

Practical use of RGSep

Chapter 3 introduced RGSep, a logic that combines rely/guarantee and separation logic. This enabled us to prove that lock-coupling list preserves the sorted list structure and does not leak memory (see §3.5). Proving the same property in concurrent separation logic is extremely difficult, if at all possible. In this chapter, we will consider peripheral issues, which are important in the practical application of RGSep.

- §4.1 considers three variant proof systems that attach the stability checks to different proof rules. The last of these systems, *mid stability*, enables programs consisting of a single atomic block to be specified concisely.

- §4.2 contains additional rules for checking that a program satisfies its guarantee condition. These rules facilitate reasoning about programs with complex atomic actions such as CAS.

- §4.3 observes that the standard proof rules for procedures and higher-order separation logic carry across to RGSep. Moreover, we can define multiple disjoint regions of shared state and statically scoped interference.

- §4.3.3 sketches an extension to actions with a local guard restricting when an action can be applied. This extension permits us to reason about locks without having to introduce thread identifiers.

- Finally, §4.4 observes that non-atomic accesses to shared memory can be treated as sequences of atomic accesses. It presents generic encondings for non-atomic reads and writes, and derives two proof rules for non-atomic reads.

4.1 When to check stability

When it comes to stability checks, the proof rules of Chapter 3 are too conservative. They check that assertions are stable on the entries and the exits of atomic blocks, at the frame rule and at the Skip axiom. Hence, given a sequence of atomic commands each intermediate assertion is checked for stability twice: once at the exit of an atomic command and at the entry of its successor.

Here, we shall see three alternative proof systems with slightly different properties. Instead of requiring that all assertions are stable by construction, we can also assign a meaning to unstable assertions. There are two sensible choices: an unstable assertion could stand either for the strongest stable assertion it entails or for the weakest stable assertion that it is entailed by. This is represented in the following definitions:

Definition 30 (Weakest stable stronger assertion)**.**

- $\mathsf{wssa}_R(q) \Rightarrow q$*;*
- $\mathsf{wssa}_R(q)$ stable under R*; and*
- *for all* p*, if* p stable under R *and* $p \Rightarrow q$*, then* $p \Rightarrow \mathsf{wssa}_R(q)$*.*

Definition 31 (Strongest stable weaker assertion)**.**

- $p \Rightarrow \mathsf{sswa}_R(p)$*;*
- $\mathsf{sswa}_R(p)$ stable under R*; and*
- *for all* q*, if* q stable under R *and* $p \Rightarrow q$*, then* $\mathsf{sswa}_R(p) \Rightarrow q$*.*

Given a binary relation $R : M \times M$, $\mathsf{wssa}_R(p)$ stands for the weakest assertion that is stronger than p and whose description of the shared state is stable under R. Dually, let $\mathsf{sswa}_R(p)$ be the strongest assertion that is weaker than p and stable under R.

It is easy to see that for every p and R, $\mathsf{wssa}_R(p)$ and $\mathsf{sswa}_R(p)$ are well defined: Both *true* and *false* are stable under R; and for every p_1 and p_2, if p_1 and p_2 are both stable under R, then so are $p_1 \wedge p_2$ and $p_1 \vee p_2$.

These definitions provide the semantic basis for the following three new proof systems.

Early stability (at the forks of parallel composition and the exits of atomic blocks):
$$\vDash C\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q) \iff^{\text{def}} \forall R' \subseteq R.\ \vDash C\ \mathbf{sat}\ (\mathsf{wssa}_{R'}(p), R', G, \mathsf{wssa}_{R'}(q))$$

Late stability (at the exits of atomic blocks and at the joins of parallel composition):
$$\vDash C\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q) \iff^{\text{def}} \forall R' \subseteq R.\ \vDash C\ \mathbf{sat}\ (\mathsf{sswa}_{R'}(p), R', G, \mathsf{sswa}_{R'}(q))$$

Mid stability (at sequencing and at the forks and the joins of parallel composition):
$$\vDash C\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q) \iff^{\text{def}} \forall R' \subseteq R.\ \vDash C\ \mathbf{sat}\ (\mathsf{wssa}_{R'}(p), R', G, \mathsf{sswa}_{R'}(q))$$

Since stability depends on the rely condition and during the proof the rely condition can be strengthened according to the Weaken rule, the definitions quantify over all stronger

rely conditions R'. The fourth combination, C **sat** $(\mathsf{sswa}_{R'}(p), R', G, \mathsf{wssa}_{R'}(q))$, places a stronger requirement on C than (p, R, G, q), but because of Lemma 27 of §3.3 it gives rise to the same proof rules as C **sat** (p, R, G, q).

Lemma 32. $(\forall R' \subseteq R.\ \vDash C \textbf{ sat } (\mathsf{sswa}_{R'}(p), R, G, \mathsf{wssa}_{R'}(q))) \implies \vDash C \textbf{ sat } (p, R, G, q)$.

Proof. Apply the WEAKEN rule, as $p \Rightarrow \mathsf{sswa}_{R'}(p)$ and $\mathsf{wssa}_{R'}(q) \Rightarrow q$. □

The rest of this section presents the proof rules for early, late, and mid stability and proves their soundness. In order to carry out these soundness proofs, we shall use the following properties of $\mathsf{wssa}_R(\_)$ and $\mathsf{sswa}_R(\_)$.

Lemma 33 (Properties of weakest stable stronger assertions)**.**

- $\mathsf{wssa}_\emptyset(p) \iff p$
- $(p \Rightarrow q) \implies (\mathsf{wssa}_R(p) \Rightarrow \mathsf{wssa}_R(q))$
- $\mathsf{wssa}_R(p * q) \iff \mathsf{wssa}_R(p) * \mathsf{wssa}_R(q)$

Lemma 34 (Properties of strongest stable weaker assertions)**.**

- $\mathsf{sswa}_\emptyset(p) \iff p$
- $(p \Rightarrow q) \implies (\mathsf{sswa}_R(p) \Rightarrow \mathsf{sswa}_R(q))$
- $\mathsf{sswa}_R(p * q) \iff \mathsf{sswa}_R(p) * \mathsf{sswa}_R(q)$

4.1.1 Early stability checks

One way to avoid duplicating the stability checks is to check stability on the forks of parallel compositions and only on the exits of atomic blocks. The test at the fork of parallel compositions ensures that the first assertion in a proof outline is stable. And since shared assertions are only modified by atomic commands, the checks at the exits of atomic commands ensures that all the other assertions in the proof outline are stable.

Figure 4.1 contains the relevant proof rules. The difference from the standard proof rules is evident in the first two rules: E-ATOMR drops the stability check on $\boxed{P}$, whereas E-PAR now checks that p_1 and p_2 are stable. The other proof rules are more standard. Unlike SKIP, E-SKIP does not have the awkward stability check on p. The rules E-SEQ, E-WEAKEN, E-LOOP, and E-CHOICE are the same as in the standard semantics. Finally, we can weaken the frame rule (E-FRAME) so that it does not check stability if the command contains no atomic blocks. This stronger frame rule is not admissible in the standard semantics.

Theorem 35 (Soundness)**.** *If* $\vdash C \textbf{ sat}_{\mathsf{ES}} (p, R, G, q)$ *then* $\vDash C \textbf{ sat}_{\mathsf{ES}} (p, R, G, q)$.

Proof. By induction on the proof rules.

$$\frac{\vdash C_1\ \mathbf{sat}_{\mathsf{ES}}\ (p_1, R \cup G_2, G_1, q_1) \qquad p_1 \text{ stable under } R \cup G_2 \quad \vdash C_2\ \mathbf{sat}_{\mathsf{ES}}\ (p_2, R \cup G_1, G_2, q_2) \qquad p_2 \text{ stable under } R \cup G_1}{\vdash C_1 \| C_2\ \mathbf{sat}_{\mathsf{ES}}\ (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2)} \text{ (E-Par)}$$

$$\frac{q \text{ stable under } R \quad \vdash C\ \mathbf{sat}_{\mathsf{ES}}\ (p, \emptyset, G, q)}{\vdash \langle C \rangle\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q)} \text{ (E-AtomR)} \qquad \frac{P, Q \text{ precise} \quad (P \rightsquigarrow Q) \subseteq G \quad \vdash C\ \mathbf{sat}_{\mathsf{ES}}\ (P * P', \emptyset, \emptyset, Q * Q')}{\vdash \langle C \rangle\ \mathbf{sat}_{\mathsf{ES}}\ (\boxed{P*F} * P', \emptyset, G, \boxed{Q*F} * Q')} \text{ (Atom)}$$

$$\frac{}{\vdash \mathbf{skip}\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, p)} \text{ (E-Skip)} \qquad \frac{\vdash C\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, p)}{\vdash C^*\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, p)} \text{ (E-Loop)}$$

$$\frac{\vdash C_1\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, r) \quad \vdash C_2\ \mathbf{sat}_{\mathsf{ES}}\ (r, R, G, q)}{\vdash (C_1; C_2)\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q)} \text{ (E-Seq)} \qquad \frac{\vdash C_1\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q) \quad \vdash C_2\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q)}{\vdash (C_1 + C_2)\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q)} \text{ (E-Choice)}$$

$$\frac{R \subseteq R' \quad p \Rightarrow p' \quad G' \subseteq G \quad q' \Rightarrow q \quad \vdash C\ \mathbf{sat}_{\mathsf{ES}}\ (p', R', G', q')}{\vdash C\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q)} \text{ (E-Weaken)} \qquad \frac{\vdash C\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q) \quad \begin{pmatrix} r \text{ stable under } (R \cup G) \\ \vee\ C \text{ contains no } \langle \_ \rangle \end{pmatrix}}{\vdash C\ \mathbf{sat}_{\mathsf{ES}}\ (p * r, R, G, q * r)} \text{ (E-Frame)}$$

Figure 4.1: Early stability proof rules

- (E-AtomR): The proof proceeds as for the usual semantics, but $\mathsf{wssa}_{R'}(\_)$ removes needing to check that the precondition is stable.

- (Atom) and (Prim): The proof is the same as for the usual semantics.

- (E-Skip): Trivial, as $\mathsf{wssa}(p)$ is preserved by environment interference.

- (E-Par): Follows from Lemma 23.

- (E-Seq): Follows from Lemmas 24 and 25.

- (E-Weaken): Follows from Lemmas 20 and 21, because the early stability semantics quantifies over all $R' \subseteq R$.

- (E-Loop): The proof is the same as for the usual semantics.

- (E-Frame): As C and $C \| \mathbf{skip}$ are equivalent with respect to the operational semantics (cf. Lemma 26), we can derive the frame rule from the parallel rule. We consider two cases separately. If r stable under $R \cup G$, abbreviated to (†) below, then

$$\frac{\vdash C_1\ \mathbf{sat}_{\mathsf{LS}}\ (p_1, R \cup G_2, G_1, q_1) \qquad q_1 \text{ stable under } R \cup G_2 \quad \vdash C_2\ \mathbf{sat}_{\mathsf{LS}}\ (p_2, R \cup G_1, G_2, q_2) \qquad q_2 \text{ stable under } R \cup G_1}{\vdash C_1 \| C_2\ \mathbf{sat}_{\mathsf{LS}}\ (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2)}\ (\text{L-Par})$$

$$\frac{p \text{ stable under } R \quad \vdash \langle C \rangle\ \mathbf{sat}_{\mathsf{LS}}\ (p, \emptyset, G, q)}{\vdash \langle C \rangle\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q)}\ (\text{L-AtomR}) \qquad \frac{P, Q \text{ precise} \quad (P \rightsquigarrow Q) \subseteq G \quad \vdash C\ \mathbf{sat}_{\mathsf{LS}}\ (P * P', \emptyset, \emptyset, Q * Q')}{\vdash \langle C \rangle\ \mathbf{sat}_{\mathsf{LS}}\ (\boxed{P*F} * P', \emptyset, G, \boxed{Q*F} * Q')}\ (\text{Atom})$$

$$\frac{}{\vdash \mathbf{skip}\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, p)}\ (\text{L-Skip}) \qquad \frac{\vdash C\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, p)}{\vdash C^*\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, p)}\ (\text{L-Loop})$$

$$\frac{\vdash C_1\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, r) \quad \vdash C_2\ \mathbf{sat}_{\mathsf{LS}}\ (r, R, G, q)}{\vdash (C_1; C_2)\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q)}\ (\text{L-Seq}) \qquad \frac{\vdash C_1\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q) \quad \vdash C_2\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q)}{\vdash (C_1 + C_2)\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q)}\ (\text{L-Choice})$$

$$\frac{R \subseteq R' \quad p \Rightarrow p' \quad G' \subseteq G \quad q' \Rightarrow q \quad \vdash C\ \mathbf{sat}_{\mathsf{LS}}\ (p', R', G', q')}{\vdash C\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q)}\ (\text{L-Weaken}) \qquad \frac{\vdash C\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q) \quad \begin{pmatrix} r \text{ stable under } (R \cup G) \\ \vee\ C \text{ contains no } \langle \_ \rangle \end{pmatrix}}{\vdash C\ \mathbf{sat}_{\mathsf{LS}}\ (p * r, R, G, q * r)}\ (\text{L-Frame})$$

Figure 4.2: Late stability proof rules

$$\dfrac{\dfrac{\dfrac{\models C\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q)}{\models C\ \mathbf{sat}_{\mathsf{ES}}\ (\mathsf{wssa}_R(p), R, G, q)} \qquad \dfrac{}{\models \mathbf{skip}\ \mathbf{sat}_{\mathsf{ES}}\ (r, R \cup G, \emptyset, r)}\ (\dagger)}{\models (C \| \mathbf{skip})\ \mathbf{sat}_{\mathsf{ES}}\ (\mathsf{wssa}_R(p) * r, R, G, q * r)}}{\models (C \| \mathbf{skip})\ \mathbf{sat}_{\mathsf{ES}}\ (p * r, R, G, q * r)}}{\models C\ \mathbf{sat}_{\mathsf{ES}}\ (p * r, R, G, q * r)}$$

If C does not contain critical regions, we can always take $G = \emptyset$. In this case,

$$\dfrac{\dfrac{\dfrac{\dfrac{\models C\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, \emptyset, q)}{\models C\ \mathbf{sat}_{\mathsf{ES}}\ (\mathsf{wssa}_R(p), R, \emptyset, q)} \qquad \dfrac{\dfrac{}{\models \mathbf{skip}\ \mathbf{sat}_{\mathsf{ES}}\ (r, R, \emptyset, r)}}{\models \mathbf{skip}\ \mathbf{sat}_{\mathsf{ES}}\ (\mathsf{wssa}_R(r), R, \emptyset, r)}}{\models (C \| \mathbf{skip})\ \mathbf{sat}_{\mathsf{ES}}\ (\mathsf{wssa}_R(p) * \mathsf{wssa}_R(r), R, \emptyset, q * r)}}{\models (C \| \mathbf{skip})\ \mathbf{sat}_{\mathsf{ES}}\ (p * r, R, \emptyset, q * r)}}{\models C\ \mathbf{sat}_{\mathsf{ES}}\ (p * r, R, \emptyset, q * r)}}{\models C\ \mathbf{sat}_{\mathsf{ES}}\ (p * r, R, G, q * r)}$$

□

4.1.2 Late stability checks

Figure 4.2 contains the 'late stability' proof rules. In this system, the atomic rule (L-AtomR) requires that the precondition is stable, but avoids checking that the postcondition is stable. The stability of the postcondition arises as a proof obligation of the

$$\frac{\vdash C_1\ \mathbf{sat}_{\mathsf{MS}}\ (p_1, R \cup G_2, G_1, q_1) \quad p_1, q_1 \text{ stable under } R \cup G_2 \qquad \vdash C_2\ \mathbf{sat}_{\mathsf{MS}}\ (p_2, R \cup G_1, G_2, q_2) \quad p_2, q_2 \text{ stable under } R \cup G_1}{\vdash (C_1 \| C_2)\ \mathbf{sat}_{\mathsf{MS}}\ (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2)} \quad \text{(M-Par)}$$

$$\frac{P, Q \text{ precise} \qquad \vdash C\ \mathbf{sat}_{\mathsf{MS}}\ (P * P', \emptyset, \emptyset, Q * Q') \qquad (P \rightsquigarrow Q) \subseteq G}{\vdash \langle C \rangle\ \mathbf{sat}_{\mathsf{MS}}\ (\boxed{P*F} * P', R, G, \boxed{Q*F} * Q')} \quad \text{(M-Atom)}$$

$$\vdash \mathbf{skip}\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, p) \quad \text{(M-Skip)} \qquad \frac{\vdash C_1\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q) \qquad \vdash C_2\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q)}{\vdash (C_1 + C_2)\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q)} \quad \text{(M-Choice)}$$

$$\frac{r \text{ stable under } R \qquad \vdash C_1\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, r) \qquad \vdash C_2\ \mathbf{sat}_{\mathsf{MS}}\ (r, R, G, q)}{\vdash (C_1; C_2)\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q)} \quad \text{(M-Seq)} \qquad \frac{R \subseteq R' \quad p \Rightarrow p' \qquad G' \subseteq G \quad q' \Rightarrow q \qquad \vdash C\ \mathbf{sat}_{\mathsf{MS}}\ (p', R', G', q')}{\vdash C\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q)} \quad \text{(M-Weaken)}$$

$$\frac{p \text{ stable under } R \qquad \vdash C\ \mathbf{sat}\ (p, R, G, p)}{\vdash C^*\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, p)} \quad \text{(M-Loop)} \qquad \frac{\vdash C\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q) \qquad \begin{pmatrix} r \text{ stable under } (R \cup G) \\ \vee\ C \text{ contains no } \langle \_ \rangle \end{pmatrix}}{\vdash C\ \mathbf{sat}_{\mathsf{MS}}\ (p * r, R, G, q * r)} \quad \text{(M-Frame)}$$

Figure 4.3: Mid stability proof rules

surrounding program. For instance, if there is a following atomic block, then from the sequence rule, the postcondition must imply the (stable) precondition of that atomic block. Otherwise, if it is the final postcondition of a thread, then L-Par checks that it is stable.

The other proof rules (L-Skip, L-Seq, L-Loop, L-Choice, L-Weaken, and L-Frame) are the same as those for early stability.

Theorem 36 (Soundness)**.** *If* $\vdash C\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q)$*, then* $\vDash C\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q)$*.*

The proof is analogous to the proof of Theorem 35.

4.1.3 Mid stability checks

Figure 4.3 contains the proof rules with the mid stability checks. Unlike all the other proof systems seen so far, the mid stability rule for atomic commands (M-Atom) does not require any assertions to be stable. Instead, we check stability at sequential composition and the forks and joins of parallel composition. Since loops involve sequencing of commands, we also need to check that the loop invariant r is stable under interference.

Because mid stability delays checking stability to the sequencing operator, it enables atomic operations to have much simpler specifications. For example, consider the `lock` and `unlock` commands from §3.5. Using the mid stability rules, we can derive the following

much simpler specifications:

$$\vdash \texttt{lock}(x)\ \mathbf{sat}_{\mathsf{MS}}\ (\boxed{N_{\_}(\mathrm{x}, v, n) * F}, \emptyset, \{(\mathsf{Lock})\}, \boxed{L_{\mathrm{tid}}(\mathrm{x}, v, n) * F})$$
$$\vdash \texttt{unlock}(x)\ \mathbf{sat}_{\mathsf{MS}}\ (\boxed{L_{\mathrm{tid}}(x, v, n) * F}, \emptyset, \{(\mathsf{Unlock})\}, \boxed{N_{\_}(x, v, n) * F})$$

These specifications are not as concise as possible: they still have the explicit shared frame F. This suggests that one could perhaps extend RGSep with another operator $**$ such that $\boxed{P} ** \boxed{Q} \iff \boxed{P * Q}$, and have an additional framing rule with the $**$ operator.

Lemma 37. *If p stable under R and q stable under R, then*
(i) $\vDash C\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q) \iff \vDash C\ \mathbf{sat}\ (p, R, G, q)$, *and*
(ii) $\vdash C\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q) \iff \vdash C\ \mathbf{sat}\ (p, R, G, q)$.

Proof. (i) Trivial, as for all $R' \subseteq R$, $\mathsf{wssa}_{R'}(p) = p$ and $\mathsf{sswa}_{R'}(q) = q$.
(ii) The $\Longrightarrow$ direction is a straightforward induction on the mid stability proof rules; all cases are trivial. To prove the $\Longleftarrow$ direction, do an induction on the standard proof rules. Case Seq: From Lemma 27, $\exists r'.\ (r' \Rightarrow r) \wedge (r \text{ stable under } R) \wedge \vdash C_1\ \mathbf{sat}\ (p, R, G, r')$. From the Weaken rule, we also have $\vdash C_2\ \mathbf{sat}\ (r', R, G, q)$. From the induction hypothesis, $\vdash C_1\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, r')$ and $\vdash C_2\ \mathbf{sat}_{\mathsf{MS}}\ (r', R, G, q)$. Therefore, by M-Seq, $\vdash (C_1; C_2)\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q)$. The other cases are trivial. □

Theorem 38 (Soundness)**.** *If $\vdash C\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q)$, then $\vDash C\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q)$, i.e. for all $R' \subseteq R$, $\vDash C\ \mathbf{sat}\ (\mathsf{wssa}_{R'}(p), R', G, \mathsf{sswa}_{R'}(q))$.*

Proof. By induction on the proof rules. Prove each rule separately.

- M-Skip: Trivial, as $\mathsf{wssa}_{R'}(p)$ is preserved by environment interference.
- Prim: Follows directly from Lemma 37 and the standard semantics, as P and Q are stable under R.
- M-Weaken: Trivial.
- M-Atom:

  For notational convenience in the following proof, let $\boxed{\exists \overline{y}.\ P_1} = \mathsf{wssa}_{R'}(\boxed{\exists \overline{y}.\ P * F})$ and $\boxed{Q_1} = \mathsf{sswa}_{R'}(\boxed{Q * F})$. (Since $\boxed{\_}$ describes only the shared state, $\mathsf{wssa}_{R'}(\boxed{\_})$ can be written as a top-level boxed assertion.)

1. $(P' \rightsquigarrow Q') \subseteq G$ — assumption
2. $\models C\ \mathbf{sat}_{\mathsf{MS}}\ (P' * P'', \emptyset, \emptyset, Q' * Q'')$ — assumption
3. $\models C\ \mathbf{sat}\ (P' * P'', \emptyset, \emptyset, Q' * Q'')$ — from 3, Lemma 37
4. $(P_1 \Rightarrow P' * F) \wedge (Q \Rightarrow Q_1 * F)$ — Def. 30 and 31
5. $\boxed{\exists \overline{y}.\ P_1}$ stable under R — Def. 30
6. $\boxed{Q_1}$ stable under R — Def. 31
7. $\models \langle C \rangle\ \mathbf{sat}\ (\boxed{\exists \overline{y}.\ P_1} * P'', R', G, \exists \overline{y}.\ \boxed{Q_1} * Q'')$ — from 1, 4, 5, 6, 7, Atomic
8. $\models \langle C \rangle\ \mathbf{sat}\ (\mathsf{wssa}_{R'}(\boxed{\exists \overline{y}.\ P' * F} * P''), R', G, \mathsf{sswa}_{R'}(\exists \overline{y}.\ \boxed{Q' * F} * Q''))$ — from 7, Lemmas 33, 34

- M-Par:
  1. $\models C_1\ \mathbf{sat}_{\mathsf{MS}}\ (p_1, R \cup G_2, G_1, q_1)$ — assumption
  2. $\models C_2\ \mathbf{sat}_{\mathsf{MS}}\ (p_2, R \cup G_1, G_2, q_2)$ — assumption
  3. p_1, q_1 stable under $R \cup G_2$ — assumption
  4. p_2, q_2 stable under $R \cup G_1$ — assumption
  5. $\models C_1\ \mathbf{sat}\ (p_1, R \cup G_2, G_1, q_1)$ — from 1, 3, Lem 37
  6. $\models C_2\ \mathbf{sat}\ (p_2, R \cup G_1, G_2, q_2)$ — from 2, 4, Lem 37
  7. $\models (C_1 \| C_2)\ \mathbf{sat}\ (p_1 * p_2, R, G, q_1 * q_2)$ — from 5, 6, Par, Thm 28
  8. $\models (C_1 \| C_2)\ \mathbf{sat}\ (\mathsf{wssa}_{R'}(p_1 * p_2), R', G, \mathsf{sswa}_{R'}(q_1 * q_2))$ — from 7, Weaken, Thm 28

  Case C_1 contains no $\langle \_ \rangle$.
  1. $\models C_1\ \mathbf{sat}_{\mathsf{MS}}\ (p_1, R \cup G_2, G_1, q_1)$ — assumption
  2. $\models C_2\ \mathbf{sat}_{\mathsf{MS}}\ (p_2, R \cup G_1, G_2, q_2)$ — assumption
  3. p_1, q_1 stable under $R \cup G_2$ — assumption
  4. C_1 contains no $\langle \_ \rangle$ — assumption
  5. $\models C_1\ \mathbf{sat}_{\mathsf{MS}}\ (p_1, R \cup G_2, \emptyset, q_1)$ — from 1, 4
  6. $\models C_2\ \mathbf{sat}_{\mathsf{MS}}\ (p_2, R, G_2, q_2)$ — from 2, M-Weaken
  7. $\models C_1\ \mathbf{sat}\ (p_1, R' \cup G_2, \emptyset, q_1)$ — from 3, 5
  8. $\models C_2\ \mathbf{sat}\ (\mathsf{wssa}_{R'}(p_2), R', G_2, \mathsf{sswa}_R(q_2))$ — from 6
  9. $\models (C_1 \| C_2)\ \mathbf{sat}\ (p_1 * \mathsf{wssa}_{R'}(p_2), R', G, q_1 * \mathsf{sswa}_{R'}(q_2))$ — from 5, 6, Par, Thm 28
  10. $\models (C_1 \| C_2)\ \mathbf{sat}\ (\mathsf{wssa}_{R'}(p_1 * p_2), R', G, \mathsf{sswa}_{R'}(q_1 * q_2))$ — from 9, Lem 33, Lem 34

  Case C_2 contains no $\langle \_ \rangle$. Symmetric.

- M-Seq:
  1. $\models C_1\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, r)$ — assumption
  2. $\models C_2\ \mathbf{sat}_{\mathsf{MS}}\ (r, R, G, q)$ — assumption
  3. r stable under R — assumption
  4. $\models C_1\ \mathbf{sat}\ (\mathsf{wssa}_{R'}(p), R', G, r)$ — from 1, 3
  5. $\models C_2\ \mathbf{sat}\ (r, R', G, \mathsf{sswa}_{R'}(q))$ — from 2, 3
  6. $\models (C_1; C_2)\ \mathbf{sat}\ (\mathsf{wssa}_{R'}(p), R', G, \mathsf{sswa}_{R'}(q))$ — from 4, 5, Seq, Thm 28

Rules	Pessimistic	Early	Late	Mid
Skip	p	–	–	–
Seq	–	–	–	q
Choice	–	–	–	–
Loop	–	–	–	p
Par	–	p_1, p_2	q_1, q_2	p_1, p_2, q_1, q_2
Atomic	p, q	q	p	–

Table 4.1: Stability checks at a glance

- M-Loop:
  1. $\vdash C$ **sat**$_{\mathsf{MS}}$ (p, R, G, p) — assumption
  2. p stable under R — assumption
  3. $\vdash C$ **sat** (p, R', G, p) — from 1, 2
  4. $\models C^*$ **sat** (p, R', G, p) — from 3, Loop, Thm 28
  5. $\models C^*$ **sat** $(\mathsf{wssa}_{R'}(p), R', G, \mathsf{sswa}_{R'}(p))$ — from 2, 4
- M-Frame: Do a case split on the disjunction in the premise.

  Case r stable under $R \cup G$.
  1. $\models C$ **sat**$_{\mathsf{MS}}$ (p, R, G, p) — assumption
  2. r stable under $(R \cup G)$ — assumption
  3. $\models C$ **sat** $(\mathsf{wssa}_{R'}(p), R', G, \mathsf{sswa}_{R'}(q))$ — from 1
  4. $\models C$ **sat** $(\mathsf{wssa}_{R'}(p) * r, R', G, \mathsf{sswa}_{R'}(q) * r)$ — from 3, Frame, Thm 28
  5. r stable under R' — from 2, as $R' \subseteq R$
  6. $\mathsf{wssa}_{R'}(p) * r \iff \mathsf{wssa}_{R'}(p * r)$ — from 5, Lemma 33
  7. $\mathsf{sswa}_{R'}(q * r) \iff \mathsf{sswa}_{R'}(q) * r$ — from 5, Lemma 34
  8. $\models C$ **sat** $(\mathsf{wssa}_{R'}(p * r), R', G, \mathsf{sswa}_{R'}(q * r))$ — from 4, 6, 7

  Case C contains no $\langle \_ \rangle$.
  1. $\models C$ **sat**$_{\mathsf{MS}}$ (p, R, G, q) — assumption
  2. C contains no $\langle \_ \rangle$ — assumption
  3. $\models C$ **sat**$_{\mathsf{MS}}$ $(p, R, \emptyset, q)$ — from 1, 2
  4. $\models C$ **sat** $(\mathsf{wssa}_{R'}(p), R', \emptyset, \mathsf{sswa}_{R'}(q))$ — from 3
  5. $\models C$ **sat** $(\mathsf{wssa}_{R'}(p) * \mathsf{wssa}_{R'}(r), R', G, \mathsf{sswa}_{R'}(q) * \mathsf{wssa}_{R'}(r))$ — from 4, Frame
  6. $\models C$ **sat** $(\mathsf{wssa}_{R'}(p) * \mathsf{wssa}_{R'}(r), R', G, \mathsf{sswa}_{R'}(q) * \mathsf{sswa}_{R'}(r))$ — from 5, Weaken
  7. $\models C$ **sat** $(\mathsf{wssa}_{R'}(p * r), R', G, \mathsf{sswa}_{R'}(q * r))$ — from 6, Lem 33, 34

□

4.1.4 The stability lattice

Table 4.1 summarises the stability checks done by each of the four proof systems we have met. Evidently the mid-stability rules appear more complex than the other three proof

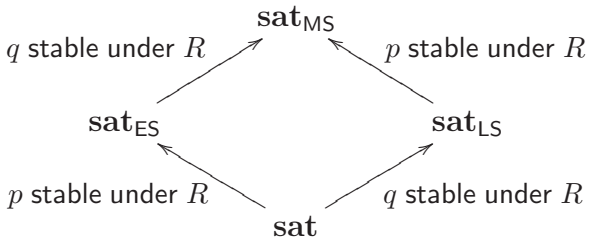

Figure 4.4: Stability lattice

systems, but this complexity makes them more powerful.

With the mid stability rules, we can prove strictly more specifications than with either of the other systems. To prove this assertion, first prove the following theorem.

Theorem 39.

- $\vdash C\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q) \implies \vdash C\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q)$,
- $\vdash C\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q) \implies \vdash C\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q)$,
- $\vdash C\ \mathbf{sat}\ (p, R, G, q) \implies \vdash C\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q)$, *and*
- $\vdash C\ \mathbf{sat}\ (p, R, G, q) \implies \vdash C\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q)$.

This states that (i) mid stability is at least as powerful as early stability and late stability and that (ii) all the variant systems enable us to prove at least as many specifications as the standard system. Proving this theorem consists of straightforward inductions over the proof rules.

In addition, we can prove that $\vdash \langle \mathtt{x} := \mathtt{x} + 1 \rangle\ \mathbf{sat}_{\mathsf{MS}}\ (\boxed{\mathtt{x} = X}, R, R, \boxed{\mathtt{x} = X + 1})$ where $R = \{x \mapsto N \rightsquigarrow x \mapsto N{+}1\}$. This specification, however, does not hold in any of the other systems because the precondition and the postcondition are unstable. Hence, the mid stability rules one can prove strictly more specifications than either of the other systems. Similarly, we can specify an atomic command (i) with a stable precondition and an unstable postcondition, or (ii) with an unstable precondition and a stable postcondition. Thus, we can show that early and late stability are incomparable in strength and that the standard system is the weakest of all.

These results are presented in Figure 4.4. An arrow from A to B in the diagram says that the proof system A can prove fewer specifications than B.

In addition, if the precondition is stable, then $\mathbf{sat}$ and $\mathbf{sat}_{\mathsf{ES}}$ are equivalent, and so are $\mathbf{sat}_{\mathsf{LS}}$ and $\mathbf{sat}_{\mathsf{MS}}$. Dually, if the postcondition is stable, then $\mathbf{sat}$ and $\mathbf{sat}_{\mathsf{LS}}$ are equivalent, and so are $\mathbf{sat}_{\mathsf{ES}}$ and $\mathbf{sat}_{\mathsf{MS}}$. If both the precondition and the postcondition of a specification are stable, then provability is the same for all four systems.

Theorem 40.
(a) *If* p stable under R, *then*

- $\vdash C\ \mathbf{sat}\ (p, R, G, q) \iff \vdash C\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q)$, *and*

- $\vdash C\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q) \iff \vdash C\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q)$.

(*b*) *If* q stable under R, *then*

- $\vdash C\ \mathbf{sat}\ (p, R, G, q) \iff \vdash C\ \mathbf{sat}_{\mathsf{LS}}\ (p, R, G, q)$, *and*
- $\vdash C\ \mathbf{sat}_{\mathsf{ES}}\ (p, R, G, q) \iff \vdash C\ \mathbf{sat}_{\mathsf{MS}}\ (p, R, G, q)$.

Proof outline. The $\Longrightarrow$ directions follow from Theorem 39. For $\Longleftarrow$ directions, take each case separately and do an induction on the proof rules of the right hand side. Each case can be transformed to an application of the respective rule of the left hand side using the additional assumption p stable under R or q stable under R. In some cases, we also need to use the relevant consequence rule (Weaken, L-Weaken, etc.). □

4.2 Satisfying the guarantee

So far, each time we want to reason about an atomic block, we must use the Atom rule. We have to invent an $P' \rightsquigarrow Q'$ that represents what the atomic block does and satisfies the guarantee. (In the semantics, the atomic block is actually annotated with this action.) When the body of the atomic block is large, guessing this action might be difficult.

Since, however, the program $\langle C_1\rangle; \langle C_2\rangle$ has strictly more behaviours than $\langle C_1; C_2\rangle$, if we prove that $\langle C_1\rangle; \langle C_2\rangle$ satisfies the guarantee, then $\langle C_1; C_2\rangle$ will also satisfy the guarantee. The hope is that proving the former might be easier than proving the latter by applying the Atom rule directly.

The following rules formalise this kind of reasoning.

$$\frac{}{\vdash \langle \mathbf{skip}\rangle\ \mathbf{sat}\ (p, \emptyset, G, p)}\ \text{(A-Skip)} \qquad \frac{\vdash \langle C\rangle\ \mathbf{sat}\ (p, \emptyset, G, p)}{\vdash \langle C^*\rangle\ \mathbf{sat}_{\mathsf{MS}}\ (p, \emptyset, G, p)}\ \text{(A-Loop)}$$

$$\frac{\begin{array}{c}\vdash \langle C_1\rangle\ \mathbf{sat}\ (p, \emptyset, G, q)\\ \vdash \langle C_2\rangle\ \mathbf{sat}\ (q, \emptyset, G, r)\end{array}}{\vdash \langle C_1; C_2\rangle\ \mathbf{sat}\ (p, \emptyset, G, r)}\ \text{(A-Seq)} \qquad \frac{\begin{array}{c}\vdash \langle C_1\rangle\ \mathbf{sat}\ (p, \emptyset, G, q)\\ \vdash \langle C_2\rangle\ \mathbf{sat}\ (p, \emptyset, G, q)\end{array}}{\vdash \langle C_1 + C_2\rangle\ \mathbf{sat}_{\mathsf{MS}}\ (p, \emptyset, G, q)}\ \text{(A-Choice)}$$

The rules A-Seq, A-Loop and A-Choice are also valid if we replace $\emptyset$ by an arbitrary rely R (A-Skip would also need to test that p is stable). However, applying these rules with any R other than $\emptyset$ would be unwise because it adds useless stability checks in the middle of atomic blocks. Instead, one first applies AtomR to check stability, and then applies the "A-" rules with $R = \emptyset$ and Atom to complete the proof.

For example, consider the following program, a slight generalisation of CAS (compare and swap):

$$\mathtt{CAD}(\mathtt{x}, \mathtt{o}, C) \stackrel{\text{def}}{=} \langle \mathtt{t} := [\mathtt{x}];\ \mathtt{if}\ (\mathtt{t} = \mathtt{o})\ \{C\}\rangle$$

If $[\mathtt{x}] = \mathtt{o}$, it executes C atomically; otherwise it does nothing. The invented name CAD stands for *compare and do*. We can derive the following two rules.

$$\frac{\vdash \langle C\rangle \ \mathbf{sat}\ (p * \boxed{\mathtt{x}\mapsto\mathtt{o} * \mathit{true}}, \emptyset, G, q) \qquad \mathtt{t} \notin \mathsf{vars}(p, C)}{\vdash \mathtt{CAD}(\mathtt{x}, \mathtt{o}, C)\ \mathbf{sat}\ (p * \boxed{\mathtt{x}\mapsto\_ * \mathit{true}}, \emptyset, G, (p \wedge \mathtt{t} \neq \mathtt{o}) \vee (q \wedge \mathtt{t} = \mathtt{o}))} \quad (\textsc{Cad})$$

$$\frac{\begin{array}{c}\models_{\mathsf{RGSep}} p \Rightarrow \boxed{\mathtt{x}\mapsto\_ * \mathit{true}} * \mathit{true} \qquad p \text{ stable under } R \qquad q \text{ stable under } R \\ \models_{\mathsf{RGSep}} p * \boxed{\mathtt{x}\mapsto\mathtt{o} * \mathit{true}} \Rightarrow p' \qquad \vdash \langle C\rangle \ \mathbf{sat}\ (p', \emptyset, G, q) \qquad \mathtt{t} \notin \mathsf{vars}(p, C)\end{array}}{\vdash \mathtt{CAD}(\mathtt{x}, \mathtt{o}, C)\ \mathbf{sat}\ (p, R, G, (p \wedge \mathtt{t} \neq \mathtt{o}) \vee (q \wedge \mathtt{t} = \mathtt{o}))} \quad (\textsc{Cad2})$$

The second rule is an immediate corollary of Cad, AtomR, and Conseq. These derived rules simplify the verification of algorithms involving CAS (for example, see §5.3.1).

Derivation of Cad. If we remove the syntactic sugar, $\mathtt{CAD}(\mathtt{x}, \mathtt{o}, C)$ becomes

$$\langle \mathtt{t} := [\mathtt{x}]; ((\mathsf{assume}(\mathtt{t} = \mathtt{o}); C) + \mathsf{assume}(\mathtt{t} \neq \mathtt{o}))\rangle.$$

First, prove that

$$\dfrac{\dfrac{\mathtt{t} \notin \mathsf{vars}(p, C) \qquad \vdash \langle C\rangle \ \mathbf{sat}\ (p * \boxed{\mathtt{x}\mapsto\mathtt{o} * \mathit{true}}, \emptyset, G, q)}{\vdash \langle C\rangle \ \mathbf{sat}\ (p * \boxed{\mathtt{x}\mapsto\mathtt{o} * \mathit{true}} * (\mathit{emp} \wedge \mathtt{t} = \mathtt{o}), \emptyset, G, q * (\mathit{emp} \wedge \mathtt{t} = \mathtt{o}))}\ \textsc{Frame}}{\vdash \langle C\rangle \ \mathbf{sat}\ (p * \boxed{\mathtt{x}\mapsto\mathtt{t} * \mathit{true}} * (\mathit{emp} \wedge \mathtt{t} = \mathtt{o}), \emptyset, G, (p \wedge \mathtt{t} \neq \mathtt{o}) \vee (q \wedge \mathtt{t} = \mathtt{o}))}\ \textsc{Conseq}$$

and that

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{assume}(\mathtt{t} = \mathtt{o})\ \mathbf{sat}\ (\mathit{emp}, \emptyset, G, \mathit{emp} \wedge \mathtt{t} = \mathtt{o})}\ \textsc{Prim}}{\vdash \langle \mathsf{assume}(\mathtt{t} = \mathtt{o})\rangle\ \mathbf{sat}\ (\mathit{emp}, \emptyset, G, \mathit{emp} \wedge \mathtt{t} = \mathtt{o})}\ \textsc{Atom}}{\vdash \langle \mathsf{assume}(\mathtt{t}{=}\mathtt{o})\rangle\ \mathbf{sat}\ (p * \boxed{\mathtt{x}\mapsto\mathtt{t} * \mathit{true}}, \emptyset, G, p * \boxed{\mathtt{x}\mapsto\mathtt{t} * \mathit{true}} * (\mathit{emp} \wedge \mathtt{t}{=}\mathtt{o}))}\ \textsc{Frame}$$

Hence, applying A-Seq, the first branch of the conditional becomes:

$$\vdash \langle \mathsf{assume}(\mathtt{t} = \mathtt{o}); C\rangle\ \mathbf{sat}\ (p * \boxed{\mathtt{x}\mapsto\mathtt{t} * \mathit{true}}, \emptyset, G, (p \wedge \mathtt{t} \neq \mathtt{o}) \vee (q \wedge \mathtt{t} = \mathtt{o}))$$

The second branch of the conditional is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{assume}(\mathtt{t} \neq \mathtt{o})\ \mathbf{sat}\ (\mathit{emp}, \emptyset, G, \mathit{emp} \wedge \mathtt{t} \neq \mathtt{o})}\ \textsc{Prim}}{\vdash \langle \mathsf{assume}(\mathtt{t} \neq \mathtt{o})\rangle\ \mathbf{sat}\ (\mathit{emp}, \emptyset, G, \mathit{emp} \wedge \mathtt{t} \neq \mathtt{o})}\ \textsc{Atom}}{\vdash \langle \mathsf{assume}(\mathtt{t} \neq \mathtt{o})\rangle\ \mathbf{sat}\ (p, \emptyset, G, p * (\mathit{emp} \wedge \mathtt{t} \neq \mathtt{o}))}\ \textsc{Frame}}{\vdash \langle \mathsf{assume}(\mathtt{t} \neq \mathtt{o})\rangle\ \mathbf{sat}\ (p * \boxed{\mathtt{x}\mapsto\mathtt{t} * \mathit{true}}, \emptyset, G, (p \wedge \mathtt{t} \neq \mathtt{o}) \vee (q \wedge \mathtt{t} = \mathtt{o}))}\ \textsc{Conseq}$$

Hence, from rule A-Choice we get that $(\langle \mathsf{assume}(\mathtt{t} = \mathtt{o}); C\rangle + \langle \mathsf{assume}(\mathtt{t} \neq \mathtt{o})\rangle)$ satisfies

$(p * \boxed{\mathtt{x}\mapsto\mathtt{t} * \mathit{true}}, \emptyset, G, (p \wedge \mathtt{t} \neq \mathtt{o}) \vee (q \wedge \mathtt{t} = \mathtt{o}))$. Moreover,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\mathtt{x}\mapsto b \rightsquigarrow \mathtt{x}\mapsto b) \in G}\ \text{G-Exact} \quad \dfrac{}{\vdash \langle \mathtt{t} := [\mathtt{x}] \rangle\ \mathbf{sat}\ (\mathtt{x}\mapsto b, \emptyset, G, \mathtt{x}\mapsto b \wedge b = \mathtt{t})}\ \text{Prim}}{\vdash \langle \mathtt{t} := [\mathtt{x}] \rangle\ \mathbf{sat}\ (\boxed{\mathtt{x}\mapsto b * \mathit{true}}, \emptyset, G, \boxed{\mathtt{x}\mapsto b * \mathit{true}} * (b = \mathtt{t} \wedge \mathit{emp}))}\ \text{Atom}}{\vdash \langle \mathtt{t} := [\mathtt{x}] \rangle\ \mathbf{sat}\ (\boxed{\mathtt{x}\mapsto b * \mathit{true}}, \emptyset, G, \exists b.\ \boxed{\mathtt{x}\mapsto b * \mathit{true}} * (b = \mathtt{t} \wedge \mathit{emp}))}\ \text{Conseq}}{\vdash \langle \mathtt{t} := [\mathtt{x}] \rangle\ \mathbf{sat}\ (\exists b.\ \boxed{\mathtt{x}\mapsto b * \mathit{true}}, \emptyset, G, \exists b.\ \boxed{\mathtt{x}\mapsto b * \mathit{true}} * (b = \mathtt{t} \wedge \mathit{emp}))}\ \text{Ex}}{\vdash \langle \mathtt{t} := [\mathtt{x}] \rangle\ \mathbf{sat}\ (\boxed{\mathtt{x}\mapsto \_ * \mathit{true}}, \emptyset, G, \boxed{\mathtt{x}\mapsto \mathtt{t} * \mathit{true}})}\ \text{Conseq}}{\vdash \langle \mathtt{t} := [\mathtt{x}] \rangle\ \mathbf{sat}\ (p * \boxed{\mathtt{x}\mapsto \_ * \mathit{true}}, \emptyset, G, p * \boxed{\mathtt{x}\mapsto \mathtt{t} * \mathit{true}})}\ \text{Frame}$$

Finally, apply rule A-Seq to derive the required conclusion. □

4.3 Modularity

4.3.1 Procedures

Reasoning about procedures is orthogonal to rely/guarantee reasoning. We shall allow mutually recursive function definitions, but for simplicity assume they have no parameters. This is sufficiently general because, if necessary, parameters can be passed through a stack implemented in the heap.

First, extend the grammar of commands as follows:

$$\begin{array}{lll} C ::= & \ldots & \\ & \mathbf{let}\ \mathit{proc}_1{=}C_1, \ldots \mathit{proc}_n{=}C_n\ \mathbf{in}\ C & \text{Procedure definitions} \\ & \mathit{proc} & \text{Procedure call} \end{array}$$

The syntax declares multiple procedures together to permit mutual recursion.

The operational semantics now uses a context that maps procedure names to their definitions; for example $\eta = \{\mathit{proc}_1 \mapsto C_1, \ldots, \mathit{proc}_n \mapsto C_n\}$. This context is just passed around by the existing rules. Here are the additional rules for procedure definitions and procedure calls:

$$\frac{\eta, \eta_1 \vdash (C, \sigma) \xrightarrow[\mathbf{p}]{R} (C', \sigma)}{\eta \vdash (\mathbf{let}\,\eta_1\,\mathbf{in}\,C, \sigma) \xrightarrow[\mathbf{p}]{R} (\mathbf{let}\,\eta_1\,\mathbf{in}\,C', \sigma)} \qquad \frac{}{\eta \vdash (\mathbf{let}\,\eta_1\,\mathbf{in}\,\mathbf{skip}, \sigma) \xrightarrow[\mathbf{p}]{R} (\mathbf{skip}, \sigma)}$$

$$\frac{\eta, \eta_1 \vdash (C, \sigma) \xrightarrow[\mathbf{p}]{R} \mathit{fault}}{\eta \vdash (\mathbf{let}\,\eta_1\,\mathbf{in}\,C, \sigma) \xrightarrow[\mathbf{p}]{R} \mathit{fault}} \qquad \frac{\eta(\mathit{proc}) = C}{\eta \vdash (\mathit{proc}, \sigma) \xrightarrow[\mathbf{p}]{R} (C, \sigma)} \qquad \frac{\mathit{proc} \notin \mathit{dom}(\eta)}{\eta \vdash (\mathit{proc}, \sigma) \xrightarrow[\mathbf{p}]{R} \mathit{fault}}$$

where η, η_1 returns the union of η and η_1 giving precedence to entries in η_1.

The proof rules are extended with a context Γ of procedure specifications. The rules presented so far just pass Γ around. When calling a procedure, we must check that the

procedure has been defined; otherwise the call might fail. If the procedure has been defined, the procedure call meets any specification against which we have checked its body.

$$\frac{}{\Gamma, \mathit{proc}\ \mathbf{sat}\ (p, R, G, q) \vdash \mathit{proc}\ \mathbf{sat}\ (p, R, G, q)} \quad \text{(Call)}$$

Handling mutual recursive procedure definitions is completely standard. Assuming all the procedures have the required specifications, we prove that each body, C_i, has the right specification. Under the same assumptions, we finally check that the command C has the right specification.

$$\frac{\begin{array}{c}\Gamma, \mathit{proc}_1\ \mathbf{sat}\ (p_1,R_1,G_1,q_1), \ldots \mathit{proc}_n\ \mathbf{sat}\ (p_n,R_n,G_n,q_n) \vdash C_1\ \mathbf{sat}\ (p_1,R_1,G_1,q_1) \\ \vdots \\ \Gamma, \mathit{proc}_1\ \mathbf{sat}\ (p_1,R_1,G_1,q_1), \ldots \mathit{proc}_n\ \mathbf{sat}\ (p_n,R_n,G_n,q_n) \vdash C_n\ \mathbf{sat}\ (p_n,R_n,G_n,q_n) \\ \Gamma, \mathit{proc}_1\ \mathbf{sat}\ (p_1,R_1,G_1,q_1), \ldots \mathit{proc}_n\ \mathbf{sat}\ (p_n,R_n,G_n,q_n) \vdash C\ \mathbf{sat}\ (p,R,G,q)\end{array}}{\Gamma \vdash (\mathbf{let}\ \mathit{proc}_1 = C_1, \ldots \mathit{proc}_n = C_n\ \mathbf{in}\ C)\ \mathbf{sat}\ (p, R, G, q)} \quad \text{(Defn)}$$

Proving the soundness of the extended logic is straightforward. First, extend the definitions in §3.3 with a context η that is just passed around.

Definition 41 (Guarantee)**.** *$\eta \vDash (C, \sigma, R)$ $\mathbf{guarantees}\, G$ if and only if for all n, $\eta \vDash (C, \sigma, R)$ $\mathbf{guarantees}_n\, G$, where $\eta \vDash (C, (l, s, o), R)$ $\mathbf{guarantees}_0\, G$ holds always; and $\eta \vDash (C, (l, s, o), R)$ $\mathbf{guarantees}_{n+1}\, G$ holds if, and only if, if $\eta \vDash (C, (l, s, o)) \xrightarrow[\lambda]{R}$ Config, then there exist C', l', s' and o' such that*

1. *Config $= (C', (l', s', o'))$;*
2. *$\eta \vDash (C', (l', s', o'), R)$ $\mathbf{guarantees}_n\, G$; and*
3. *if $\lambda = \mathbf{p}$, then $(s, s') \in G$.*

Definition 42. *$\eta \models C$ $\mathbf{sat}$ (p, R, G, q) if and only if for all σ, σ', if $\sigma \models p$, then*

1. *$\eta \vDash (C, \sigma, R)$ $\mathbf{guarantees}\, G$; and*
2. *if $\eta \vDash (C, \sigma) \xrightarrow{R}{}^{*} (\mathbf{skip}, \sigma')$, then $\sigma' \models q$.*

Similarly, extend all the lemmas in §3.3 with the context, η, of procedure definitions. The additional effort required to prove the extended lemmas amounts to just passing η around. Finally, define $\Gamma \vDash C$ $\mathbf{sat}$ (p, R, G, q) by quantifying over all contexts η that satisfy Γ.

Definition 43. *Let $\Gamma \vDash C$ $\mathbf{sat}$ (p, R, G, q) if and only if for all procedure contexts η, if $\eta \vDash \eta(\mathit{proc})$ $\mathbf{sat}$ $\Gamma(\mathit{proc})$ for all $\mathit{proc} \in \mathit{dom}(\Gamma)$, then $\eta \vDash C$ $\mathbf{sat}$ (p, R, G, q).*

Hence, we can state and prove soundness of the proof rules for procedures:

Theorem 44 (Soundness)**.** *If* $\Gamma \vdash C$ **sat** (p, R, G, q)*, then* $\Gamma \vDash C$ **sat** (p, R, G, q)*.*

The proof is identical to the proof of Theorem 28, except that we have to pass the procedure context (η) around and that we have to prove the soundness of the two additional rules (Call and Defn).

This extension is orthogonal to when the stability is checked. The same proof rules apply to early, late, and mid stability.

4.3.2 Multiple regions

Concurrent separation logic [58] has multiple resource names, each protecting a different disjoint part of memory. Chapter 3 presented RGSep acting on a single region of shared state. This section extends RGSep to multiple regions of shared state.

Assume a countably infinite set of region names $\mathsf{RegionName}$ and let ϱ range over that set. The assertions are the same as before, except that shared (boxed) assertions are annotated with the name of the region they describe.

$$p, q, r ::= P \mid \boxed{P}_{\varrho} \mid p * q \mid p \Rightarrow q \mid \exists x.\ p$$

As before, states consist of three components: the local state (l), the shared state (s), and the local state of the other threads (o). The difference is that the shared state is no longer an element of the separation algebra M, but rather a mapping from region names to M; in other words, $s : \mathsf{RegionName} \rightharpoonup M$.

Definition 45. $\mathsf{States} \stackrel{\text{def}}{=} \left\{ (l, s, o) \;\middle|\; \begin{array}{l} \{l, o\} \subseteq M \wedge s \in (\mathsf{RegionName} \rightharpoonup M) \\ \wedge\ (l \odot o \odot \bigodot_{\rho \in dom(s))} s(\varrho)) \text{ is defined} \end{array} \right\}$

Two states may be combined provided they agree on their shared states, have disjoint local states, and identical total states.

Assertions simply ignore the third component of the state (the state of the other threads), and are defined in terms of the local state (l), the shared state (s), and an interpretation (i) for the logical variables. The semantics of assertions does not change except for boxes:

$$l, s, i \models_{\mathsf{RGSep}} \boxed{P}_{\varrho} \stackrel{\text{def}}{\iff} (l = u) \wedge (s(\varrho), i \models_{\mathsf{SL}} P)$$

The operational semantics is mostly unaffected. The transition relation changes from $\mathit{Config}_1 \xrightarrow[\lambda]{R} \mathit{Config}_2$ into $\mathit{Config}_1 \xrightarrow[\lambda]{\mathcal{R}} \mathit{Config}_2$, where $\mathcal{R}$ is a mapping from region names to rely conditions (binary relations on M). In the operational semantics in Section 3.2, the only rule that depends on R is the rule for environment transitions. This becomes:

$$\frac{\forall \varrho \in dom(\mathcal{R}).\ \ \mathcal{R}(\varrho)(s(\varrho), s'(\varrho)) \qquad (l, s', o') \in \mathsf{States}}{(C, (l, s, o)) \xrightarrow[\mathbf{e}]{\mathcal{R}} (C, (l, s', o'))}$$

The other semantic rules just pass $\mathcal{R}$ around as before.

Judgements are extended into $\vdash C\ \mathbf{sat}\ (p, \mathcal{R}, \mathcal{G}, q)$, where $\mathcal{R}$ and $\mathcal{G}$ map region names to rely and guarantee conditions respectively.

Here is a simple rule for atomic blocks that access a single shared region ρ:

$$\frac{\begin{array}{c}\vdash C\ \mathbf{sat}\ (P' * P'', \emptyset, \emptyset, Q' * Q'') \\ P' \text{ stable under } \mathcal{R}(\varrho) \qquad Q' \text{ stable under } \mathcal{R}(\varrho) \\ P' \rightsquigarrow Q' \Longrightarrow P \rightsquigarrow Q \qquad (P' \rightsquigarrow Q') \subseteq \mathcal{G}(\varrho)\end{array}}{\vdash \langle C \rangle\ \mathbf{sat}\ (\boxed{P}_{\varrho} * P'', \mathcal{R}, \mathcal{G}, \boxed{Q}_{\varrho} * Q'')} \qquad (\textsc{AtomicI})$$

We can also define a slightly more complex rule that allows the same atomic blocks to access state from multiple shared regions.

$$\frac{\begin{array}{c}\vdash C\ \mathbf{sat}\ (P'_1 * \ldots P'_n * P'', \emptyset, \emptyset, Q'_1 * \ldots Q'_n * Q'') \\ \forall i.\ \begin{pmatrix} P'_i \text{ stable under } \mathcal{R}(\varrho_i) & Q'_i \text{ stable under } \mathcal{R}(\varrho_i) \\ P'_i \rightsquigarrow Q'_i \Longrightarrow P_i \rightsquigarrow Q_i & (P'_i \rightsquigarrow Q'_i) \subseteq \mathcal{G}(\varrho_i) \end{pmatrix}\end{array}}{\vdash \langle C \rangle\ \mathbf{sat}\ (\boxed{P_1}_{\varrho_1} * \ldots * \boxed{P_n}_{\varrho_n} * P'', \mathcal{R}, \mathcal{G}, \boxed{Q_1}_{\varrho_1} * \ldots * \boxed{Q_n}_{\varrho_n} * Q'')} \qquad (\textsc{AtomicII})$$

In both these rules, the boxed assertions of the postcondition must be *precise* assertions.

Proving the soundness of this extension is not too difficult. First, we must extend the definitions and the lemmas of §3.3 to use $\mathcal{R}$ and $\mathcal{G}$ instead of R and G. For example, Definition 18 becomes:

Definition 46. $(C, \sigma, \mathcal{R})$ **guarantees** $\mathcal{G} \iff^{\text{def}} \forall n.\ (C, \sigma, \mathcal{R})$ **guarantees**$_n$ $\mathcal{G}$, *where* $(C, (l, s, o), \mathcal{R})\,\mathbf{guarantees}_0\,\mathcal{G}$ *holds always; and* $(C, (l, s, o), \mathcal{R})\,\mathbf{guarantees}_{n+1}\,\mathcal{G}$ *holds if, and only if, if* $(C, (l, s, o)) \xrightarrow[\lambda]{\mathcal{R}} Config$, *then there exist* C', l', s', o' *such that*

1. $Config = (C', (l', s', o'))$;
2. $(C', (l', s', o'), \mathcal{R})\,\mathbf{guarantees}_n\,\mathcal{G}$; *and*
3. *if* $\lambda = \mathbf{p}$, *then for all* $\varrho \in dom(\mathcal{G})$, $(s(\varrho), s'(\varrho)) \in \mathcal{G}(\varrho)$.

Proving this extension sound is almost identical to the soundness proof for the single shared region. Again, we can delay checking stability and derive early, late, and mid stability proof rules.

4.3.3 Local guards

In the lock-coupling list proof of §3.5, the mutexes recorded the thread identifier of the thread which has acquired the lock. Although the actual implementation may store a single bit, it is customary in rely/guarantee proofs to add some auxiliary state to specify which thread has acquired the lock. Hence, we can assert that only the thread that has acquired a lock can later release it.

This extension achieves the same effect without needing to introduce any auxiliary state. We can instead use separation logic permissions. Locking a node retains half the permission for a node; unlocking a node restores the missing permission.

$$x \mapsto \{.lock{=}0\} \rightsquigarrow x \overset{1/2}{\mapsto} \{.lock{=}1\}$$
$$x \overset{1/2}{\mapsto} \{.lock{=}1\} \rightsquigarrow x \mapsto \{.lock{=}0\}$$

Using these actions, when a thread acquires a lock, it gets a token that (i) enables it to release lock, (ii) ensures that no other thread releases the lock. If the thread forgets to release the lock, this will show up as a memory leak: any provable precise post-condition will contain the token. The thread holding the token can also pass it to another thread via some communication mechanism. Hence, it can delegate the knowledge that the lock is acquired and the responsibility that it is later released.

In this example, the local state of a thread plays quite a subtle role in controlling interference. It acts as a token, a permission to perform a certain action, and as a guard, a prohibition that the environment does some action.

In the operational semantics, an environment transition (see Figure 3.2, 6th rule) requires that the resulting state is well formed, that the new shared state is disjoint from the local state. In essence, the existence of the local state *restricts* what the environment can do (e.g. it cannot allocate an existing memory address).

Besides its prohibitive role as to what other threads can do, the local state has a permissive role. Its presence allows a thread to do more actions than it would otherwise be able to do (e.g. in some algorithms, a mutex can be unlocked only by the thread that locked it).

So far, our proof rules have ignored these two roles of the local state that are present in the semantics. We can, however, extend our assertion language with guarded boxed assertions $\boxed{L|S}$, where L is an assertion about the local state, whose presence is used in the stability proof of S. Similarly, guarded actions $G \mid P \rightsquigarrow Q$ use the guard G to stand for the local state that must be owned for the action to occur.

Definition 47. *The assertion* $\boxed{L|S}$ *is stable under interference from* $G \mid P \rightsquigarrow Q$ *if and only if*

$$(P \mathrel{-\!\!\circledast} S) * Q \Rightarrow S \text{ or } \neg(P * G * L) \text{ or } \neg(Q * L) \text{ or } \neg(S * L)$$

The three new cases are when the action cannot execute. The local state that protects the stability of a shared assertion cannot be updated directly because the shared assertion might become unstable. Each time it is updated we need to recheck stability.

4.4 Non-atomic accesses to shared memory

The semantics, as presented so far, restricts accesses to shared memory to be atomic. The only form of commands that can access shared state is the atomic command, $\langle C \rangle$. However, it is well-known that non-atomic accesses can be simulated by a sequence of multiple atomic accesses. For instance, we can encode a non-atomic shared read as two atomic reads, and a check that both reads returned the same value. If the values differ, then we could have read a corrupted value; so we assign a random value to x. Another possibility is to forbid races altogether. In this case, if the two values read differ, reading just fails.

	Race $\Longrightarrow$ corrupted value	Race $\Longrightarrow$ error

$x := [e]$	`local` *temp*;
	$\langle temp := [\mathsf{e}] \rangle$;
	$\langle \mathtt{x} := [\mathsf{e}] \rangle$;
	`if` $(\mathtt{x} \neq temp)$
	  $\mathtt{x} :=$ nondet;
		`local` *temp*;
		$\langle temp := [\mathsf{e}] \rangle$;
		$\langle \mathtt{x} := [\mathsf{e}] \rangle$;
		`if` $(\mathtt{x} \neq temp)$
		  fault;

For both implementations of the non-atomic read, we can derive the following rule:

$$\frac{P = (Q * X \mapsto Y) \qquad \boxed{P} \text{ stable under } R \qquad x \notin fv(P)}{\vdash x := [e] \textbf{ sat } (\boxed{P} \wedge e = X, R, G, \boxed{P} \wedge x = Y)}$$

Given a stable assertion $\boxed{P}$ assigning a constant value Y to the memory cell at address e, we can read this cell non-atomically and get the value Y. The logical variable X is used because e could mention x.

For the non-faulting implementation, we can derive a simple rule for racy reads. If the shared state contains the memory cell at address e, then we can read $[e]$. We cannot, however, assert anything about the value we read because a race condition could have occurred.

$$\frac{P = (Q * X \mapsto \_) \qquad \boxed{P} \text{ stable under } R \qquad x \notin fv(P)}{\vdash x := [e] \textbf{ sat } (\boxed{P} \wedge e = X, R, G, \boxed{P})}$$

Proof. In the following proof outlines, the boxed assertions are stable because of the assumption $\boxed{P}$ stable under R.

$\left\{\boxed{Q * X \mapsto Y} \wedge e = X\right\}$	$\left\{\boxed{Q * X \mapsto \_} \wedge e = X\right\}$
`local` $temp$;	`local` $temp$;
$\left\{\boxed{Q * X \mapsto Y} \wedge e = X\right\}$	$\left\{\boxed{Q * X \mapsto \_} \wedge e = X\right\}$
$\langle temp := [\mathsf{e}]\rangle$;	$\langle temp := [\mathsf{e}]\rangle$;
$\left\{\boxed{Q * X \mapsto Y} \wedge e = X \wedge temp = Y\right\}$	$\left\{\boxed{Q * X \mapsto \_} \wedge e = X\right\}$
$\langle \mathtt{x} := [\mathsf{e}]\rangle$;	$\langle \mathtt{x} := [\mathsf{e}]\rangle$;
$\left\{\boxed{Q * X \mapsto Y} \wedge x = Y \wedge x = temp\right\}$	$\left\{\boxed{Q * X \mapsto \_}\right\}$
`if` $(\mathtt{x} \neq temp)$ $fault$;	`if` $(\mathtt{x} \neq temp)$ $\mathtt{x} :=$ nondet;
$\left\{\boxed{Q * X \mapsto Y} \wedge x = Y \wedge x = temp\right\}$	$\left\{\boxed{Q * X \mapsto \_}\right\}$

□

We can also encode non-atomic operations with side-effects as a sequence of atomic operations that fails if a race condition occurred. For instance, a non-atomic write is just a sequence of arbitrary atomic writes, which will eventually write the entire value provided that there were no conflicting writes.

$$[e] := e' \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathtt{local}\ \mathtt{t}_1, \mathtt{t}_2; \\ \left(\begin{array}{l} \mathtt{t}_1 := \mathsf{nondet}; \\ \langle [\mathsf{e}] := \mathtt{t}_1 \rangle; \\ \langle \mathtt{t}_2 := [\mathsf{e}] \rangle; \\ \mathtt{if}\ (\mathtt{t}_1 \neq \mathtt{t}_2)\ fault; \end{array} \right)^* ; \\ \langle [\mathsf{e}] := \mathsf{e}' \rangle; \end{array} \right\}$$

More generally, given a non-atomic primitive command and a suitable memory model, we can devise an encoding into a sequence of atomic commands that guarantees the same properties. Based on that encoding, we can derive custom proof rules.

Chapter 5

Reasoning about linearisability

Linearisability is the standard correctness condition for fine-grained concurrent data structure implementations. Informally, a procedure is linearisable in a context if and only if it appears to execute atomically in that context. A concurrent data structure is linearisable if all the operations it supports are linearisable.

Linearisability is widely used in practice because atomic code can be specified much more accurately and consicely than arbitrary code. For instance, consider we want to specify the following procedure, which increments the shared variable $\mathtt{x}$ atomically:

$$\mathtt{inc()}\ \{\mathtt{int\ t;\ do}\ \{\mathtt{t} := \mathtt{x};\}\ \mathtt{while}(\neg\mathtt{CAS}(\&\mathtt{x}, \mathtt{t}, \mathtt{t}+1));\}$$

Using the rely/guarantee proof rules, we can prove that `inc()` satisfies the specifications $(\mathtt{x}{=}N, \mathtt{x}{=}\overleftarrow{\mathtt{x}}, G, \mathtt{x}{=}N{+}1)$, $(\mathtt{x}{\leq}N, \mathtt{x}{\leq}\overleftarrow{\mathtt{x}}, G, \mathtt{x}{\leq}N{+}1)$, $(\mathtt{x}{\geq}N, \mathtt{x}{\geq}\overleftarrow{\mathtt{x}}, G, \mathtt{x}{\geq}N{+}1)$, and $(\mathsf{true}, \mathsf{True}, G, \mathsf{true})$, where $G = (\mathtt{x}{\geq}\overleftarrow{\mathtt{x}})$. Each of these four specifications is useful in a different context, but there is no single best specification we can give to `inc()`.

A better way to specify `inc()` is to prove that it is observationally equivalent to $\langle \mathtt{x} := \mathtt{x} + 1; \rangle$. Then, using the mid-stability proof rules, we can derive the specification $(\mathtt{x} = N, \mathsf{True}, G, \mathtt{x} = N{+}1)$, which encompasses the previous four specifications.

This chapter, first, defines linearisability in two ways: the standard one due to Herlihy and Wing [45], and an alternative one that is more suitable for verification. Then, we shall consider how to prove linearisability, illustrated by linearisability proof sketches of a number of fine-grained algorithms. The chapter concludes by discussing related work.

5.1 Definition of linearisability

5.1.1 Historical definition

Herlihy and Wing [45] define linearisability in terms of histories of I/O automata. A history is a finite sequence of events describing the execution of a concurrent program.

There are two types of events: invocations of methods and (matching) responses. For each invocation, there may or may not be a matching response later in the sequence (depending on whether the method call has returned or is still running); but for each response, there is a matching invocation earlier in the history. Hence, all non-empty histories start with an invocation event.

A history H induces an irreflexive partial order $<_H$ on operations,

$$op_1 <_H op_2, \text{ if and only if } res(op_1) \text{ precedes } inv(op_2) \text{ in } H.$$

Two operations that are unrelated by this order are said to be concurrent.

A history H is *sequential* if every invocation in H, except possibly the last, is immediately followed by a matching response. Each response is immediately preceded by a matching invocation. Or equivalently, a history H is sequential, if and only if, $<_H$ is a total order.

For a history H, $complete(H)$ is the maximal subsequence of H consisting only of invocations and matching responses. In other words, $complete(H)$ is obtained from H by removing any unmatched invocations. A history is *legal* if it satisfies all the internal data invariants.

Definition 48. *A history H is linearisable, if it can be extended (by appending zero or more response events) to some history H' such that:*

- *$complete(H')$ is equivalent to some legal sequential history S*
- *$<_H \subseteq <_S$.*

An object is linearisable if all its possible execution histories are linearisable.

5.1.2 Alternative definition

Herlihy and Wing [45] acknowledge that the first definition of linearisability is somewhat 'awkward' for verification. Instead, it is better to define under what conditions a method is linearisable, and to say that an object is linearisable if and only if all its methods are linearisable.

A method call is linearisable if it is observationally equivalent to an atomic execution of the method at some point between its invocation and its return. Phrased differently, a method call is linearisable if there exists an instant between the invocation and the return of the method at which the entire externally visible effect of the method took place. The instant when according to an external observer the entire effect of the method takes place is known as the *linearisation point* of that method.

Of course, this definition only makes sense in a context where the terms "externally visible" and "atomic" are defined. We consider them in the context of a simple module

that *owns* some private memory, inaccessible from outside the module. An *externally visible effect* is any observable update to the global memory and the result of a call to a public method of the module. Similarly, *atomic* will be with respect to all public methods on the module and with any code outside the module. When a module implements a data structure and provides update and access methods, it is often useful to define an abstract version of the data structure and assume that it is globally visible. Even though it may not be visible directly, all its data would be available through the module's access methods.

5.1.3 Properties

Linearisability is a local notion: a history consisting of calls to multiple separate objects is linearisable if and only if it is linearisable with respect to each of the objects separately. Hence, we can partition the task of proving linearisability and consider one object at a time.

In addition, linearisability enables concise specifications: a linearisable method is specified by just the precondition and the postcondition of its linearisation point. Since all interesting concurrency is hidden within the module, the method can be given a simple specification summarising its sequential effect.

5.2 Proving linearisability

In theory, a method is linearisable if and only if modulo termination it is equivalent to a single atomic block performing its abstract operation. Therefore, to prove that a method is linearisable it is sufficient to be able to reason about program equivalence in a possibly restricted context.

The standard way for reasoning about program equivalence is to represent programs as transition systems and to define appropriate simulations between them. Unfortunately, this approach is not practical. Transition systems and simulation relations are unintuitive to the average programmer. Wanting to reason about his C program, the programmer has to translate it to a different formalism (a transition system) and to define special simulation relations between objects of that new system.

This complexity is, however, unnecessary. Proving linearisability may be a special case of proving program equivalence, but reasoning about this special case is much simpler than about the general case. Instead, we can embed the specification within the program as auxiliary code. Writing these annotations is a form of programming, familiar to the average programmer. As the examples in §5.3 will demonstrate, most algorithms require very few annotations. More advanced algorithms (e.g. RCDSS) require more auxiliary code, but still the designers of the algorithm should be able to come up with these

annotations if they believe that their algorithm is correct.

Before presenting further details, it is useful to recall how we prove the correctness of sequential programs. Assume we are given a low-level implementation (e.g. a sorted linked list) and a high-level specification (e.g. a set of integers), and we want to prove that the implementation conforms to the specification. First, assume that concrete program and the abstract program operate on disjoint states: the concrete and the abstract state respectively. If not, we can rename the variables of either program to satisfy this condition. Then, define an *abstraction map* that relates the concrete state to the abstract state. Execute the two programs in sequence, and show that if the abstraction map holds initially, it also holds when both programs end. For an object with multiple methods, repeat this process for each method of the object.

For concurrent programs, we can take a similar approach.

1. For each method, locate the *linearisation point* as a point in the concrete source code. The intended meaning is that when this program point is executed, it is a valid linearisation point for the particular method call.

2. Embed the abstract operation as an atomic command at that program point.

3. Define an *abstraction map* that relates the concrete and the abstract states.

4. Prove that if the abstraction map holds initially, then it holds continuously throughout the program, and that the concrete method and the abstract method return the same results.

Unfortunately, the first step may fail. Sometimes, we cannot identify the linearisation point as as a point in the program's control-flow graph because it depends on future behaviour.

Herlihy and Wing [45] gave an artificial example of a concurrent queue, whose linearisation point could not be specified directly as a point in the source code. They went on to reject this proof method and proposed a more elaborate method whereby each legal concrete state is mapped to a set of abstract states.

Their approach, however, is unnecessarily complex. Instead, we shall introduce auxiliary variables to capture the location of the linearisation point. The amount of auxiliary state needed to carry out the proof of a program is a good measure for the program's synchronisation intricacy. For common synchronisation patterns, there is a systematic way of introducing such auxiliary state.

5.2.1 Single assignment variables

To prove that the embedded abstract operation at the linearisation point was executed exactly once, it is useful to have special variables that during execution get assigned at

most one time. The proof technique is to introduce one such variable per method, initially unset, to assign to it at the linearisation point, and to have the postcondition check that the variable has been set. This ensures that the abstract operation was executed exactly once, and, if the linking invariant is satisfied, the method is linearisable.

A write-once variable is a variable that at its definition does not contain any value and that dynamically gets assigned to at most once. Formally, we can define write-once variables as an abstract data type with four operations:

- $\texttt{new}_{\texttt{single}}$: It creates a new, uninitialised write-once memory cell.
- $\texttt{read}_{\texttt{single}}$, which returns the value stored in the write-once memory cell.
- $\texttt{write}_{\texttt{single}}$: This has a precondition saying that the variable is uninitialised, so that it cannot be assigned multiple times.
- $\texttt{dispose}_{\texttt{single}}$: This simply deallocates a single-assignment variable.

These operations have specifications:

$$\begin{aligned} \{emp\} \quad & \texttt{x} := \texttt{new}_{\texttt{single}}; \quad \{\texttt{x} \overset{s}{\mapsto} undef\} \\ \{y \overset{s}{\mapsto} z \wedge e = y\} \quad & \texttt{x} := [e]_{\texttt{single}}; \quad \{y \overset{s}{\mapsto} z \wedge \texttt{x} = z\} \\ \{e_1 \overset{s}{\mapsto} undef\} \quad & [e_1]_{\texttt{single}} := e_2; \quad \{e_1 \overset{s}{\mapsto} e_2\} \\ \{e \overset{s}{\mapsto} \_\} \quad & \texttt{dispose}_{\texttt{single}}(e); \quad \{emp\} \end{aligned}$$

For convenience, we shall use $[e]_{\texttt{single}}$ notation for accessing the value stored in the write-once memory location e. We will use the keyword `single` to declare such variables, and we will omit the subscripts when they are obvious from the context.

We can impose a syntactic restriction that assignments to a certain write-once variable occur only together with the abstract operation at potential linearisation points. Hence, if after the execution of a method that write-once variable has been set, a linearisation point was executed; moreover, the abstract operation took place exactly once.

An alternative approach is to keep a counter per method call counting how many times the abstract operation is executed (by a similar syntactic restriction). At the beginning set the counter to 0, and at the end prove it contains 1. This approach is less convenient because it requires more state to be carried around in the proofs.

5.2.2 Proof technique for a class of lock-free algorithms

Lock-free algorithms are a class of non-blocking algorithms which get rid of deadlock and livelock, but not starvation. At all times, if some thread participating in the algorithm is scheduled, then global progress is made in the sense that some outstanding operation makes a non-Zeno step towards completion, even if some threads have failed or are

descheduled for arbitrarily long. Clearly, lock-free algorithms cannot use locks to synchronise between threads, because if one thread acquires a lock and then gets descheduled and another thread waits for that lock, then no thread makes progress.

Instead, a large class of lock-free algorithms splits the operation in two phases: The thread executing a method, first, announces its intention to perform operation k by atomically placing a record in memory; then, it proceeds to complete the operation. This record contains all the information necessary in order to complete k. Therefore, if another concurrent thread interferes with operation k, it will first help the first thread complete k before performing its own operation. Therefore, the linearisation point of a method call is not necessarily in the code of the method itself, because another thread could have intervened and finished off the operation on behalf of the caller.

Typically, these algorithms create a record in the heap describing the operation to be performed. This descriptor record becomes visible to any interfering thread so that they can 'help' the outstanding method to finish. For the proof, it is helpful to extend this descriptor record with a single-assignment auxiliary field `AbsResult`. (In algorithms that do not have descriptor records, such as "lazy contains" [73], we can introduce an auxiliary record containing just the field `AbsResult`.) At each linearisation point, we perform the *abstract effect* of the operation, and assign its result to `AbsResult`. Then, we verify that:

$$\{p \wedge d.\texttt{AbsResult} \overset{s}{\mapsto} \mathit{undef}\}\ \text{ConcreteOp}(d)\ \{d.\texttt{AbsResult} \overset{s}{\mapsto} \texttt{Result}\},$$

which means that there was precisely one assignment to $d.\texttt{AbsResult}$ during the execution of ConcreteOp(d). By construction, we know that there is a one-to-one correspondence between assignments to $d.\texttt{AbsResult}$ and linearisation points. This entails that the entire abstract operation was executed precisely once at some moment between the invocation and the return of the concrete operation. Finally, the abstract and the concrete operations return the same value; so, they have the same externally-visible behaviour.

5.2.3 Proof technique for read-only methods

If a method does not have any side-effects, we can employ a simpler approach. All we need to prove that there is a possible linearisation point. We can relax the requirement that the abstract operation was executed exactly once to at least once. Since the operation does not have any side-effects, executing it multiple times does not matter. This adjustment often simplifies the annotations to such methods, as well as the corresponding proofs.

5.2.4 Common annotation patterns

For each method call, we associate a descriptor record with one field for each argument of the method and one additional field, `AbsResult`, which is assigned at the linearisation

point. Calling a method implicitly allocates a new such record in the heap. Within the method's body, let the program variable '`this`' point to that record.

For those algorithms that are passed a descriptor record as a parameter, we do not create a new record. Instead we extend the existing record with an `AbsResult` field.

We define the following syntactic sugar for linearisation point annotations. $\texttt{Lin}_{e_1,\ldots,e_n}$ declares that the descriptors $e_1, \ldots, e_n$ are linearised at this point in the order they were given. The most common pattern is to write $\texttt{Lin}_{\texttt{this}}$ at the projected linearisation points. $\texttt{Lin}_{e_1,\ldots,e_n}(b)$ is a conditional linearisation point: if b holds, then e_1 up to e_n are linearised; otherwise, they're not.

$$\begin{aligned} \texttt{Lin}_{e_1,\ldots,e_n} &\stackrel{\text{def}}{=} e_1.\texttt{AbsResult} := \texttt{AbsOp}(e_1);\ \ldots;\ e_n.\texttt{AbsResult} := \texttt{AbsOp}(e_n); \\ \texttt{Lin}_{e_1,\ldots,e_n}(b) &\stackrel{\text{def}}{=} \texttt{if}\ (b)\ \{\texttt{Lin}_{e_1,\ldots,e_n}\} \end{aligned}$$

Finally, a very common idiom is for successful compare-and-swap (CAS) operations to be linearisation points. Therefore, we annotate the CAS instructions with the sequence of operations whose linearisation points take place if the CAS is successful.

$$\begin{aligned} \texttt{CAS}_{e_1,\ldots,e_n}(a,o,n) \stackrel{\text{def}}{=}\ & \langle \\ & \texttt{bool b} := \texttt{CAS}(a,o,n); \\ & \texttt{Lin}_{e_1,\ldots,e_n}(\texttt{b}); \\ & \texttt{return b}; \\ & \rangle \end{aligned}$$

The angle brackets $\langle C \rangle$ denote that the command C is executed atomically.

5.2.5 Prophecy variables

In some advanced algorithms, the location of the linearisation point depends on unpredictable future behaviour. For example, the linearisation point of RDCSS (see §5.3.3) is the read operation, but only if a later CAS succeeds.

Fortunately, we can circumvent this problem by introducing an auxiliary prophecy variable. Prophecy variables, introduced by Abadi and Lamport [1], are an analogue to history variables and capture finite knowledge about the future. Introducing such prophecy variables is definitively not intuitive, but Abadi and Lamport showed that it is sound under some finiteness conditions. A prophecy variable amounts to an oracle that guesses some condition about the future.

Here, we will be interested only in whether a certain future CAS succeeds; so, a single boolean prophecy variable is sufficient (and it trivially satisfies the required finiteness conditions).

5.3 Examples

This section demonstrates the techniques described in §5.2 by verifying a number of fine-grained concurrent algorithms. Its aim is to show recurring auxiliary code patterns rather than complete proofs. Hence, for brevity, most details will be omitted and only proof sketches presented. Where more details are given, the examples act as further case studies for RGSep. We shall consider the following algorithms:

1. concurrent stack implementations:
   - simple non-blocking stack
   - elimination-based non-blocking stack of Hendler et al. [39]
2. list-based set implementations:
   - lock-coupling list-based set
   - optimistic list-based set
   - lazy concurrent list-based set of Heller et al. [38]
3. restricted double-compare single-swap (RDCSS) of Harris et al. [34]
4. multiple compare-and-swap (MCAS) of Harris et al. [34]

The hope is that the actual proof can be generated automatically from a program annotated with the necessary auxiliary code and its rely and guarantee conditions. Such an automatic tool has not yet been implemented: Chapter 6 describes a less powerful tool that can reason about the shape of memory-allocated data structures, but not about their contents.

5.3.1 Stack algorithms

First, we shall consider two non-blocking stack algorithms. Both algorithms represent the stack as a singly-linked list starting from a known address `S`. The stack interface consists of two operations, `push` and `pop`, with the following abstract implementations:

$$
\begin{array}{l}
\texttt{Abs\_push}(e) \stackrel{\text{def}}{=} \langle\ \texttt{Abs} := e{\cdot}\texttt{Abs};\ \texttt{AbsResult} := e;\ \rangle \\
\texttt{Abs\_pop}() \stackrel{\text{def}}{=} \langle\ \textbf{case}\ (\texttt{Abs}) \\
\quad \mid \epsilon \Longrightarrow \quad \texttt{AbsResult} := \texttt{EMPTY}; \\
\quad \mid v{\cdot}A \Longrightarrow \{\ \texttt{Abs} := A;\ \texttt{AbsResult} := v;\ \}\ \rangle
\end{array}
$$

This pseudocode uses sequences and pattern matching notation. We write ϵ for the empty sequence, and $A{\cdot}B$ for the concatenation of sequences A and B. `Abs_push` just adds a new element at the beginning of the sequence. `Abs_pop` first checks if the sequence is empty. If it is empty, it returns the reserved value `EMPTY`; otherwise, it removes the first element

```
class Cell {Cell next;  value_t data;}
Cell S, Abs;

void push (value v) {
  Cell t, x;
  x := new Cell();
  x.data := v;
  do {
    ⟨t := S;⟩
    x.next := t;
  } while (¬CAS_this(&S, t, x));
}
```

```
value pop () {
  Cell t, x;
  do {
    ⟨t := S; Lin_this(t = null);⟩
    if(t = null)
      return EMPTY;
    x := t.next;
  } while(¬CAS_this(&S, t, x));
  return t.data;
}
```

Figure 5.1: Annotated simple stack implementation

from the sequence and returns it. Normally pushing a value does not return anything, but here we assume that `Abs_push`(e) always returns e. This simplifies the specifications of the second stack algorithm.

Simple stack

Figure 5.1 contains a simple lock-free implementation of a concurrent stack. The stack is stored as a linked list, and is updated by CAS instructions. The code illustrates a common design pattern in lock-free algorithms. Each method (i) reads the state's current value; (ii) computes an updated value depending on the value read; and (iii) atomically updates the state swapping the new value for its old value. If the state has changed between (i) and (iii) and has not been restored to its initial value, the CAS will fail, and we repeat the algorithm until the CAS succeeds.

The highlighted code is the auxiliary code that is needed for the linearisability proof. In this case, the auxiliary code just consists of program annotations defining the linearisation points. The linearisation point of `push` is its CAS when it succeeds; the linearisation point of `pop` is reading `S`, if it was empty, or the CAS, if that succeeds.

RGSep proof The proof is straightforward. To avoid 'variables as resource' [64], we treat the shared global variables `S` and `Abs` as memory cells at addresses &`S` and &`Abs` respectively. Any use of `S` in the code is just a shorthand for [&`S`]. We have two actions: pushing an element onto the stack, and removing an element from the stack.

$$\&\mathtt{S}{\mapsto}y * \&\mathtt{Abs} \mapsto A \;\rightsquigarrow\; \&\mathtt{S}{\mapsto}x * x{\mapsto}\mathsf{Cell}(v,y) * \&\mathtt{Abs} \mapsto v{\cdot}A \quad (\mathsf{Push})$$

$$\&\mathtt{S}{\mapsto}x * x{\mapsto}\mathsf{Cell}(v,y) * \&\mathtt{Abs} \mapsto v{\cdot}A \;\rightsquigarrow\; \&\mathtt{S}{\mapsto}y * x{\mapsto}\mathsf{Cell}(v,y) * \&\mathtt{Abs} \mapsto A \quad (\mathsf{Pop})$$

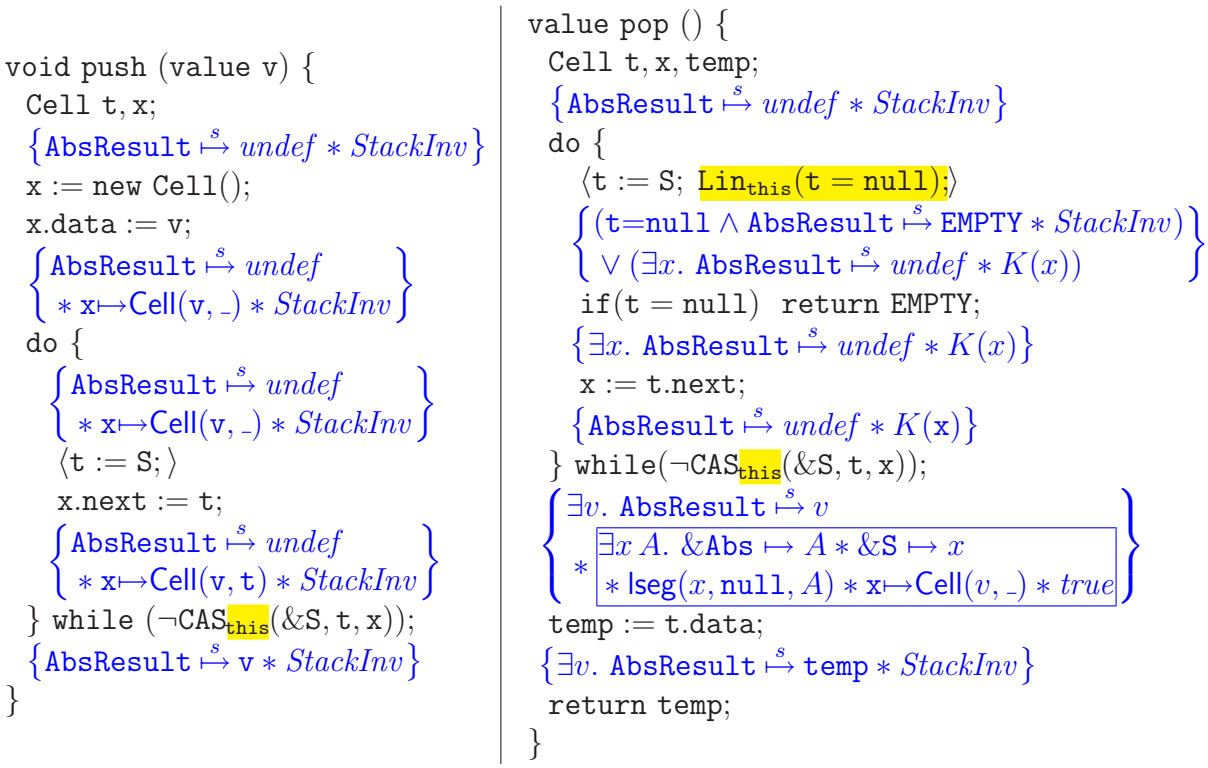

Figure 5.2: Proof outline for the simple stack

These actions also push or pop a value from the abstract stack. Thus, if the concrete and the abstract stack are equivalent at the beginning, they are still equivalent after one atomic operation. Formally, we can define the following list segment predicate:

$$\begin{aligned}\mathsf{lseg}(x, y, A) \stackrel{\text{def}}{=}\ & (x = y \wedge A = \epsilon \wedge \mathit{emp}) \\ & \vee\ (x \neq y \wedge \exists v\, z\, B.\ x{\mapsto}\mathsf{Cell}(v, z) * \mathsf{lseg}(z, y, B) \wedge A = v{\cdot}B)\end{aligned}$$

This represents a singly-linked list segment from x until y, whose values form the sequence A. We write $A{\cdot}B$ for sequence concatenation and ϵ for the empty sequence. It is straightforward to prove that the following assertion is stable under the two possible actions, (Push) and (Pop).

$$\mathit{StackInv} \stackrel{\text{def}}{=} \boxed{\exists x\, A.\ \&\mathtt{S} \mapsto x * \&\mathtt{Abs} \mapsto A * \mathsf{lseg}(x, \mathtt{null}, A) * \mathit{true}}$$

Informally, this assertion says that `S` points to the head of a list (x) which represents the sequence (A) stored in `Abs`. The "$* \mathit{true}$" conjunct indicates that there may be other garbage nodes in the shared heap. This happens because of the Pop action: in its postcondition, the cell x remains in the shared heap although it is not reachable from `S`.

Figure 5.2 shows the proof outline. Besides the invariant, the other crucial assertion

about the shared state is:

$$K(y) \stackrel{\text{def}}{=} \boxed{\begin{array}{l}\left(\begin{array}{l}\exists x\, v\, A\, B.\ \&\mathtt{Abs} \mapsto A{\cdot}v{\cdot}B * \&\mathtt{S} \mapsto x * \mathsf{lseg}(x, \mathtt{t}, A) \\ * \ \mathtt{t}{\mapsto}\mathsf{Cell}(v, y) * \mathsf{lseg}(y, \mathtt{null}, B) * \mathit{true}\end{array}\right) \\ \vee\ (\exists x\, A.\ \&\mathtt{Abs} \mapsto A * \&\mathtt{S} \mapsto x * \mathsf{lseg}(x, \mathtt{null}, A) * \mathtt{t}{\mapsto}\mathsf{Cell}(\_, \_) * \mathit{true})\end{array}}$$

This asserts that either the cell pointed to by `t` is in the stack and the next cell is y, or it is not in the stack but still in the shared state. This assertion is also stable under the two permitted actions Push and Pop. Further, note that $K(y) \Rightarrow \mathit{StackInv}$.

The assertion $K(y)$ is crucial for the safety of the algorithm. In order to prove that `pop` is linearisable, we must ensure that when the CAS succeeds, `t.next` = `x`. If between the assignment `x := t.next` and the following CAS, the node `t` could have been popped from the stack, the stack changed, and the node `t` pushed back onto the top of the stack (so that `t.next` $\neq$ `x`), then the CAS would succeed, but would do something different than popping `t` from the stack. This is known as an ABA problem: the stack was in state A, then changed to state B, but the CAS does not distinguish between states A and B; so it thinks it is still in state A. Note how the actions rule out this from happening: Push adds a *new* node to the shared state and Pop leaves popped nodes in the shared state. That is essentially why the assertion $K(y)$ is stable under the rely.

Now, consider the atomic blocks in more detail. Following the exact details below is not central to understanding the algorithm or the linearisability proofs. It is just an example of a RGSep proof that could be automatically generated by a moderate extension of SmallfootRG (see Chapter 6 for more about mechanisation).

1. `CAS` of push We have to prove the triple:

$$\left\{\begin{array}{l}\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} \\ * \ \mathtt{x}{\mapsto}\mathsf{Cell}(\mathtt{v}, \mathtt{t}) * \mathit{StackInv}\end{array}\right\}\ \mathtt{b} := \mathtt{CAS}_{\mathtt{this}}(\&\mathtt{S}, \mathtt{t}, \mathtt{x});\ \left\{\begin{array}{l}(\mathtt{b} \wedge \mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{v} * \mathit{StackInv}) \\ \vee \left(\begin{array}{l}\neg\mathtt{b} \wedge \mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} \\ * \ \mathtt{x}{\mapsto}\mathsf{Cell}(\mathtt{v}, \mathtt{t}) * \mathit{StackInv}\end{array}\right)\end{array}\right\}$$

Applying Cad (see §4.2), Ex, and Conseq reduces the problem to showing:

$$\left\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * \mathtt{x}{\mapsto}\mathsf{Cell}(\mathtt{v}, \mathtt{t}) * \boxed{\exists A.\ \&\mathtt{S}{\mapsto}\mathtt{t} * \&\mathtt{Abs}{\mapsto}A * \mathsf{lseg}(\mathtt{t}, \mathtt{null}, A) * \mathit{true}}\right\}$$
$$\left\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * \mathtt{x}{\mapsto}\mathsf{Cell}(\mathtt{v}, \mathtt{t}) * \boxed{\&\mathtt{S} \mapsto \mathtt{t} * \&\mathtt{Abs} \mapsto A * \mathsf{lseg}(\mathtt{t}, \mathtt{null}, A) * \mathit{true}}\right\}$$
$$\langle[\&\mathtt{S}] := \mathtt{x};\ \mathtt{Abs\_push}(\mathtt{v})\rangle$$
$$\left\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{v} * \boxed{\&\mathtt{S} \mapsto \mathtt{x} * \&\mathtt{Abs} \mapsto \mathtt{v}{\cdot}A * \mathtt{x}{\mapsto}\mathsf{Cell}(\mathtt{v}, \mathtt{t}) * \mathsf{lseg}(\mathtt{t}, \mathtt{null}, A) * \mathit{true}}\right\}$$
$$\left\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{v} * \boxed{\exists x A.\ \&\mathtt{S} \mapsto x * \&\mathtt{Abs} \mapsto A * \mathsf{lseg}(x, \mathtt{null}, A) * \mathit{true}}\right\}$$

Finally, apply the rule Atom with

- $P = \&\mathtt{S} \mapsto \mathtt{t} * \&\mathtt{Abs} \mapsto A$,

- $Q = \&\mathtt{S} \mapsto \mathtt{x} * \&\mathtt{Abs} \mapsto \mathtt{v}{\cdot}A * \mathtt{x} \mapsto \mathsf{Cell}(\mathtt{v}, \mathtt{t})$, and
- $F = \mathsf{lseg}(\mathtt{t}, \mathtt{null}, A) * \mathit{true}$.

Note that the action $P \rightsquigarrow Q$ is an instance of Push.

2. CAS of pop We have to prove the triple:

$$\left\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * K(\mathtt{x})\right\}$$
$$\mathtt{b} := \mathtt{CAS}_{\mathtt{this}}(\&\mathtt{S}, \mathtt{t}, \mathtt{x});$$
$$\left\{\begin{array}{l}\left(\begin{array}{l}\mathtt{b} \wedge \exists v.\ \mathtt{AbsResult} \overset{s}{\mapsto} v * \\ \qquad \boxed{\begin{array}{l}\exists x\ A.\ \&\mathtt{Abs} \mapsto A * \&\mathtt{S} \mapsto x * \mathsf{lseg}(x, \mathtt{null}, A) \\ * \mathtt{phead} \mapsto \mathsf{Cell}(v, \_) * \mathit{true}\end{array}}\end{array}\right) \\ \vee\ (\neg\mathtt{b} \wedge \mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * K(\mathtt{x}))\end{array}\right\}$$

The proof is analogous to the previous case and proceeds by applying Cad, Ex, Conseq, and finally Atom with

- $P = \&\mathtt{S} \mapsto \mathtt{t} * \mathtt{t} \mapsto \mathsf{Cell}(v, \mathtt{x}) * \&\mathtt{Abs} \mapsto v{\cdot}A$
- $Q = \&\mathtt{S} \mapsto \mathtt{x} * \mathtt{t} \mapsto \mathsf{Cell}(v, \mathtt{x}) * \&\mathtt{Abs} \mapsto A$
- $F = \mathsf{lseg}(\mathtt{pnext}, \mathtt{null}, A) * \mathit{true}$

3. Reading S in pop We must prove the following specification:

$$\left\{\begin{array}{l}\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} \\ * \mathit{StackInv}\end{array}\right\} - \left\{\begin{array}{l}(\mathtt{t} = \mathtt{null} \wedge \mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{EMPTY} * \mathit{StackInv}) \\ \vee\ (\exists x.\ \mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * K(x))\end{array}\right\}$$

Apply Atomic with the following parameters:

- $\{\overline{y}\} = \emptyset$
- $P' = Q' = \left(\begin{array}{l}(b \wedge \&\mathtt{S} \mapsto \mathtt{null} * \&\mathtt{Abs} \mapsto \epsilon) \\ \vee\ (\neg b \wedge \&\mathtt{S} \mapsto x * x \mapsto \mathsf{Cell}(v, y) * \&\mathtt{Abs} \mapsto A)\end{array}\right)$
- $P'' = \mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef}$
- $Q'' = \left(\begin{array}{l}(b \wedge \mathtt{t} = \mathtt{null} \wedge \mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{EMPTY}) \\ \vee\ (\neg b \wedge \mathtt{t} \neq \mathtt{null} \wedge \mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef})\end{array}\right)$
- $F = (b \wedge \mathsf{lseg}(x, \mathtt{null}, A) * \mathit{true}) \vee (\neg b \wedge \mathsf{lseg}(y, \mathtt{null}, A) * \mathit{true})$

$$\frac{\begin{array}{ccccc}(P' \rightsquigarrow Q') \subseteq G & & \vdash C\ \mathbf{sat}\ (P' * P'', \emptyset, \emptyset, Q' * Q'') & & \boxed{\exists \overline{y}.P}\ \text{stable under } R \\ FV(P'') \cap \{\overline{y}\} = \emptyset & \models_{\mathsf{SL}} P \Rightarrow P' * F & & \models_{\mathsf{SL}} Q' * F \Rightarrow Q & \boxed{Q}\ \text{stable under } R\end{array}}{\vdash \langle C \rangle\ \mathbf{sat}\ (\boxed{\exists \overline{y}.\ P} * P'', R, G, \exists \overline{y}.\ \boxed{Q} * Q'')}$$

Note that we inserted the logical variable b to record whether the stack is empty or not. The premises of Atomic follow because:

- P and Q are stable (proof omitted).
- $\models_{\mathsf{SL}} P \Rightarrow (P' * F)$.

- $\models_{\mathsf{SL}} (Q' * F) \Rightarrow Q$.
- $(P' \rightsquigarrow Q') \subseteq G$. This holds because $P' = Q'$ and P' is exact.
- The body of the atomic satisfies $\{P' * P''\}$ _ $\{Q' * Q''\}$, namely:

$$\left\{\begin{array}{l}(b \wedge \&\mathtt{S} \mapsto \mathtt{null} * \&\mathtt{Abs} \mapsto \epsilon * \mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef}) \\ \vee (\neg b \wedge \&\mathtt{S} \mapsto x * x{\mapsto}\mathsf{Cell}(v, y) * \&\mathtt{Abs} \mapsto A * \mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef})\end{array}\right\}$$
$$\mathtt{t} := [\&\mathtt{S}];$$
$$\mathtt{if}\ (\mathtt{t} = \mathtt{null})\ \ [\mathtt{AbsResult}] := \mathtt{Abs\_pop}();$$
$$\left\{\begin{array}{l}(b \wedge \mathtt{t} = \mathtt{null} \wedge \&\mathtt{S} \mapsto \mathtt{null} * \&\mathtt{Abs} \mapsto \epsilon * \mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{EMPTY}) \\ \vee (\neg b \wedge \mathtt{t} \neq \mathtt{null} \wedge \&\mathtt{S} \mapsto x * x{\mapsto}\mathsf{Cell}(v, y) * \&\mathtt{Abs} \mapsto A * \mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef})\end{array}\right\}$$

Therefore, we have proved that `push` and `pop` have the following specifications:

$$\left\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * \mathit{StackInv}\right\}\ \mathtt{push(v)}\ \left\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{v} * \mathit{StackInv}\right\}$$
$$\left\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * \mathit{StackInv}\right\}\ \mathtt{x := pop()}\ \left\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{x} * \mathit{StackInv}\right\}$$

with $R = G = \{(\mathsf{Push}), (\mathsf{Pop})\}$. Furthermore, as `AbsResult` was written only at valid linearisation points, these specifications entail that `push` and `pop` are linearisable, hence observationally equivalent (in any context that satisfies R and does not access `S` directly) to the atomic operations `Abs_push` and `Abs_pop`.

HSY elimination-based stack

Hendler, Shavit, and Yerushalmi [39] presented an improved version of a stack that performs better on higher workloads. Figure 5.3 shows an adapted version of their algorithm. The highlighted bits in the algorithm is the auxiliary code needed to specify and verify the linearisability of the algorithm.

The implementation is moderately complex because it combines two algorithms: a central singly-linked list, `S`, and an elimination layer. The elimination layer consists of two global arrays: `loc`[1..threadNum] which has one element per thread storing a pointer to a `ThreadInfo` record, and `coll`[1..size] which stores the identifiers of the threads trying to collide.

A *push* or a *pop* operation first tries to perform the operation on the central stack object, by doing a CAS to change the shared top-of-the-stack pointer. If it is successful then it is the linearisation point of the operation. Otherwise, if the CAS fails (because of contention), the thread backs off to the elimination scheme. If this scheme fails, it tries again the top-of-the stack pointer and so on until one of the two schemes succeeds.

The elimination scheme works as follows: Thread `p` first announces its arrival at the collision layer by writing its descriptor in the `loc` array. Then it selects a random slot in the `coll` array, and it atomically reads that slot and overwrites it with its own identifier.

```
class Cell {Cell next;  value_t data;}

class ThreadInfo {
  int id;       // thread identifier
  int op;       // PUSH or POP
  value_t data;
  single int AbsResult; }

Cell S, Abs;    // shared stack object(s)
ThreadInfo loc[1..threadNum];
int coll[1..size];

void StackOp(ThreadInfo p)
 int him, pos;
 while (true){
  if (TryStackOp(p)) return;
  ⟨loc[mypid] := p;⟩
  pos := GetPosition(p);
  him := coll[pos];
  while(¬CAS(&coll[pos], him, mypid))
   him := coll[pos];
  if (1 ≤ him ≤ threadNum){
   ThreadInfo q := loc[him];
   if (q≠null ∧ q.id=him ∧ q.op≠p.op)
    if (CAS(&loc[mypid], p, null))
     if (TryCollision(p, q))
      return;
     else
      continue;
    else
     FinishCollision(p); return;
  }
  delay();
  if (¬CAS(&loc[mypid], p, null))
   FinishCollision(p); return;
 }
```

```
/* Private methods */

bool TryStackOp(ThreadInfo p)
 Cell phead, pnext;
 if (p.op = PUSH)
  phead := S;
  pnext := new Cell(p.data, phead);
  return CAS_p(&S, phead, pnext);
 else if (p.op = POP)
  ⟨phead := S; Lin_p(phead = null);⟩
  if (phead = null)
   p.data := EMPTY;
   return true;
  pnext := phead.next;
  if (CAS_p(&S, phead, pnext))
   p.data := phead.data;
   return true;
  else
   return false;

bool TryCollision(ThreadInfo p, q)
 bool b;
 if (p.op = PUSH)
  b := CAS_{p,q}(&loc[q.id], q, p);
 else if (p.op = POP)
  b := CAS_{q,p}(&loc[q.id], q, null);
  if(b) p.data := q.data;
 return b;

void FinishCollision(ThreadInfo p)
 if (p.op = POP)
  p.data := loc[mypid].data;
  ⟨loc[mypid] := null;⟩
```

Figure 5.3: Annotated HSY stack implementation

Now there are two possible scenarios depending on the value read from the `coll` array.

If p reads the identifier of another thread q executing an opposite operation, it attempts to eliminate itself with it: First, it does a CAS to remove its entry from the `loc` array, so that no other threads might eliminate themselves with p, and then tries to remove q from the `loc` array with another CAS. If this last CAS succeeds, then the two operations have eliminated each other. This is the linearisation point of both the PUSH and the POP,

with the PUSH operation happening immediately before the POP. If the first CAS failed, this is because `p` was eliminated by some other thread in the meantime; so we just finish the collision and return.

Otherwise, the thread delays for a small amount of time in the hope that some other thread will collide with it. Then it does a CAS to remove its entry from the `loc` array. If this CAS fails, `p` was eliminated by some other thread; so we just finish the collision and return. Otherwise we just go round the loop and try again the operation on the shared top-of-the-stack pointer.

Linearisation points Following the methodology in §5.2.2, we introduced the auxiliary "single-assignment" field `AbsResult`, and annotated the linearisation points with assignments to it. This algorithm has two types of linearisation points. Those in `TryStackOp` are quite standard and linearise the operation `p` that is performed. On the other hand, the compare-and-swaps in `TryCollision`, if successful, linearise both `p` and `q`, doing the PUSH before the POP.

Proof structure In this proof, we will exploit the modularity of RGSep to reason separately about the shared stack object stored in `S` (code within `TryStackOp`) and separately about the elimination scheme (the rest of the algorithm). The benefits of this approach are that:

- The proof is simpler because the collision layer proof does not need not worry about the shared stack, and vice versa.
- The proof explains the elimination scheme orthogonally to the stack implementation.
- The proof is more robust. If we change the implementation of the collision layer or of the shared stack, we need to repeat only part of the proof.

Standard rely/guarantee cannot achieve this modularity in the proof because interference is global. Concurrent separation logic enables this modularity, but it cannot tractably carry out either of the subproofs.

In fact, we can decompose the collision layer proof further. The part involving the collision array is just an optimisation for selecting an identifier that is likely to be waiting for collision. Assigning any random value to `him` is sufficient for the safety of the algorithm.

The proofs use three disjoint regions of shared state (cf. §4.3.2). Stack contains the shared stack object and the abstract stack object; Loc contains the `loc`[..] array together with any `ThreadInfo` records that have become shared through that array. Finally, Coll just contains the `coll`[..] array.

Predicates The proof uses the simplest case of fractional permissions (see §2.4) with just two fractions: 1/2 (for a read permission) and 1 (for a full permission). Adding two half permissions gives a full permission: $x \overset{1/2}{\mapsto} v_1 * x \overset{1/2}{\mapsto} v_2 \iff x \mapsto v_1 \wedge v_1 = v_2$.

To describe `ThreadInfo` records, we shall use the following predicates:

$$\begin{aligned}
\mathsf{T}(p, id, op, v, r) &\overset{\text{def}}{=} p{\mapsto}\texttt{ThreadInfo}\ \{\texttt{.id}\overset{1/2}{=}id;\ \texttt{.op}\overset{1/2}{=}op;\ \texttt{.data}{=}v;\ \texttt{.AbsResult}\overset{s}{=}r\} \\
&\quad \wedge\ id \in T \wedge op \in \{\mathsf{PUSH}, \mathsf{POP}\} \\
\mathsf{T_U}(p, id, op, v) &\overset{\text{def}}{=} \mathsf{T}(p, id, op, v, \mathit{undef}) \\
\mathsf{T_R}(p, id, op) &\overset{\text{def}}{=} p{\mapsto}\texttt{ThreadInfo}\ \{\texttt{.id}\overset{1/2}{=}id;\ \texttt{.op}\overset{1/2}{=}op;\} \\
&\quad \wedge\ id \in T \wedge op \in \{\mathsf{PUSH}, \mathsf{POP}\}
\end{aligned}$$

The predicate $\mathsf{T}(p, id, op, v, r)$ describes a record at address p containing a valid thread identifier (i.e. belonging to the set of all thread identifiers, T), a descriptor (PUSH or POP) denoting which operation is performed, a data value and the result of the abstract operation at the linearisation point. The read permissions enable the `id` and `op` fields to be shared. The predicate $\mathsf{T_U}(p, id, op, v)$ just describes an operation before its linearisation point. Finally, $\mathsf{T_R}(p, id, op)$ contains the remaining permissions of a `ThreadInfo` record.

These permissions are used to make the proof modular. At some points in the algorithm both the central stack and the elimination layer need to know that a certain `ThreadInfo` record p exists, but only one of the two components actually need to update its `data` and `AbsResult` fields. With this encoding, we give a $\mathsf{T}(p, \_, \_, \_, \_)$ permission to the first component and a $\mathsf{T_R}(p, \_, \_)$ to the second.

Central stack object The shared stack object proof is almost identical to the proof of the simple stack in the beginning of this section. Again, we treat `S` and `Abs` as allocated in the heap addresses &`S` and &`Abs` respectively. The actions and the invariant are identical:

$$\&\mathtt{S}{\mapsto}y * \&\mathtt{Abs} \mapsto A \ \rightsquigarrow\ \&\mathtt{S}{\mapsto}x * x{\mapsto}\mathsf{Cell}(v, y) * \&\mathtt{Abs} \mapsto v{\cdot}A \qquad (\mathsf{Push})$$

$$\&\mathtt{S}{\mapsto}x * x{\mapsto}\mathsf{Cell}(v, y) * \&\mathtt{Abs} \mapsto v{\cdot}A \ \rightsquigarrow\ \&\mathtt{S}{\mapsto}y * x{\mapsto}\mathsf{Cell}(v, y) * \&\mathtt{Abs} \mapsto A \qquad (\mathsf{Pop})$$

As before, the invariant says that `Abs` stores the sequence that `S` represents. The `Stack` subscript indicates the region described by the boxed assertion.

$$\mathit{StackInv} \overset{\text{def}}{=} \boxed{\exists x\ A.\ \&\mathtt{S} \mapsto x * \mathtt{Abs} \mapsto A * \mathsf{lseg}(x, \mathtt{null}, A) * \mathit{true}}_{\mathtt{Stack}}$$

For the function `TryStackOp`, we prove the following specification:

$$\left\{\begin{array}{l}\mathsf{T_U}(\mathsf{p}, \mathsf{mypid}, \_, \_) \\ * \mathit{StackInv}\end{array}\right\}\ \mathtt{x} := \mathtt{TryStackOp}(\mathtt{p});\ \left\{\begin{array}{l}(\mathsf{x} \wedge \exists v.\ \mathsf{T}(\mathsf{p}, \mathsf{mypid}, \_, v, v) * \mathit{StackInv}) \\ \vee\ (\neg\mathsf{x} \wedge \mathsf{T_U}(\mathsf{p}, \mathsf{mypid}, \_, \_) * \mathit{StackInv})\end{array}\right\}$$

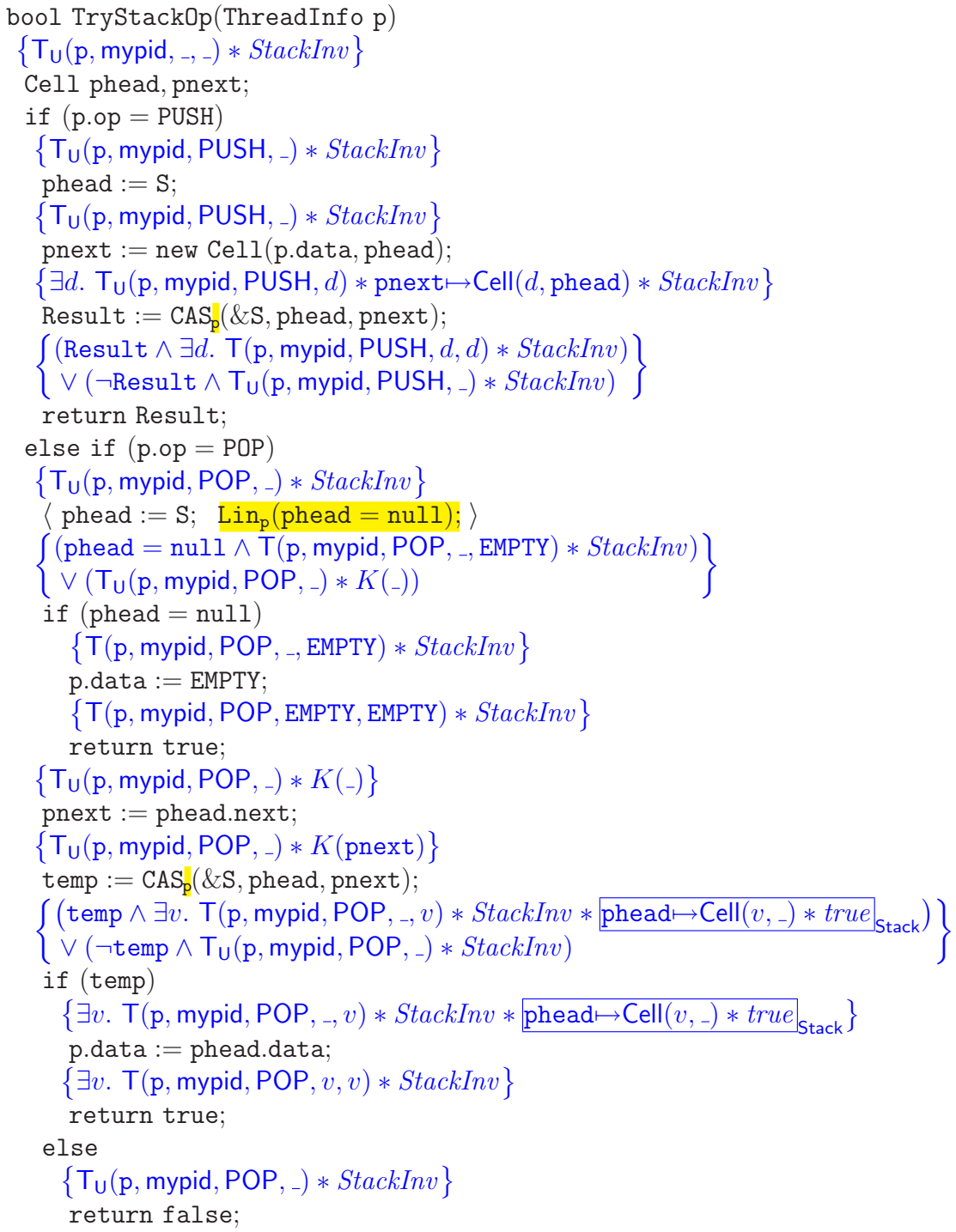

Figure 5.4: Proof outline of `TryStackOp`.

This specification says that if the stack invariant holds initially and `p` has not been linearised, then at the end, `p` will be linearised if and only if the function returns *true*. Moreover, if the function returns *true* then `p.data` will contain the return value of the abstract method call at the linearisation point.

Figure 5.4 shows the proof outline. The formula $K(y)$ is the same as in the simple stack proof and asserts that either the cell pointed to by `phead` is in the stack and the next cell is y, or it is not in the stack (but still in the shared state). This avoids the ABA problems discussed previously.

$$K(y) \stackrel{\text{def}}{=} \boxed{\begin{array}{l}\left(\begin{array}{l}\exists x\, v\, A\, B.\ \&\mathtt{Abs} \mapsto A{\cdot}v{\cdot}B * \&\mathtt{S} \mapsto x * \mathsf{lseg}(x, \mathtt{phead}, A)\\ * \mathtt{phead}{\mapsto}\mathsf{Cell}(v, y) * \mathsf{lseg}(y, \mathtt{null}, B) * \mathit{true}\end{array}\right)\\ \vee\ (\exists x\, A.\ \&\mathtt{Abs} \mapsto A * \&\mathtt{S} \mapsto x * \mathsf{lseg}(x, \mathtt{null}, A) * \mathtt{phead}{\mapsto}\mathsf{Cell}(\_, \_) * \mathit{true})\end{array}}_{\mathtt{Stack}}$$

Note that this proof does not mention the data structures of the elimination layer.

Elimination layer Similarly, the collision layer proof does not rely on the shared stack implementation, except for the existence of abstract stack object, `Abs`. It assumes that the stack implementation cannot deallocate `Abs`. This is a trivial proof obligation because only the auxiliary code can access `Abs`. It amounts to the assertion $\boxed{\mathtt{Abs} \mapsto \_ * \mathit{true}}_{\mathtt{Stack}}$ which is implied by *StackInv* and is trivially stable. This property can also be described by an 'existence' permission. In particular, while the elimination layer knows something about the `Stack` region, it does not modify it: its $G_{\mathtt{Stack}} = \emptyset$.

The proof uses fractional permissions in special manner. We always have half of a descriptor owned by the local state of a thread and half of it belonging to the shared state. When calling `StackOp`, we first put a $\mathsf{T_R}(\_, \_, \_)$ permission of the argument in the shared region `Elim`. This action initialises our logical splitting of the permissions.

$$\mathit{emp} \rightsquigarrow \mathsf{T_R}(p, \_, \_) \qquad \text{(Publish)}$$

Thereafter, all actions except for Publish respect this splitting and assign do not transfer full ownership to either party, although they may swap which halves belong to the shared state and which to the local state of a single thread.

A thread $t \in T$ effectively owns the location $\mathtt{loc}[t]$ and all descriptors whose thread identifier is t. It is allowed to place or remove a descriptor from $\mathtt{loc}[t]$ provided that the descriptor has been 'published.' This protocol ensures that if a thread ever reads a pointer to a descriptor record from the `loc` array, the record is not deallocated afterwards.

$$\mathtt{loc}[t] \mapsto \_ * \mathsf{T_R}(p, x, op) \rightsquigarrow \mathtt{loc}[t] \mapsto p * \mathsf{T}(p, x, op, \_, \_) \qquad \text{(Place)}$$

$$\mathtt{loc}[t] \mapsto p * \mathsf{T}(p, x, op, \_, \_) \rightsquigarrow \mathtt{loc}[t] \mapsto \mathsf{null} * \mathsf{T_R}(p, x, op) \qquad \text{(Remove)}$$

Moreover, as a result of an elimination, thread t can update a different entry, i, of the `loc` array by overwriting an unlinearised operation by its own descriptor and linearising both operations. If pushing a value, it must also make its descriptor available to the other thread so that the POP can return the value that was pushed.

$$\begin{array}{rl} & \texttt{loc}[i] \mapsto p * \mathsf{T_U}(p, i, \mathsf{PUSH}, v) \wedge i \neq t \\ \rightsquigarrow & \texttt{loc}[i] \mapsto \texttt{null} * \mathsf{T}(p, i, \mathsf{PUSH}, v, v) \qquad \text{(Elim1)} \\ & \texttt{loc}[i] \mapsto p * \mathsf{T_U}(p, i, \mathsf{POP}, x) * \mathsf{T_R}(q, t, \mathsf{PUSH}) \wedge i \neq t \\ \rightsquigarrow & \texttt{loc}[i] \mapsto q * \mathsf{T}(p, i, \mathsf{POP}, x, v) * \mathsf{T}(q, t, \mathsf{PUSH}, v, v) \qquad \text{(Elim2)} \end{array}$$

Finally, there is an action that allows a thread to remove its descriptor from the shared state when it notices that it has been eliminated by another thread:

$$\mathsf{T}(p, t, op, \_, \_) \rightsquigarrow \mathsf{T_R}(p, t, op) \quad \text{provided} \quad \circledast_{t \in T} (\exists v.\ \texttt{loc}[t] \mapsto v \wedge v \neq p) \qquad \text{(NoticeElim)}$$

The side-condition of this action requires that p does not occur in the location array. This ensures that the representation invariant (defined below) is stable under this action.

To define the representation invariant, we introduce the predicate $my\_loc(t)$ to describe the properties of a single entry t of the `loc` array. The representation invariant is just the separating conjunction of all entries in the `loc` array:

$$\begin{array}{rcl} my\_loc(t) & \stackrel{\text{def}}{\iff} & \texttt{loc}[t] \mapsto \mathsf{null} \\ & & \vee\ \exists q.\ \texttt{loc}[t] \mapsto q * \mathsf{T_U}(q, t, \_, \_) \\ & & \vee\ \exists q\, t'.\ \texttt{loc}[t] \mapsto q * \mathsf{T_R}(q, t', \mathsf{POP}) \wedge t \neq t' \\ & & \vee\ \exists q\, v\, t'.\ \texttt{loc}[t] \mapsto q * \mathsf{T}(q, t', \mathsf{PUSH}, v, v) \wedge t \neq t' \\ J & \stackrel{\text{def}}{\iff} & \circledast_{t \in T}\, my\_loc(t) * true \end{array}$$

Note that $my\_loc(t') * true$ (for all t') and J are stable under the actions defined so far.

Figures 5.5, 5.6 and 5.7 contain the proof outlines of `TryCollision`, `FinishCollision` and `StackOp` respectively. The proof outline of `StackOp` treats the code involving the `coll`[..] array as simply assigning a random value to the variable `him`.

The most interesting case in the proof are the two $\texttt{CAS}(\&\texttt{loc}[\mathsf{mypid}], \mathtt{p}, \texttt{null})$ operations in `StackOp`. If they succeed, they simply do the action Remove. If, however, the `CAS`es fail then we know that we are in the second or third disjunct of A. Hence, $\texttt{loc}[\mathsf{mypid}] \neq \mathtt{p}$ and also $\texttt{loc}[t] \neq \mathtt{p}$ for all $t \in T \setminus \{\mathsf{mypid}\}$ since $\mathsf{T_R}(\mathtt{p}, \ldots)$ and $\mathsf{T}(\mathtt{p}, \ldots)$ are $*$-conjoined with $my\_loc(t')$. Therefore, we can do a NoticeElim action and remove our descriptor from the shared state thereby getting C as a postcondition.

Finally, the collision array (`coll`) hardly affects the safety of the algorithm. It is trivial to reason about it: the resource invariant $\circledast_{i \in [1..\texttt{size}]} \texttt{coll}[i] \mapsto \_$ suffices.

Define $x \not\mapsto y = (\exists z.\ x \mapsto z \wedge z \neq y)$. Let $D = \boxed{\&\texttt{Abs} \mapsto \_ * true}_{\texttt{Stack}}$.
Frame out: $\boxed{\circledast_{t' \in T}\, my\_loc(t') * true}_{\texttt{Elim}}$. Fix t.

$\texttt{bool TryCollision(ThreadInfo p, q)}$
$\texttt{bool b;}$
$\left\{ \boxed{\begin{array}{l}(\texttt{loc}[t] \mapsto \texttt{q} * \mathsf{T_U}(\texttt{q}, t, op_2, \_) * true) \\ \vee\, (\texttt{loc}[t] \not\mapsto \texttt{q} * \mathsf{T_R}(\texttt{q}, t, op_2) * true)\end{array}}_{\texttt{Elim}} * D * \mathsf{T_U}(\texttt{p}, \mathsf{mypid}, op_1, \_) \wedge op_1 \neq op_2 \right\}$
$\texttt{if (p.op} = \texttt{PUSH) \{}$
$\left\{ \boxed{\begin{array}{l}(\texttt{loc}[t] \mapsto \texttt{q} * \mathsf{T_U}(\texttt{q}, t, \mathsf{POP}, \_) * true) \\ \vee\, (\texttt{loc}[t] \not\mapsto \texttt{q} * \mathsf{T_R}(\texttt{q}, t, \mathsf{POP}) * true)\end{array}}_{\texttt{Elim}} * D * \mathsf{T_U}(\texttt{p}, \mathsf{mypid}, \mathsf{PUSH}, \_) \right\}$
$\texttt{b} := \texttt{CAS}_{\texttt{p,q}}(\&\texttt{loc[q.id]}, \texttt{q}, \texttt{p});$
$\left\{ \begin{array}{l}\exists v.\ (b \wedge \boxed{\mathsf{T}(\texttt{p}, \mathsf{mypid}, \mathsf{PUSH}, v, v) * true}_{\texttt{Elim}} * \mathsf{T_R}(\texttt{p}, t, \mathsf{PUSH})) \\ \vee\, (\neg b \wedge \mathsf{T_U}(\texttt{p}, \mathsf{mypid}, \mathsf{PUSH}, \_))\end{array} \right\}$
$\texttt{\} else if (p.op} = \texttt{POP) \{}$
$\left\{ \boxed{\begin{array}{l}(\texttt{loc}[t] \mapsto \texttt{q} * \mathsf{T_U}(\texttt{q}, t, \mathsf{PUSH}, \_) * true) \\ \vee\, (\texttt{loc}[t] \not\mapsto \texttt{q} * \mathsf{T_R}(\texttt{q}, t, \mathsf{PUSH}) * true)\end{array}}_{\texttt{Elim}} * D * \mathsf{T_U}(\texttt{p}, \mathsf{mypid}, \mathsf{POP}, \_) \right\}$
$\texttt{b} := \texttt{CAS}_{\texttt{q,p}}(\&\texttt{loc[q.id]}, \texttt{q}, \texttt{null});$
$\left\{ \begin{array}{l}(b \wedge \exists v.\ \boxed{\mathsf{T}(\texttt{q}, t, \mathsf{PUSH}, v, v) * true}_{\texttt{Elim}} * \mathsf{T}(\texttt{p}, \mathsf{mypid}, \mathsf{POP}, \_, v)) \\ \vee\, (\neg b \wedge \mathsf{T_U}(\texttt{p}, \mathsf{mypid}, \mathsf{POP}, \_))\end{array} \right\}$
$\texttt{if(b)}\ \ \texttt{p.data} := \texttt{q.data;}$
$\{\exists v.\ (b \wedge \mathsf{T}(\texttt{p}, \mathsf{mypid}, \mathsf{POP}, v, v)) \vee (\neg b \wedge \mathsf{T_U}(\texttt{p}, \mathsf{mypid}, \mathsf{POP}, \_))\}$
$\texttt{\}}$
$\left\{ \begin{array}{l}(b \wedge \exists v.\ \boxed{\mathsf{T}(\texttt{p}, \mathsf{mypid}, \_, v, v) * true}_{\texttt{Elim}} * \mathsf{T_R}(\texttt{p}, t, \mathsf{PUSH})) \\ \vee\, (b \wedge \exists v.\ \mathsf{T}(\texttt{p}, \mathsf{mypid}, \_, v, v)) \\ \vee\, (\neg b \wedge \mathsf{T_U}(\texttt{p}, \mathsf{mypid}, \_, \_))\end{array} \right\}$
$\texttt{return b;}$

Figure 5.5: Proof outline of `TryCollision`.

$\texttt{void FinishCollision(ThreadInfo p)}$
$\left\{ \begin{array}{l}\exists v.\ \mathsf{T}(\texttt{p}, \mathsf{mypid}, \mathsf{PUSH}, v, v) * \boxed{\circledast_{t \in T}\, my\_loc(t) * true}_{\texttt{Elim}} \\ \vee\, \exists v\, q.\ \mathsf{T}(\texttt{p}, \mathsf{mypid}, \mathsf{POP}, \_, v) * \boxed{\begin{array}{l}\texttt{loc}[\mathsf{mypid}] \mapsto q * \mathsf{T}(q, \_, \mathsf{PUSH}, v, v) \\ * \circledast_{t \in T \setminus \{\mathsf{mypid}\}}\, my\_loc(t) * true\end{array}}_{\texttt{Elim}}\end{array} \right\}$
$\texttt{if (p.op} = \texttt{POP)}$
$\texttt{p.data} := \texttt{loc[mypid].data;}$
$\langle \texttt{loc[mypid]} := \texttt{null;} \rangle$
$\left\{ \exists v.\ \mathsf{T}(\texttt{p}, \mathsf{mypid}, \_, v, v) * \boxed{\circledast_{t \in T}\, my\_loc(t) * true}_{\texttt{Elim}} \right\}$

Figure 5.6: Proof outline of `FinishCollision`.

Let $A = \boxed{\begin{pmatrix} \texttt{loc[mypid]} \mapsto \mathsf{p} * \mathsf{T_U}(\mathsf{p}, \mathsf{mypid}, op_1, \_) \\ \lor (\exists q\, v.\ \texttt{loc[mypid]} \mapsto q * \mathsf{T}(q, \_, \mathsf{PUSH}, v, v) * \mathsf{T}(\mathsf{p}, \mathsf{mypid}, \mathsf{POP}, \_, v) \\ \lor (\exists q\, v.\ \texttt{loc[mypid]} \mapsto q * \mathsf{T_R}(q, \_, \mathsf{POP}) * \mathsf{T}(\mathsf{p}, \mathsf{mypid}, \mathsf{PUSH}, v, v) \end{pmatrix} * \circledast_{t \in (T \setminus \{\mathsf{mypid}\})} my\_loc(t) * true}_{\mathtt{Elim}}$

Let $B = \boxed{\texttt{loc[him]} \mapsto \mathsf{q} * \mathsf{T_U}(\mathsf{q}, \mathtt{him}, op_2, \_) * true \lor \exists q'.\ q \neq \mathsf{q} \land \texttt{loc[him]} \mapsto q' * \mathsf{T_R}(\mathsf{q}, \mathtt{him}, op_2) * true}_{\mathtt{Elim}}$

Let $C = \begin{pmatrix} \exists v\, q.\ \mathsf{T}(\mathsf{p}, \mathsf{mypid}, \mathsf{POP}, \_, v) * \boxed{\texttt{loc[mypid]} \mapsto q * \mathsf{T}(q, \_, \mathsf{PUSH}, v, v) * \circledast_{t \in (T \setminus \{\mathsf{mypid}\})} my\_loc(t) * true}_{\mathtt{Elim}} \\ \lor \exists v.\ \mathsf{T}(\mathsf{p}, \mathsf{mypid}, \mathsf{PUSH}, v, v) * \boxed{\circledast_{t \in T} my\_loc(t) * true}_{\mathtt{Elim}} \end{pmatrix}$

```
void StackOp(ThreadInfo p) {  int him, pos;
 while (true){
  {T_U(p, mypid, op1, _) * [T_R(p, mypid, op1) * J]_Elim * StackInv}
  if (TryStackOp(p))
    {∃v. T(p, mypid, op1, v, v) * [T_R(p, mypid, op1) * J]_Elim * StackInv}
    return;
  {T_U(p, mypid, op1, _) * [T_R(p, mypid, op1) * J]_Elim * StackInv}
  {T_U(p, mypid, op1, _) * [T_R(p, mypid, op1) * J]_Elim * D}
  ⟨loc[mypid] := p;⟩
  {T_R(p, mypid, op1) * A * D}
  him := nondet();
  if (1 ≤ him ≤ threadNum) {
   ThreadInfo q := loc[him];
   if (q≠null ∧ q.id=him ∧ q.op≠p.op)
     {op1 ≠ op2 ∧ T_R(p, mypid, op1) * A * B * D}
     if (CAS(&loc[mypid], p, null))
      { op1 ≠ op2 ∧ T_U(p, mypid, op1, _) * B * D
        * [T_R(p, mypid, op1) * J]_Elim }
      if (TryCollision(p, q))
       { ∃v. T_R(p, mypid, op1)
             * [T(p, mypid, op1, v, v) * true ∧ J]_Elim
         ∨ T(p, mypid, op1, v, v) * [J]_Elim }
       return;
      else {T_U(p, mypid, _, _) * [T_R(p, mypid, op1) * J]_Elim}
       continue;
     else {StackInv * C}
      FinishCollision(p); return;
  }
  {T_R(p, mypid, op1) * A}
  delay();
  if (¬CAS(&loc[mypid], p, null)) {C}
   FinishCollision(p); return;
  {T_U(p, mypid, _, _) * [J]_Elim}
} }
```

Let $D = \boxed{Abs \mapsto \_ * true}_{\mathtt{Stack}}$

$G_{\mathtt{Stack}} = \emptyset$

Frame out $StackInv$ which is stable under $R_{\mathtt{Stack}} \cup G_{\mathtt{Stack}}$.

Figure 5.7: Proof outline of `StackOp`.

```
locate(e) {
  local p, c;
  p := Head;
  lock(p);
  c := p.next;
  while (c.value < e) {
    lock(c);
    unlock(p);
    p := c;
    c := p.next;
    lock(c);
  }
  return(p, c);
}
```

```
add(e) {
  local x, y, z;
  (x, z) := locate(e);
  if (z.value ≠ e) {
    y := new Node();
    y.lock := 0;
    y.value := e;
    y.next := z;
    x.next := y;
  }
  Lin_this;
  unlock(x);
}
```

```
remove(e) {
  local x, y, z;
  (x, y) := locate(e);
  if (y.value = e) {
    lock(y);
    z := y.next;
    x.next := z; Lin_this;
    unlock(x);
    dispose(y);
  } else { Lin_this;
    unlock(x);
  }
}
```

Figure 5.8: Lock-coupling list algorithm

5.3.2 List-based set algorithms

This collection of algorithms represent an integer set as a sorted linked list with two sentinel nodes at the two ends of the list, containing the values $-\infty$ and $+\infty$ respectively. The set is equipped with a lookup operation `contains` and two destructive operations `add` and `remove` that change the set's cardinality by one. Their specifications are given by the following abstract code.

```
Abs_contains(e) ≝ ⟨
  AbsResult := (e ∈ Abs);
⟩
```

```
Abs_add(e) ≝ ⟨
  AbsResult := (e ∉ Abs);
  Abs := Abs ∪ {e};
⟩
```

```
Abs_remove(e) ≝ ⟨
  AbsResult := (e ∈ Abs);
  Abs := Abs \ {e};
⟩
```

Pessimistic list (Lock-coupling list)

The first algorithm (see Figure 5.8) is pessimistic in its concurrency management: it always locks a node before accessing it. `locate` traverses the list using *lock coupling*: the lock on some node is not released until the next one is locked, somewhat like a person climbing a rope "hand-over-hand." The methods `add` and `remove` call `locate` to traverse the list and lock the appropriate nodes; then they update the data structure locally.

In §3.5, we proved that this algorithm is safe, does not leak memory, and maintains the structure of a sorted list. Here, we will prove that it is linearisable. To achieve this, we embed the abstract implementations `Abs_Add` and `Abs_Remove` at the candidate linearisation points and derive the post-condition `Result` = `AbsResult` for `add` and `remove`.

The proof is a direct adaptation of the proof in §3.5. First, introduce an auxiliary variable `Abs` to hold the abstract set that the linked list represents. When a node is

```
locate(e) {
  local p, c, s;
  while (true) {
    p := Head;
    ⟨c := p.next;⟩
    while (c.value < e) {
      p := c;
      ⟨c := c.next;⟩
    }
    lock(p);
    s := Head;
    while (s.value < e)
      ⟨s := s.next;⟩
    if (s = c ∧ p.next = c)
      return (p, c);
    else
      unlock(p);
} }
```

```
add(e) {
  local x, y, z;
  (x, z) := locate(e);
  if (z.value ≠ e) {
    y := new Node();
    y.lock := 0;
    y.value := e;
    y.next := z;
    ⟨x.next := y;
     Lin_this;⟩
  } else {
    Lin_this;
  }
  unlock(x);
}
```

```
remove(e) {
  local x, y, z;
  (x, y) := locate(e);
  if (y.value = e) {
    lock(y);
    ⟨z := y.next;⟩
    ⟨x.next := z;
     Lin_this;⟩
    unlock(y);
  } else {
    Lin_this;
  }
  unlock(x);
}
```

Figure 5.9: Optimistic list-based set implementation

inserted or removed from the list, this is a linearisation point and the (instrumented) algorithm also updates `Abs`. Thread *tid* is allowed to perform the following actions:

$$U(x,v,n) \rightsquigarrow L_{tid}(x,v,n) \qquad \text{(Lock)}$$

$$L_{tid}(x,v,n) \rightsquigarrow U(x,v,n) \qquad \text{(Unlock)}$$

$$L_{tid}(x,u,n) * \&\texttt{Abs} \mapsto A \rightsquigarrow L_{tid}(x,u,m) * U(m,v,n) * \&\texttt{Abs} \mapsto A \cup \{v\}$$
$$\text{provided } (u < v < w) \wedge N_s(n,w,y) \qquad \text{(Add)}$$

$$L_{tid}(x,u,n) * L_{tid}(n,v,m) * \&\texttt{Abs} \mapsto A \rightsquigarrow L_{tid}(x,u,m) * \&\texttt{Abs} \mapsto A \setminus \{v\}$$
$$\text{provided } v < +\infty \qquad \text{(Remove)}$$

The assertions in the proof outlines are exactly the same as in §3.5, provided we redefine the predicate $s(A)$ to also require that `Abs` contains the mapping of the sequence A to a set:

$$s(A) \stackrel{\text{def}}{=} \exists B.\ A = -\infty \cdot B \cdot +\infty \wedge \mathit{sorted}(A) \wedge \&\texttt{Abs} \mapsto \mathit{elems}(B)$$

Optimistic list

Now consider the algorithm in Figure 5.9, which implements `locate` differently. The new implementation is optimistic: it traverses the list without taking any locks, then locks two candidate nodes, and re-traverses the list to check whether the nodes are still present in the list and adjacent. If either test fails, the nodes are unlocked and the algorithm is

restarted.

While one thread has locked part of the list and is updating it, another thread may optimistically traverse it. The success of the optimistic traversal clearly depends on some properties of locked nodes (e.g., that they point to valid next nodes). In particular, updating the `next` field of a shared node and reading the `next` field of an unlocked node must be both atomic. Otherwise, because of a possible race condition, the read could return a corrupted value.

The optimistic algorithm has the same linearisation points as the lock-coupling algorithm. The set of permitted actions is similar: we can lock, unlock, add, and remove a node. The actions Lock, Unlock, and Add are the same as for the lock coupling list. The action Remove is, however, different. Whereas in lock-coupling the node n was removed from the shared state, here it remains in the shared state, because it may be accessed by concurrent optimistic list traversals.

$$U(x, v, n) \rightsquigarrow L_{tid}(x, v, n) \qquad \text{(Lock)}$$

$$L_{tid}(x, v, n) \rightsquigarrow U(x, v, n) \qquad \text{(Unlock)}$$

$$L_{tid}(x, u, n) * \&\texttt{Abs} \mapsto A \rightsquigarrow L_{tid}(x, u, m) * U(m, v, n) * \&\texttt{Abs} \mapsto A \cup \{v\}$$
$$\text{provided } (u < v < w) \wedge N_s(n, w, y) \qquad \text{(Add)}$$

$$L_{tid}(x, u, n) * L_{tid}(n, v, m) * \&\texttt{Abs} \mapsto A \rightsquigarrow L_{tid}(x, u, m) * L_{tid}(n, v, m) * \&\texttt{Abs} \mapsto A \setminus \{v\}$$
$$\text{provided } v < +\infty \qquad \text{(Remove)}$$

Lazy list (lazy concurrent list-based set implementation)

The final list algorithm we will consider is due to Heller et al. [38], and combines optimistic and lazy concurrency techniques. Under common work loads, their algorithm is more efficient than the previous versions, because checking for membership in the set avoids locking. The concrete representation is the same as the one used by the earlier algorithms. In addition, however, nodes have a `marked` field, which is set when the node is deleted.

An element is added as before. An element is removed in two stages: first, the node is *logically* removed by setting the `marked` field; then it is *physically* removed by redirecting reference fields. Concurrent membership tests traverse the list without checking the `marked` flag. This flag is checked only when a candidate node is found. Similarly, `locate` ignores the flag while traversing the list. When the method locates and locks the two candidate nodes, it *validates* them by checking they are adjacent and unmarked. If validation fails, the `locate` operation is restarted.

Because `contains` is completely wait-free, this algorithm crucially depends on global invariants, such as the list being sorted, which *must hold at all times*, even when part of the list is locked and local updates are performed. RGSep is good at describing these.

```
locate(e) :
  while (true) {
    pred := Head;
    ⟨curr := pred.next;⟩
    while (curr.val < e) {
      pred := curr;
      ⟨curr := curr.next;⟩
    }
    lock(pred);
    lock(curr);
    if (¬⟨pred.marked⟩
        ∧ ¬⟨curr.marked⟩
        ∧ ⟨pred.next⟩ = curr)
      return (pred, curr);
    else {
      unlock(pred);
      unlock(curr); }
  }

contains(e) :
  ⟨Lin_this(e ∉ Abs);⟩
  ⟨OutOps := OutOps ∪ {this};⟩
  curr := Head;
  while (curr.val < e)
    ⟨curr := curr.next;⟩
  ⟨b := curr.marked; Lin_this(¬b);⟩
  ⟨OutOps := OutOps \ {this};⟩
  if (b)
    return false;
  else
    return curr.val = e;
```

```
add(e) :
  (n1, n3) := locate(e);
  if (n3.val ≠ e) {
    n2 := new Node(e);
    n2.next := n3;
    ⟨n1.next := n2; Lin_this;⟩
    Result := true;
  } else {
    Result := false; Lin_this;
  }
  unlock(n1);
  unlock(n3);
  return Result;

remove(e) :
  (n1, n2) := locate(e);
  if (n2.val = e) {
    ⟨n2.marked := true; Lin_this;
     foreach(p ∈ OutOps)
       Lin_p(p.arg_1 = e); ⟩
    ⟨n3 := n2.next;⟩
    ⟨n1.next := n3;⟩
    Result := true
  } else {
    Result := false; Lin_this;
  }
  unlock(n1);
  unlock(n2);
  return Result;
```

Figure 5.10: Lazy concurrent list-based set implementation

Linearisation points The linearisation points of `add` and `remove` are annotated in the source code. Just note that a successful `remove` is linearised at the point where the node is logically deleted, not when it is physically removed from the list. Physically removing the node from the list does not change the abstract set the list represents; indeed, it can be seen as an optimisation for improving future list accesses.

The linearisation point of `contains` is much subtler. If `contains` returns *true*, the last assignment to `curr` in the loop and the read of the `marked` bit are both valid linearisation points. If, however, `contains` returns *false*, the proof is more subtle. Here is an informal argument which will motivate the formal proof.

Initially, when `contains(e)` starts executing, either `e` is in the list or it is not. If `e`

is not in the list, then clearly the linearisation point can be identified to be that initial point. If `e` was in the list, then there exists a (unique) unmarked node m that is reachable from `Head` and contains the value `e`.

If the node m never gets marked while `contains(e)` executes, `contains(e)` will find that node and return `true`. So, by contradiction, the node m must have been marked (and perhaps even physically removed) during the execution of `contains(e)`.

This, however, does not mean that e will not be in the list at the next scheduling of `contains(e)`; indeed, another node containing e could have been added in the meantime. But since marking a node and inserting a new node cannot be done atomically, there is a point (just after the node was marked) when e was not in the list. Take that point to be the linearisation point.

To capture this point in a formal proof, we introduce an auxiliary record for each outstanding `contains` operation, and a global auxiliary variable `OutOps` containing the set of all outstanding `contains` records. Each record contains three fields: the thread identifier of the method invoking `contains`, the argument `e` of `contains`, and an initially unset field `AbsResult` for holding the result of the linearisable function. As `contains` is a read-only operation, it is sufficient that a linearisation point exists; it does not need to be unique. The freedom to linearise `contains` multiple times simplifies the auxiliary code significantly.

Node predicates The nodes of this list algorithm consist of four fields `lock`, `value`, `next`, and `marked`. We use the following shorthand notation to describe nodes:

$$\begin{aligned}
N_s(x,v,n) &\stackrel{\text{def}}{=} x \mapsto \{.\texttt{lock} = s; .\texttt{value} = v; .\texttt{next} = n; .\texttt{marked} = _\} \\
M_s(x,v,n) &\stackrel{\text{def}}{=} x \mapsto \{.\texttt{lock} = s; .\texttt{value} = v; .\texttt{next} = n; .\texttt{marked} = \mathit{true}\} \\
U(x,v,n) &\stackrel{\text{def}}{=} N_0(x,v,n) \\
L_{tid}(x,v,n) &\stackrel{\text{def}}{=} N_{tid}(x,v,n) \wedge tid \neq 0
\end{aligned}$$

Actions The actions are slightly more involved than for the previous algorithms, because they also specify the operations affecting the auxiliary state. The actions for locking a node, releasing a lock, and adding a node are standard:

$$U(x,v,n) \rightsquigarrow L_{tid}(x,v,n) \qquad \text{(Lock)}$$

$$L_{tid}(x,v,n) \rightsquigarrow U(x,v,n) \qquad \text{(Unlock)}$$

$$\begin{aligned}
L_{tid}(x,u,n) * \texttt{Abs} \mapsto A \rightsquigarrow\ & L_{tid}(x,u,m) * U(m,v,n) * \texttt{Abs} \mapsto A \cup \{v\} \\
& \text{provided } (u < v < w) \wedge N_s(n,w,y) \qquad \text{(Add)}
\end{aligned}$$

Physically removing a node from the list requires the node to be marked; moreover, it does not affect the auxiliary state.

$$L_{tid}(x,u,n) * M_{tid}(n,v,m) \rightsquigarrow L_{tid}(x,u,m) * M_{tid}(n,v,m) \quad \text{provided } v < +\infty \qquad \text{(Remove)}$$

Marking a node is a more complex action, because it does several tasks at once: (*i*) it marks node x; (*ii*) it removes v from the abstract set; and (*iii*) it linearises all outstanding $\texttt{contains}(v)$ calls.

$$\begin{pmatrix} L_{tid}(x,v,n) * \texttt{Abs} \mapsto A \\ * \circledast_{c \in C} \mathcal{R}_s(c,v,\_) \end{pmatrix} \rightsquigarrow \begin{pmatrix} M_{tid}(x,v,n) * \texttt{Abs} \mapsto A \setminus \{v\} \\ * \circledast_{c \in C} \mathcal{R}_s(c,v,\mathit{false}) \end{pmatrix} \quad \text{provided } \begin{pmatrix} \&\texttt{OutOps} \mapsto B \cup C \\ * \circledast_{b \in B} \exists w.\ \mathcal{R}_\_(b,w,\_) \wedge w \neq v \\ \wedge -\infty < v < +\infty \end{pmatrix} \qquad \text{(Mark)}$$

where $\mathcal{R}_s(c,v,w)$ represents a record for $\texttt{contains}(v)$ that currently has its `AbsResult` set to w ($w \in \{\mathit{undef}, \mathit{false}, \mathit{true}\}$):

$$\mathcal{R}_s(c,v,w) \stackrel{\text{def}}{=} c \mapsto \mathsf{Record}\{.\texttt{threadid} = s, .\texttt{val} = v, .\texttt{AbsResult} = w\}$$

Finally, there is a symmetric action enabling a thread to add and remove an outstanding record of a contains(e) it initiated.

$$\begin{aligned} (c \notin B) \wedge (\&\texttt{OutOps} \mapsto B) &\rightsquigarrow (\&\texttt{OutOps} \mapsto B \cup \{c\}) * \mathcal{R}_{tid}(c,v,w)) \\ (\&\texttt{OutOps} \mapsto B \cup \{c\}) * \mathcal{R}_{tid}(c,v,w)) &\rightsquigarrow (c \notin B) \wedge (\&\texttt{OutOps} \mapsto B) \end{aligned}$$

5.3.3 Restricted double-compare single-swap (RDCSS)

Restricted double-compare single-swap (`RDCSS`, see Figure 5.11) is an intermediate abstract operation defined by Harris et al. [34] in their implementation of multiple compare-and-swap (MCAS). `RDCSS` takes as an argument two addresses with their expected values and one new value. If both addresses contain their expected values, then the new value $\texttt{n}_2$ is stored at the second address $\texttt{a}_2$. The specification of `RDCSS` (`RDCSS_spec`) requires that this entire operation is atomic. It is restricted in the sense that the address space is logically split in two disjoint domains, A and B. The first address ($\texttt{a}_1$) must belong to A, whereas the second address ($\texttt{a}_2$) to B. Hence, any address passed as an $\texttt{a}_1$ cannot be passed as an $\texttt{a}_2$ in a later call to `RDCSS` and vice versa.

On A-type addresses, all the standard memory operations are permitted; whereas on B-type addressed, only two operations are permitted: `RDCSS_read` and `RDCSS`. We can relax

```
class Descriptor {
  address_t      a1, a2;
  word_t         o1, o2, n2;
  single word_t AbsResult;
  single word_t r2;  }

RDCSS_spec(Descriptor d) = ⟨
  local r := [d.a2];
  if ([d.a1] = d.o1 ∧ [a.a2] = d.o2)
       [d.a2] := d.n2;
  result r ⟩

Complete(Descriptor d) {
       local r;
C1 :   ⟨r := [d.a1];⟩
       if (r = d.o1)
C2 :     CAS1(d.a2, d, d.n2);
       else
C3 :     CAS1(d.a2, d, d.o2);
}
```

```
RDCSS_read(address_t a2) {
       local r;
R1 :   ⟨r := [a2];⟩
       while(IsDesc(r)) {
         Complete(r);
R2 :     ⟨r := [a2];⟩
       }
       return r;
}

RDCSS(Descriptor d) {
       local r;
A1 :   r := CAS1(d.a2, d.o2, d);
       while (IsDesc(r)) {
         Complete(r);
A2 :     r := CAS1(d.a2, d.o2, d);
       }
       if (r = d.o2)  Complete(d);
       return r;
}
```

Figure 5.11: RDCSS implementation

this restriction by allowing standard atomic memory writes to B-type addresses provided they do not interfere with the RDCSS implementation. These additional operations, actually used in MCAS, are formalised in the action WriteB.

Each RDCSS operation has a unique descriptor: this is a record in memory containing the parameters of the method call (a_1, o_1, a_2, o_2, n_2). For the sake of the proof, we add two auxiliary fields (AbsResult and r_2) to these descriptors.

The implementation of RDCSS uses a variant of CAS, which instead of a boolean indicating whether it succeeded, it returns the (old) value stored in the memory address:

```
value_t CAS1(value_t *addr, value_t exp, value_t new) {
  value_t v;
  atomic { v = *addr;
           if (v == exp) *addr = new; }
  return v;
}
```

Each RDCSS is performed in two steps: first, it places its descriptor at the memory address a_2. This essentially 'locks' that location. Any thread that reads a memory location and finds it contains a descriptor must either wait until the RDCSS operation completes or complete it itself by calling the helper function Complete. RDCSS_read reads location a, after committing any conflicting outstanding RDCSS operations.

The function call $\texttt{IsDesc}(\texttt{r})$ checks whether $\texttt{r}$ is a pointer to a descriptor or a normal value. The implementation distinguishes between the two by reserving one bit in the representation of values. Hence, the implementation $\texttt{IsDesc}(\texttt{r})$ can just check whether the reserved bit is set.

Linearisation points The linearisation point of RDCSS is among the lines A1, A2, and C1. If the CASes at A1 or A2 encounter a different value for $[\texttt{d.a}_2]$ that is not a descriptor, then these are valid linearisation points because RDCSS fails at that point, returning $\texttt{r}$. Therefore, we annotate A1 and A2 as follows:

$$\langle \texttt{r} := \texttt{CAS1}(\texttt{d.a}_2, \texttt{d.o}_2, \texttt{d});\ \texttt{Lin}_{\texttt{d}}(\texttt{r} \neq \texttt{d.o}_2 \wedge \neg\texttt{IsDesc}(\texttt{r}));\rangle$$

If, however, the descriptor $\texttt{d}$ gets placed in $[\texttt{d.a}_2]$, then the linearisation point is within Complete. In fact it is line C1 provided that the CAS on line C2 or C3 succeeds. Since with 'helping', any thread can call Complete, the linearisation point is the line C1 of the $\texttt{Complete}(\texttt{d})$ that succeeds in its CAS at C2 or C3. This linearisation point depends on future behaviour (the success of a CAS) that is not known at the linearisation point. To capture this formally, the proof will introduce a prophecy variable. Note that C2 and/or C3 are not valid linearisation points because the value of $[\texttt{d.a}_1]$ could have changed since last read at C1.

The linearisation point of RDCSS_read is the last R1 or R2 line executed. The proof annotates these lines as follows:

$$\langle \texttt{r} := [\texttt{a}_2];\ \texttt{Lin}_{\texttt{this}}(\neg\texttt{IsDesc}(\texttt{r}));\rangle$$

Descriptors The predicate $\mathsf{D}_d(a_1, a_2, o_1, o_2, n_2, r_2)$ describes a valid RDCSS descriptor stored at address d.

$$\begin{aligned}&\mathsf{D}_d(a_1, a_2, o_1, o_2, n_2, a, r_2) \stackrel{\text{def}}{=}\\ &\quad d \mapsto \{.\texttt{a}_1{=}a_1;\ .\texttt{o}_1{=}o_1;\ .\texttt{a}_2{=}a_2;\ .\texttt{o}_2{=}o_2;\ .\texttt{n}_2{=}n_2;\ .\texttt{AbsResult}{=}a;\ .\texttt{r}_2{=}r_2\}\\ &\quad \wedge (a_1 \in A) \wedge (a_2 \in B) \wedge \texttt{IsDesc}(d) \wedge \neg\texttt{IsDesc}(o_2) \wedge \neg\texttt{IsDesc}(n_2)\end{aligned}$$

It says that d must be an address of type 'descriptor', that a_1 must be a type A address and that a_2 must be a type B address. Moreover, the values o_2 and n_2 must be normal values (and not descriptors).

When the AbsResult field is undefined, $\texttt{r}_2$ is also undefined. The following shorthand notation describes such records:

$$\mathsf{U}_d(a_1, a_2, o_1, o_2, n_2) \stackrel{\text{def}}{=} \mathsf{D}_d(a_1, a_2, o_1, o_2, n_2, \mathit{undef}, \mathit{undef})$$

Abstraction map Nodes of type A store normal values; their abstract value is simply the value stored at the node. Nodes of type B, however, can store either a normal value or a pointer to a descriptor. In the latter case, their abstract value depends on whether the described operation has 'committed' or not. If it has 'committed', that is if it has past its linearisation point, then the abstract value is the one stored in its $\mathtt{r_2}$ field. Otherwise, if it is before the linearisation point, the abstract value is still the recorded old value, $\mathtt{o_2}$.

Let $K(x)$ be the following assertion mapping the concrete value of location x to its abstract value.

$$K(x) \stackrel{\text{def}}{=} (x \in B) \wedge \exists d\, v\, w.\ \begin{pmatrix} Abs[x] \mapsto v * x \mapsto v \wedge \neg \mathtt{IsDesc}(v) \\ \vee\ Abs[x] \mapsto v * x \mapsto d * \mathsf{U}_d(\_, x, \_, v, \_) \\ \vee\ Abs[x] \mapsto w * x \mapsto d * \mathsf{D}_d(\_, x, \_, v, \_, v, w) \end{pmatrix}$$

The overall invariant, *RDCSS_Inv*, asserts that (*i*) all the locations in A exist, (*ii*) every location in B has a matching concrete and abstract values, and (*iii*) there may be some more garbage state (completed RDCSS descriptors).

$$\mathit{RDCSS\_Inv} \stackrel{\text{def}}{=} \exists D.\ \circledast_{x \in A}\, x \mapsto \_ * \circledast_{x \in B}\, K(x) * \circledast_{x \in D}\, \exists o_2.\ \mathsf{D}_d(\_, \_, \_, o_2, \_, o_2, \_)$$

Actions The first two actions describe what code outside the RDCSS module is allowed to do. It can write any value to A-type cells, and it can update B-type cells atomically provided that both the old and the new value are not RDCSS descriptors.

$$(x \in A) \wedge x \mapsto v \rightsquigarrow x \mapsto w \qquad (\mathsf{WriteA})$$

$$(x \in B) \wedge \neg \mathtt{IsDesc}(v) \wedge x \mapsto v \rightsquigarrow x \mapsto w \wedge \neg \mathtt{IsDesc}(w) \qquad (\mathsf{WriteB})$$

These actions together with the ones below form the rely condition. They need not be included in the guarantee because the algorithm itself does not perform either of the actions above. Nevertheless, the proof goes through whether we include them in the guarantee or not.

The next three actions describe what the algorithm itself does and correspond very closely to the code. The first action allows a descriptor to be placed at a memory cell provided it has a matching old value (cf. lines A1 and A2). The second action allows a descriptor to be removed if its linearisation point has been passed (cf. lines C2 and C3). The final action involves only changes to the abstract state and happens at the

$\{\boxed{RDCSS\_Inv} * \mathsf{U_d}(a_1, a_2, o_1, o_2, n_2)\}$
local r;
$\langle \mathtt{r} := \mathtt{CAS1}(\mathtt{d.a_2}, \mathtt{d.o_2}, \mathtt{d});\ \mathsf{Lin_d}(\mathtt{r} \neq \mathtt{d.o_2} \wedge \neg \mathtt{IsDesc(r)}); \rangle$
$\left\{ \begin{array}{l} (\mathtt{r} = o_2 \wedge \boxed{RDCSS\_Inv} \wedge \mathsf{D_d}(a_1, a_2, o_1, o_2, n_2, \_, \_) * true) \\ \vee\ (\boxed{RDCSS\_Inv} \wedge \mathsf{D_r}(\_, a_2, \_, \_, \_, \_, \_) * true * \mathsf{U_d}(a_1, a_2, o_1, o_2, n_2)) \\ \vee\ (\neg \mathtt{IsDesc(r)} \wedge \mathtt{r} \neq o_2 \wedge \boxed{RDCSS\_Inv} * \mathsf{D_d}(a_1, a_2, o_1, o_2, n_2, \mathtt{r}, undef)) \end{array} \right\}$
while (IsDesc(r)) {
  $\{\boxed{RDCSS\_Inv} \wedge \mathsf{D_r}(\_, a_2, \_, \_, \_, \_, \_) * true * \mathsf{U_d}(a_1, a_2, o_1, o_2, n_2)\}$
  Complete(r);
  $\{\boxed{RDCSS\_Inv} * \mathsf{U_d}(a_1, a_2, o_1, o_2, n_2)\}$
  $\langle \mathtt{r} := \mathtt{CAS1}(\mathtt{d.a_2}, \mathtt{d.o_2}, \mathtt{d});\ \mathsf{Lin_d}(\mathtt{r} \neq \mathtt{d.o_2} \wedge \neg \mathtt{IsDesc(r)}); \rangle$
}
$\left\{ \begin{array}{l} (\mathtt{r} = o_2 \wedge \boxed{RDCSS\_Inv} \wedge \mathsf{D_d}(a_1, a_2, o_1, o_2, n_2, \_, \_) * true) \\ \vee\ (\neg \mathtt{IsDesc(r)} \wedge \mathtt{r} \neq o_2 \wedge \boxed{RDCSS\_Inv} * \mathsf{D_d}(a_1, a_2, o_1, o_2, n_2, \mathtt{r}, undef)) \end{array} \right\}$
if (r = d.o2) {
  $\{(\mathtt{r} = o_2 \wedge \boxed{RDCSS\_Inv} \wedge \mathsf{D_d}(a_1, a_2, o_1, o_2, n_2, \_, \_) * true)\}$
  Complete(d);
  $\{(\mathtt{r} = o_2 \wedge \boxed{RDCSS\_Inv} \wedge \mathsf{D_d}(a_1, a_2, o_1, o_2, n_2, o_2, \_) * true)\}$
}
$\left\{ \begin{array}{l} \boxed{RDCSS\_Inv} * \mathsf{D_d}(a_1, a_2, o_1, o_2, n_2, \mathtt{r}, \_) \\ \vee\ \boxed{RDCSS\_Inv} \wedge \mathsf{D_d}(a_1, a_2, o_1, o_2, n_2, \mathtt{r}, \_) * true \end{array} \right\}$
return r;

Figure 5.12: `RDCSS(d)` proof outline

linearisation point of `RDCSS` (i.e. on *some* executions of line C1).

$$x \mapsto v \rightsquigarrow x \mapsto d * \mathsf{U}_d(a_1, x, o_1, v, n_1) \qquad \text{(PlaceD)}$$

$$x \mapsto d * \mathsf{D}_d(a_1, x, o_1, o_2, n_2, a, y) \rightsquigarrow x \mapsto y * \mathsf{D}_d(a_1, x, o_1, o_2, n_2, a, y) \qquad \text{(RemoveD)}$$

$$Abs[a_2] \mapsto o_2 * \mathsf{U}_d(a_1, o_1, a_2, o_2, n_2) \rightsquigarrow Abs[a_2] \mapsto x * \mathsf{D}_d(a_1, o_1, a_2, o_2, n_2, o_2, x) \qquad \text{(Lin)}$$

Note that all these actions preserve the abstraction map, *RDCSS_Inv*. We exploit the symmetry of algorithm by having identical rely and guarantee conditions.

Proof outline The proof outlines of `RDCSS` and `RDCSS_read` are relatively straightforward. In the proof of the atomic sections in lines A1, A2, R1, and R2 we do a case split depending on the value stored in $\mathtt{a_2}$.

In contrast, verifying `Complete` is much more involved (see proof outline in Figure 5.13). The complexity arises because we have to specify the linearisation point, which depends on future behaviour. To do so, we introduce a *boolean prophecy variable* `futSucc`. We assume there is an oracle that tells us whether the value of $[\mathtt{d.a_2}]$ at the time of the appropriate CAS below will be `d` or not, and hence whether the CAS will succeed or

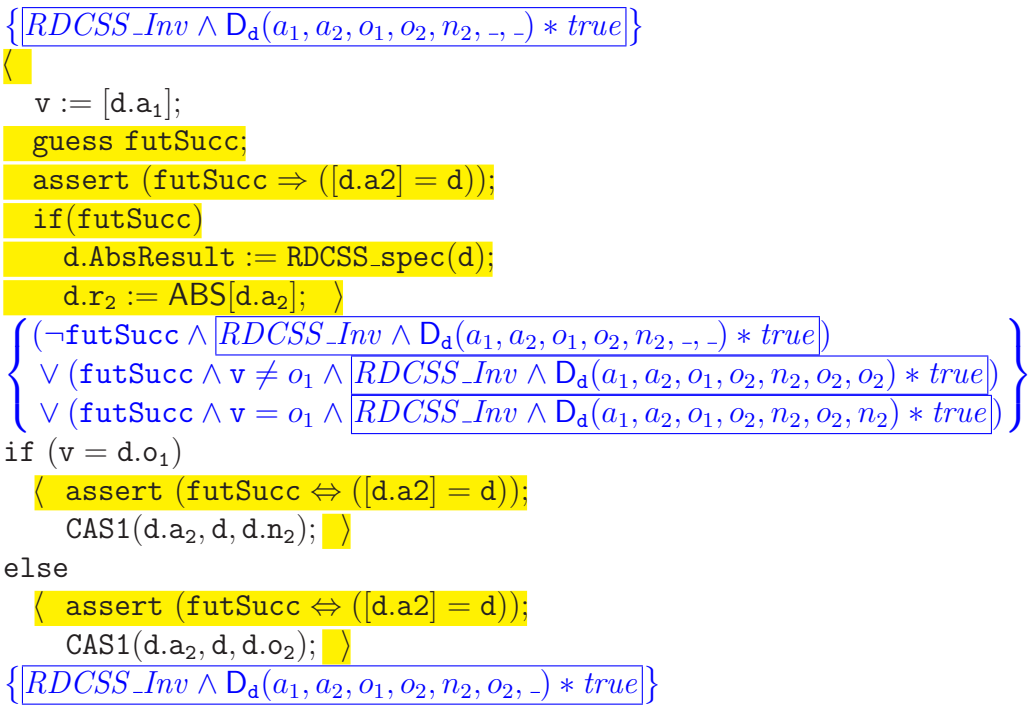

Figure 5.13: `Complete(d)` proof outline

not. Abadi and Lamport [1] proved that assuming the existence of such an oracle is sound under some finiteness conditions, which are trivially satisfied by this boolean prophecy variable. The 'guessed' value of the prophecy variable is such that the two introduced `assert` commands immediately before the two compare-and-swap instructions succeed.

As RGSep does not have a modality to express properties about the future, the proof requires an additional `assert` command just after the `guess` statement. We can justify that this `assert` succeeds informally with a short proof by contradiction. Assume the `assert` at C1 does not hold. Then, `futSucc` is true and $[\mathtt{d.a_2}] \neq \mathtt{d}$ at C1. Hence, $[\mathtt{d.a2}] \neq \mathtt{d}$ at C2/C3 because $[\mathtt{d.a_2}] \neq \mathtt{d}$ is stable under the rely condition. Therefore, as the asserts at C2/C3 hold, `futSucc` is false which contradicts our assumption.

Now that we have established that all the `assert` statements are satisfied, we can convert them to `assume` statements, and the proof is straightforward.

5.3.4 Multiple compare-and-swap (MCAS)

Figure 5.14 contains the implementation of multiple compare-and-swap (`MCAS`) due to Harris et al. [34] which has been annotated with the linearisation points. Each MCAS operation uses a fresh descriptor record (`cd`) with its arguments, and a status flag which is initially `UNDECIDED`. The algorithm progresses by placing the descriptor in all the memory locations $\mathtt{a_i}$ that contain the expected value $\mathtt{o_i}$. If one of these contains a different value, then `MCAS` fails, and the status is changed to `FAILED`. If we encounter the descriptor of

```
class CASNDescriptor {
 enum {UNDECIDED, SUCCEEDED, FAILED}  status;
 int k;
 address_t a_1, ..., a_k;
 word_t o_1, ..., o_k;
 word_t n_1, ..., n_k;
 single bool AbsResult;
}

MCAS_spec(CASNDescriptor cd) ≝ ⟨
  if ([cd.a_1] = cd.o_1 ∧ ... ∧ [cd.a_cd.k] = cd.o_cd.k])
    [ca.a_1] := cd.n_1; ...;
    [ca.a_cd.k] := cd.n_cd.k;
    AbsResult := true;
  else
    AbsResult := false;
⟩

void MCAS(CASNDescriptor cd) {
L:  if (cd.status = UNDECIDED)
      s := SUCCEEDED;
      for (i := 1; i ≤ cd.k; i++)
A1:     ⟨v := RDCSS(&cd.status, UNDECIDED, cd.a_i, cd.o_i, cd);
        Lin_cd(¬IsCASNDesc(v) ∧ v ≠ cd.o_i ∧ cd.AbsResult = undef);⟩
        if(IsCASNDesc(v))
          if(v ≠ cd)
            MCAS_impl(v); goto L;
        else if (v ≠ cd.o_i)
          s := FAILED; break;
A2:   ⟨Lin_cd(cd.status = UNDECIDED ∧ s = SUCCEEDED);
       CAS(&cd.status, UNDECIDED, s);⟩
    if (cd.status = SUCCEEDED)
F1:   for (i := 1; i ≤ cd.k; i++) CAS(cd.a_i, cd, cd.n_i);
    else
F2:   for (i := 1; i ≤ cd.k; i++) CAS(cd.a_i, cd, cd.o_i);
}
```

Figure 5.14: MCAS implementation

another concurrent invocation to `MCAS`, we 'help' that operation complete and restart our own operation. (If we encounter our own descriptor, it does not matter, it means some other thread has 'helped' us.) On exiting the `for` loop successfully, we have placed our descriptor on all relevant memory locations: then, we simply change our state to `SUCCEEDED` (or `FAILED`, if one of the `RDCSS` operations failed). Finally, we update the memory contents to their new or old values depending on whether `MCAS` succeeded or not.

Similar to `RDCSS`, `MCAS` must distinguish its descriptors from normal values. To do so, the implementation reserves a second bit in the domain of values and calls the function `IsCASNDesc(v)` to test that bit.

Linearisation points Treating the RDCSS call as atomic, the linearisation points are straightforward. If `MCAS` succeeds, the CAS at line A2 is the linearisation point. If it fails, the linearisation point is the RDCSS at line A1 that noticed that $[\texttt{cd.a}_\texttt{i}] \neq \texttt{o}_\texttt{i}$. Because multiple threads could have noticed that the `MCAS` failed, we take the linearisation point be the time that this was first noticed.

Predicates Let $\mathsf{CD}_{cd}(s, k, addr, old, new, r)$ denote a valid MCAS descriptor with status s, width k, an array $addr$ of addresses, an array old of expected values, an array new of new values, and abstract result r. Valid descriptors have a few constraints on their arguments: the three arrays must have length k, there must not be any duplicates in the a array, and there is a correspondence between the `status` and `AbsResult` fields.

$$
\begin{aligned}
&\mathsf{CD}_{cd}(s, k, addr, old, new, r) \stackrel{\text{def}}{\iff} \\
&\qquad cd \mapsto \{\texttt{.status}{=}s;\ \texttt{.k}{=}k;\ \texttt{.AbsResult}{=}r\} \\
&\quad * \circledast_{1 \le i \le k} (cd \mapsto \{\texttt{.a}_i{=}addr(i);\ \texttt{.o}_i{=}old(i);\ \texttt{.n}_i{=}new(i)\} \wedge addr(i) \in A) \\
&\wedge\ dom(addr) = dom(old) = dom(new) = \{1, \ldots, k\} \\
&\wedge\ (\forall i, j \in \{1, \ldots, k\}.\ addr(i) = addr(j) \Rightarrow i = j) \\
&\wedge\ (r = \texttt{true} \Leftrightarrow s = \mathsf{SUCCEEDED}) \\
&\wedge\ (r = undef \Rightarrow s = \mathsf{UNDECIDED})
\end{aligned}
$$

Abstraction map The abstraction map is straightforward. If an address contains a descriptor, then *abstractly* it holds the old value if the MCAS has not (yet) succeeded, or the new value if the MCAS has succeeded. Informally, we can write the following abstraction map:

$$
\mathsf{ABS}[a] = \begin{cases}
v & \text{if } \mathsf{MEM}[a] = v \wedge \neg\texttt{IsCASNDesc}(v) \\
cd.\texttt{o}_i & \text{if } \mathsf{MEM}[a] = cd \wedge cd.\texttt{a}_i = a \wedge cd.\texttt{status} \neq \mathsf{SUCCEEDED} \\
cd.\texttt{n}_i & \text{if } \mathsf{MEM}[a] = cd \wedge cd.\texttt{a}_i = a \wedge cd.\texttt{status} = \mathsf{SUCCEEDED}
\end{cases}
$$

This abstraction map is a function because each a_i field of the descriptor cd contains a different address.

Unfortunately, expressing this abstraction map in separation logic is tricky. The problem is that a given descriptor cd may be needed to define the abstraction map of several addresses. Therefore we cannot simply take the separate conjunction of the abstraction maps of all the addresses. Preferably, we would like a connective that treats the descriptors additively and the addresses multiplicatively. We can simulate this effect using the following pattern involving the magic wand:

Definition 49. $\langle P \rangle Q \stackrel{\text{def}}{\iff} P \mathrel{-\!\!*} (P * Q)$

Informally, one can understand this formula as asserting that Q holds in the current heap provided its context satisfies P. This reading is justified by the following properties:

$$P * \langle P \rangle Q \iff P * Q$$

$$P * \circledast_{i \in \mathcal{I}} \langle P \rangle Q_i \iff P * \circledast_{i \in \mathcal{I}} Q_i$$

Hence, we can represent the abstraction map as the following separation logic assertion:

$$J \stackrel{\text{def}}{\iff} \circledast_{x \in A} \left(\begin{array}{l} \exists v.\ x \mapsto v * Abs[x] \mapsto v \wedge \neg \texttt{IsCASNDesc}(v) \\ \vee\ \exists v\, a\, o\, i.\ x \mapsto v * \langle \mathsf{CD}_v(\mathsf{FAILED}, \_, a, o, \_, \_) \wedge a_i = x \rangle Abs[x] \mapsto o_i \\ \vee\ \exists v\, a\, o\, i.\ x \mapsto v * \langle \mathsf{CD}_v(\mathsf{UNDECIDED}, \_, a, o, \_, \_) \wedge a_i = x \rangle Abs[x] \mapsto o_i \\ \vee\ \exists v\, a\, n\, i.\ x \mapsto v * \langle \mathsf{CD}_v(\mathsf{SUCCEEDED}, \_, a, \_, n, \_) \wedge a_i = x \rangle Abs[x] \mapsto n_i \end{array} \right)$$

$$* \circledast_{cd \in D} \mathsf{CD}_{cd}(\_, \_, \_, \_, \_, \_)$$

J says that for each address x in A, if x contains a concrete value then its abstract value is the same as that concrete value. If, however, it contains a descriptor and the descriptor is in the context then x's abstract value is n_i or o_i depending on whether the MCAS has succeeded. Finally, we separately have a collection of descriptors D.

Actions First, we have three actions that do not change the abstract state.

$$a(i) \mapsto o(i) \rightsquigarrow a(i) \mapsto cd \quad \text{provided} \quad \mathsf{CD}_{cd}(\mathsf{UNDECIDED}, k, a, o, n, r) \tag{5.1}$$

$$a(i) \mapsto cd \rightsquigarrow a(i) \mapsto n(i) \quad \text{provided} \quad \mathsf{CD}_{cd}(\mathsf{SUCCEEDED}, k, a, o, n, r) \tag{5.2}$$

$$a(i) \mapsto cd \rightsquigarrow a(i) \mapsto o(i) \quad \text{provided} \quad \mathsf{CD}_{cd}(\mathsf{FAILED}, k, a, o, n, r) \tag{5.3}$$

These are performed by lines A1, F1, and F2 respectively.

There are another three actions that change the fields of an MCAS descriptor.

$$\mathsf{CD}_{cd}(\mathsf{UNDECIDED}, k, a, o, n, \mathit{undef}) \rightsquigarrow \mathsf{CD}_{cd}(\mathsf{UNDECIDED}, k, a, o, n, \texttt{false}) \tag{5.4}$$

$$\mathsf{CD}_{cd}(\mathsf{UNDECIDED}, k, a, o, n, \texttt{false}) \rightsquigarrow \mathsf{CD}_{cd}(\mathsf{FAILED}, k, a, o, n, \texttt{false}) \tag{5.5}$$

$$\begin{pmatrix}\mathsf{CD}_{cd}(\mathsf{UNDECIDED}, k, a, o, n, \mathit{undef})\\ * \circledast_i (a_i \mapsto cd * Abs[a_i] \mapsto o_i)\end{pmatrix} \rightsquigarrow \begin{pmatrix}\mathsf{CD}_{cd}(\mathsf{SUCCEEDED}, k, a, o, n, \texttt{true})\\ * \circledast_i (a_i \mapsto cd * Abs[a_i] \mapsto n_i)\end{pmatrix} \tag{5.6}$$

The first is performed by line A1 when the RDCSS fails; the other two are performed by line A2.

Note how all these actions satisfy the rely condition of the implementation of RDCSS. The first three actions are just restricted versions of WriteB and the next three actions are restricted versions of WriteA.

Proof outline Besides the abstraction map, the following important assertion is preserved by the rely condition:

$$K(m) \stackrel{\text{def}}{=} \mathsf{CD}_{\mathtt{cd}}(s, a, o, n, r) * (s = \mathsf{UNDECIDED} \Rightarrow \circledast_{1 \le j \le m} a_j \mapsto cd * Abs[a_j] \mapsto o_j * \mathit{true})$$

This assertion is part of the loop invariant for the first `for` loop of `MCAS`. It states that if the status of the operation is UNDECIDED then the first m addresses in the descriptor contain a pointer to the descriptor. Informally, this is stable under the rely condition because in order to remove a descriptor from a memory location (action 5.2 or 5.3), the status must be SUCCEEDED or FAILED.

5.4 Related work

This section briefly discusses the two alternative ways of proving linearisability: *reduction* and *simulation*.

Reduction-based techniques

A simple way to reason about atomicity is Lipton's theory of left and right movers [55]. An action a is a *right mover* if in any execution trace where a occurs immediately before an action b of a different thread, we can swap the execution order of a and b without changing the resulting state. Similarly, an action is a *left mover* if it left-commutes with operations of other threads. Lock acquisitions are right mover actions, lock releases are left movers; reading a variable is a *both mover* provided there is no race condition on that variable, otherwise it is an non-commutative atomic action. A composite action is

Let $B_j \stackrel{\text{def}}{=} (x = \mathsf{UNDECIDED} \Rightarrow a(j) \mapsto \mathtt{cd} * Abs[a(j)] \mapsto o(j) * true)$.

```
void MCAS(CASNDescriptor cd) {
L: {J ∧ CD_cd(_, k, a, o, n, _) * true}
 if (cd.status = UNDECIDED) {
   s := SUCCEEDED;
   {J ∧ CD_cd(_, k, a, o, n, _) * true ∧ s = SUCCEEDED}
   for (i := 1; i ≤ cd.k; i++) {
     {J ∧ ∃x. CD_cd(x, k, a, o, n, _) * ⊛_{1≤j<i} B_j ∧ s = SUCCEEDED}
     ⟨v := RDCSS(&cd.status, UNDECIDED, cd.a_i, cd.o_i, cd);
     Lin_cd(¬IsCASNDesc(v) ∧ v ≠ cd.o_i ∧ cd.AbsResult = undef);⟩
     { ∃x r. CD_cd(x, k, a, o, n, r) * ⊛_{1≤j<i} B_j * ((v = o_i ∨ v = cd) ⇒ B_i)
       ∧ (IsCASNDesc(v) ⇒ CD_v(_, _, _, _, _, _) * true)
       ∧ ((¬IsCASNDesc(v) ∧ v ≠ o_i) ⇒ r = false) ∧ J
       ∧ s = SUCCEEDED }
     if(IsCASNDesc(v))
       if(v ≠ cd)
         {J ∧ CD_cd(_, k, a, o, n, _) * CD_v(_, _, _, _, _, _) * true}
         MCAS_impl(v); goto L;
     else if (v ≠ cd.o_i)
       {J ∧ CD_cd(s, k, a, o, n, false) * true}
       s := FAILED; break;
     {J ∧ ∃x. CD_cd(x, k, a, o, n, _) * ⊛_{1≤j≤i} B_j ∧ s = SUCCEEDED}
   }
   { s = SUCCEEDED ∧ J ∧ ∃x. CD_cd(x, k, a, o, n, _) * ⊛_{1≤j≤k} B_j
     ∨ s = FAILED ∧ J ∧ ∃x. CD_cd(x, k, a, o, n, false) }
   ⟨Lin_cd(cd.status = UNDECIDED ∧ s = SUCCEEDED);
    CAS(&cd.status, UNDECIDED, s);⟩
 }
 { J ∧ CD_cd(SUCCEEDED, k, a, o, n, true) * true
   ∨ J ∧ CD_cd(FAILED, k, a, o, n, false) * true }
 if (cd.status = SUCCEEDED)
   for (i := 1; i ≤ cd.k; i++) CAS(cd.a_i, cd, cd.n_i);
 else
   for (i := 1; i ≤ cd.k; i++) CAS(cd.a_i, cd, cd.o_i);
 { J ∧ CD_cd(SUCCEEDED, k, a, o, n, true) * true
   ∨ J ∧ CD_cd(FAILED, k, a, o, n, false) * true }
}
```

Figure 5.15: Proof outline for MCAS

deemed atomic if it consists of a sequence of right mover actions, followed by a single atomic action, followed by a sequence of left mover actions.

Cohen and Lamport [16] showed how reduction-based proofs can be done in TLA. More recently, Flanagan and Qadeer [29, 28] have defined a type and effect system, which assigns atomicity effects to expressions based on whether the expressions are left or right movers. So far, their work has been focused on a lock-based paradigm and relies on a separate race condition detector. Wang and Stoller [76] extended Flanagan and Qadeer's work to verify the atomicity simple algorithms using CAS and LL/CS. Their approach is very good at verifying quickly the atomicity of simple algorithms that consist of a single CAS loop, but it is limited to algorithms that conform to this rigid pattern.

Simulation-based techniques

Another approach to verifying concurrent algorithms is based on I/O automata or UNITY. The verifier must manually translate the algorithm to an automaton, and then prove that this automaton is observationally equivalent to a simpler automaton that represents the specification. To do so, one constructs a forward or a backward simulation between the two automata. Using these techniques, Doherty et al. [25] and Colvin et al. [18] verify a few simple stack and queue algorithms. In more complex cases, such as the 'lazy list' algorithm [38] presented in §5.3.2, Colvin et al. [19] define an intermediate automaton and combine forward and backward simulation.

This approach may be useful for small complex algorithms, but it is unlikely to scale for larger programs because it lacks modularity. Another weakness of this method is the manual translation of the algorithm into an I/O automaton. While this translation could perhaps be automated, one still needs to prove that it is sound.

Chapter 6

Mechanisation

A program logic, such as RGSep, can be mechanised at two levels:

- By embedding it in a generic theorem prover (such as Isabelle, HOL, or Coq), proving soundness of the logic and generating a useful reasoning system.
- By developing a specialised tool that checks that a program adheres to a specification belonging to subset of the assertion language.

Here, the latter approach is followed. The result is a tool, SmallfootRG, that takes a lightly annotated program as its input and proves that it is correct with respect to its user-supplied specification or reports an error. SmallfootRG is based on Smallfoot [6], a simple theorem prover that handles a subset of separation logic, and contains three additional decision procedures: septraction elimination, symbolic execution of atomic commands, and stabilization. Had the former approach been followed, these procedures would have been encoded as tactics in the theorem prover.

First, we will go over the three new decision procedures inside SmallfootRG: how to eliminate the septraction operator (see §6.2.2), how to execute an atomic command on a symbolic state (see §6.4.2), and how to get a stable assertion implied by a given unstable assertion (see §6.4.3). Then, in §6.5 we will see how these procedures are applied as SmallfootRG verifies the lock-coupling list algorithm. Finally, §6.6 presents some experimental results.

6.1 SmallfootRG assertions

SmallfootRG assertions are a subset of RGSep assertions chosen to facilitate entailment checking and symbolic execution. Recall that the state is split into thread-local state and shared state. Hence, the assertions specify a state consisting of two heaps with disjoint

domains: the local heap (visible to a single thread), and the shared heap (visible to all threads). Normal formulae, P, specify the local heap, whereas boxed formulae, $\boxed{P}$, specify the shared heap.[1] Boxes cannot be nested.

SmallfootRG accepts assertions written in the following grammar:

$$\begin{array}{rl} A, B ::= & e_1{=}e_2 \mid e_1{\neq}e_2 \mid e \mapsto \rho \mid \mathsf{lseg}(e_1, e_2) \mid \mathsf{junk} \\ P, Q, R, S ::= & A \mid P \vee Q \mid P * Q \mid P \mathrel{-\!\!\circledast} Q \mid P\lfloor_{e_1,\ldots,e_n} \\ p, q ::= & p \vee q \mid P * \boxed{Q} \end{array}$$

where e is a pure expression, an expression that does not depend on the heap. All variables starting with an underscore (e.g., $\_\mathtt{x}$) are implicitly existentially quantified at the top level. In the assertion $\bigvee_i(P_i * \boxed{Q_i})$, if X is the set of existential free variables of $\bigvee_i(P_i * \boxed{Q_i})$ then their scope is $\exists X.\, \bigvee_i(P_i * \boxed{Q_i})$.

The first line contains the usual atomic assertions of separation logic: pure predicates (that do not depend on the heap), heap cells ($e \mapsto \rho$), list segments ($\mathsf{lseg}(e_1, e_2)$), and junk. The formula $e \mapsto \rho$ asserts that the heap consists of a single memory cell with address e and contents ρ, where ρ is a mapping from field names to values (pure expressions); $\mathsf{lseg}(e_1, e_2)$ says that the heap consists of an acyclic linked list segment starting at e_1 and ending at e_2; junk asserts the heap may contain inaccessible state. For notational convenience, let pure assertions hold only on the empty heap. Technically, $e_1{=}e_2$ is an abbreviation for the formula $(e_1{=}_{\mathsf{SL}}e_2) \wedge \mathit{emp}$ where $=_{\mathsf{SL}}$ is the usual definition of equality in separation logic. This way, we can write $P * (e_1 = e_2)$ instead of $P \wedge (e_1 =_{\mathsf{SL}} e_2)$.

The second line contains operators for building larger assertions:

- Disjunction, $P \vee Q$, asserts that the heap satisfies P or Q.
- Separating conjunction, $P * Q$, asserts that the heap can be divided into two (disjoint) parts, one satisfying P and the other satisfying Q.
- *Septraction* ($-\!\!\circledast$) is defined as $h \models (P \mathrel{-\!\!\circledast} Q) \iff \exists h_1\, h_2.\ h_2 = h * h_1$ and $h_1 \models P$ and $h_2 \models Q$. This operation is similar to subtraction or differentiation, as it achieves the effect of subtracting heap h_1 satisfying P from the bigger heap h_2 satisfying Q.
- The "dangling" operator, $P\lfloor_D$, asserts that P holds and that all locations in the set D are not allocated. This can be defined in separation logic as $P\lfloor_{(E_1,\ldots,E_n)} \iff P \wedge \neg((E_1 \mapsto \_) * \mathsf{junk}) \wedge \cdots \wedge \neg((E_n \mapsto \_) * \mathsf{junk})$, but for analysis it is better treated as a built-in assertion form, because it is much easier to analyse than $\wedge$ and $\neg$.

[1] More generally, SmallfootRG supports multiple disjoint regions of shared state, and boxes are annotated with the name ϱ of the region: $\boxed{P}_\varrho$ (cf. §4.3.2). For clarity of exposition, this Chapter presents the analysis with respect to a single resource, and omits the subscript.

$$
\begin{aligned}
(e \mapsto \rho)\lfloor_D &\iff e \notin D * (e \mapsto \rho) \\
&\text{where } e \notin \{e_1, \ldots, e_n\} \stackrel{\text{def}}{=} e \neq e_1 * \cdots * e \neq e_n \\
\mathsf{lsegi}_{tl,\rho}(e_1, e_2, D')\lfloor_D &\iff \mathsf{lsegi}_{tl,\rho}(e_1, e_2, D \cup D') \\
(P * Q)\lfloor_D &\iff P\lfloor_D * Q\lfloor_D \\
(P \vee Q)\lfloor_D &\iff P\lfloor_D \vee Q\lfloor_D
\end{aligned}
$$

Figure 6.1: Elimination rules for $P\lfloor_D$.

Finally, the third line introduces $P * \boxed{Q}$, the novel assertion of RGSep, which does not exist in separation logic. It asserts that the shared state satisfies Q and that the local state is separate and satisfies P.

6.2 Entailment checking

Given a procedure that checks separation logic entailments, it is relatively easy to extend it to handle the dangling operator ($\lfloor_D$) and septraction ($-\circledast$): see §6.2.1 and §6.2.2 for details.

Reasoning about assertions with boxes is straightforward. First, assertions containing boxes are always written in a canonical form, $\bigvee_i (P_i * \boxed{Q_i})$. Given an implication between formulas in this form, we can essentially check implications between normal separation logic formulae, by the following lemma:

$$(P \vdash P') \wedge (Q \vdash Q') \implies (P * \boxed{Q} \vdash P' * \boxed{Q'})$$

Furthermore, we can deduce from $P * \boxed{Q}$ all the heap-independent facts, such as $x \neq y$, which are consequences of $P * Q$, since shared and local states are always disjoint.

6.2.1 Reasoning about the dangling operator ($\lfloor_D$)

Extending any separation logic theorem prover to handle the dangling operator ($\lfloor_D$) is simple. As it distributes over disjunction ($\vee$) and separating conjunction ($*$), it can be eliminated from all terms (see Figure 6.1) except for those containing recursive predicates, such as lseg. Recursive predicates require the dangling set D to be passed as a parameter,

$$\mathsf{lsegi}_{tl,\rho}(E_1, E_2, D) \stackrel{\text{def}}{=} (E_1 = E_2) \vee \exists x.\, E_1 \mapsto (tl{=}x, \rho)\lfloor_D * \mathsf{lsegi}_{tl,\rho}(x, E_2, D).$$

The list segment is subscripted with the name of the linking field, tl, and any common fields, ρ, that all the nodes in the list segment have. Being able to remember any common fields is important for the `lazy list` algorithm (see §6.6) because its invariant involves

a list segment where all the nodes are marked as deleted (have a `marked` field set to 1). We omit the subscript when the linking field is tl and ρ is empty.

Note that the above definition of lsegi is *imprecise* (cf. Def. 8 in §2.4): $\mathsf{lsegi}(E, E, \emptyset)$ describes both the empty heap and a cyclic list. So far program analyses based on separation logic [7, 24] have used precise list segments. Using imprecise list segments simplifies the theorem prover, as the side-conditions for appending list segments are not needed. In contrast, frame inference (i.e. given P and Q find a frame R such that $P \vdash Q * R$) becomes harder as there may be multiple solutions. A precise list segment, $\mathsf{lseg}(E_1, E_2)$, is just a special case of our imprecise list segment, $\mathsf{lsegi}(E_1, E_2, \{E_2\})$.

Another benefit of the dangling operator is that some proof rules can be strengthened, removing some causes of incompleteness. For instance, the following application of the proof rule for deallocating a memory cell $\{P * \mathtt{x} \mapsto \_\}$ `dispose(x)` $\{P\}$ can be strengthened by rewriting the precondition and obtain $\{P\lfloor_{\mathtt{x}} * \mathtt{x} \mapsto \_\}$ `dispose(x)` $\{P\lfloor_{\mathtt{x}}\}$.

6.2.2 Reasoning about septraction ($-\!\circledast$)

This section describes a few general properties of septraction and then shows how SmallfootRG reasons about this connective. Septraction can be defined in terms of the separating implication ($-\!*$) as follows:

$$P -\!\circledast Q \iff \neg(P -\!* \neg Q)$$

This definition, however, is useless for automated reasoning because negation is hard to reason about in separation logic. In contrast, septraction has much nicer properties. The following properties are direct consequences of the definitions.

$$\begin{aligned} \mathit{emp} -\!\circledast P &\iff P \\ (P * Q) -\!\circledast R &\iff P -\!\circledast (Q -\!\circledast R) \\ P -\!\circledast Q &\iff P -\!\circledast (Q \wedge (P * \mathit{true})) \end{aligned}$$

In addition, septraction distributes over $\vee$, and semi-distributes over $\wedge$.

$$\begin{aligned} P -\!\circledast (Q \vee R) &\iff (P -\!\circledast Q) \vee (P -\!\circledast R) \\ (Q \vee R) -\!\circledast P &\iff (Q -\!\circledast P) \vee (R -\!\circledast P) \\ P -\!\circledast (Q \wedge R) &\implies (P -\!\circledast Q) \wedge (P -\!\circledast R) \\ (Q \wedge R) -\!\circledast P &\implies (Q -\!\circledast P) \wedge (R -\!\circledast P) \end{aligned}$$

If P is *exact*, the last two properties become equivalences.

If P is *precise*, then $(P -\!\circledast (P * Q)) \implies Q$.

$$
\begin{array}{rcl}
(e_1 \mapsto \rho_1) \mathrel{-\!\!\circledast} (e_2 \mapsto \rho_2) & \iff & e_1 = e_2 * \rho_1 \simeq \rho_2 \\
e_1{\mapsto}(tl{=}e_2, \rho) \mathrel{-\!\!\circledast} \mathsf{lsegi}_{tl,\rho'}(e, e', D) & \iff & \\
\multicolumn{3}{r}{e_1{\neq}0 * e_1{\notin}D * \rho{\simeq}\rho' * \mathsf{lsegi}_{tl,\rho'}(e, e_1, D)\lfloor_{e_1} * \mathsf{lsegi}_{tl,\rho'}(e_2, e', D)\lfloor_{e_1}} \\
(e \mapsto \rho) \mathrel{-\!\!\circledast} (P * Q) & \iff & P\lfloor_e * (e \mapsto \rho \mathrel{-\!\!\circledast} Q) \\
\multicolumn{3}{r}{\vee \;\; (e \mapsto \rho \mathrel{-\!\!\circledast} P) * Q\lfloor_e} \\
(e \mapsto \rho) \mathrel{-\!\!\circledast} (P \vee Q) & \iff & (e \mapsto \rho \mathrel{-\!\!\circledast} P) \;\vee\; (e \mapsto \rho \mathrel{-\!\!\circledast} Q) \\
(P * Q) \mathrel{-\!\!\circledast} R & \iff & P \mathrel{-\!\!\circledast} (Q \mathrel{-\!\!\circledast} R) \\
(P \vee Q) \mathrel{-\!\!\circledast} R & \iff & (P \mathrel{-\!\!\circledast} R) \vee (Q \mathrel{-\!\!\circledast} R)
\end{array}
$$

Figure 6.2: Elimination rules for septraction ($-\!\!\circledast$).

When we are septracting a single memory cell, $e \mapsto \rho$, then further properties hold:

$$
\begin{array}{rcl}
(e \mapsto \rho) \mathrel{-\!\!\circledast} P & \iff & ((e \mapsto \rho) \mathrel{-\!\!\circledast} P)\lfloor_e \\
(e \mapsto \rho) \mathrel{-\!\!\circledast} P\lfloor_D & \iff & ((e \mapsto \rho) \mathrel{-\!\!\circledast} P)\lfloor_D * e \notin D \\
(e \mapsto \rho) \mathrel{-\!\!\circledast} (e' \mapsto \rho') & \iff & e = e' * \rho \simeq \rho' \\
(e \mapsto \rho) \mathrel{-\!\!\circledast} (P * Q) & \iff & (((e \mapsto \rho) \mathrel{-\!\!\circledast} P) * Q\lfloor_x) \vee (((e \mapsto \rho) \mathrel{-\!\!\circledast} Q) * P\lfloor_x) \\
(e \mapsto \rho) \mathrel{-\!\!\circledast} emp & \iff & false
\end{array}
$$

where $e \notin \{e_1, \ldots, e_n\} \stackrel{\text{def}}{\iff} e{\neq}e_1 * \ldots * e{\neq}e_n$, and
$\rho \simeq \rho' \stackrel{\text{def}}{\iff} \circledast_{x \in (dom(\rho) \cap dom(\rho'))} \rho(x){=}\rho'(x)$.

Intuitively, if we remove a memory cell from P, the result does not contain the removed cell. If we remove a memory cell from $P\lfloor_D$, then D must not contain the address of the memory cell; otherwise, the result is false. If we remove a memory cell from another memory cell, the two memory cells must be identical and the resulting state is empty. Removing a memory cell from a separating conjunction of two formulae generates a case split: the cell could belong either to the first conjunct or to the second. This equivalence is reminiscent of the chain rule of differentiation ($\frac{d(yz)}{dx} = \frac{dy}{dx}z + y\frac{dz}{dx}$). Finally, removing a cell from the empty heap is impossible.

List segments and septraction mix very well. Removing a node from a list produces two list segments: one from the beginning up to the removed node, and one starting after that node until the end of the list.

$$
e_1{\mapsto}(tl{=}e_2, \rho) \mathrel{-\!\!\circledast} \mathsf{lsegi}_{tl,\rho'}(e, e') \iff e_1{\neq}0 * \rho{\simeq}\rho' * \mathsf{lsegi}_{tl,\rho'}(e, e_1)\lfloor_{e_1} * \mathsf{lsegi}_{tl,\rho'}(e_2, e')\lfloor_{e_1}
$$

Hence, we can eliminate the septraction operator from $P \mathrel{-\!\!\circledast} Q$, provided that P and Q belong to the fragment of separation logic accepted by SmallfootRG, and P does not contain any lseg or junk predicates (see Figure 6.2). Had we allowed further inductive predicates, such as $\mathsf{tree}(E_1)$, we would have needed an additional rule for computing $(E \mapsto \rho) \mathrel{-\!\!\circledast} \mathsf{tree}(E_1)$.

Aside: weakest precondition versus strongest postcondition

Given an unknown command C specified as $\{P\}\ C\ \{Q\}$, $P * (Q \mathrel{-\!\!*} R)$ is the weakest *safe* precondition of executing C and deriving the postcondition R. Similarly, given a precondition R, provided $R \Rightarrow (P * true)$, then $Q * (P \mathrel{-\!\!\circledast} R)$ is the strongest postcondition of executing C from an initial state satisfying R

Given an action $P \rightsquigarrow Q$, there are two obvious ways to check that an assertion R is stable under the action: we can either use the weakest precondition or the strongest postcondition formulation. If P is *precise*, the two ways are equivalent:

$$(R \wedge P * true) \Rightarrow (P * (Q \mathrel{-\!\!*} R)) \iff ((P \mathrel{-\!\!\circledast} R) * Q \Rightarrow R)),$$

but the latter is generally easier to compute than the former.

6.3 Programs in SmallfootRG

SmallfootRG programs are written in an untyped toy language whose syntax resembles C. Expressions, E, consist of integer constants, `null` (which is the same as 0), variables, field dereferences, and arithmetic operations (plus, minus, times, divide, and logical xor).

$$E ::= n \mid \texttt{null} \mid var \mid E\texttt{->field} \mid E_1\texttt{+}E_2 \mid E_1\texttt{-}E_2 \mid E_1\texttt{*}E_2 \mid E_1\texttt{/}E_2 \mid E_1\texttt{\^{}}E_2$$

Boolean expressions, B, consist of the boolean constants `true` and `false`, equalities and inequalities between expressions, and the short-cut evaluation `&&` and `||` operators. There are no boolean variables, but their effect can be simulated by an integer variable that holds either zero or non-zero.

$$\begin{aligned} B ::= {} & \texttt{true} \mid \texttt{false} \mid E_1\texttt{==}E_2 \mid E_1\texttt{!=}E_2 \mid E_1\texttt{<=}E_2 \mid E_1\texttt{<}E_2 \mid E_1\texttt{>=}E_2 \mid E_1\texttt{>}E_2 \\ & \mid B_1\texttt{\&\&}B_2 \mid B_1\texttt{||}B_2 \end{aligned}$$

Finally, commands, C, are given by the following grammar:

$$\begin{aligned} C ::= {} & var\texttt{=}E; \mid E_1\texttt{->field=}E_2; \mid \texttt{assume}(B); \mid \texttt{assert}(B); \mid \{C_1 \ldots C_n\} \\ & \mid \texttt{procName}(E_1, \ldots, E_m; E_{m+1}, \ldots, E_{m+n}); \\ & \mid \texttt{if(*)}\ C_1\ \texttt{else}\ C_2 \mid \texttt{if}(B)\ C_1\ \texttt{else}\ C_2 \\ & \mid \texttt{while(*)}\ C \mid \texttt{while}(B)\ C \\ & \mid \texttt{atomic}\ C \mid \texttt{atomic}\ C\ \texttt{as actName}(E_1, \ldots, E_n); \\ & \mid \texttt{atomic}(B)\ C \mid \texttt{atomic}(B)\ C\ \texttt{as actName}(E_1, \ldots, E_n); \end{aligned}$$

These consist of assignments to variables or to fields, `assume` and `assert` statements, procedure calls, sequencing, conditionals, loops, and atomic blocks. As expressions can

contain multiple memory dereferences and the axioms of separation logic can cope only with a single memory read at a time, we convert a complex expression into a sequence of memory reads into fresh temporary variables followed by a simple expression that does not contain any memory dereferences. Similarly, we desugar complex boolean expressions into sequences of assignments and `assume` or `assert` statements, taking care of the short-circuit evaluation of `&&` and `||`. Procedure calls take two types of arguments: those before the semicolon are passed by reference, those after the semicolon are passed by value. If all arguments are to be passed by value, we can omit the semicolon. The `*` in `if(*)` and `while(*)` denotes non-deterministic choice.

Atomic blocks have an optional action annotation which is used by the symbolic execution engine (see §6.4.2). An atomic block with a guard B is a conditional critical region, which blocks until B becomes true and then executes its body atomically. This is just syntactic sugar for `atomic{assume(`B`);` C`}`.

This syntax is largely inherited from Smallfoot with a few extensions (multiple memory dereferences in an expression, short-circuit evaluation, 'less than', etc., non-deterministic choice, and action annotations).

6.4 Reasoning about programs

6.4.1 Describing interference

Recall that RGSep abstracts interference by two relations: the rely condition (R) and the guarantee condition (G), which are compactly represented as sets of actions (updates) to the shared state. Because in many systems, every thread executes the same code, we assume that the rely and guarantee conditions of all threads are identical up a parameter, `TID`, representing the 'current' thread.

SmallfootRG does not attempt to infer such actions; instead it provides convenient syntax for the user to define them. The declaration `action name(params) [`P`] [`Q`]` defines the action $P \rightsquigarrow Q$, giving it a name and some parameters. Formally, the parameters are just existentially quantified variables whose scope extends over both P and Q.

SmallfootRG uses the names and the parameters of the actions to minimise the annotation burden, and to simplify the guarantee checks. SmallfootRG requires the user to annotate every command that changes the shared state with the name of the action it performs and with concrete instantiations for the parameters. Hence, checking that the command performs a permitted action is trivial.

For example, consider the following two action declarations:

```
action Lock(x)     [x|->lk=0  ]  [x|->lk=TID]
action Unlock(x)   [x|->lk=TID]  [x|->lk=0  ]
```

`Lock(x)` takes a location `x` whose `lk` field is zero, and replaces it with `TID`, which stands for the current thread identifier (which is unique for each thread and always non-zero). `Unlock(x)` takes a location `x` whose `lk` field contains the current thread identifier (`TID`) and replaces it with zero. Crucially, the precondition and the postcondition delimit the overall footprint of the action on the shared state. They assert that the action does not modify any *shared* state other than `x`.

As a *guarantee* condition for thread t, the action $P \rightsquigarrow Q$ (which may contain `TID`) stands for $P[t/\texttt{TID}] \rightsquigarrow Q[t/\texttt{TID}]$. As a *rely* condition for thread t, the same action means $(\_t \neq 0 * \_t \neq t * P[\_t/\texttt{TID}]) \rightsquigarrow Q[\_t/\texttt{TID}]$ where $\_t$ is a fresh existential variable.

6.4.2 Symbolic execution of atomic blocks

We discharge verification conditions by performing a form of symbolic execution [52, 7] on symbolic states, and then check that the result implies the given postcondition. Symbolic heaps are formulae of the form

$$A_1 * \ldots * A_m * \boxed{\bigvee_i B_{i,1} * \ldots * B_{i,n_i}}$$

where each A_i and $B_{i,j}$ is an atomic formula. The A part of a symbolic heap describes the local state of the thread, and the B part (inside the box) describes the shared part. Disjunctions within boxed assertions represent more compactly the result of stabilization, and avoid duplicating the local part of the assertion for each disjunct of the shared part. Symbolic states are finite sets of symbolic heaps, representing their disjunction.

As commands can contain non-pure (accessing the heap) expressions in guards and assignments, SmallfootRG translates them into a series of reads to temporary variables followed by an assignment or a conditional that uses only pure expressions. For example, `assume(x->tl==0)` would be translated to `temp = x->tl; assume(temp==0)`, for a fresh variable `temp`.

Except for atomic blocks, symbolic execution is pretty standard: the shared component is just passed around. For atomic blocks more work is needed. Consider executing the atomic block `atomic(B) {C} as Act(x)` starting from symbolic precondition $X * \boxed{S}$. Intuitively, the command is executed atomically when the condition `B` is satisfied. The annotation `as Act(x)` specifies that command C performs shared action `Act` with the parameter instantiated with `x`. Suppose that `Act` was declared as `action Act(x) [P] [Q]`. Our task is to find the postcondition Ψ in the following Hoare triple:

$$\{X * \boxed{S}\}\ \texttt{atomic(B) C as Act(x);}\ \{\Psi\}$$

Our algorithm consists of 4 steps, corresponding to the premises of the following inference rule.

$$\frac{\begin{array}{c}\{X * S\}\ \texttt{assume(B)}\ \{X * P * F\} \\ \{X * P\}\ \texttt{C}\ \{X'\} \qquad X' \vdash Q * Y \qquad \mathit{stabilize}(Q * F) = R\end{array}}{\{X * \boxed{S}\}\ \texttt{atomic(B) C as Act(x);}\ \{Y * \boxed{R}\}}$$

Step 1. Add shared state S to the local state, and call the symbolic execution engine and theorem prover to infer the frame F such that $\{X * S\}$ `assume(B)` $\{X * P * F\}$. This step has the dual function of checking that the action's precondition P is implied, and also inferring the leftover state F, which should not be accessed during the execution of `C`. The symbolic execution of `assume(B)` removes cases where `B` evaluates to false. Note that the evaluation of `B` can access the shared state. If this step fails, the action's precondition cannot be met, and we report an error.

Step 2. Execute the body of the atomic block symbolically starting with $X * P$. Notice that F is not mentioned in the precondition: because of the semantics of Hoare triples in separation logic, this ensures that command `C` does not access the state described by F, as required by the specification of `Act`.

Step 3. Call the theorem prover to infer the frame Y such that $X' \vdash Q * Y$. As before, this has the effect of checking that the postcondition Q is true at the end of the execution, and inferring the leftover state Y. This Y becomes the local part of the postcondition. If the implication fails, the postcondition of the annotated action cannot be met, and we report an error.

Step 4. Combine the shared leftover F computed in the first step with the shared postcondition Q, and stabilize the result $Q * F$ with respect to the execution of actions by the environment as described in §6.4.1.

Read-only atomics Atomic regions that do not write to the heap do not need an action annotation. For such regions we can use the following simplified atomic rule:

$$\frac{\{S\}\ \texttt{C}\ \{X'\} \qquad \mathrm{stab}(X') = R \qquad \texttt{C is read-only.}}{\{\boxed{S}\}\ \texttt{atomic C}\ \{\boxed{R}\}}$$

6.4.3 Inferring stable assertions

Most often, the postcondition of a critical section obtained by symbolic execution is not stable under interference; therefore, we must find a stable postcondition which is weaker than the original.

Assume for the time being that the rely contains a single action `Act` with precondition P and postcondition Q; later, we will address the general case. Mathematically, inferring

a stable assertion from an unstable assertion S is a straightforward fix-point computation

$$S_0 = S \qquad S_{n+1} = S_n \vee (P \mathrel{-\!\!\circledast} S_n) * Q,$$

where S_n is the result of at most n executions of `Act` starting from S. This computation, however, does not always terminate because the domain of assertions is infinite.

Instead, we can approximate the fix-point by using abstract interpretation [20]. Take the concrete domain to be the set of syntactic Smallfoot assertions, and the abstract domain to be a finite subset of normalised Smallfoot assertions that contain a bounded number of existentially quantified variables. Both domains are lattices ordered by implication, with *true* as $\top$ and *false* as $\bot$; $\vee$ is join.

We have a lossy abstraction function α : Assertion $\rightarrow$ RestrictedAssertion that converts a Smallfoot assertion to a restricted assertion, and a concretisation function γ : RestrictedAssertion $\rightarrow$ Assertion which is just the straightforward inclusion (i.e., the identity) function. In our implementation, the abstraction function α is computed by applying a set of abstraction rules, an adaptation of the rules of Distefano et al. [24]. The details are at the end of this section. Nevertheless, the technique is parametric to any suitable abstraction function.

The fix-point can be computed in the abstract domain as follows:

$$S_0 = \alpha(S) \qquad S_{n+1} = S_n \vee \alpha((P \mathrel{-\!\!\circledast} S_n) * Q).$$

In the general case we have n actions $\mathtt{act}_1, \ldots, \mathtt{act}_n$. Two natural algorithms are to interleave the actions during the fix-point computation, or to stabilize one action at a time and repeat the process until we get an assert stable under all actions. The latter strategy tends to reach the fix-point quicker.

As an example, consider stabilizing the assertion $x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$ with the `Lock` and `Unlock` actions from §6.4.1. Before stabilizing, we replace variable `TID` in the specification of the actions with a fresh existentially quantified variable *_tid*, and add assumptions $\_tid \neq 0$ and $\_tid \neq \mathtt{TID}$. The idea is that any thread might be executing in parallel with our thread, and all we know is that the thread identifier cannot be 0 (by design choice) and it cannot be `TID` (because `TID` is the thread identifier of our thread). In this case, stabilization involves the first and third rules in Figure 6.2. In most cases, an inconsistent assertion would be generated by adding one of the following equalities: $0 = 1$, $0 = \_tid$, $\mathtt{TID} = \_tid$. The following fix-point computation does not list those cases.

$$
\begin{aligned}
S_0 &\iff \alpha(x \mapsto lk{=}0 * y \mapsto lk{=}\texttt{TID}) = x \mapsto lk{=}0 * y \mapsto lk{=}\texttt{TID} \\
&\qquad\qquad \text{action } \texttt{lock} \\
S_1 &\iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \texttt{TID} * x \mapsto lk{=}\_tid * y \mapsto lk{=}\texttt{TID}) \\
&\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \texttt{TID} * x \mapsto lk{=}\_tid * y \mapsto lk{=}\texttt{TID}) \\
&\iff \_tid \neq \texttt{TID} * x \mapsto lk = \_tid * y \mapsto lk{=}\texttt{TID} \\
&\qquad\qquad \text{action } \texttt{lock} \\
S_2 &\iff S_1 \vee \alpha(\_tid' \neq 0 * \_tid' \neq \texttt{TID} * x \mapsto lk{=}\_tid' * y \mapsto lk{=}\texttt{TID}) \\
&\iff S_1 \vee (\_tid' \neq 0 * \_tid' \neq \texttt{TID} * x \mapsto lk{=}\_tid' * y \mapsto lk{=}\texttt{TID}) \\
&\iff (\_tid \neq \texttt{TID} * x \mapsto lk{=}\_tid * y \mapsto lk{=}\texttt{TID}) \iff S_1 \\
&\qquad\qquad \text{action } \texttt{unlock} \\
S_3 &\iff S_2 \vee \alpha(x \mapsto lk{=}0 * y \mapsto lk{=}\texttt{TID}) \\
&\iff S_2 \vee (x \mapsto lk{=}0 * y \mapsto lk{=}\texttt{TID}) \\
&\iff S_2
\end{aligned}
$$

In this case, we do not need to stabilize with respect to `lock` again, since `unlock` produced no changes.

Adaptation of Distefano et al.'s abstraction

The domain of separation logic assertions, P, is infinite because assertions can contain an unbounded number of existentially quantified variables. Distefano et al. [24] proposed an abstraction function that restricts the number of existential variables. Their insight was to forbid assertions such as $x \mapsto \_y * \_y \mapsto z$ and to combine the two $\mapsto$ terms into a list segment. Their abstract domain is finite, which is sufficient to guarantee that fix-point computations terminate.

Assume that variable names are ordered and that existential variables are smaller than normal variables in this order. The following algorithm abstracts a formula $P = (A_1 * \ldots A_n)$ where each A_i is an atomic formula.

1. Rewrite all equalities $e_1 = e_2$, so that e_1 is a single variable, which is 'smaller' than all the variables in e_2. This is always possible for simple equalities. If we get an equality between more complex expressions, we can simply drop that equality.

2. For each equality $e_1 = e_2$ in P, substitute any other occurrences of e_1 in P by e_2; if e_1 is an existential variable, discard the equality.

3. For each A_i describing a memory structure (i.e., a cell or a list segment) whose starting address is an existential variable, find all other terms that can point to that address. If there are none, A_i is unreachable; replace it with junk. If there is only one, then try to combine them into a list segment. If there are multiple terms that point to A_i, leave them as they are.

To combine two terms into a list segment, use the following implication:

$$L_{\mathtt{tl},\rho_1}(e_1, \_\mathtt{x})\lfloor_{D_1} * L_{\mathtt{tl},\rho_2}(\_\mathtt{x}, e_2)\lfloor_{D_2} \implies \mathsf{lseg}_{\mathtt{tl},\rho_1\cap\rho_2}(e_1, e_2)\lfloor_{D_1\cap D_2}$$

where $L_{\mathtt{tl},\rho}(e_1, e_2)\lfloor_D$ is $lseg_{\mathtt{tl},\rho}(e_1, e_2)\lfloor_D$ or $e_1 \mapsto (\mathtt{tl} = e_2, \rho) * e_1 \neq D$. This is a generalisation of Distefano et al.'s rule, because our list segments record common fields ρ of nodes, and the set D of disjoint memory locations. In addition, because our list segments are imprecise, we do not need Distefano et al.'s side condition that e_2 is `nil` or allocated separately.

4. Put the formulae in a canonical order by renaming their existential variables. This is achieved by first, ordering atomic formulas by only looking at their shape, while ignoring the ordering between existential variables, and then renaming the existential variables based on the order they appear in the ordered formula.

Simply running this analysis as described above would forget too much information and could not prove even the simplest programs. This is because the analysis would abstract $x{\mapsto}(lk{=}\mathtt{TID}, tl{=}\_y) * \mathsf{lseg}(\_y, z)$ into $\mathsf{lseg}(x, z)$, forgetting that the node x was locked! To avoid this problem, before starting the fix-point calculation, replace existential variables in such $\mapsto$ assertions containing occurrences of `TID` with normal variables to stop the abstraction rules from firing. At the end of the fix-point calculation, replace them back with existential variables. Note that the number of normal variables does not increase during the fix-point computation and hence the analysis still terminates. Experiments indicate that this simple heuristic gives enough precision in practice. In addition, turning dead program variables into existential variables before starting the fix-point calculation significantly reduces the number of cases and speeds up the analysis.

6.5 Example: lock coupling list

This section demonstrates, by example, that SmallfootRG can verify the safety of a fine-grained concurrent linked list. This is the same algorithm as in §3.5, but here we prove a weaker property: that the algorithm does not access unallocated memory and does not leak memory.

Figure 6.3 contains the annotated input to SmallfootRG. Next, we informally describe the annotations required, and also the symbolic execution of our tool. In the tool the assertions about shared states are enclosed in `[ ... ]` brackets, rather than a box. For example, in the assertion `x|->hd=9 * [y|->hd=10]`, the cell at `x` is local whereas that at `y` is shared.

Note that SmallfootRG calculates loop invariants with a standard fix-point computation using the same abstraction function as that for stabilization.

```
action Lock(x)      [x|->lk=0,tl=_w  ] [x|->lk=TID,tl=_w]
action Unlock(x)    [x|->lk=TID,tl=_w] [x|->lk=0,tl=_w]
action Add(x,y)     [x|->lk=TID,tl=_w] [x|->lk=TID,tl=y * y|->tl=_w]
action Remove(x,y) [x|->lk=TID,tl=y*y|->lk=TID,tl=_z]
                    [x|->lk=TID,tl=_z]

ensures: [a!=0 * lseg(a,0)]
init() { a = new(); a->tl = 0; a->lk = 0; }

lock(x) { atomic(x->lk == 0) { x->lk = TID; } as Lock(x); }
unlock(x) { atomic { x->lk = 0; } as Unlock(x); }
```

```
requires: [a!=0 * lseg(a,0)]
ensures:  [a!=0 * lseg(a,0)]
add(e) { local prev,curr,temp;
  prev = a;
  lock(prev);                          } (a)
  atomic { curr = prev->tl; }          } (b)
  if (curr!=0)
    atomic { temp = curr->hd; }
  while(curr!=0 && temp<e) {
    lock(curr);
    unlock(prev);                      } (c)
    prev = curr;
    atomic { curr = prev->tl; }
    if (curr!=0)
      atomic { temp = curr->hd; }
  }
  temp = new();
  temp->lk= 0;
  temp->hd = e;                        } (d)
  temp->tl = curr;
  atomic { prev->tl = temp; }          } (e)
      as Add(prev,temp);
  unlock(prev);
}
```

```
requires: [a!=0 * lseg(a,0)]
ensures:  [a!=0 * lseg(a,0)]
remove(e) { local prev,curr,temp;
  prev = a;
  lock(prev);
  atomic { curr = prev->tl; }
  if (curr!=0)
    atomic { temp = curr->hd; }
  while(curr!=0 && temp!=e) {
    lock(curr);
    unlock(prev);
    prev = curr;
    atomic { curr = prev->tl; }
    if (curr!=0)
      atomic { temp = curr->hd; }
  }
  if (curr!=0) {
    lock(curr);
    atomic { temp = prev->tl; }
    atomic { prev->tl = temp; }        } (f)
        as Remove(prev,curr);
    dispose(curr);
  }
  unlock(prev);
}
```

Figure 6.3: Lock-coupling list. Annotations are in italic font.

The rest of this section explains the highlighted parts of the verification (a)–(f).

Executing an atomic block (a) To illustrate the execution of an atomic block, consider the first `lock` in the `add` function, following the rule in §6.4.2.

(Step 1) Execute the guard and find the frame.

```
prev==a * a!=0 * lseg(a,0)
  assume(prev->lk == 0);
prev==a * a!=0 * prev↦lk:0,tl:_z * lseg(_z,0)
```

The execution unrolls the list segment because `a!=0` ensures that the list is not empty. Then, check that the annotated action's precondition holds, namely `prev↦lk=0,tl=_w`. (Recall that any variable starting with an underscore, such as `_w`, is an existential variable quantified across the pre- and post-condition of the action.) The checking procedure computes the leftover formula – the *frame* – obtained by removing cell `prev`. For this atomic block the frame is `lseg(_z,0)`. The frame is not used by the atomic block, and hence remains true at the exit of the atomic block.

(Step 2) Execute the body of the atomic block starting with the separate conjunction of the local state and the precondition of the action: `prev==a * a!=0 * prev↦lk:0,tl:_z * _w=_z` in total. At the end, we get `prev==a * a!=0 * prev↦lk:TID,tl:_z * _w==_z`.

(Step 3) Try to prove that this assertion implies the postcondition of the action plus some local state. In this case, all the memory cells were consumed by the postcondition; hence, when exiting the atomic block, no local state is left.

(Step 4) So far we have derived the postcondition $\boxed{\texttt{prev}\mapsto(\texttt{lk}{=}\texttt{TID}, \texttt{tl}{=}\texttt{\_z}) * \mathsf{lseg}(\texttt{\_z}, 0)}$, but we have not finished. We must *stabilize* the postcondition to take into account the effect of other threads onto the resulting state. Following the fix-point computation of §6.4.3, compute a weaker assertion that is stable under interference from all possible actions of other threads. In this case, the initial assertion was already stable.

Executing a read-only atomic block (b) The next atomic block only reads the shared state without updating it. Hence, no annotation is necessary, as this action causes no interference. Symbolic execution proceeds normally, allowing the code to access the shared state. Again, when we exit the region, we need to stabilize the derived postcondition.

Stabilization (c) This case illustrates how stabilization forgets information. Consider unlocking the `prev` node within the loop. Just before unlocking `prev`, we have the shared assertion:

$$\boxed{\mathsf{lseg}(\texttt{a}, \texttt{prev}) * \texttt{prev}\mapsto(lk{=}\texttt{TID}, tl{=}\texttt{curr}) * \texttt{curr}\mapsto(lk{=}\texttt{TID}, tl{=}\texttt{\_z}) * \mathsf{lseg}(\texttt{\_z}, 0)}.$$

This says that the shared state consists of a list segment from `a` to `prev`, two adjacent locked nodes `prev` and `curr`, and a list segment from `_z` to `nil`. Just after unlocking the node, before stabilization, we get:

$$\boxed{\mathsf{lseg}(\texttt{a}, \texttt{prev}) * \texttt{prev}\mapsto(lk{=}0, tl{=}\texttt{curr}) * \texttt{curr}\mapsto(lk{=}\texttt{TID}, tl{=}\texttt{\_z}) * \mathsf{lseg}(\texttt{\_z}, 0)}.$$

Stabilization first forgets that $\texttt{prev}{\rightarrow}lk = 0$, because another thread could have locked the node; moreover, it forgets that `prev` is allocated, because it could have been deleted by another thread. The resulting stable assertion is:

$$\boxed{\mathsf{lseg}(\mathsf{a}, \mathtt{curr}) * \mathtt{curr}{\mapsto}(lk{=}\mathtt{TID}, tl{=}\mathtt{\_z}) * \mathsf{lseg}(\mathtt{\_z}, 0)}.$$

Local updates (d) Next we illustrate that local updates do not need to consider the shared state. Consider the code after the loop in `add`. As `temp` is local, the creation of the new cell and the two field updates affect only the local state. These commands cannot affect the shared state. Additionally, as `temp` is local state, we know that no other thread can alter it. Therefore, we get the following symbolic execution:

```
[a!=0 * lseg(a,prev) * prev↦lk=TID,tl=curr * lseg(curr,0)]
  temp = new();  temp->lk = 0;  temp->val = e; temp->tl = z;
[a!=0 * lseg(a,prev) * prev↦lk=TID,tl=curr * lseg(curr,0)]
             * temp↦lk=0,val=e,tl=curr
```

Transferring state from local to shared (e) Next we illustrate the transfer of state from local ownership to shared ownership. Consider the atomic block with the `Add` annotation:

```
[a!=0 * lseg(a,prev) * prev↦lk=TID,tl=curr * lseg(curr,0)]
      * temp↦lk=0,tl=curr
  atomic { prev->tl = temp } as Add(prev,temp);
[a!=0 * lseg(a,prev) * prev↦lk=TID,tl=temp
             * temp↦tl=curr * lseg(curr,0)]
```

We execute the body of the atomic block with the separate conjunction of the local state and the precondition of the action, so $\texttt{prev}{\mapsto}\texttt{lk=TID,tl=curr} * \texttt{temp}{\mapsto}\texttt{lk=0,tl=curr}$ in total. At the end, we get $\texttt{prev}{\mapsto}\texttt{lk=TID,tl=temp} * \texttt{temp}{\mapsto}\texttt{lk=0,tl=prev}$ and we try to prove that this implies the postcondition of the action plus some local state. In this case, all the memory cells were consumed by the postcondition; hence, when exiting the atomic block, no local state is left. Hence the cell `temp` is transferred from local state to shared state.

Transferring state from shared to local (f) This illustrates the transfer of state from shared ownership to local ownership, and hence that shared state can safely be disposed. Consider the atomic block with a `Remove` annotation.

```
[lseg(a,prev) * prev↦lk=TID,tl=curr
          * curr↦lk=TID,tl=temp * lseg(temp,0)]
```

Program	LOC	LOA	Act	#Iter	#Calls	Mem(Mb)	Time(s)
lock coupling	50	9	4	365	3879	0.47	3.9
lazy list	58	16	6	246	8254	0.70	13.5
optimistic list	59	13	5	122	4468	0.47	7.1
blocking stack	36	7	2	30	123	0.23	0.06
Peterson	17	24	10	136	246	0.47	1.35

Table 6.1: Experimental results

```
  atomic {  prev->tl = temp; } as Remove(x,y);
[lseg(a,prev) * prev↦lk=TID,tl=temp * lseg(temp,0)]
            * curr↦lk=TID,tl=temp
```

Removing the action's precondition, `prev↦lk=TID,tl=curr * curr↦lk=TID,tl=temp`, from the shared state leaves a frame of `lseg(a,prev) * lseg(temp,0)`. Executing the body gives `prev↦lk=TID,tl=temp * curr↦lk=TID,tl=temp` and we try to prove that this implies the postcondition of the action plus some local state. The action's postcondition requires `prev↦lk=TID,tl=temp`, so the remaining `curr↦lk=TID,tl=temp` is returned as local state. This action has taken shared state, accessible by every thread, and made it *local* to a single thread. Importantly, this means that the thread is free to dispose this memory cell as no other thread will attempt to access it.

```
[lseg(a,x) * x↦lk=TID,tl=z * lseg(z,0)] * y↦lk=TID,tl=z
  dispose(y);
[lseg(a,x) * x↦lk=TID,tl=z * lseg(z,0)]
```

Summary This example has illustrated fine-grained locking, in particular

- dynamically allocated locks
- non-nested lock/unlock pairs
- disposal of memory (including locks)

Other examples SmallfootRG handles include optimistic reads from shared memory and lazy deletions.

6.6 Experimental results

SmallfootRG extends the separation logic tool called Smallfoot [6]. The tests were executed on a Powerbook G4 1.33 GHz with 786MB memory running OSX 10.4.8. The results are reported in Figure 6.1. For each example we report: the number of lines of code (LOC) and of annotation (LOA); the number of user-provided actions (Actions);

the total number of iterations for all the fix-point calculations for stabilization (#Iter); the number of calls to the underlying theorem prover during stabilization (#Calls); the maximum memory allocated during execution (Mem (Mb)) in megabytes, and the total execution time (Time (s)) in seconds.

We have tested our tool on a number of fine-grained concurrency examples. The first three (lock coupling, lazy list, optimistic list) all implement the data structure of a set as a singly linked list with a lock per node.

- `lock coupling` The main part of the algorithm was described in Section 6.5. When traversing the list, locks are acquired and released in a "hand over hand" fashion.

- `lazy list` An algorithm by Heller et al. [38], which traverses the list without acquiring any locks; at the end it locks the relevant node and validates the node is still in the list. Deletions happen in two steps: nodes are first marked as deleted, then they are physically removed from the list.

- `optimistic list` Similar to lazy list, it traverses the list without acquiring any locks; at the end it locks the relevant node and re-traverses the list to validate that the node is still in the list.

The next two examples are simpler: `blocking stack` simply acquires a lock before modifying the shared stack; and `Peterson` [66] is a well-known mutual exclusion algorithm.

We have a final example of Simpson's `4Slot` [71], which implements a wait-free atomic memory cell with a single reader and a single writer. This algorithm has been verified in both our tool, and Smallfoot. In our new tool it takes under 4 minutes, while in the original Smallfoot it took just under 25 minutes. Also, the specification of the invariant for Smallfoot is over twice as long as the action specification for SmallfootRG.[2]

Program	**Lines of Annotation**	**Time (s)**
`4Slot` (SmallfootRG)	42	221
`4Slot` (Smallfoot)	80	1448

Smallfoot requires the same invariant about shared state at every program point. In contrast, SmallfootRG calculates only the pertinent shared states at each atomic block, so when it enters an atomic block it does not need to consider as many possibilities as Smallfoot. This example demonstrates that using binary relations instead of simply invariants leads to shorter proofs.

Apart from the `4Slot` algorithm, we believe our tool takes an acceptable amount of time to verify the algorithms discussed in this section. Our examples have demonstrated

[2]The specification for both could be simplified if Smallfoot directly supported arrays in the assertion language.

the disposal of memory (lock-coupling list and blocking stack), the optimistic reading of values and leaking memory (lazy and optimistic list algorithms), and classic mutual exclusion problems (Peterson's and Simpson's algorithm).

Chapter 7

Conclusion

7.1 Summary

The dissertation has covered three main topics:

- a new logic, RGSep, that enables concise reasoning about fine-grained concurrency;
- a set of techniques for proving linearisability demonstrated by proofs of important practical algorithms such as MCAS;
- a tool, SmallfootRG, based on RGSep that checks safety properties about fine-grained concurrent algorithms operating on linked lists.

There are many topics that this dissertation did not address: liveness, termination, fairness, starvation—issues well known and widely discussed. It did not consider distributed systems and the additional problems caused by communication delays, process or link failures, and security concerns.

At least, it has shown that reasoning about fine-grained shared-memory concurrency can be done formally, concisely, modularly, and compositionally. And in some cases, even automatically.

7.2 Future work

Dynamic modularity RGSep and separation logic both permit multiple disjoint regions of shared state, but the number of regions is determined statically. Recently, Gotsman et al. [30] described a way of lifting this restriction in Separation Logic thereby permitting locks to be stored in the heap. Their work can probably be recast to RGSep.

Separation logic permissions [8] are also dynamic in nature. Their drawback is that they protect a single memory cell and enforce a very simple property: no writes happen to that memory cell. If integrated with RGSep boxed assertions, permissions could enforce much more complex properties. In addition, RGSep would gain good dynamic modularity and become ideal for reasoning about object-oriented languages.

Liveness/termination Proving termination in a concurrent setting is another interesting open problem. Except for the simplest concurrent programs, termination of a concurrent operation depends heavily on its environment and often on the scheduler's fairness. Terminology such as 'wait-free,' 'lock-free', and 'obstruction-free' describe various degrees of context-dependent termination. Judging from the informal proofs of these properties, their formal proofs will be global, but tractable and, in fact, relatively easy.

Weak memory models Most proof methods, including RGSep, assume that parallel composition has an interleaving semantics. This assumption is false for the so called "weak memory models" provided by modern processors. In these models, there is no globally consistent view of the shared state, and each thread may observe writes to shared variables happening in a different order. Is there a suitable logic for verifying programs running in weak memory models?

Tool support This dissertation has already presented a prototype checker based on RGSep demonstrating that automated reasoning about fine-grain concurrency is possible. There are a few limitations to the current version of SmallfootRG, which can be resolved in the future. Interference actions can be inferred, arithmetic can be taken into account.

But more importantly, tools should be developed to encompass the techniques for proving linearisability and automating these proofs. This will enable us to verify concurrent libraries involving intricate internal concurrency, so that their users may safely assume they expose a simple sequential interface. Proving linearisability automatically is not very difficult, because the respective hand-crafted proofs involve surprisingly simple mathematics, and simple annotations can be used to determine the linearisation point. The recent work by Amit et al. [3] is a promising step in that direction.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. (Cited on pages 17, 86, and 111.)

[2] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):507–534, May 1995. (Cited on pages 18 and 21.)

[3] Daphna Amit, Noam Rinetzky, Thomas Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearisability. In *19th Computer Aided Verification (CAV)*, 2007. (Cited on page 137.)

[4] Edward A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1):110–135, January 1975. (Cited on pages 8 and 32.)

[5] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. (Cited on page 33.)

[6] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO (Formal Methods for Components and Objects)*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005. (Cited on pages 118 and 133.)

[7] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005. (Cited on pages 121 and 125.)

[8] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *32nd ACM Symposium on Principles of Programming Languages (POPL 2005)*, pages 259–270. ACM, 2005. (Cited on pages 8, 29, and 137.)

[9] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in separation logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006. (Cited on page 31.)

[10] John Boyland. Checking interference with fractional permissions. In *10th International Static Analysis Symposium (SAS 2003)*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003. (Cited on page 29.)

[11] Stephen D. Brookes. A semantics for concurrent separation logic. In *15th International Conference on Concurrency Theory (CONCUR)*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2004. (Cited on pages 8 and 32.)

[12] Stephen D. Brookes. Variables as resource for shared-memory programs: Semantics and soundness. *Electronic Notes in Theoretical Computer Science*, 158:123–150, 2006. (Cited on page 32.)

[13] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378. IEEE Computer Society, 2007. (Cited on pages 22 and 32.)

[14] Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In *14th International Static Analysis Symposium (SAS 2007)*, volume 4634 of *Lecture Notes in Computer Science*. Springer, August 2007. (No citations.)

[15] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988. (Cited on page 18.)

[16] Ernie Cohen and Leslie Lamport. Reduction in TLA. In *9th International Conference on Concurrency Theory (CONCUR)*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 1998. (Cited on pages 8 and 117.)

[17] Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. Technical Report CS-TR-987, School of Computing Science, Newcastle University, October 2006. (Cited on pages 19 and 22.)

[18] Robert Colvin, Simon Doherty, and Lindsay Groves. Verifying concurrent data structures by simulation. *Electronic Notes in Theoretical Computer Science*, 137(2):93–110, 2005. (Cited on pages 9 and 117.)

[19] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set. In *18th Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 475–488. Springer, 2006. (Cited on page 117.)

[20] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, 1977. (Cited on page 127.)

[21] Karl Crary, David Walker, and J. Gregory Morrisett. Typed memory management in a calculus of capabilities. In *26th ACM Symposium on Principles of Programming Languages (POPL 1999)*, pages 262–275, 1999. (Cited on page 34.)

[22] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 59–69. ACM Press, 2001. (Cited on page 34.)

[23] Jürgen Dingel. Computer-assisted assume/guarantee reasoning with VeriSoft. In *Int. Conference on Software Engineering (ICSE-25)*, pages 138–148, Portland, Oregon, USA, May 2003. IEEE Computer. (Cited on page 18.)

[24] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302, 2006. (Cited on pages 121, 127, and 128.)

[25] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *Int. Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2004)*, volume 3235 of *Lecture Notes in Computer Science*, pages 97–114, Madrid, Spain, September 2004. IFIP WG 6.1, Springer. (Cited on pages 9 and 117.)

[26] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation (PLDI 2002)*, pages 13–24, New York, NY, USA, 2002. ACM Press. (Cited on page 34.)

[27] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *16th European Symposium on Programming (ESOP 2007)*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007. (Cited on page 59.)

[28] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *ACM Conference on Programming Language Design and Implementation (PLDI 2003)*, pages 338–349. ACM, June 2003. (Cited on pages 8 and 117.)

[29] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *ACM Workshop on Types in Language Design and Implementation*, pages 1–12. ACM, January 2003. (Cited on page 117.)

[30] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. Technical Report MSR-TR-2007-39, Microsoft Research, April 2007. (Cited on page 136.)

[31] Michael Greenwald. *Non-Blocking Synchronization and System Design*. Ph.D. thesis, Stanford University, Palo Alto, CA, USA, 1999. (Cited on page 13.)

[32] Dan Grossman. Type-safe multithreading in Cyclone. In *ACM Workshop on Types in Language Design and Implementation*, pages 13–25, New York, NY, USA, 2003. ACM Press. (Cited on page 34.)

[33] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402. ACM, October 2003. (Cited on page 14.)

[34] Tim Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *16th International Symposium on Distributed Computing*, pages 265–279, October 2002. (Cited on pages 10, 87, 106, and 111.)

[35] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton Jones. Composable memory transactions. In *10th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP 2005)*, 2005. (Cited on page 13.)

[36] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *ACM Conference on Programming Language Design and Implementation (PLDI 2006)*, pages 14–25. ACM, June 2006. (Cited on page 14.)

[37] Jonathan Hayman and Glynn Winskel. Independence and concurrent separation logic. In *21st IEEE Symposium on Logic in Computer Science (LICS 2006)*, pages 147–156. IEEE Computer Society, 2006. (Cited on page 32.)

[38] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *9th International Conference on Principles of Distributed Systems (OPODIS)*, December 2005. (Cited on pages 87, 103, 117, and 134.)

[39] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 206–215, New York, NY, USA, 2004. ACM Press. (Cited on pages 87 and 92.)

[40] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In *10th Computer Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer, June 1998. (Cited on page 18.)

[41] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *ICCAD '00: Int. Conference on Computer-Aided Design*, pages 245–253. IEEE Press, 2000. (Cited on page 18.)

[42] Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. (Cited on page 13.)

[43] Maurice P. Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd IEEE International Conference on Distributed Computing Systems*, May 2003. (Cited on page 13.)

[44] Maurice P. Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Annual International Symposium on Computer Architecture*, 1993. (Cited on page 14.)

[45] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. (Cited on pages 80, 81, and 83.)

[46] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. (Cited on page 34.)

[47] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM Symposium on Principles of Programming Languages (POPL 2001)*, pages 14–26, 2001. (Cited on page 22.)

[48] Bart Jacobs, K. Rustan M. Leino, and Wolfram Schulte. Verification of multithreaded object-oriented programs with invariants. In *Workshop on Specification and Verification of Component-Based Systems*, October 2004. (Cited on page 8.)

[49] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *3rd IEEE Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 137–147. IEEE Computer Society, 2005. (Cited on pages 8 and 33.)

[50] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 420–439. Springer, 2006. (Cited on pages 8 and 33.)

[51] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983. (Cited on pages 9, 18, and 19.)

[52] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. (Cited on page 125.)

[53] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974. (Cited on page 14.)

[54] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994. (Cited on page 9.)

[55] Richard J. Lipton. Reduction: A new method of proving properties of systems of processes. In *2nd ACM Symposium on Principles of Programming Languages (POPL 1975)*, pages 78–86, 1975. (Cited on page 115.)

[56] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981. (Cited on page 18.)

[57] Carroll Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988. (Cited on page 37.)

[58] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *15th International Conference on Concurrency Theory (CONCUR)*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67. Springer, 2004. (Cited on pages 8, 31, 53, 59, and 75.)

[59] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999. (Cited on pages 22 and 24.)

[60] Peter W. O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *15th Int. Workshop on Computer Science Logic (CSL 2001)*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. (Cited on page 59.)

[61] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976. (Cited on page 18.)

[62] Susan S. Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976. (Cited on pages 16 and 18.)

[63] Matthew J. Parkinson. *Local reasoning for Java*. Ph.D. dissertation, University of Cambridge Computer Laboratory, 2005. Also available as Technical Report UCAM-CL-TR-654. (Cited on page 29.)

[64] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. In *21st IEEE Symposium on Logic in Computer Science (LICS 2006)*, pages 137–146. IEEE Computer Society, 2006. (Cited on pages 31 and 88.)

[65] Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In *34th ACM Symposium on Principles of Programming Languages (POPL 2007)*, 2007. (Cited on page 8.)

[66] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981. (Cited on pages 14 and 134.)

[67] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981. (Cited on page 9.)

[68] Leonor Prensa Nieto. The rely-guarantee method in Isabelle/HOL. In *12th European Symposium on Programming (ESOP 2003)*, volume 2618 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2003. (Cited on pages 18, 19, and 22.)

[69] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002. (Cited on pages 22 and 59.)

[70] Nir Shavit and Dan Touitou. Software transactional memory. In *Proc. Symp. on Principles of Distributed Computing*, pages 204–213. ACM, August 1995. (Cited on page 14.)

[71] H. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings*, 137(1):17–30, January 1990. (Cited on page 134.)

[72] Colin Stirling. A compositional reformulation of Owicki-Gries's partial correctness logic for a concurrent while language. In *ICALP*, volume 226 of *Lecture Notes in Computer Science*, pages 407–415. Springer, 1986. (Cited on page 18.)

[73] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *11th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP 2006)*, pages 129–136. ACM, 2006. (Cited on page 85.)

[74] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. Technical Report UCAM-CL-TR-687, University of Cambridge, Computer Laboratory, June 2007. (Cited on page 59.)

[75] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *18th International Conference on Concurrency Theory (CONCUR)*, volume 4703 of *Lecture Notes in Computer Science*. Springer, September 2007. (Cited on page 43.)

[76] Liqiang Wang and Scott D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *10th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP 2005)*, pages 61–71, New York, NY, USA, 2005. ACM Press. (Cited on pages 8 and 117.)

[77] Qiwen Xu, Willem P. de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997. (Cited on page 18.)

[78] Uri Zarfaty and Philippa Gardner. Local reasoning about tree update. *Electronic Notes in Theoretical Computer Science*, 158:399–424, 2006. (Cited on page 24.)

Glossary

$<_H$	order between operations in H	81
$\rightsquigarrow$	syntax for defining actions	37
$*$	separating conjunction	23
$-\!\!*$	separating implication (magic wand)	23
$-\!\!\circledast$	septraction	23
$\mapsto$	a single memory cell assertion	27
$\overset{k}{\mapsto}$	memory cell with permission k	30
$\overset{s}{\mapsto}$	write-once memory cell	84
$\top$	full permission	29
$\bot$	zero permission	29
$\odot$	composition of resources	23
$\oplus$	addition of permissions	29
B	boolean expression	15, 27
C	command	15
c	primitive command	15, 27
E	head-reading expression	15, 26
e	pure expression	26
emp	empty heap assertion	23
G	guarantee	19
H	history	80
i	interpretation for logical variables	24
K	permission algebra	29
k	an accounting figure, a permission	29
l	local state	35
M	resource algebra	23
m	element of a resource algebra	23
P, Q, S	separation logic assertions	23, 24

R	rely	19
s	shared state	35
σ	(total) state	35
u	empty resource	23