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:

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

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.