University of Science and Technology of China

A dissertation submitted for the degree of PhD

Refinement Verification of Concurrent Programs and Its Applications

Author : Hongjin Liang

Supervisor : Xinyu Feng and Zhong Shao

Finished Time : May, 2014

Abstract

Many verification problems can be reduced to refinement verification, i.e., proving that a concrete program has no more behaviors than a more abstract program. This dissertation explores the applications of refinement verification of concurrent programs, and proposes compositional verification techniques that support these applications. It makes several contributions to understanding and verifying concurrent program refinement.

First, it shows a *R*ely-*G*uarantee-based *Sim*ulation (RGSim) as a general proof technique for concurrent program refinement. The novel simulation relation is parameterized with the interference between threads and their parallel environments. It is compositional and supports modular verification. RGSim can incorporate the assumptions about environments made by specific refinement applications, thus is flexible and practical. We apply RGSim to reason about optimizations in parallel contexts. We also reduce the verification of concurrent garbage collectors (GCs) to refinement verification, and propose a general GC verification framework based on RGSim. Using the framework, we verify the Boehm et al. concurrent mark-sweep GC algorithm.

Second, it shows a Hoare-style program logic for modular and effective verification of linearizability of concurrent objects, which is an important application of concurrent program refinement verification. Our logic with a lightweight instrumentation mechanism supports objects with non-fixed linearization points (LPs), including the most challenging ones that use the helping mechanism to achieve lock-freedom (as in HSY elimination-based stack), or have LPs depending on unpredictable future executions (as in the lazy set algorithm), or involve both features (as in the RDCSS algorithm). We generalize RGSim with the support for non-fixed LPs as the meta-theory of our logic, and show it implies a contextual refinement which is equivalent to linearizability. Using our logic we successfully verify 12 well-known algorithms, two of which are used in the `java.util.concurrent` package.

Finally, it shows a unified framework that characterizes the full correctness (i.e., linearizability and progress properties) of concurrent objects via contextual

refinements. We prove that for linearizable objects, each progress property is equivalent to a certain form of termination-sensitive contextual refinement. The framework unifies linearizability and all the five most common progress properties: wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, and deadlock-freedom. It enables modular verification of safety and liveness properties of client programs, and also makes it possible to borrow ideas from existing proof methods for contextual refinements to verify linearizability and a progress property together.

Keywords: Concurrency, Program Verification, Refinement, Simulation, Program Logic, Program Correctness

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Program refinement has been used as a theoretical foundation for program verification. Refinement establishes a relation $\sqsubseteq$ between a concrete program C and a more abstract one (or a specification) $\mathbb{C}$. Informally, $C \sqsubseteq \mathbb{C}$ requires that C have no more observable behaviors than $\mathbb{C}$. It enables us to soundly substitute C for $\mathbb{C}$ in any program context. We say C and $\mathbb{C}$ are equivalent if the other direction $\mathbb{C} \sqsubseteq C$ holds as well. As an example, consider the following program which is an implementation of `x++`.

```
local t;  t := x;  x := t + 1;
```

It first stores the value of `x` in a local variable `t`, and then writes the computation result back to `x`. It refines `x++` if we do not care about the value of the local variable `t`. It is also refined by `x++`, and hence the two programs are equivalent.

Studying refinement and its verification techniques is not only of theoretical interest. In fact, many verification problems can be reduced to refinement verification. Below we list some typical applications.

- *Correctness of compilation and optimizations.* A compiler translates the source program $\mathbb{C}$ into the target C. It has to ensure the semantics preservation, which requires C be a refinement of $\mathbb{C}$ [42]. If every target program is a refinement of its source, we can conclude the correctness of the compiler.

- *Correctness of programs and algorithms.* A correct program or algorithm C has to satisfy its specification, which can be viewed as an abstract program $\mathbb{C}$. Thus the correctness of C can be characterized by a refinement $C \sqsubseteq \mathbb{C}$. For instance, verifying the implementation of an abstract data structure requires us to prove a refinement from an abstract operation to a concrete and executable program (also known as data refinement [36]).

In concurrent settings, we give another two examples showing the deep relationship between the program correctness and refinement.

  - *Concurrent objects.* A concurrent object or library provides a set of methods that allow clients to manipulate the shared data structure with abstract atomic behaviors [33]. Its correctness can be reduced to a refinement from abstract atomic operations to concrete and executable method implementations in a concurrent context. In fact, Filipović et al. [21] have proved that linearizability as a standard functional correctness criterion of concurrent objects is *equivalent* to a contextual refinement.
  - *Implementations of Software Transactional Memory (STM).* Many languages supporting STM provide a high-level atomic block **atomic**$\{\mathbb{C}\}$ as a transaction, so that programmers can assume the atomicity of the execution of $\mathbb{C}$. Atomic blocks are implemented using some STM protocol (e.g., TL2 [16]) that allows very fine-grained interleavings. Verifying that the fine-grained program respects the semantics of atomic blocks gives us the correctness of the STM implementation.

- *Correctness of Garbage Collectors (GCs).* High-level garbage-collected languages (e.g., Java) allow programmers to work at an abstract level without knowledge of the underlying GC algorithm. However, the concrete and executable low-level program involves interactions between the mutators and the collector. If we view the GC algorithm as a transformation from high-level mutators to low-level ones with a concrete GC, the GC safety can be reduced naturally to the correctness of the transformation, which can be characterized by a refinement between the low-level executable implementations and the high-level mutators.

- *Correctness of Operating System (OS) kernels.* A kernel provides an abstract programming model, which hides the details of the underlying hardware and simplifies the development of the high-level user applications. The correctness of the kernel requires that the real user behaviors in the concrete machine model be compatible with their expected behaviors at the abstract model [40]. Thus the verification of OS kernels can be reduced to verifying refinement between the two machines.

There has been a large body of work on refinement verification. However, it is often difficult to apply existing work to verify *concurrent* program refinement.

1.1 Challenges in Verifying Concurrent Program Refinement

This dissertation studies general proof techniques for concurrent program refinement, and mainly focus on the applications on verifying concurrent objects. Below we analyze the key challenges in these two aspects.

1.1.1 General Problems

An effective and general proof technique for refinement $\sqsubseteq$ between concurrent programs needs to satisfy the following requirements.

- *Independence of language details.* In many refinement applications, the concrete program C could be in a different language from the abstract $\mathbb{C}$. For instance, the input of a compiler is usually written in a high-level programming language, while the target may be machine code or in assembly. Also the source and the target may take different views of program states.

- *Support of different granularities of concurrency.* To verify fine-grained implementations of abstract operations, the refinement proof should support different granularities of state accesses at the concrete and the abstract levels. For instance, a concurrent object allows interleavings between concrete methods executed by parallel threads, though it is expected to generate atomic behaviors at the abstract level.

- *Compositionality.* A compositional proof technique is of particular importance for modular verification of refinement. In a concurrent setting, we should be able to know $C_1 \,\|\, C_2 \sqsubseteq \mathbb{C}_1 \,\|\, \mathbb{C}_2$ if we have $C_1 \sqsubseteq \mathbb{C}_1$ and $C_2 \sqsubseteq \mathbb{C}_2$. Here we use $\|$ for the parallel composition of two threads.

The most challenging requirement among the above is to admit compositionality with respect to parallel compositions. Since the refinement or equivalence relation between sequential threads cannot be preserved in general with parallel compositions, we cannot simply adapt existing proof methods for sequential refinement (e.g. [28, 36, 37, 42]) to verify refinement of concurrent programs. Proof techniques based on fully abstract semantics of concurrent programs (e.g., [2, 10]) are compositional, but they assume arbitrary program contexts, which is too strong for practical refinement applications mentioned above. We will explain the challenges in detail in Section 2.1.

1.1.2 On the Correctness of Concurrent Objects

Verifying implementations of concurrent objects is an important application of refinement verification of concurrent programs. Linearizbaility [35] is a widely accepted functional correctness criterion for concurrent objects. It requires the fine-grained implementation of an object operation to have the same effect with (i.e., to "refine") an instantaneous atomic operation. To prove linearizability, the most common approach (e.g., see [3, 14, 65, 66]) is to find a linearization point (LP) in the code of the implementation, and show that it is the single point where the effect of the operation takes place.

However, it is difficult to apply this idea when the LPs are not fixed in the code of object methods. For a large class of lock-free algorithms with *helping* mechanism (e.g., HSY elimination-based stack [30]), the LP of one method might be in the code of some other method. In these algorithms, each thread maintains a descriptor recording all the information required to fulfill its intended operation. When a thread A detects conflicts with another thread B, A may access B's descriptor and help B finish its intended operation first before finishing its own. In this case, B's operation takes effect at a step from A. Thus its LP should *not* be in its own code, but in the code of thread A.

Besides, in optimistic algorithms and lazy algorithms (e.g., Heller et al.'s lazy set [29]), the LPs might depend on unpredictable future interleavings. In those algorithms, a thread may access the shared states as if no interference would occur, and validate the accesses later. If the validation succeeds, it finishes the operation; otherwise it rolls back and retries. Its LP is usually at a prior state access, but only if the later validation succeeds.

Reasoning about algorithms with non-fixed LPs has been a long-standing problem. Most existing work either supports only simple objects with static LPs in the implementation code (e.g., [3, 14, 64]), or lacks formal soundness arguments (e.g., [66]). We will explain in detail the challenges for linearizability verification with non-fixed LPs in Section 5.1.

In addition to linearizability that describes the correctness on functionality, concurrent objects are also expected to satisfy progress properties. The most important progress properties are wait-freedom, lock-freedom and obstruction-freedom for non-blocking implementations, and starvation-freedom and deadlock-freedom for lock-based implementations. These properties describe conditions under which method calls are guaranteed to successfully complete in an execution. For example, lock-freedom guarantees that "infinitely often some method call finishes in a finite number of steps" [33].

Nevertheless, the common informal or semi-formal definitions of the progress properties are difficult to use in a modular and layered program verification because they fail to describe how the progress properties affect clients. In a modular verification of client threads, the concrete implementation C of the object methods should be replaced by the corresponding abstract atomic operations $\mathbb{C}$. The progress properties should then characterize whether and how the behaviors of a client program will be affected if using C instead of $\mathbb{C}$. In particular, we are interested in whether the termination of a client will be preserved.

Filipović et al. [21] proved that linearizability and a contextual refinement are equivalent. Informally, C is a contextual refinement of $\mathbb{C}$, if every observable behavior of any client program using C can also be observed when the client uses $\mathbb{C}$ instead. To obtain equivalence to linearizability, the observable behaviors include I/O events but not divergence (i.e., non-termination). Gotsman and Yang [24] showed that a client program that diverges using a linearizable and *lock-free* object must also diverge when using the abstract operations instead. Their work reveals a connection between lock-freedom and a form of contextual refinement which preserves termination as well as safety properties. But it is unclear how other progress guarantees affect termination of client programs and how they are related to contextual refinements.

1.2 Contributions and Dissertation Outline

By addressing the above problems, this dissertation mainly makes the following contributions. First, we propose a *R*ely-*G*uarantee-based *Sim*ulation (RGSim) as a general proof technique for concurrent program refinement. The novel simulation relation satisfies all the requirements mentioned in Section 1.1.1.

- RGSim parameterizes the simulation with rely/guarantee conditions [38], which specify the interference between threads and their parallel environments. This makes it compositional with respect to parallel compositions, allowing us to decompose refinement proofs for multi-threaded programs into proofs for individual threads. Also, the rely/guarantee conditions can incorporate the assumptions about environments made by specific refinement applications, so RGSim can be applied to solve many practical refinement problems.

- Based on the simulation technique, RGSim focuses on comparing externally observable behaviors (e.g., I/O events) only, which gives us considerable

leeway in the implementations of related programs. The relation is mostly independent of the language details. It can be used to relate programs in different languages with different views of program states and different granularities of atomic state accesses.

Second, as an important application of RGSim, we reduce the problem of verifying concurrent garbage collectors to verifying program transformations, and propose a general GC verification framework, which combines unary Rely-Guarantee reasoning [38] with relational proofs based on RGSim. We have verified the Boehm et al. concurrent garbage collection algorithm [9] using our framework. As far as we know, it is the first time to formally prove the correctness of this algorithm.

Third, we design a Hoare-style program logic for modular and effective verification of linearizability of concurrent objects. It is the first program logic that has a formal soundness proof for linearizability and supports the most challenging objects with non-fixed LPs. Our logic is built upon the unary program logic LRG [20], but we give a relational interpretation of assertions and rely/guarantee conditions. We introduce a lightweight instrumentation mechanism specifically for linearizability proofs and design new logic rules for these auxiliary commands.

- To support the helping mechanism, we collect a pending thread pool as auxiliary state, which is a set of threads and their abstract operations that might be helped. The thread that is currently being verified can use auxiliary commands to help execute the abstract operations in the pending thread pool.

- For future-dependent LPs, we introduce a try-commit mechanism. The **try** clause guesses whether the corresponding abstract operation should be executed and keeps all possibilities, while **commit** chooses a specific possible case when we know which guess is correct later. The try-commit clauses allow us to reason about future-dependent uncertainty without resorting to prophecy variables [1, 66], whose existing semantics (e.g., [1]) is unsuitable for Hoare-style verification.

We also generalize the RGSim relation with the support for non-fixed LPs as the meta-theory of our logic, and show it implies a contextual refinement, which is equivalent to linearizability. Using our logic we successfully verify 12 well-known algorithms, some of which are used in the `java.util.concurrent` package.

Finally, we present a unified framework that characterizes progress properties via contextual refinements. We prove that for linearizable objects, each

progress property is equivalent to a specific type of termination-sensitive contextual refinement. The framework unifies all the five most common progress properties: wait-freedom, lock-freedom, obstruction-freedom, deadlock-freedom and starvation-freedom. It shows exactly how progress properties affect the termination behaviors of client programs, and hence enables modular verification. It also ensures that the verification of a specific contextual refinement guarantees both linearizability and the corresponding progress property for a concurrent object.

Outline of this dissertation.

- Chapter 2 presents the formal definition of RGSim and prove its compositionality.
- Chapter 3 studies some simple applications of RGSim, such as reasoning about optimizations in parallel contexts.
- Chapter 4 proposes a general concurrent GC verification framework based on RGSim, and verify the Boehm et al. concurrent mark-sweep GC [9].
- Chapter 5 describes the program logic and the lightweight instrumentation mechanism for verifying linearizability with non-fixed LPs.
- Chapter 6 presents the unified contextual refinement framework for observing the progress of concurrent objects.

Chapter 2

Rely-Guarantee-Based Simulation

This chapter describes RGSim, a new simulation parameterized with rely/guarantee conditions. It is a general and compositional proof technique for refinement $\sqsubseteq$ of concurrent programs. Following Leroy's approach [42], we define the refinement relation $C \sqsubseteq \mathbb{C}$ saying that C has no more observable behaviors than $\mathbb{C}$.

Below we first analyze the challenges for compositional verification of concurrent program refinement, and explain our approach informally in Section 2.1. Then we give the basic technical settings in Section 2.2, including a formal definition of program refinement. We formulate the RGSim relation in Section 2.3, and prove its compositionality in Section 2.4. Finally we discuss a simple example in Section 2.5. More serious examples can be found in Chapter 3.

2.1 Challenges and Our Approach

The major challenge in verifying refinement $\sqsubseteq$ between concurrent programs is to allow compositional proofs, i.e., we should be able to know $C_1 \parallel C_2 \sqsubseteq \mathbb{C}_1 \parallel \mathbb{C}_2$ if we have $C_1 \sqsubseteq \mathbb{C}_1$ and $C_2 \sqsubseteq \mathbb{C}_2$.

2.1.1 Sequential Refinement Loses Parallel Compositionality

Observable behaviors of sequential imperative programs usually refer to their control effects (e.g., termination and exceptions) and final program states. However, refinement relations defined correspondingly cannot be preserved after parallel compositions. It has been a well-known fact in the compiler community that sound optimizations for sequential programs may change the behaviors of multi-threaded programs [7]. The Dekker's algorithm shown in Figure 2.1(a) has been

```
local r1;              local r2;
x  := 1;               y  := 1;
r1 := y;          ||   r2 := x;
if (r1 = 0) then       if (r2 = 0) then
  critical region        critical region
```

(a) Dekker's mutual exclusion algorithm (initially x = y = 0)

```
x++; || x++;
```

vs.

```
local r1;          local r2;
r1 := x;       ||  r2 := x;
x  := r1 + 1;      x  := r2 + 1;
```

(b) different granularities of atomic operations

Figure 2.1 Equivalence lost after parallel composition.

widely used to demonstrate the problem. Reordering the first two assignment statements of the thread on the left preserves its sequential behaviors, but the whole program can no longer ensure exclusive access to the critical region.

In addition to instruction reordering, the different granularities of atomic operations between the concrete and the abstract programs can also break the compositionality of program equivalence in a concurrent setting. In Figure 2.1(b), the concrete program at the bottom behaves differently from the abstract one at the top (assuming each statement is executed atomically), although the individual threads at the two levels have the same behaviors.

2.1.2 Assuming Arbitrary Environments is Too Strong

The problem with the refinement for sequential programs is that it does not consider the effects of threads' intermediate state accesses on their parallel environments. Previous work on fully abstract semantics of concurrent programs (e.g., [2, 10]) suggests an alternative definition (and a proof method) for refinement. The semantics of a program is modeled as a set of execution traces. Each trace is an interleaving of state transitions made by the program itself and *arbitrary* transitions made by the environment. Then the refinement between programs can be defined as the subset relation between the corresponding trace sets. Since it considers all possible environments, the refinement relation has very nice compositionality, and hence supports compositional verification straightforwardly. But unfortunately it is too strong to formulate and prove the refinement for many

well-known applications. Here are some examples.

- Many concurrent languages (e.g., C++ [8]) do not give semantics to programs with data races (like the examples shown in Figure 2.1). Therefore the compilers only need to guarantee the semantics preservation of data-race-free programs.
- When we prove that a fine-grained implementation of a concurrent object is a refinement of an abstract atomic object, we can assume that all accesses to the object are made through the object's methods only, e.g., a stack object can only be accessed through push and pop methods, and its internal data cannot be arbitrarily updated.
- Usually the implementation of STM (e.g., TL2 [16]) ensures the atomicity of a transaction **atomic**$\{\mathbb{C}\}$ only when there are no data races. Therefore, the refinement from high-level atomic blocks to fine-grained concurrent code assumes data-race-freedom at the abstract level.
- Many garbage-collected languages are type-safe and prohibit operations such as pointer arithmetic. Therefore the garbage collector could make corresponding assumptions about the mutators that run in parallel.

In all these cases, individual threads are allowed to make various assumptions about their environments in the refinement. We do not have to ensure semantics preservation within all contexts.

2.1.3 Languages at the Two Levels May Be Different

The use of different languages at the concrete and the abstract levels makes the formulation and verification of the refinement more difficult. If the concrete and the abstract languages have different views of program states and different atomic primitives, we cannot directly compare the state transitions made by the concrete and the abstract programs. This is another reason that makes the aforementioned subset relation between sets of program traces in fully abstract semantics infeasible. For the same reason, many existing techniques for proving refinement or equivalence of programs in the same language cannot be applied either.

2.1.4 Different Observers Make Different Observations

Concurrency introduces tensions between two kinds of observers: human beings (as external observers) and the parallel program contexts. External observers do not

care about the implementation details of the concrete and the abstract programs. For them, intermediate state accesses (such as memory reads and writes) are silent steps (unobservable), and only external events (such as I/O operations) are observable. On the other hand, state accesses have effects on the parallel program contexts, and are not silent to them.

If the refinement relation relates externally observable event traces only, it cannot have parallel compositionality, as we explained in Section 2.1.1. On the other hand, relating all state accesses of programs is too strong. Any reordering of state accesses or change of atomicity would fail the refinement.

2.1.5 Our Approach

In this paper we propose a *R*ely-*G*uarantee-based *Sim*ulation (RGSim) $\preceq$ between the concrete and the abstract programs. It establishes a weak simulation, ensuring that for every externally observable event made by the concrete program there is a corresponding one at the abstract side. We choose to view intermediate state accesses as silent steps, thus we can relate programs with different implementation details. This also makes our simulation independent of language details.

To support parallel compositionality, our relation takes into account explicitly the expected interference between threads and their parallel environments. Inspired by the Rely-Guarantee verification method [38], we specify the interference using rely/guarantee conditions. In Rely-Guarantee reasoning, the rely condition R of a thread specifies the permitted state transitions that its environment may have, and its guarantee G specifies the possible transitions made by the thread itself. To ensure parallel threads can collaborate, we need to check the interference constraint, i.e., the guarantee of each thread is permitted in the rely of every other. Then we can verify their parallel composition by separately verifying each thread, showing its behaviors under the rely condition indeed satisfy its guarantee. After parallel composition, the threads should be executed under their common environment (i.e., the intersection of their relies) and guarantee all the possible transitions made by them (i.e., the union of their guarantees).

Parameterized with rely/guarantee conditions for the two levels, our relation $(C, \mathcal{R}, \mathcal{G}) \preceq (\mathbb{C}, \mathbb{R}, \mathbb{G})$ talks about not only the concrete program C and the abstract program $\mathbb{C}$, but also the interference $\mathcal{R}$ and $\mathcal{G}$ between C and its low-level environment, and $\mathbb{R}$ and $\mathbb{G}$ between $\mathbb{C}$ and its environment at the abstract level. Informally, $(C, \mathcal{R}, \mathcal{G}) \preceq (\mathbb{C}, \mathbb{R}, \mathbb{G})$ says the executions of C under the environment $\mathcal{R}$ do not exhibit more observable behaviors than the executions of $\mathbb{C}$ under the environment $\mathbb{R}$, and the state transitions of C and $\mathbb{C}$ satisfy $\mathcal{G}$ and $\mathbb{G}$ respectively.

RGSim is now compositional, as long as the threads are composed with well-behaved environments only. The parallel compositionality lemma is in the following form. If we know $(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq (\mathbb{C}_1, \mathbb{R}_1, \mathbb{G}_1)$ and $(C_2, \mathcal{R}_2, \mathcal{G}_2) \preceq (\mathbb{C}_2, \mathbb{R}_2, \mathbb{G}_2)$, and also the interference constraints are satisfied, i.e., $\mathcal{G}_2 \subseteq \mathcal{R}_1$, $\mathcal{G}_1 \subseteq \mathcal{R}_2$, $\mathbb{G}_2 \subseteq \mathbb{R}_1$ and $\mathbb{G}_1 \subseteq \mathbb{R}_2$, we could get

$$(C_1 \,\|\, C_2, \mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2) \preceq (\mathbb{C}_1 \,\|\, \mathbb{C}_2, \mathbb{R}_1 \cap \mathbb{R}_2, \mathbb{G}_1 \cup \mathbb{G}_2)\,.$$

The compositionality of RGSim gives us a proof theory for concurrent program refinement.

Also different from fully abstract semantics for threads, which assumes arbitrary behaviors of environments, RGSim allows us to instantiate the interference $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$ and $\mathbb{G}$ differently for different assumptions about environments, therefore it can support the aforementioned refinement applications. For instance, if we want to prove a refinement that preserves the behaviors of data-race-free programs, we can specify the data-race-freedom in $\mathbb{R}$ and $\mathbb{G}$. Then we are no longer concerned with the examples in Figure 2.1, both of which have data races.

Example. Next we give an example of loop invariant hoisting to illustrate how RGSim works. The following source program C is transformed to the target C_1. We want to prove in a concurrent setting that the transformation is correct, i.e., C_1 is a refinement of C. The formal proofs are given in Section 3.1.2.

Target Code (C_1)		Source Code (C)
`local t;`		`local t;`
`t := x + 1;`	$\Leftarrow$	`while(i < n) {`
`while(i < n) {`		`  t := x + 1;`
`  i := i + t;`		`  i := i + t;`
`}`		`}`

Benton [6] has proved that the optimized code C_1 preserves the *sequential* behaviors of the source C. In a concurrent setting, this optimization is incorrect within arbitrary environments. For instance, if other threads may update `x`, the final values of `i` might be different at the two levels. In fact, this optimization works only when the environments $\mathcal{R}$ at both levels do not update `x` nor `t`. The guarantees $\mathcal{G}$ of both C_1 and C can be specified as arbitrary transitions. Then we can prove the RGSim relation $(C_1, \mathcal{R}, \mathcal{G}) \preceq (C, \mathcal{R}, \mathcal{G})$ and conclude the correctness of the transformation.

$$(\mathit{Events})\quad e \;::=\; \ldots \qquad\qquad (\mathit{Labels})\quad \iota \;::=\; e \mid \tau$$

(a) events and transition labels

$$\begin{array}{llll}
(\mathit{LState}) & \sigma & ::= & \ldots \\
(\mathit{LExpr}) & E & \in & \mathit{LState} \to \mathit{Int}_\perp \\
(\mathit{LBExp}) & B & \in & \mathit{LState} \to \{\mathbf{true}, \mathbf{false}\}_\perp \\
(\mathit{LInstr}) & c & \in & \mathit{LState} \rightharpoonup \mathscr{P}((\mathit{Labels} \times \mathit{LState}) \cup \{\mathbf{abort}\}) \\
(\mathit{LStmt}) & C & ::= & \mathbf{skip} \mid c \mid C_1; C_2 \mid \mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2 \\
 & & \mid & \mathbf{while}\ (B)\ C \mid C_1 \,\|\, C_2
\end{array}$$

(b) low-level language

$$\begin{array}{llll}
(\mathit{HState}) & \Sigma & ::= & \ldots \\
(\mathit{HExpr}) & \mathbb{E} & \in & \mathit{HState} \to \mathit{Int}_\perp \\
(\mathit{HBExp}) & \mathbb{B} & \in & \mathit{HState} \to \{\mathbf{true}, \mathbf{false}\}_\perp \\
(\mathit{HInstr}) & \mathbb{c} & \in & \mathit{HState} \rightharpoonup \mathscr{P}((\mathit{Labels} \times \mathit{HState}) \cup \{\mathbf{abort}\}) \\
(\mathit{HStmt}) & \mathbb{C} & ::= & \boldsymbol{skip} \mid \mathbb{c} \mid \mathbb{C}_1;;\mathbb{C}_2 \mid \boldsymbol{if}\ \mathbb{B}\ \boldsymbol{then}\ \mathbb{C}_1\ \boldsymbol{else}\ \mathbb{C}_2 \\
 & & \mid & \boldsymbol{while}\ \mathbb{B}\ \boldsymbol{do}\ \mathbb{C} \mid \mathbb{C}_1 \|\!| \mathbb{C}_2
\end{array}$$

(c) high-level language

Figure 2.2 Generic languages at concrete and abstract levels.

2.2 Basic Technical Settings

In this section, we present the programming languages at the concrete and abstract levels. Then we define the basic refinement relation $\sqsubseteq$, which naturally says the concrete program has no more externally observable event traces than the abstract one. Our RGSim relation, which will be formally defined in Section 2.3, is proposed as a proof technique for this simple and intuitive refinement $\sqsubseteq$.

2.2.1 Languages

Following standard simulation techniques, we model the semantics of concrete and abstract programs as labeled transition systems. Before showing the languages, we first define events and labels in Figure 2.2(a). We leave the set of (externally observable) events e unspecified here. It can be instantiated by program verifiers, depending on their interest (e.g., input/output events). A label ι that will be associated with a state transition is either an event or τ, which means the corresponding transition does not generate any event (i.e., a silent step).

$$\frac{(\iota, \Sigma') \in \mathbb{c}\ \Sigma}{(\mathbb{c}, \Sigma) \xrightarrow{\iota} (\textbf{\textit{skip}}, \Sigma')} \qquad \frac{\textbf{abort} \in \mathbb{c}\ \Sigma}{(\mathbb{c}, \Sigma) \longrightarrow \textbf{abort}} \qquad \frac{\Sigma \notin dom(\mathbb{c})}{(\mathbb{c}, \Sigma) \longrightarrow (\mathbb{c}, \Sigma)}$$

$$\frac{}{(\textbf{\textit{skip}} \,|||\, \textbf{\textit{skip}}, \Sigma) \longrightarrow (\textbf{\textit{skip}}, \Sigma)} \qquad \frac{(\mathbb{C}_1, \Sigma) \xrightarrow{\iota} (\mathbb{C}_1', \Sigma')}{(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma) \xrightarrow{\iota} (\mathbb{C}_1' \,|||\, \mathbb{C}_2, \Sigma')}$$

$$\frac{(\mathbb{C}_2, \Sigma) \xrightarrow{\iota} (\mathbb{C}_2', \Sigma')}{(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma) \xrightarrow{\iota} (\mathbb{C}_1 \,|||\, \mathbb{C}_2', \Sigma')} \qquad \frac{(\mathbb{C}_1, \Sigma) \longrightarrow \textbf{abort} \quad \text{or} \quad (\mathbb{C}_2, \Sigma) \longrightarrow \textbf{abort}}{(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma) \longrightarrow \textbf{abort}}$$

Figure 2.3 Selected operational semantics rules of the high-level language.

The language for the concrete programs, which we also call the low-level language, is shown in Figure 2.2(b). We abstract away the forms of states, expressions and primitive instructions in the language. An arithmetic expression E is modeled as a function from states to integers lifted with an undefined value $\perp$. The boolean expression B is modeled similarly. An instruction c is a partial function from states to sets of label and state pairs, describing the state transitions and the events it generates. We use $\mathscr{P}(\_)$ to denote the power set. Unsafe executions lead to **abort**. Note that the semantics of an instruction could be non-deterministic. Moreover, it might be undefined on some states, making it possible to model blocking operations such as requesting a lock.

Statements C are either primitive instructions or compositions of them. **skip** is a special statement used as a flag to show the end of executions. When it is sequentially composed with other statements, it has no computational effects. A single-step execution of statements is modeled as a labeled transition $(C, \sigma) \xrightarrow{\iota}_L (C', \sigma')$. The step **abort**s if an unsafe instruction is executed.

The high-level language (i.e., the language for the abstract programs) is defined similarly in Figure 2.2(c), but it is important to note that its states and primitive instructions may be different from those in the low-level language. The compound statements are almost the same as their low-level counterparts. $\mathbb{C}_1;;\mathbb{C}_2$ and $\mathbb{C}_1|||\mathbb{C}_2$ are sequential and parallel compositions of $\mathbb{C}_1$ and $\mathbb{C}_2$ respectively. Note that we choose to use the same set of compound statements in the two languages for simplicity only. This is not required by our simulation relation, although the analogous program constructs of the two languages (e.g., parallel compositions $C_1 \parallel C_2$ and $\mathbb{C}_1 \,|||\, \mathbb{C}_2$) make it convenient for us to discuss the compositionality later.

Figure 2.3 shows part of the definition of $(\mathbb{C}, \Sigma) \xrightarrow{\iota}_H (\mathbb{C}', \Sigma')$, which gives the high-level operational semantics of statements. We often omit the subscript

H (or L) in the labeled transition and the label on top of the arrow when it is τ. The semantics is mostly standard. We only show the rules for primitive instructions and parallel compositions here. Note that when a primitive instruction $\mathbb{c}$ is blocked at state Σ (i.e., $\Sigma \notin dom(\mathbb{c})$), we let the program configuration $(\mathbb{c}, \Sigma)$ reduce to itself. For example, the instruction `lock(l)` would be blocked when `l` is not 0, making it be repeated until `l` becomes 0; whereas `unlock(l)` simply sets `l` to 0 at any time and would never be blocked. Primitive instructions in the high-level and low-level languages are *atomic* in the interleaving semantics. The operational semantics of the low-level language is defined similarly and omitted. Below we use $(\mathbb{C}, \Sigma) \longrightarrow^* (\mathbb{C}', \Sigma')$ for zero or multiple-step transitions with no events generated, and $(\mathbb{C}, \Sigma) \xrightarrow{e}{}^* (\mathbb{C}', \Sigma')$ for multiple-step transitions with *only one* event e generated. Both notations are overloaded at the low level.

2.2.2 Event Trace Refinement

Now we can formally define the refinement relation $\sqsubseteq$ that relates the set of externally observable event traces generated by the low-level and the high-level programs. A trace $\mathcal{E}$ is a sequence of events e, and may end with a termination marker **term** or a fault marker **abort**. We write ϵ for the empty sequence and $::$ for the concatenation of two sequences.

$$(\mathit{EvtTrace}) \quad \mathcal{E} \ ::= \ \epsilon \mid \mathbf{term} \mid \mathbf{abort} \mid e :: \mathcal{E}$$

Definition 2.1 (Event Trace Set)**.** $\mathit{ETrSet}_n(C, \sigma)$ represents a set of external event traces produced by C in n steps from the state σ.

1. $\mathit{ETrSet}_0(C, \sigma) \stackrel{\text{def}}{=} \{\epsilon\}$;
2. $\mathit{ETrSet}_{n+1}(C, \sigma) \stackrel{\text{def}}{=}$
$$\{\mathcal{E} \mid (C, \sigma) \longrightarrow (C', \sigma') \wedge \mathcal{E} \in \mathit{ETrSet}_n(C', \sigma')$$
$$\vee\ (C, \sigma) \xrightarrow{e} (C', \sigma') \wedge \mathcal{E}' \in \mathit{ETrSet}_n(C', \sigma') \wedge \mathcal{E} = e :: \mathcal{E}'$$
$$\vee\ (C, \sigma) \longrightarrow \mathbf{abort} \wedge \mathcal{E} = \mathbf{abort}$$
$$\vee\ C = \mathbf{skip} \wedge \mathcal{E} = \mathbf{term}\}.$$

We define $\mathit{ETrSet}(C, \sigma)$ as $\bigcup_n \mathit{ETrSet}_n(C, \sigma)$.

We overload the notation and use $\mathit{ETrSet}(\mathbb{C}, \Sigma)$ for the high-level language. Note that we treat **abort** as a specific behavior instead of undefined arbitrary behaviors. The choices should depend on applications. The ideas in the paper should also apply for the latter setting, though we need to change our refinement and simulation relations defined below.

Then we define an event trace refinement as the subset relation between event trace sets, which is similar to Leroy's refinement property [42].

Definition 2.2 (Event Trace Refinement)**.** We say (C, σ) is an event trace refinement of $(\mathbb{C}, \Sigma)$, written as $(C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)$, if and only if

$$ETrSet(C, \sigma) \subseteq ETrSet(\mathbb{C}, \Sigma)\,.$$

The refinement is defined for program configurations (i.e., pairs of code and states) instead of for code only because the initial states may affect the behaviors of programs. Suppose a state transformation $\mathbf{T}$ translates the initial high-level state Σ to the initial low-level state σ, i.e., $\sigma = \mathbf{T}(\Sigma)$ holds. We can define the refinement $C \sqsubseteq_{\mathbf{T}} \mathbb{C}$ as follows.

$$C \sqsubseteq_{\mathbf{T}} \mathbb{C} \quad \stackrel{\text{def}}{=} \quad \forall \sigma, \Sigma.\ \sigma = \mathbf{T}(\Sigma) \implies (C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)\,.$$

If the transformation $\mathbf{T}$ translates code as well, we can formulate its correctness in Eq. (2.1). Here we overload the notation and use $\mathbf{T}(\mathbb{C})$ to represent the code transformation.

$$\mathsf{Correct}(\mathbf{T}) \quad \stackrel{\text{def}}{=} \quad \forall C, \mathbb{C}.\ C = \mathbf{T}(\mathbb{C}) \implies C \sqsubseteq_{\mathbf{T}} \mathbb{C}\,. \tag{2.1}$$

From the event trace refinement definition, we can derive an *event trace equivalence* relation by requiring both directions hold.

$$(C, \sigma) \approx (\mathbb{C}, \Sigma) \quad \stackrel{\text{def}}{=} \quad (C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma) \wedge (\mathbb{C}, \Sigma) \sqsubseteq (C, \sigma)\,.$$

For the state transformation $\mathbf{T}$, we define

$$C \approx_{\mathbf{T}} \mathbb{C} \quad \stackrel{\text{def}}{=} \quad \forall \sigma, \Sigma.\ \sigma = \mathbf{T}(\Sigma) \Longrightarrow (C, \sigma) \approx (\mathbb{C}, \Sigma)\,.$$

2.3 Definition of the RGSim Relation

The event trace refinement is defined directly over the externally observable behaviors of programs. It is intuitive, and also abstract in that it is independent of language details. However, as we explained before, it is *not* compositional with respect to parallel compositions. In this section we propose RGSim, which can be viewed as a compositional proof technique that allows us to derive the simple event trace refinement.

Our co-inductively defined RGSim relation is in the form of $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, which is a simulation between program configurations (C, σ) and $(\mathbb{C}, \Sigma)$. It is parametrized with the rely and guarantee conditions at the low level and the high level, which are binary relations over states.

(a) α-related transitions (b) side condition of TRANS

Figure 2.4 Related transitions.

$$\mathcal{R}, \mathcal{G} \in \mathscr{P}(\mathit{LState} \times \mathit{LState}), \quad \mathbb{R}, \mathbb{G} \in \mathscr{P}(\mathit{HState} \times \mathit{HState}).$$

The simulation also takes two additional parameters: the *step invariant* α and the *postcondition* γ, which are both relations between the low-level and the high-level states.

$$\alpha, \gamma \in \mathscr{P}(\mathit{LState} \times \mathit{HState}).$$

Before we formally define RGSim in Definition 2.4, we first introduce the α-related transitions as follows.

Definition 2.3 (α-Related Transitions)**.** $\langle \mathcal{R}, \mathbb{R} \rangle_\alpha \stackrel{\text{def}}{=}$

$$\{((\sigma, \sigma'), (\Sigma, \Sigma')) \mid (\sigma, \sigma') \in \mathcal{R} \wedge (\Sigma, \Sigma') \in \mathbb{R} \wedge (\sigma, \Sigma) \in \alpha \wedge (\sigma', \Sigma') \in \alpha\}.$$

$\langle \mathcal{R}, \mathbb{R} \rangle_\alpha$ represents a set of the α-related transitions in $\mathcal{R}$ and $\mathbb{R}$, putting together the corresponding transitions in $\mathcal{R}$ and $\mathbb{R}$ that can be related by α, as illustrated in Figure 2.4(a). $\langle \mathcal{G}, \mathbb{G} \rangle_\alpha$ is defined in the same way.

Definition 2.4 (RGSim)**.** Whenever $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, then $(\sigma, \Sigma) \in \alpha$ and the following are true.

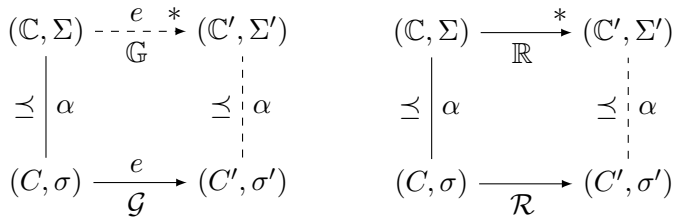
1. If $(C, \sigma) \longrightarrow (C', \sigma')$, then there exist $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma) \longrightarrow^* (\mathbb{C}', \Sigma')$, $((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha$ and $(C', \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}', \Sigma', \mathbb{R}, \mathbb{G})$.

2. If $(C, \sigma) \xrightarrow{e} (C', \sigma')$, then there exist $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma) \xrightarrow{e}{}^* (\mathbb{C}', \Sigma')$, $((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha$ and $(C', \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}', \Sigma', \mathbb{R}, \mathbb{G})$.

3. If $C = \mathbf{skip}$, then there exists Σ' such that $(\mathbb{C}, \Sigma) \longrightarrow^* (\boldsymbol{skip}, \Sigma')$, $((\sigma, \sigma), (\Sigma, \Sigma')) \in \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha$, $(\sigma, \Sigma') \in \gamma$ and $\gamma \subseteq \alpha$.

4. If $(C, \sigma) \longrightarrow \mathbf{abort}$, then $(\mathbb{C}, \Sigma) \longrightarrow^* \mathbf{abort}$.

5. If $((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha$, then $(C, \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma', \mathbb{R}, \mathbb{G})$.



(a) program steps (b) environment steps

Figure 2.5 Simulation diagrams of RGSim.

Then, $(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})$ iff
for any σ and Σ, if $(\sigma, \Sigma) \in \zeta$, then $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$. Here the *precondition* $\zeta \in \mathscr{P}(\mathit{LState} \times \mathit{HState})$ is used to relate the initial states σ and Σ.

Informally, $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$ says the low-level configuration (C, σ) is simulated by the high-level configuration $(\mathbb{C}, \Sigma)$ with behaviors $\mathcal{G}$ and $\mathbb{G}$ respectively, no matter how their environments $\mathcal{R}$ and $\mathbb{R}$ interfere with them. It requires the following hold for every execution of C.

- Starting from α-related states, each step of C corresponds to zero or multiple steps of $\mathbb{C}$, and the resulting states are α-related too. If an external event is produced in the step of C, the same event should be produced by $\mathbb{C}$. We show the simulation diagram with events generated by the program steps in Figure 2.5(a), where solid lines denote hypotheses and dashed lines denote conclusions, following Leroy's notations [42].

- The α relation reflects the abstractions from the low-level machine model to the high-level one, and is preserved by the related transitions at the two levels (so it is an *invariant*). For instance, when verifying a fine-grained implementation of sets, the α relation may relate a concrete representation in memory (e.g., a linked-list) at the low level to the corresponding abstract mathematical set at the high level.

- The corresponding transitions of C and $\mathbb{C}$ need to be in $\langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha$. That is, for each step of C, its state transition should satisfy the guarantee $\mathcal{G}$, and the corresponding transition made by the multiple steps of $\mathbb{C}$ should be in the transitive closure of $\mathbb{G}$. The guarantees are abstractions of the programs' behaviors. As we will show later in the PAR rule in Figure 2.7, they will serve as the rely conditions of the sibling threads at the time of

parallel compositions. Note that we do not need each step of $\mathbb{C}$ to be in $\mathbb{G}$, although we could do so. This is because we only care about the coarse-grained behaviors (with mumbling) of $\mathbb{C}$ that are used to simulate C. We will explain more by the example of Eq. (2.2) in Section 2.4.

- If C terminates, then $\mathbb{C}$ terminates as well, and the final states should be related by the postcondition γ. We require $\gamma \subseteq \alpha$, i.e., the final state relation is not weaker than the step invariant.

- C is not safe only if $\mathbb{C}$ is not safe either. This means the safety of the high-level program should be preserved at the low level.

- Whatever the low-level environment $\mathcal{R}$ and the high-level one $\mathbb{R}$ do, as long as the state transitions are α-related, they should not affect the simulation between C and $\mathbb{C}$, as shown in Figure 2.5(b). Here a step in $\mathcal{R}$ may correspond to zero or multiple steps of $\mathbb{R}$. Note that different from the program steps, some steps of $\mathcal{R}$ may not correspond to steps of $\mathbb{R}$. On the other hand, only requiring that $\mathcal{R}$ be simulated by $\mathbb{R}$ (i.e., every step of $\mathcal{R}$ should correspond to steps of $\mathbb{R}$, as shown in (2.3) in Section 2.4) is not sufficient for parallel compositionality, which we will explain later in Section 2.4.

Then based on the simulation, we hide the states by the precondition ζ and define the RGSim relation between programs only. By the definition we know $\zeta \subseteq \alpha$ if $(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})$, i.e., the precondition needs to be no weaker than the step invariant. Usually in practice α is very weak and naturally implied by the pre- and post-conditions ζ and γ, e.g., ζ and γ are the same as α in most examples in Chapter 3.

RGSim is adequate with respect to the event trace refinement (Definition 2.2). That is, $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$ ensures that (C, σ) does not have more observable behaviors than $(\mathbb{C}, \Sigma)$.

Theorem 2.5 (Adequacy of RGSim)**.** *If there exist $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$, $\mathbb{G}$, α and γ such that $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, then $(C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)$.*

The adequacy theorem shows that RGSim is a proof technique for the simple and natural refinement $\sqsubseteq$, which is what we ultimately care about. The theorem can be proved by first strengthening the relies to the identity transitions and weakening the guarantees to the universal relations. Then we prove that the resulting simulation under identity environments implies the event trace refinement. The mechanized proof in the Coq proof assistant [13] is available online [44].

$\mathsf{InitRel}_{\mathbf{T}}(\zeta) \stackrel{\text{def}}{=} \forall \sigma, \Sigma.\ \sigma = \mathbf{T}(\Sigma) \implies (\sigma, \Sigma) \in \zeta$

$B \Leftrightarrow \mathbb{B} \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid B\ \sigma = \mathbb{B}\ \Sigma\} \qquad B \mathbb{A} \mathbb{B} \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid B\ \sigma \wedge \mathbb{B}\ \Sigma\}$

$\mathsf{Intuit}(\alpha) \stackrel{\text{def}}{=} \forall \sigma, \Sigma, \sigma', \Sigma'.\ (\sigma, \Sigma) \in \alpha \wedge \sigma \subseteq \sigma' \wedge \Sigma \subseteq \Sigma' \implies (\sigma', \Sigma') \in \alpha$

$\eta \# \alpha \stackrel{\text{def}}{=} (\eta \cap \alpha) \subseteq (\eta \uplus \alpha) \qquad \alpha^{-1} \stackrel{\text{def}}{=} \{(\Sigma, \sigma) \mid (\sigma, \Sigma) \in \alpha\}$

$\alpha \uplus \beta \stackrel{\text{def}}{=} \{(\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2) \mid (\sigma_1, \Sigma_1) \in \alpha \wedge (\sigma_2, \Sigma_2) \in \beta\}$

$\beta \circ \alpha \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \exists \theta.\ (\sigma, \theta) \in \alpha \wedge (\theta, \Sigma) \in \beta\}$

$\mathsf{Id} \stackrel{\text{def}}{=} \{(\sigma, \sigma) \mid \sigma \in \mathit{LState}\} \qquad \mathsf{True} \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma, \sigma' \in \mathit{LState}\}$

$\mathrm{R_M}\ \mathsf{isMidOf}\ (\alpha, \beta; \mathcal{R}, \mathbb{R}) \stackrel{\text{def}}{=}$
$\forall \sigma, \sigma', \Sigma, \Sigma'.\ ((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{R}, \mathbb{R} \rangle_{\beta \circ \alpha} \Longrightarrow \forall \theta.\ (\sigma, \theta) \in \alpha \wedge (\theta, \Sigma) \in \beta \implies$
$\exists \theta'.\ ((\sigma, \sigma'), (\theta, \theta')) \in \langle \mathcal{R}, \mathrm{R_M} \rangle_\alpha \wedge ((\theta, \theta'), (\Sigma, \Sigma')) \in \langle \mathrm{R_M}, \mathbb{R} \rangle_\beta$

Figure 2.6 Auxiliary definitions for RGSim.

When the initial state for the low-level program is transformed from the initial high-level state, we have the following Corollary 2.6. We use $\mathsf{InitRel}_{\mathbf{T}}(\zeta)$ (defined in Figure 2.6) to say that the transformation $\mathbf{T}$ over states ensures the binary precondition ζ.

Corollary 2.6. *If there exist* $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$, $\mathbb{G}$, α, ζ *and* γ *such that* $\mathsf{InitRel}_{\mathbf{T}}(\zeta)$ *and* $(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha; \zeta \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})$, *then* $C \sqsubseteq_{\mathbf{T}} \mathbb{C}$.

2.4 Compositionality Rules

RGSim is compositional with respect to various program constructs, including parallel compositions. We present the compositionality rules in Figure 2.7, which gives us a relational proof method for concurrent program refinement.

As in the Rely-Guarantee logic [38], we require that the pre- and post-conditions be *stable* under the interference from the environments. Here we introduce the concept of stability of a relation ζ with respect to a set of transition pairs $\Lambda \in \mathscr{P}((\mathit{LState} \times \mathit{LState}) \times (\mathit{HState} \times \mathit{HState}))$.

Definition 2.7 (Stability)**.** $\mathsf{Sta}(\zeta, \Lambda)$ holds iff
for any σ, σ', Σ and Σ', if $(\sigma, \Sigma) \in \zeta$ and $((\sigma, \sigma'), (\Sigma, \Sigma')) \in \Lambda$, then $(\sigma', \Sigma') \in \zeta$.

Usually we need $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$, which says whenever ζ holds initially and $\mathcal{R}$ and $\mathbb{R}^*$ perform related actions, the resulting states still satisfy ζ. By unfolding $\langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha$, we could see α itself is stable with respect to any α-related transitions, i.e., $\mathsf{Sta}(\alpha, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$ always holds. Another simple example is given below, where

$$\frac{\zeta \subseteq \alpha}{(\mathbf{skip}, \mathcal{R}, \mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\zeta} (\boldsymbol{skip}, \mathbb{R}, \mathsf{Id})} \text{ (SKIP)}$$

$$\frac{(C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}_1, \mathbb{R}, \mathbb{G}) \qquad (C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma\ltimes\eta} (\mathbb{C}_2, \mathbb{R}, \mathbb{G})}{(C_1; C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\eta} (\mathbb{C}_1; ; \mathbb{C}_2, \mathbb{R}, \mathbb{G})} \text{ (SEQ)}$$

$$\frac{\begin{array}{c}(C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_1\ltimes\gamma} (\mathbb{C}_1, \mathbb{R}, \mathbb{G}) \qquad (C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_2\ltimes\gamma} (\mathbb{C}_2, \mathbb{R}, \mathbb{G}) \\ \zeta \subseteq (B \Leftrightarrow \mathbb{B}) \qquad \zeta_1 = (\zeta \cap (B \mathbb{\wedge} \mathbb{B})) \qquad \zeta_2 = (\zeta \cap (\neg B \mathbb{\wedge} \neg \mathbb{B})) \qquad \zeta \subseteq \alpha\end{array}}{(\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\boldsymbol{if}\ \mathbb{B}\ \boldsymbol{then}\ \mathbb{C}_1\ \boldsymbol{else}\ \mathbb{C}_2, \mathbb{R}, \mathbb{G})} \text{ (IF)}$$

$$\frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma_1\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \\ \gamma \subseteq (B \Leftrightarrow \mathbb{B}) \qquad \gamma_1 = (\gamma \cap (B \mathbb{\wedge} \mathbb{B})) \qquad \gamma_2 = (\gamma \cap (\neg B \mathbb{\wedge} \neg \mathbb{B}))\end{array}}{(\mathbf{while}\ (B)\ C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma\ltimes\gamma_2} (\boldsymbol{while}\ \mathbb{B}\ \boldsymbol{do}\ \mathbb{C}, \mathbb{R}, \mathbb{G})} \text{ (WHILE)}$$

$$\frac{\begin{array}{c}(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma_1} (\mathbb{C}_1, \mathbb{R}_1, \mathbb{G}_1) \qquad (C_2, \mathcal{R}_2, \mathcal{G}_2) \preceq_{\alpha;\zeta\ltimes\gamma_2} (\mathbb{C}_2, \mathbb{R}_2, \mathbb{G}_2) \\ \mathcal{G}_1 \subseteq \mathcal{R}_2 \qquad \mathcal{G}_2 \subseteq \mathcal{R}_1 \qquad \mathbb{G}_1 \subseteq \mathbb{R}_2 \qquad \mathbb{G}_2 \subseteq \mathbb{R}_1\end{array}}{(C_1 \parallel C_2, \mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2) \preceq_{\alpha;\zeta\ltimes(\gamma_1\cap\gamma_2)} (\mathbb{C}_1 \|\| \mathbb{C}_2, \mathbb{R}_1 \cap \mathbb{R}_2, \mathbb{G}_1 \cup \mathbb{G}_2)} \text{ (PAR)}$$

$$\frac{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \qquad (\zeta \cup \gamma) \subseteq \alpha' \subseteq \alpha \qquad \mathsf{Sta}(\alpha', \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha)}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha';\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})} \text{ (STREN-}\alpha\text{)}$$

$$\frac{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \qquad \alpha \subseteq \alpha' \qquad \mathsf{Sta}(\alpha, \langle \mathcal{R}, \mathbb{R}^* \rangle_{\alpha'})}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha';\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})} \text{ (WEAKEN-}\alpha\text{)}$$

$$\frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \\ \zeta' \subseteq \zeta \qquad \gamma \subseteq \gamma' \subseteq \alpha \qquad \mathcal{R}' \subseteq \mathcal{R} \qquad \mathbb{R}' \subseteq \mathbb{R} \qquad \mathcal{G} \subseteq \mathcal{G}' \qquad \mathbb{G} \subseteq \mathbb{G}'\end{array}}{(C, \mathcal{R}', \mathcal{G}') \preceq_{\alpha;\zeta'\ltimes\gamma'} (\mathbb{C}, \mathbb{R}', \mathbb{G}')} \text{ (CONSEQ)}$$

$$\frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \qquad \eta \subseteq \beta \qquad \eta \# \{\zeta, \gamma, \alpha\} \\ \mathsf{Intuit}(\{\alpha, \zeta, \gamma, \beta, \eta, \mathcal{R}, \mathbb{R}, \mathcal{R}_1, \mathbb{R}_1\}) \qquad \mathsf{Sta}(\eta, \{\langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha, \langle \mathcal{R}_1, \mathbb{R}_1^* \rangle_\beta\})\end{array}}{(C, \mathcal{R} \uplus \mathcal{R}_1, \mathcal{G} \uplus \mathcal{G}_1) \preceq_{\alpha\uplus\beta;(\zeta\uplus\eta)\ltimes(\gamma\uplus\eta)} (\mathbb{C}, \mathbb{R} \uplus \mathbb{R}_1, \mathbb{G} \uplus \mathbb{G}_1)} \text{ (FRAME)}$$

$$\frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathrm{C_M}, \mathrm{R_M}, \mathrm{G_M}) \\ (\mathrm{C_M}, \mathrm{R_M}, \mathrm{G_M}) \preceq_{\beta;\delta\ltimes\eta} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \qquad \mathrm{R_M}\ \mathsf{isMidOf}\ (\alpha, \beta; \mathcal{R}, \mathbb{R}^*)\end{array}}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\beta\circ\alpha;(\delta\circ\zeta)\ltimes(\eta\circ\gamma)} (\mathbb{C}, \mathbb{R}, \mathbb{G})} \text{ (TRANS)}$$

Figure 2.7 Compositionality rules for RGSim. At each proof rule, we implicitly assume that the pre- and post-conditions are stable under the environments' interference (Definition 2.7), and the relies and guarantees are closed over identity transitions.

both environments could increment $\mathtt{x}$ and the unary stable assertion $\mathtt{x} \geq 0$ is lifted to the relation ζ. We can prove $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$.

$$\zeta \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x}) \wedge \sigma(\mathtt{x}) \geq 0\} \qquad \alpha \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}$$
$$\mathcal{R} \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma' = \sigma\{\mathtt{x} \rightsquigarrow \sigma(\mathtt{x}) + 1\}\} \qquad \mathbb{R} \stackrel{\text{def}}{=} \{(\Sigma, \Sigma') \mid \Sigma' = \Sigma\{\mathtt{x} \rightsquigarrow \Sigma(\mathtt{x}) + 1\}\}$$

Stability of the pre- and post-conditions under the environments' interference is assumed as an implicit side condition at every proof rule in Figure 2.7, e.g., we assume $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$ in the SKIP rule. We also require implicitly that the relies and guarantees are closed over identity transitions, since stuttering steps will not affect observable event traces.

In Figure 2.7, the rules SKIP, SEQ, IF and WHILE reveal a high degree of similarity to the corresponding inference rules in Hoare logic. In the SEQ rule, γ serves as the postcondition of C_1 and $\mathbb{C}_1$ and the precondition of C_2 and $\mathbb{C}_2$ at the same time. The IF rule requires the boolean conditions of both sides to be evaluated to the same value under the precondition ζ. The definitions of the sets $B \Leftrightarrow \mathbb{B}$ and $B \nLeftrightarrow \mathbb{B}$ are given in Figure 2.6. The rule also requires the precondition ζ to imply the step invariant α. In the WHILE rule, the γ relation is viewed as a loop invariant preserved at the loop entry point, and needs to ensure $B \Leftrightarrow \mathbb{B}$.

Parallel compositionality. The PAR rule shows parallel compositionality of RGSim. The interference constraints say that two threads can be composed in parallel if one thread's guarantee implies the rely of the other. After parallel composition, they are expected to run in the common environment and their guaranteed behaviors contain each single thread's behaviors.

Note that, although RGSim does not require every step of the high-level program to be in its guarantee (see the first two conditions in Definition 2.4), this relaxation does not affect the parallel compositionality. This is because the low-level program could have less behaviors than the high-level one. To let $\mathbb{C}_1 \,|||\, \mathbb{C}_2$ simulate $C_1 \parallel C_2$, we only need a subset of the interleavings of $\mathbb{C}_1$ and $\mathbb{C}_2$ to simulate those of C_1 and C_2. Thus the high-level relies and guarantees need to ensure the existence of those interleavings only. Below we give a simple example to explain this subtle issue. We can prove

$$(\mathtt{x:=x+2}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathtt{x:=x+1;x:=x+1}, \mathbb{R}, \mathbb{G})\,. \tag{2.2}$$

The relies and the guarantees $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$ and $\mathbb{G}$ say $\mathtt{x}$ can be increased by 2. And α, ζ and γ relate $\mathtt{x}$ of the two sides.

$$\mathcal{R} = \mathcal{G} \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma' = \sigma \vee \sigma' = \sigma\{\mathtt{x} \rightsquigarrow \sigma(\mathtt{x}) + 2\}\};$$
$$\mathbb{R} = \mathbb{G} \stackrel{\text{def}}{=} \{(\Sigma, \Sigma') \mid \Sigma' = \Sigma \vee \Sigma' = \Sigma\{\mathtt{x} \rightsquigarrow \Sigma(\mathtt{x}) + 2\}\};$$
$$\alpha = \zeta = \gamma \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}.$$

Note that the high-level program is actually finer-grained than its guarantee, but to prove Eq. (2.2) we only need the execution in which it goes two steps to the end without interference from its environment. Also we prove $(\texttt{print(x)}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\texttt{print(x)}, \mathbb{R}, \mathbb{G})$. Here we use the instruction $\mathbf{print}(E)$ to observe the value of $\mathtt{x}$, which will produce an external event $\mathbf{out}(n)$ if E evaluates to n. Then by the PAR rule, we get

$$(\texttt{x:=x+2} \,\|\, \texttt{print(x)}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\texttt{(x:=x+1;x:=x+1)} \,\|\|\, \texttt{print(x)}, \mathbb{R}, \mathbb{G}).$$

The result does not violate the natural meaning of refinement. All the possible external events produced by the low-level side can also be produced by the high-level side, although the latter could have more external behaviors due to its finer granularity.

Another subtlety in the RGSim definition is with the fifth condition over the environments, which is crucial for parallel compositionality. One may think a more natural alternative to this condition is to require that $\mathcal{R}$ be simulated by $\mathbb{R}$, as shown below.

$$\begin{array}{l}\text{If } (\sigma, \sigma') \in \mathcal{R}\text{, then there exists } \Sigma' \text{ such that} \\ (\Sigma, \Sigma') \in \mathbb{R}^* \text{ and } (C, \sigma', \mathcal{R}, \mathcal{G}) \preceq'_{\alpha;\gamma} (\mathbb{C}, \Sigma', \mathbb{R}, \mathbb{G}).\end{array} \tag{2.3}$$

We refer to this modified simulation definition as $\preceq'$. Unfortunately, $\preceq'$ does not have parallel compositionality. As a counter-example, suppose the invariant α says the low-level $\mathtt{x}$ is not greater than the high-level $\mathtt{x}$.

$$\alpha \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) \leq \Sigma(\mathtt{x})\},$$

Then we could prove the following.

$$(\texttt{x:=x+1},\ \mathsf{Id},\ \mathsf{True}) \preceq'_{\alpha;\alpha\ltimes\alpha} (\texttt{x:=x+2},\ \mathsf{Id},\ \mathsf{True}); \tag{2.4}$$

$$(\texttt{x:=0;print(x)},\ \mathsf{True},\ \mathsf{Id}) \preceq'_{\alpha;\alpha\ltimes\alpha} (\texttt{x:=0;print(x)},\ \mathsf{True}, \mathsf{Id}). \tag{2.5}$$

Here we use Id and True (defined in Figure 2.6) for the sets of identity transitions and arbitrary transitions respectively, and overload the notations at the low level to the high level. However, the following refinement does *not* hold after parallel composition.

$$\begin{array}{l}(\texttt{x:=x+1} \,\|\, (\texttt{x:=0;print(x)}),\ \mathsf{Id},\ \mathsf{True}) \\ \preceq'_{\alpha;\alpha\ltimes\alpha} (\texttt{x:=x+2} \,\|\|\, (\texttt{x:=0;print(x)}),\ \mathsf{Id},\ \mathsf{True}).\end{array}$$

This is because the rely $\mathcal{R}$ (or $\mathbb{R}$) is an abstraction of all the permitted behaviors in the environment of a thread $\mathtt{t}$. Any thread $\mathtt{t}'$ whose behaviors are allowed in $\mathcal{R}$ (or $\mathbb{R}$) can run in parallel with $\mathtt{t}$. Thus to obtain parallel compositionality, we have to ensure that the simulation is preserved with *any* possible sibling thread $\mathtt{t}'$. With *our* definition $\preceq$, the simulation of Eq. (2.5) is not provable, because after some α-related transitions of environments, the low-level program may print out a value smaller than the one printed at the high level.

Other rules. We also develop some other useful rules about RGSim. For example, the STREN-α rule allows us to replace the invariant α by a stronger invariant α'. We need to check that α' is indeed an invariant preserved by the α-related program steps, i.e., $\mathsf{Sta}(\alpha', \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha)$ holds. Symmetrically, the WEAKEN-α rule requires α to be preserved by environment steps related by the weaker invariant α'. As in the Rely-Guarantee logic [38], the pre- and post-conditions, the relies and the guarantees can be strengthened or weakened by the CONSEQ rule.

The FRAME rule allows us to use local specifications [59]. When verifying the simulation between C and $\mathbb{C}$, we need to only talk about the locally-used resource in α, ζ and γ, and the local relies and guarantees $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$ and $\mathbb{G}$. Then the proof can be reused in contexts where some extra resource η is used, and the accesses of it respect the invariant β and $\mathcal{R}_1$, $\mathcal{G}_1$, $\mathbb{R}_1$ and $\mathbb{G}_1$. We give the auxiliary definitions in Figure 2.6. The disjoint union $\uplus$ between states is lifted to state pairs. A state relation α is intuitionistic, written as $\mathsf{Intuit}\alpha$, if it is monotone with respect to the extension of states. The disjointness $\eta \# \alpha$ says that any state pair satisfying both η and α can be split into two disjoint state pairs satisfying η and α respectively. For example, let $\eta \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma(\mathtt{y}) = \Sigma(\mathtt{y})\}$ and $\alpha \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}$ where $\mathtt{x}$ and $\mathtt{y}$ are two distinct variables, then both η and α are intuitionistic and $\eta \# \alpha$ holds. The FRAME rule also requires η to be stable under interference from the programs (i.e., the programs do not change the extra resource) and the extra environments. We use $\eta \# \{\zeta, \gamma, \alpha\}$ as a shorthand for $(\eta \# \zeta) \wedge (\eta \# \gamma) \wedge (\eta \# \alpha)$. Similar representations are used in this rule.

Finally, the transitivity rule TRANS allows us to verify refinement by introducing an intermediate level as a bridge. The intermediate environment $\mathrm{R_M}$ should be chosen with caution so that the $(\beta \circ \alpha)$-related transitions can be decomposed into β-related and α-related transitions, as illustrated in Figure 2.4(b). Here $\circ$ defines the composition of two relations and $\mathsf{isMidOf}$ defines the side condition over the environments, as shown in Figure 2.6. We use θ for a middle-level state.

Soundness. All the rules in Figure 2.7 are sound, i.e., for each rule the premises imply the conclusion. We prove their soundness by co-induction, directly following the definition of RGSim. The proofs [44] are checked in the Coq proof assistant [13].

Instantiations of relies and guarantees. We can derive the sequential refinement and the fully-abstract-semantics-based refinement by instantiating the rely conditions in RGSim. For example, the simulation of Eq. (2.6) over closed programs assumes identity environments, making the interference constraints in the PAR rule unsatisfiable. This confirms the observation in Section 2.1.1 that the sequential refinement loses parallel compositionality.

$$(C, \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathsf{Id}, \mathsf{True}) \tag{2.6}$$

The simulation of Eq. (2.7) assumes arbitrary environments, which makes the interference constraints in the PAR rule trivially true. But this assumption is too strong: usually Eq. (2.7) cannot be satisfied in practice.

$$(C, \mathsf{True}, \mathsf{True}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathsf{True}, \mathsf{True}) \tag{2.7}$$

2.5 A Simple Example

Below we give a simple example to illustrate the use of RGSim and its parallel compositionality in verifying concurrent program refinement. The high-level program $\mathbb{C}_1 ||| \mathbb{C}_2$ is transformed to $C_1 \parallel C_2$, using a lock `l` to synchronize the accesses of the shared variable `x`. We aim to prove $C_1 \parallel C_2 \sqsubseteq_{\mathbf{T}} \mathbb{C}_1 ||| \mathbb{C}_2$. That is, although `x:=x+2` is implemented by two steps of incrementing `x` in C_2, the parallel observer C_1 will not print unexpected values. Here we view output events as externally observable behaviors.

```
print(x);   |||  x := x + 2;
            ⇓
lock(l);         lock(l);
print(x);   ||   x := x+1; x := x+1;
unlock(l);       ⟨unlock(l); X := x; ⟩
```

To facilitate the proof, we introduce an auxiliary shared variable `X` at the low level to record the value of `x` at the time when releasing the lock. It specifies the

value of x outside every critical section, thus should match the value of the high-level x after every corresponding action. Here $\langle C \rangle$ means C is executed atomically. Its semantics follows RGSep [66] (or see Section 5.2). The auxiliary variable is write-only and would not affect the external behaviors of the program [1]. Thus below we can focus on the instrumented low-level program with the auxiliary code.

By the adequacy and compositionality of RGSim, we only need to prove simulations over individual threads, providing appropriate relies and guarantees. We first define the invariant α, which only cares about the value of x when the lock is free.

$$\alpha \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma(\mathtt{X}) = \Sigma(\mathtt{x}) \wedge (\sigma(\mathtt{l}) = 0 \implies \sigma(\mathtt{x}) = \sigma(\mathtt{X}))\}.$$

We let the pre- and post-conditions be α as well.

The high-level threads can be executed in arbitrary environments with arbitrary guarantees: $\mathbb{R} = \mathbb{G} \stackrel{\text{def}}{=} \mathsf{True}$. The low-level threads use the lock to protect every access of x, thus their relies and guarantees are not arbitrary.

$$\begin{aligned}
\mathcal{R} \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid{}& \sigma(\mathtt{l}) = \mathtt{cid} \implies \\
& \sigma(\mathtt{x}) = \sigma'(\mathtt{x}) \wedge \sigma(\mathtt{X}) = \sigma'(\mathtt{X}) \wedge \sigma(\mathtt{l}) = \sigma'(\mathtt{l})\}; \\
\mathcal{G} \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid{}& \sigma' = \sigma \vee \sigma(\mathtt{l}) = 0 \wedge \sigma' = \sigma\{\mathtt{l} \rightsquigarrow \mathtt{cid}\} \\
& \vee \sigma(\mathtt{l}) = \mathtt{cid} \wedge \sigma' = \sigma\{\mathtt{x} \rightsquigarrow _\} \\
& \vee \sigma(\mathtt{l}) = \mathtt{cid} \wedge \sigma' = \sigma\{\mathtt{l} \rightsquigarrow 0, \mathtt{X} \rightsquigarrow _\}\}.
\end{aligned}$$

Every low-level thread guarantees that it updates x only when the lock is acquired. Its environment cannot update x or l if the current thread holds the lock. Here cid is the identifier of the current thread. When acquired, the lock holds the identifier of the owner thread.

Following the definition of RGSim, we can prove $(C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha\times\alpha} (\mathbb{C}_1, \mathbb{R}, \mathbb{G})$ and $(C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha\times\alpha} (\mathbb{C}_2, \mathbb{R}, \mathbb{G})$. By applying the PAR rule and from the adequacy of RGSim (Corollary 2.6), we know $C_1 \parallel C_2 \sqsubseteq_{\mathbf{T}} \mathbb{C}_1 ||| \mathbb{C}_2$ holds for any transformation $\mathbf{T}$ that respects α.

Perhaps interestingly, if we omit the lock and unlock operations in C_1, then $C_1 \parallel C_2$ would have more externally observable behaviors than $\mathbb{C}_1 ||| \mathbb{C}_2$. This does *not* indicate the unsoundness of our PAR rule (which is sound!). The reason is that x might have different values at the two levels after the environments' α-related transitions $\langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha$, so that we cannot have $(\mathtt{print(x)}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha\times\alpha} (\mathtt{print(x)}, \mathbb{R}, \mathbb{G})$, even though the code of the two sides is syntactically identical.

The use of the auxiliary variable. The auxiliary variable X helps us define the invariant α and do the proof. It is difficult to prove the refinement without

this auxiliary variable. One may wish to prove

$$(C_1, \mathcal{R}', \mathcal{G}') \preceq_{\alpha';\alpha'\ltimes\alpha'} (\mathbb{C}_1, \mathbb{R}, \mathbb{G}), \tag{2.8}$$

where α', $\mathcal{R}'$ and $\mathcal{G}'$ are defined as follows by eliminating $\mathtt{X}$ from α, $\mathcal{R}$ and $\mathcal{G}$.

$$\begin{array}{rcl}
\alpha' & \stackrel{\text{def}}{=} & \{(\sigma,\Sigma) \mid \sigma(\mathtt{l}){=}0 \implies \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}\,; \\
\mathcal{R}' & \stackrel{\text{def}}{=} & \{(\sigma,\sigma') \mid \sigma(\mathtt{l}){=}\mathtt{cid} \implies \sigma(\mathtt{x}){=}\sigma'(\mathtt{x}) \wedge \sigma(\mathtt{l}){=}\sigma'(\mathtt{l})\}\,; \\
\mathcal{G}' & \stackrel{\text{def}}{=} & \{(\sigma,\sigma') \mid \sigma'{=}\sigma \ \vee\ \sigma(\mathtt{l}){=}0 \wedge \sigma'{=}\sigma\{\mathtt{l} \rightsquigarrow \mathtt{cid}\} \\
& & \qquad \vee\, \sigma(\mathtt{l}){=}\mathtt{cid} \wedge \sigma'{=}\sigma\{\mathtt{x} \rightsquigarrow _\} \\
& & \qquad \vee\, \sigma(\mathtt{l}){=}\mathtt{cid} \wedge \sigma'{=}\sigma\{\mathtt{l} \rightsquigarrow 0\}\}\,.
\end{array}$$

But Eq. (2.8) does not hold because $\langle \mathcal{R}', \mathbb{R}^* \rangle_{\alpha'}$ permits unexpected transitions. For instance, we allow $((\sigma,\sigma'),(\Sigma,\Sigma')) \in \langle \mathcal{R}', \mathbb{R}^* \rangle_{\alpha'}$ for the following σ, σ', Σ and Σ'.

$$\sigma \;=\; \sigma' \stackrel{\text{def}}{=} \{\mathtt{x} \rightsquigarrow 0, \mathtt{l} \rightsquigarrow \mathtt{cid}\}\,; \qquad \Sigma \stackrel{\text{def}}{=} \{\mathtt{x} \rightsquigarrow 0\}\,; \qquad \Sigma' \stackrel{\text{def}}{=} \{\mathtt{x} \rightsquigarrow 1\}\,.$$

The high-level environment is allowed to change $\mathtt{x}$ even if the thread holds the lock at the low level. Then the left thread may print out different values at the two levels, breaking the simulation of Eq. (2.8).

It is possible to define the RGSim relation in another way that allows us to get rid of the auxiliary variable for this example. Instead of defining separate rely/guarantee relations at the two levels and using α to relate them, we can directly define "relational rely/guarantee" relations

$$r, g \in \mathscr{P}((\mathit{LState} \times \mathit{LState}) \times (\mathit{HState} \times \mathit{HState}))\,.$$

The new simulation is defined by substituting r and g for $\langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha$ and $\langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha$ in Definition 2.4. It is in the following form.

$$C \preceq_{r;g;\alpha;\zeta\ltimes\gamma} \mathbb{C}\,.$$

It has all the nice properties of our current RGSim relation (including parallel compositionality) and we no longer need auxiliary variables to prove the simple example. We can prove the new simulations $C_1 \preceq_{\alpha';\alpha'\ltimes\alpha';r;g} \mathbb{C}_1$ and $C_2' \preceq_{\alpha';\alpha'\ltimes\alpha';r;g} \mathbb{C}_2$. Here C_2' results from removing $\mathtt{X}$ from C_2, α' is defined as above and r and g are as follows.

$$\begin{array}{rcl}
r & \stackrel{\text{def}}{=} & \{((\sigma,\sigma'),(\Sigma,\Sigma')) \mid \sigma(\mathtt{l}){=}\mathtt{cid} \implies \\
& & \qquad \sigma(\mathtt{x}){=}\sigma'(\mathtt{x}) \wedge \sigma(\mathtt{l}){=}\sigma'(\mathtt{l}) \wedge \Sigma(\mathtt{x}){=}\Sigma'(\mathtt{x})\}\,; \\
g & \stackrel{\text{def}}{=} & \{((\sigma,\sigma'),(\Sigma,\Sigma')) \mid \sigma'{=}\sigma \wedge \Sigma'{=}\Sigma \ \vee\ \sigma(\mathtt{l}){=}0 \wedge \sigma'{=}\sigma\{\mathtt{l} \rightsquigarrow \mathtt{cid}\} \wedge \Sigma'{=}\Sigma \\
& & \qquad \vee\, \sigma(\mathtt{l}){=}\mathtt{cid} \wedge \sigma'{=}\sigma\{\mathtt{x} \rightsquigarrow _\} \wedge \Sigma'{=}\Sigma \\
& & \qquad \vee\, \sigma(\mathtt{l}){=}\mathtt{cid} \wedge \sigma'{=}\sigma\{\mathtt{l} \rightsquigarrow 0\} \wedge \Sigma' = \Sigma\{\mathtt{x} \rightsquigarrow \sigma(\mathtt{x})\}\}\,.
\end{array}$$

Note r requires that if the thread holds the lock at the low level, neither the high-level or the low-level environment can change `x`. This relational r does not permit the unexpected transitions discussed before. It is more expressive than $\langle \mathcal{R}', \mathbb{R}^* \rangle_{\alpha'}$, but also looks much heavier.

In the next two chapters, we will show more applications of RGSim. We would use the current RGSim relation because in those applications it is usually easier to define separate rely/guarantee conditions at the two levels. However, in Chapter 5 on linearizability verification, using relational r and g could make the refinement proofs more intuitive. We also define the syntax of r and g in Chapter 5 to make it easier to use them.

2.6 Discussions and Summary

We propose RGSim to verify concurrent program refinement. By describing explicitly the interference with environments, RGSim is compositional. We can use it to prove the correctness of concurrent program transformations. We give a mechanized formulation of RGSim, and prove its adequacy and compositionality in the Coq proof assistant [13]. Both the manual and mechanized proofs are available online [44].

RGSim ensures that the low-level program preserves safety properties (including the partial correctness) of the high-level program, but allows a terminating high-level program to be refined by a low-level program having infinite silent steps (e.g., `while(true) skip;`). In the example in Section 2.5, the low-level programs is allowed to be blocked forever (e.g., at the time when the lock is held but never released by some other thread). Proving the preservation of the termination behavior would require liveness proofs in a concurrent setting (e.g., proving the absence of deadlock), which we leave as future work.

The compositionality of RGSim allows us to decompose the refinement for a large program to refinements for basic refinement units (which are usually instructions). However, for those refinement units, we have to refer to the semantics of RGSim (Definition 2.4) rather than syntactic rules to verify them, since Figure 2.7 provides only compositionality rules, with no rules for primitive instructions. This makes the proofs a bit tedious and complicated. In Chapter 5, we will design a more complete set of proof rules to verify linearizability of concurrent objects. In fact, Turon et al. [63] recently designed a program logic for general refinement verification of concurrent programs. Their logic is based on similar ideas as RGSim and our logic for linearizability in Chapter 5.

Chapter 3

Simple Applications of RGSim

Chapter 2 introduced RGSim, a rely-guarantee-based simulation for compositional verification of concurrent program refinement. In this chapter, we will systematically study two kinds of applications of RGSim: reasoning about optimizations in parallel contexts (Section 3.1) and verifying fine-grained implementations of abstract algorithms and concurrent objects (Section 3.2).

3.1 Relational Reasoning about Optimizations

Verifying compiler optimizations is a natural application of refinement verification. It requires proving that the target program is a refinement of the source. RGSim establishes a relational approach to justify optimizations of concurrent programs. Below we adapt Benton's work [6] on sequential optimizations to the concurrent setting.

3.1.1 Optimization Rules

Usually optimizations depend on particular contexts, e.g., the assignment $x := E$ can be eliminated only in the context that the value of x is never used after the assignment. In a shared-state concurrent setting, we should also consider the parallel context for an optimization. RGSim enables us to specify various sophisticated requirements for the parallel contexts by rely/guarantee conditions. Based on RGSim, we provide a set of inference rules to characterize and justify common optimizations (e.g., dead code elimination) with information of both the sequential and the parallel contexts. Note in this section the target and the source programs are in the same language.

Sequential Unit Laws

$$\frac{(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (C_2, \mathcal{R}_2, \mathcal{G}_2)}{(\mathbf{skip}; C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (C_2, \mathcal{R}_2, \mathcal{G}_2)} \qquad \frac{(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (C_2, \mathcal{R}_2, \mathcal{G}_2)}{(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{skip}; C_2, \mathcal{R}_2, \mathcal{G}_2)}$$

Plus the variants with **skip** after the code C_1 or C_2. That is, **skip**s could be arbitrarily introduced and eliminated.

Common Branch

$$\frac{\begin{array}{c}\forall \sigma_1, \sigma_2.\ (\sigma_1, \sigma_2) \in \zeta \implies B\ \sigma_2 \neq \bot \\ (C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_1\ltimes\gamma} (C_1, \mathcal{R}', \mathcal{G}') \qquad \zeta_1 = (\zeta \cap (\mathbf{true} \Rrightarrow B)) \\ (C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_2\ltimes\gamma} (C_2, \mathcal{R}', \mathcal{G}') \qquad \zeta_2 = (\zeta \cap (\mathbf{true} \Rrightarrow \neg B))\end{array}}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2, \mathcal{R}', \mathcal{G}')}$$

When the if-condition can be evaluated and both branches can be optimized to the same code C, we can transform the whole if-statement to C without introducing new behaviors.

Known Branch

$$\frac{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (C_1, \mathcal{R}', \mathcal{G}') \qquad \zeta = (\zeta \cap (\mathbf{true} \Rrightarrow B))}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2, \mathcal{R}', \mathcal{G}')}$$

$$\frac{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (C_2, \mathcal{R}', \mathcal{G}') \qquad \zeta = (\zeta \cap (\mathbf{true} \Rrightarrow \neg B))}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2, \mathcal{R}', \mathcal{G}')}$$

Since the if-condition B is **true** (or **false**) initially, we can consider the then-branch (or the else-branch) only. These rules can be derived from the above rule for common branch.

Dead While

$$\frac{\zeta = (\zeta \cap (\mathbf{true} \Rrightarrow \neg B)) \qquad \zeta \subseteq \alpha \qquad \mathsf{Sta}(\zeta, \langle \mathcal{R}_1, \mathcal{R}_2^* \rangle_\alpha)}{(\mathbf{skip}, \mathcal{R}_1, \mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\zeta} (\mathbf{while}\ (B)\{C\}, \mathcal{R}_2, \mathsf{Id})}$$

We can eliminate the loop, if the loop condition is **false** (no matter how the environments update the states) at the loop entry point.

Loop Peeling

$$\frac{(\mathbf{while}\ (B)\{C\}, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{while}\ (B)\{C\}, \mathcal{R}_2, \mathcal{G}_2)}{(\mathbf{if}\ (B)\ \{C; \mathbf{while}\ (B)\{C\}\}\ \mathbf{else\ skip}, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{while}\ (B)\{C\}, \mathcal{R}_2, \mathcal{G}_2)}$$

Loop Unrolling

$$\frac{(\textbf{while } (B)\{C\}, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (\textbf{while } (B)\{C\}, \mathcal{R}_2, \mathcal{G}_2)}{(\textbf{while } (B)\{C; \textbf{if } (B)\ C \textbf{ else skip}\}, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (\textbf{while } (B)\{C\}, \mathcal{R}_2, \mathcal{G}_2)}$$

Dead Code Elimination

$$\frac{(\textbf{skip}, \mathsf{Id}, \mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\gamma} (C, \mathsf{Id}, \mathcal{G}) \qquad \mathsf{Sta}(\{\zeta, \gamma\}, \langle \mathcal{R}_1, \mathcal{R}_2^* \rangle_\alpha)}{(\textbf{skip}, \mathcal{R}_1, \mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\gamma} (C, \mathcal{R}_2, \mathcal{G})}$$

Intuitively $(\textbf{skip}, \mathsf{Id}, \mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\gamma} (C, \mathsf{Id}, \mathcal{G})$ says that the code C can be eliminated in a sequential context where the initial and the final states satisfy ζ and γ respectively. If both ζ and γ are stable with respect to the interference from the environments $\mathcal{R}_1$ and $\mathcal{R}_2$, then the code C can be eliminated in such a parallel context as well.

Redundancy Introduction

$$\frac{(c, \mathsf{Id}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\textbf{skip}, \mathsf{Id}, \mathsf{Id}) \qquad \mathsf{Sta}(\{\zeta, \gamma\}, \langle \mathcal{R}_1, \mathcal{R}_2^* \rangle_\alpha)}{(c, \mathcal{R}_1, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\textbf{skip}, \mathcal{R}_2, \mathsf{Id})}$$

As we lifted sequential dead code elimination, we can also lift sequential redundant code introduction to the concurrent setting, so long as the pre- and post-conditions are stable with respect to the environments. Note that here c is a single instruction. We should consider the interference from the environments at every intermediate state when introducing a sequence of redundant instructions.

With these rules, we can prove the correctness of many traditional compiler optimizations performed on concurrent programs in appropriate contexts. Below we give some examples of hoisting loop invariants, strength reduction and induction variable elimination.

3.1.2 Example: Invariant Hoisting

We first formally prove the example in Section 2.1.5. As we discussed, safely hoisting the invariant code `t:=x+1` requires that the environment $\mathcal{R}$ should not update `x` nor `t`.

$$\mathcal{R} \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma(\mathtt{x}) = \sigma'(\mathtt{x}) \wedge \sigma(\mathtt{t}) = \sigma'(\mathtt{t})\}\,.$$

The guarantee of the program can be specified as arbitrary transitions. Since we only care about the values of `i`, `n` and `x`, the invariant relation α can be defined as

$$\alpha \stackrel{\text{def}}{=} \{(\sigma_1, \sigma) \mid \sigma_1(\mathtt{i}) = \sigma(\mathtt{i}) \wedge \sigma_1(\mathtt{n}) = \sigma(\mathtt{n}) \wedge \sigma_1(\mathtt{x}) = \sigma(\mathtt{x})\}\,.$$

We do not need special pre- and post-conditions, thus the correctness of the optimization is formalized as follows.

$$(C_1, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (C, \mathcal{R}, \mathsf{True})\,. \tag{3.1}$$

We could prove (3.1) directly by the RGSim definition and the operational semantics of the code. But below we give a more convenient proof using the optimization rules and the compositionality rules instead. We first prove the following by the optimization rules for dead code elimination and redundancy introduction.

$$(\mathtt{t:=x+1}, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\gamma} (\mathbf{skip}, \mathcal{R}, \mathsf{True})$$
$$(\mathbf{skip}, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\gamma\ltimes\eta} (\mathtt{t:=x+1}, \mathcal{R}, \mathsf{True})$$

Here γ and η specify the states at the specific program points.

$$\gamma \stackrel{\text{def}}{=} \alpha \cap \{(\sigma_1, \sigma) \mid \sigma_1(\mathtt{t}) = \sigma_1(\mathtt{x}) + 1\}\,;$$
$$\eta \stackrel{\text{def}}{=} \gamma \cap \{(\sigma_1, \sigma) \mid \sigma(\mathtt{t}) = \sigma(\mathtt{x}) + 1\}\,.$$

Then by the compositionality rules SEQ and WHILE, we can get $(C_1', \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (C', \mathcal{R}, \mathsf{True})$, where C_1' and C' result from adding **skip**s to C_1 and C respectively.

C_1' :

```
t := x + 1;
while(i < n) {
  skip;
  i := i + t;
}
```

C' :

```
skip;
while(i < n) {
  t := x + 1;
  i := i + t;
}
```

Besides, from sequential unit laws and compositionality rules SEQ and WHILE, we can prove $(C_1, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (C_1', \mathcal{R}, \mathsf{True})$ and $(C', \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (C, \mathcal{R}, \mathsf{True})$. Finally, by the TRANS rule, we can conclude Eq. (3.1), i.e., the correctness of the optimization in appropriate contexts. Since $\mathcal{R}$ only prohibit updates of `x` and `t`, we can execute C_1 and C concurrently with other threads which update `i` and `n` or read `x`, still ensuring semantics preservation.

3.1.3 Example: Strength Reduction and Induction Variable Elimination

Target-Level C_2

```
local k, r;
k := 0;
r := 6*n;
while(k<r) {
  x := x+k;
  k := k+6;
}
```

$\Leftarrow$

Middle-Level C_1

```
local i, k;
i := 0;
k := 0;
while(i<n) {
  x := x+k;
  i := i+1;
  k := k+6;
}
```

$\Leftarrow$

Source-Level C

```
local i;
i := 0;
while(i<n) {
  x := x+6*i;
  i := i+1;
}
```

The source program C is first transformed to C_1 by strength reduction which introduces a local variable `k` and replaces multiplication by addition. The original induction variable `i` and the introduced local variable `k` cannot be updated by the environments. Then C_1 is transformed to the target C_2 by eliminating `i` and using the new induction variable `k` in the loop condition. We assume `n` and `r` will not be updated by the target environment, so we can compute the new boundary (i.e., `r:=6*n`) outside the loop. Below we give the environments $\mathcal{R}$, $\mathcal{R}_1$ and $\mathcal{R}_2$ at the source, intermediate and target levels respectively.

$$\begin{array}{lcl} \mathcal{R} & \stackrel{\text{def}}{=} & \{(\sigma,\sigma') \mid \sigma(\mathtt{i}) = \sigma'(\mathtt{i})\} \\ \mathcal{R}_1 & \stackrel{\text{def}}{=} & \{(\sigma_1,\sigma_1') \mid \sigma_1(\mathtt{i}) = \sigma_1'(\mathtt{i}) \wedge \sigma_1(\mathtt{k}) = \sigma_1'(\mathtt{k})\} \\ \mathcal{R}_2 & \stackrel{\text{def}}{=} & \{(\sigma_2,\sigma_2') \mid \sigma_2(\mathtt{k}) = \sigma_2'(\mathtt{k}) \wedge \sigma_2(\mathtt{r}) = \sigma_2'(\mathtt{r}) \wedge \sigma_2(\mathtt{n}) = \sigma_2'(\mathtt{n})\} \end{array}$$

For both optimization phases, we require that the common variables in the source and target have the same values. This is shown in the invariant relations α (between C_1 and C) and β (between C_2 to C_1) below.

$$\begin{array}{lcl} \alpha & \stackrel{\text{def}}{=} & \{(\sigma_1,\sigma) \mid \sigma_1(\mathtt{i}) = \sigma(\mathtt{i}) \wedge \sigma_1(\mathtt{n}) = \sigma(\mathtt{n}) \wedge \sigma_1(\mathtt{x}) = \sigma(\mathtt{x})\}\,; \\ \beta & \stackrel{\text{def}}{=} & \{(\sigma_2,\sigma_1) \mid \sigma_2(\mathtt{k}) = \sigma_1(\mathtt{k}) \wedge \sigma_2(\mathtt{n}) = \sigma_1(\mathtt{n}) \wedge \sigma_2(\mathtt{x}) = \sigma_1(\mathtt{x})\}\,. \end{array}$$

Thus we formalize the correctness of the two optimization phases as follows.

$$(C_2,\mathcal{R}_2,\mathsf{True}) \preceq_{\beta;\beta\ltimes\beta} (C_1,\mathcal{R}_1,\mathsf{True})\,, \quad (C_1,\mathcal{R}_1,\mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (C,\mathcal{R},\mathsf{True})\,.$$

They can be proved directly by the RGSim definition or by applying the optimization rules (for dead code elimination and redundancy introduction). The proofs are similar to those for the previous example of invariant hoisting, and hence omitted here.

Afterwards, we can compose the proofs of these two phases by the TRANS rule, and get

$$(C_2, \mathcal{R}_2, \mathsf{True}) \preceq_{\alpha\circ\beta;\alpha\circ\beta\ltimes\alpha\circ\beta} (C, \mathcal{R}, \mathsf{True})\,.$$

Here $\alpha \circ \beta = \{(\sigma_2, \sigma) \mid \sigma_2(\mathtt{n}) = \sigma(\mathtt{n}) \wedge \sigma_2(\mathtt{x}) = \sigma(\mathtt{x})\}$. That is, if the environment of the source program C does not change `i` nor `n`, we can safely apply strength reduction and induction variable elimination over C to get the target program C_2.

3.1.4 Discussions and Related Work

We apply RGSim to justify concurrent optimizations, following Benton [6] who presents a declarative set of rules for sequential optimizations. Also the proof rules of RGSim for sequential compositions (SEQ), conditional statements (IF) and loops (WHILE) coincide with those in Benton's relational Hoare logic [6] and Yang's relational separation logic [72]. Nevertheless, we have not applied RGSim to verify real-world compilation or more complicated optimization algorithms (e.g., lazy code motion) in concurrent settings, which we leave as future work.

Compiler verification for concurrent programming languages can date back to work in [23, 71], which is about functional languages using message-passing mechanisms. Recently, Lochbihler [47] presents a verified compiler for Java threads and prove semantics preservation by a weak bisimulation. He views every heap update as an observable move, thus does not allow the target and the source to have different granularities of atomic updates. To achieve parallel compositionality, he requires the relation to be preserved by any transitions of shared states, i.e., the environments are assumed arbitrary. As we explained in Section 2.1.2, this is a too strong requirement in general for many refinement applications, including the examples discussed in this section.

Burckhardt et al. [11] present a proof method for verifying concurrent program transformations on relaxed memory models. The method relies on a compositional trace-based denotational semantics, where the values of shared variables are always considered arbitrary at any program point. In other words, they also assume arbitrary environments.

Following Leroy's CompCert project [42], Ševčík et al. [61] verify compilation from a C-like concurrent language to x86 by simulations. They focus on correctness of a particular compiler, and there are two phases in their compiler whose proofs are not compositional.

Our RGSim is a general, compiler-independent and compositional proof technique for concurrent program refinement. Verifying optimizations is just one of its applications. We will show more applications of RGSim in the following section and the next chapter.

A_1 :

```
1  local d1;
2  d1 := 0;
3  while (d1 = 0) {
4    atom{
5      if (a = b)
6        d1 := 1;
7      if (a > b)
8        a := a - b;
9    }
10 }
```

$\|\|$

A_2 :

```
1  local d2;
2  d2 := 0;
3  while (d2 = 0) {
4    atom{
5      if (b = a)
6        d2 := 1;
7      if (b > a)
8        b := b - a;
9    }
10 }
```

(a) abstract code

C_1 :

```
1  local d1, t11, t12;
2  d1 := 0;
3  while (d1 = 0) {
4    t11 := a;
5    t12 := b;
6    if (t11 = t12)
7      d1 := 1;
8    if (t11 > t12)
9      a := t11 - t12;
10 }
```

$\|$

C_2 :

```
1  local d2, t21, t22;
2  d2 := 0;
3  while (d2 = 0) {
4    t21 := b;
5    t22 := a;
6    if (t21 = t22)
7      d2 := 1;
8    if (t21 > t22)
9      b := t21 - t22;
10 }
```

(b) concrete code

Figure 3.1 Concurrent GCD.

3.2 Verifying Fine-Grained Implementations of Abstract Operations

As we mentioned in Chapter 1, verifying the implementation of an abstract algorithm can be reduced to proving refinement from an abstract operation to a concrete and executable program. In a concurrent setting, we can use RGSim to verify the fine-grained implementation of an abstract program. Below we first discuss the verification of a concurrent GCD algorithm [20] which calculates the greatest common divisor of two variables.

3.2.1 Example: Concurrent GCD

Figure 3.1(b) shows the concurrent GCD implementation. The program uses two threads to compute the Greatest Common Divisor (GCD) of the shared variables

a and b. One thread executes C_1 which reads the values of a and b, but only updates a if a > b. The other thread executes C_2 which does the reverse. When a = b, the two threads terminate. This fine-grained GCD program implements the more abstract program in Figure 3.1(a), where two threads atomically update a and b respectively. Here we use $\boldsymbol{atom}\{\mathbb{C}\}$ to execute $\mathbb{C}$ atomically. Its semantics follows RGSep [66] (or see Section 5.2).

Our goal is to prove that the concrete and abstract GCD programs always obtain the same result, i.e., $(C_1 \parallel C_2); \texttt{print(a)}$ and $(A_1 \parallel\!\parallel A_2); \texttt{print(a)}$ have the same outputs. We use print(a) at the two levels to print out the results after both threads complete their computations.

By adequacy of RGSim and its compositionality, we only need to prove that the core computations for updating a (or b) are equivalent in C_1 and A_1 (or C_2 and A_2). That is, the code from line 4 to line 9 in C_1 (denoted by C_1') should be equivalent to the atomic block from line 4 to line 9 in A_1 (denoted by A_1'); and the parts of code from line 4 to line 9 in C_2 and A_2 should be equivalent too.

We first define the α relation, requiring that the common variables at the low and high levels have the same values.

$$\alpha \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma(\texttt{a}) = \Sigma(\texttt{a}) \wedge \sigma(\texttt{b}) = \Sigma(\texttt{b}) \wedge \sigma(\texttt{d1}) = \Sigma(\texttt{d1}) \wedge \sigma(\texttt{d2}) = \Sigma(\texttt{d2})\}.$$

The threads' rely and guarantee conditions can be specified as follows, where the rely of one thread is just the guarantee of the other.

$$\begin{aligned}
&\mathcal{R}_1 = \mathcal{G}_2 \stackrel{\text{def}}{=} \\
&\quad \{(\sigma, \sigma') \mid \sigma'(\texttt{t11}) = \sigma(\texttt{t11}) \wedge \sigma'(\texttt{t12}) = \sigma(\texttt{t12}) \\
&\qquad\qquad \wedge \sigma'(\texttt{d1}) = \sigma(\texttt{d1}) \wedge \sigma'(\texttt{a}) = \sigma(\texttt{a}) \wedge (\sigma(\texttt{a}) \geq \sigma(\texttt{b}) \Rightarrow \sigma'(\texttt{b}) = \sigma(\texttt{b}))\} \\
&\mathcal{R}_2 = \mathcal{G}_1 \stackrel{\text{def}}{=} \\
&\quad \{(\sigma, \sigma') \mid \sigma'(\texttt{t21}) = \sigma(\texttt{t21}) \wedge \sigma'(\texttt{t22}) = \sigma(\texttt{t22}) \\
&\qquad\qquad \wedge \sigma'(\texttt{d2}) = \sigma(\texttt{d2}) \wedge \sigma'(\texttt{b}) = \sigma(\texttt{b}) \wedge (\sigma(\texttt{b}) \geq \sigma(\texttt{a}) \Rightarrow \sigma'(\texttt{a}) = \sigma(\texttt{a}))\} \\
&\mathbb{R}_1 = \mathbb{G}_2 \stackrel{\text{def}}{=} \\
&\quad \{(\Sigma, \Sigma') \mid \Sigma'(\texttt{d1}) = \Sigma(\texttt{d1}) \wedge \Sigma'(\texttt{a}) = \Sigma(\texttt{a}) \wedge (\Sigma(\texttt{a}) \geq \Sigma(\texttt{b}) \Rightarrow \Sigma'(\texttt{b}) = \Sigma(\texttt{b}))\} \\
&\mathbb{R}_2 = \mathbb{G}_1 \stackrel{\text{def}}{=} \\
&\quad \{(\Sigma, \Sigma') \mid \Sigma'(\texttt{d2}) = \Sigma(\texttt{d2}) \wedge \Sigma'(\texttt{b}) = \Sigma(\texttt{b}) \wedge (\Sigma(\texttt{b}) \geq \Sigma(\texttt{a}) \Rightarrow \Sigma'(\texttt{a}) = \Sigma(\texttt{a}))\}
\end{aligned}$$

Then we can operationally prove the RGSim relations between C_1' and A_1'. Here α^{-1} is the inverse relation of α, as defined in Figure 2.6.

$$(C_1', \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha\ltimes\alpha} (A_1', \mathbb{R}_1, \mathbb{G}_1), \qquad (A_1', \mathbb{R}_1, \mathbb{G}_1) \preceq_{\alpha^{-1};\alpha^{-1}\ltimes\alpha^{-1}} (C_1', \mathcal{R}_1, \mathcal{G}_1).$$

By the rules WHILESEQ, we get the RGSim relations between C_1 and A_2.

$$(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha\ltimes\alpha} (A_1, \mathbb{R}_1, \mathbb{G}_1), \qquad (A_1, \mathbb{R}_1, \mathbb{G}_1) \preceq_{\alpha^{-1};\alpha^{-1}\ltimes\alpha^{-1}} (C_1, \mathcal{R}_1, \mathcal{G}_1).$$

Similarly, the relations hold between C_2 and A_2.

$$(C_2, \mathcal{R}_2, \mathcal{G}_2) \preceq_{\alpha;\alpha\ltimes\alpha} (A_2, \mathbb{R}_2, \mathbb{G}_2)\,, \qquad (A_2, \mathbb{R}_1, \mathbb{G}_1) \preceq_{\alpha^{-1};\alpha^{-1}\ltimes\alpha^{-1}} (C_2, \mathcal{R}_1, \mathcal{G}_1).$$

When C_1 and C_2 (or A_1 and A_2) are parallel composed to compute the GCD together, the environment of the whole GCD program cannot modify the shared variables `a` and `b`. We let it be the identity transition set Id. The guarantee of the whole program is just specified as True, a set of arbitrary transitions. We can prove that both $(\texttt{print}(\texttt{a}), \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (\texttt{print}(\texttt{a}), \mathsf{Id}, \mathsf{True})$ and the reverse direction hold. Then by the rules PAR and SEQ, we can get

$$((C_1 \,\|\, C_2); \texttt{print}(\texttt{a}), \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} ((A_1 \,|||\, A_2); \texttt{print}(\texttt{a}), \mathsf{Id}, \mathsf{True})\,,$$

and also the reverse direction. By the adequacy of RGSim (Corollary 2.6), we obtain the final result. For any state transformation $\mathbf{T}$ that respects α, we have

$$(C_1 \,\|\, C_2); \texttt{print}(\texttt{a}) \approx_{\mathbf{T}} (A_1 \,|||\, A_2); \texttt{print}(\texttt{a})\,.$$

Thus we have proved that the concrete fine-grained and the abstract coarse-grained GCD programs in Figure 3.1 can obtain the same results from the same inputs. It is not difficult to find that the abstract program really computes the GCD of `a` and `b`. So we can conclude that the concrete program computes their GCD as well. This example shows a way to verify a complicated program by proving that it is equivalent to a simpler program and then verifying the simpler program.

3.2.2 Verifying Concurrent Objects

A concurrent object provides a set of methods, which can be called in parallel by clients as the only way to access the object. The correctness on functionality of an object is usually characterized by linearizability [35]. Informally, linearizability describes atomic behaviors of object implementations. It requires that each method call should appear to take effect instantaneously at some moment between its invocation and return. We can define abstract atomic operations in a high-level language as object specifications. Then linearizability intuitively establishes a refinement between the concrete fine-grained implementations and the corresponding atomic operations in concurrent environments.

We have applied RGSim to verify linearizability of many object implementations, including the non-blocking counter [64], Treiber's stack algorithm [62] and the lock-coupling list [33]. However, RGSim does not support objects with

non-fixed Linearization Points (LPs), including those using the helping mechanism (e.g., HSY elimination-based stack [30]), or having LPs depending on unpredictable future executions (e.g., the lazy set algorithm [29]), or involving both features (e.g., the RGCSS algorithm [66]). As we explained in Section 1.1.2, it is quite difficult to verify those objects. In Chapter 5, we will extend the RGSim relation and design a program logic for linearizability, both of which support those challenging objects with non-fixed LPs as well as simple ones with static LPs that can be verified using the plain RGSim relation. Since the examples for the plain RGSim will be completely covered by those in Chapter 5, we omit the proofs here and encourage readers to study Chapter 5, which contains detailed and formal explanations about linearizability verification, and more effective and intuitive example proofs.

In the next chapter, we will study another important application of RGSim: verifying concurrent garbage collectors.

Chapter 4

Verifying Concurrent Garbage Collectors

In this chapter, we first explain in detail how to reduce the problem of verifying concurrent garbage collectors (GCs) to refinement verification (Section 4.1), and use RGSim to develop a general GC verification framework (Section 4.2). Then we apply the framework to prove the correctness of the Boehm et al. concurrent GC algorithm [9] (Section 4.3).

4.1 Correctness of Concurrent GCs

A concurrent GC is executed by a dedicated thread and performs the collection work in parallel with user threads (mutators), which access the shared heap via read, write and allocation operations. To ensure that the GC and the mutators share a coherent view of the heap, the heap operations from mutators may be instrumented with extra operations, which provide an interaction mechanism to allow arbitrary mutators to cooperate with the GC. These instrumented heap operations are called barriers (e.g., read barriers and write barriers).

A concurrent GC algorithm consists of the GC thread and the corresponding barriers. It provides a higher-level user-friendly programming model for garbage-collected languages (e.g., Java). In this high-level model, programmers feel they access the heap using regular memory operations, and are freed from manually disposing objects that are no longer in use. They do not need to consider the implementation details of the GC and the existence of barriers.

We could verify the GC system by using a Hoare-style logic to prove that the GC thread and the barriers satisfy their specifications. However, we say this is an indirect approach because it is unclear if the specified correct behaviors would

indeed preserve the mutators' intended behaviors and generate the abstract view for high-level programmers. Usually this part is examined by experts and then trusted.

Here we propose a more direct approach. We view a concurrent GC algorithm as a transformation $\mathbf{T}$ from a high-level garbage-collected language to a low-level language. A standard atomic memory operation at the source level is transformed into the corresponding barrier code at the target level. In the source level, we assume there is an *abstract GC thread* that magically turns unreachable objects into reusable memory. The abstract collector $AbsGC$ is transformed into the concrete GC code C_{gc} running concurrently with the target mutators. That is,

$$\mathbf{T}(\mathsf{t}_{\text{gc}}.AbsGC \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \,|||\, \mathsf{t}_n.\mathbb{C}_n) \stackrel{\text{def}}{=} \mathsf{t}_{\text{gc}}.C_{\text{gc}} \,||\, \mathsf{t}_1.\mathbf{T}(\mathbb{C}_1) \,||\, \ldots \,||\, \mathsf{t}_n.\mathbf{T}(\mathbb{C}_n)\,,$$

where $\mathbf{T}(\mathbb{C})$ simply translates some memory access instructions in $\mathbb{C}$ into the corresponding barriers, and leaves the rest unchanged. Note that here we introduce an abstract GC and assume a finite memory at the source level. This is because at the target level we assume a finite memory to model the real machine; and if the source level memory is infinite, the bijective mapping between the memory at the two levels would become much more complicated.

Then we reduce the correctness of the concurrent GC algorithm to $\mathsf{Correct}(\mathbf{T})$, saying that any mutator program will not have unexpected behaviors when executed using this GC algorithm.

4.2 A General Verification Framework Based on RGSim

The compositionality of RGSim allows us to develop a general framework to prove $\mathsf{Correct}(\mathbf{T})$, which is much more difficult using monolithic proof methods. By the parallel compositionality of RGSim (the PAR rule in Figure 2.7), we can decompose the refinement proofs into proofs for the GC thread and each mutator thread. For a mutator thread, we can further decompose the refinement proof into proof for each primitive instruction, using the compositionality of RGSim (the rules SEQ, IF and WHILE in Figure 2.7).

Verifying the GC thread. The semantics of the abstract GC thread can be defined by a binary state predicate $\mathsf{AbsGCStep}$.

$$\frac{(\Sigma, \Sigma') \in \mathsf{AbsGCStep}}{(\mathsf{t}_{\text{gc}}.AbsGC, \Sigma) \longrightarrow (\mathsf{t}_{\text{gc}}.AbsGC, \Sigma')}$$

That is, the abstract GC thread always makes AbsGCStep to change the high-level state. We can choose different AbsGCStep for different GCs, but usually AbsGCStep guarantees not modifying reachable objects in the heap.

Thus for the GC thread, we need to show that C_{gc} is simulated by $AbsGC$ when executed in their environments. This can be reduced to unary Rely-Guarantee reasoning about C_{gc} by proving $\mathcal{R}_{\text{gc}}; \mathcal{G}_{\text{gc}} \vdash \{p_{\text{gc}}\} C_{\text{gc}} \{q_{\text{gc}}\}$ in a standard Rely-Guarantee logic [38] with proper $\mathcal{R}_{\text{gc}}$, $\mathcal{G}_{\text{gc}}$, p_{gc} and q_{gc}, as long as $\mathcal{G}_{\text{gc}}$ is a concrete representation of AbsGCStep. The judgment says given an initial state satisfying the precondition p_{gc}, if the environment's behaviors satisfy $\mathcal{R}_{\text{gc}}$, then each step of C_{gc} satisfies $\mathcal{G}_{\text{gc}}$, and the postcondition q_{gc} holds at the end if C_{gc} terminates. In general, the collector never terminates, thus we can let q_{gc} be **false**. $\mathcal{G}_{\text{gc}}$ and p_{gc} should be provided by the verifier, where p_{gc} needs to be general enough so that it can be satisfied by any possible low-level initial state. $\mathcal{R}_{\text{gc}}$ encodes the possible behaviors of mutators, which can be derived, as we will show below.

Verifying mutators. For the mutator thread, since $\mathbf{T}$ is syntax-directed on $\mathbb{C}$, we can reduce the refinement verification for arbitrary mutators to verifying the refinement on each primitive instruction only, following the compositionality of RGSim. The proof needs proper rely/guarantee conditions. Let $\mathbb{G}^{\mathsf{t}}_{\mathbb{c}}$ and $\mathcal{G}^{\mathsf{t}}_{\mathbf{T}(\mathbb{c})}$ denote the guarantees of the source instruction $\mathbb{c}$ and the target code $\mathbf{T}(\mathbb{c})$ for the mutator thread t respectively. Then we can define the general guarantees for thread t.

$$\mathcal{G}(\mathsf{t}) \stackrel{\text{def}}{=} \bigcup_{\mathbb{c}} \mathcal{G}^{\mathsf{t}}_{\mathbf{T}(\mathbb{c})}\,; \qquad \mathbb{G}(\mathsf{t}) \stackrel{\text{def}}{=} \bigcup_{\mathbb{c}} \mathbb{G}^{\mathsf{t}}_{\mathbb{c}}\,. \tag{4.1}$$

Its rely conditions $\mathcal{R}(\mathsf{t})$ and $\mathbb{R}(\mathsf{t})$ should include all the possible guarantees made by other threads, and the GC's concrete and abstract behaviors respectively.

$$\mathcal{R}(\mathsf{t}) \stackrel{\text{def}}{=} \mathcal{G}_{\text{gc}} \cup \left(\bigcup_{\mathsf{t}' \neq \mathsf{t}} \mathcal{G}(\mathsf{t}')\right); \qquad \mathbb{R}(\mathsf{t}) \stackrel{\text{def}}{=} \mathsf{AbsGCStep} \cup \left(\bigcup_{\mathsf{t}' \neq \mathsf{t}} \mathbb{G}(\mathsf{t}')\right). \tag{4.2}$$

The $\mathcal{R}_{\text{gc}}$ used to verify the GC code can now be defined.

$$\mathcal{R}_{\text{gc}} \stackrel{\text{def}}{=} \bigcup_{\mathsf{t}} \mathcal{G}(\mathsf{t})\,. \tag{4.3}$$

The refinement proof also needs definitions of binary relations α, ζ and γ. The invariant α relates the low-level and the high-level states and needs to be preserved by each low-level step. In general, a high-level state Σ can be mapped to a low-level state σ by giving a concrete local store for the GC thread, adding additional structures in the heap (to record information for collection), renaming heap cells (for copying GCs), etc. The relations ζ and γ are parametrized over the thread

ID t. For each mutator thread t, $\zeta(\mathsf{t})$ and $\gamma(\mathsf{t})$ need to hold at the beginning and the end of each basic transformation unit (every high-level primitive instruction in this case) respectively. We let $\gamma(\mathsf{t})$ be the same as $\zeta(\mathsf{t})$ to support sequential compositions. They need to satisfy $\mathsf{Good}_{\mathbf{T}}$ defined below.

$$\mathsf{Good}_{\mathbf{T}}(\zeta(\mathsf{t})) \stackrel{\text{def}}{=} \mathsf{InitRel}_{\mathbf{T}}(\zeta(\mathsf{t})) \wedge \forall \mathbb{B}.\ \zeta(\mathsf{t}) \subseteq (\mathbf{T}(\mathbb{B}) \Leftrightarrow \mathbb{B})\,. \tag{4.4}$$

We require $\mathsf{InitRel}_{\mathbf{T}}(\zeta(\mathsf{t}))$ (see Figure 2.6), i.e., $\zeta(\mathsf{t})$ holds over the initial states. In addition, the target and the source boolean expressions should be evaluated to the same value under ζ-related states, as required in the IF and WHILE rules in Figure 2.7.

Theorem 4.1 (Concurrent GC Verification Framework)**.** *If there exist* $\mathbb{G}^{\mathsf{t}}_{\mathbb{c}}$, $\mathcal{G}^{\mathsf{t}}_{\mathbf{T}(\mathbb{c})}$, $\zeta(\mathsf{t})$, α, $\mathcal{G}_{\text{gc}}$ *and* p_{gc} *(for any* $\mathbb{c}$ *and* t*) such that the following hold (where* $\mathcal{G}(\mathsf{t})$, $\mathbb{G}(\mathsf{t})$, $\mathcal{R}(\mathsf{t})$, $\mathbb{R}(\mathsf{t})$ *and* $\mathcal{R}_{\text{gc}}$ *are defined in (4.1), (4.2) and (4.3), and* $\mathsf{Good}_{\mathbf{T}}(\zeta(\mathsf{t}))$ *defined in (4.4) holds):*

1. *(Correctness of* $\mathbf{T}$ *on mutator instructions)*
   $\forall \mathsf{t}, \mathbb{c}.\ (\mathbf{T}(\mathbb{c}), \mathcal{R}(\mathsf{t}), \mathcal{G}(\mathsf{t})) \preceq_{\alpha;\zeta(\mathsf{t})\ltimes\zeta(\mathsf{t})} (\mathbb{c}, \mathbb{R}(\mathsf{t}), \mathbb{G}(\mathsf{t}))$ *;*

2. *(Verification of the GC code)*
   $\mathcal{R}_{\text{gc}}; \mathcal{G}_{\text{gc}} \vdash \{p_{\text{gc}}\} C_{\text{gc}} \{\mathbf{false}\}$ *;*

3. *(Side conditions)*
   $\mathcal{G}_{\text{gc}} \circ \alpha^{-1} \subseteq \alpha^{-1} \circ (\mathsf{AbsGCStep})^*$*; and* $\forall \sigma, \Sigma.\ \sigma = \mathbf{T}(\Sigma) \implies p_{\text{gc}}\ \sigma$ *;*

then $\mathsf{Correct}(\mathbf{T})$.

That is, to verify a concurrent GC algorithm (viewed as a transformation $\mathbf{T}$), we need to do the following.

- Define the α and $\zeta(\mathsf{t})$ relations, and prove the correctness of $\mathbf{T}$ on high-level primitive instructions. Since $\mathbf{T}$ preserves the syntax on most instructions, it's often immediate to prove the target instructions are simulated by their sources. But for instructions that are transformed to barriers, we need to verify that the barriers implement both the source instructions (by simulation) and the interaction mechanism (shown in their guarantees).

- Find some proper $\mathcal{G}_{\text{gc}}$ and p_{gc}, and reason about the GC code in the Rely-Guarantee logic. We require the GC's guarantee $\mathcal{G}_{\text{gc}}$ should not contain more behaviors than $\mathsf{AbsGCStep}$ (the first side condition), and C_{gc} can start its execution from any state σ transformed from a high-level one (the second side condition).

Proof of Theorem 4.1. We first prove the following from the premises *2* and *3* of the theorem.

$$(C_{\mathrm{gc}}, \mathcal{R}_{\mathrm{gc}}, \mathcal{G}_{\mathrm{gc}}) \preceq_{\alpha;\zeta_{\mathrm{gc}} \ltimes \zeta_{\mathrm{gc}}} (AbsGC, \mathsf{True}, \mathsf{AbsGCStep})$$

Here $\zeta_{\mathrm{gc}} \stackrel{\mathrm{def}}{=} \{(\sigma, \Sigma) \mid \sigma = \mathbf{T}(\Sigma)\}$. The proof directly follows the RGSim definition. Then with the premise *1* and the compositionality of RGSim, we can get the following by induction over the program structure.

$$\begin{aligned}\forall \mathbb{C}_1, \ldots, \mathbb{C}_n.\ &(\mathsf{t}_{\mathrm{gc}}.C_{\mathrm{gc}} \,\|\, \mathsf{t}_1.\mathbf{T}(\mathbb{C}_1) \,\|\, \ldots \,\|\, \mathsf{t}_n.\mathbf{T}(\mathbb{C}_n), \mathsf{Id}, \mathsf{True}) \\ &\preceq_{\alpha;\zeta \ltimes \zeta} (\mathsf{t}_{\mathrm{gc}}.AbsGC \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \,|||\, \mathsf{t}_n.\mathbb{C}_n, \mathsf{Id}, \mathsf{True}).\end{aligned}$$

Here $\zeta \stackrel{\mathrm{def}}{=} \zeta_{\mathrm{gc}} \cap (\bigcap_{\mathsf{t}} \zeta(\mathsf{t}))$. Finally, from the adequacy of RGSim (Corollary 2.6), we can conclude $\mathsf{Correct}(\mathbf{T})$. □

4.3 Example: Boehm et al. Concurrent GC Algorithm

We illustrate the applications of the verification framework (Theorem 4.1) by proving the correctness of a mostly-concurrent mark-sweep garbage collector proposed by Boehm et al. [9]. Variants of the algorithm have been used in practice (e.g., by IBM [5]).

4.3.1 Overview of the GC Algorithm

The GC runs both the mark and sweep phases concurrently with the mutators. In the mark phase, it does a depth-first tracing and marks the objects which are reachable from the *roots* (i.e., the mutators' local pointer variables that may contain references to the heap objects). Later in the sweep phase, it scans the heap and reclaims unmarked objects. During the tracing, the connectivity between objects might be changed by the mutators, thus a write barrier is required to notify the collector of those modified objects. Boehm et al.'s algorithm gives each object a dirty bit (called a *card*) and its write barrier dirties the card of the object being updated. Then, between the mark and sweep phases, the GC runs a short stop-the-world phase, where it suspends all the mutators and re-traces from the dirty objects which have been marked (called *card-cleaning*). Thus all reachable objects have been marked before the sweep phase, ensuring the correctness of the GC.

We show the code of the GC thread in Figures 4.1 and 4.2. We assume each object contains m pointer fields $\mathrm{pt}_1, \ldots, \mathrm{pt}_m$, a `data` field, and two auxiliary `color`

```
 1  constant int WHITE, BLACK, BLUE; // colors
 2  constant int N; // total number of threads
 3  constant int M; // size of heap
 4
 5  Collection() {  local mstk;
 6    while (true) {
 7      Initialize();
 8      Trace();
 9      CleanCard();
10      atomic{ ScanRoot(); CleanCard(); }
11      Sweep();
12    }
13  }
14
15  Initialize() {
16    local i, c;  i := 1;
17    while (i <= M) {
18      i.dirty := 0;  c := i.color;
19      if (c = BLACK) { i.color := WHITE; }
20      i := i + 1;
21    }
22  }
23
24  Trace() {
25    local t, rt, i;  t := 1;
26    while (t <= N) {
27      rt := get_root(t);
28      foreach i in rt do { MarkAndPush(i); }
29      t := t + 1;
30      TraceStack();
31    }
32  }
33
34 TraceStack() {
35    local i, j;
36    while (!is_empty(mstk)) {
37      i := pop(mstk);
38      j := i.pt_1; MarkAndPush(j);
39      ...
40      j := i.pt_m; MarkAndPush(j);
41    }
42  }
```

Figure 4.1 Boehm et al. GC code.

```
43  MarkAndPush(i) {
44    local c;
45    if (i != 0) {
46      c := i.color;
47      if (c = WHITE) {
48        i.color := BLACK;  push(i, mstk);
49      }
50    }
51  }
52
53  CleanCard() {
54    local i, c, d;  i := 1;
55    while (i <= M) {
56      c := i.color;  d := i.dirty;
57      if (d = 1) {
58        i.dirty := 0;
59        if (c = BLACK) { push(i, mstk); }
60      }
61      i := i + 1;
62    }
63    TraceStack();
64  }
65
66  ScanRoot() {
67    local t, rt, i;  t := 1;
68    while (t <= N) {
69      rt := get_root(t);
70      foreach i in rt do { MarkAndPush(i); }
71      t := t + 1;
72    }
73  }
74
75  Sweep() {
76    local i, c;  i := 1;
77    while (i <= M) {
78      c := i.color;
79      if (c = WHITE) { free(i); }
80      i := i + 1;
81    }
82  }
```

Figure 4.2 Boehm et al. GC code (continued).

```
update(x, fd, E) { // fd ∈ {pt1, ..., ptm}
  atomic{ x.fd := E; aux := x; }
  atomic{ x.dirty := 1; aux := 0; }
}
```

Figure 4.3 Write barrier for Boehm et al. GC.

and `dirty` fields. The `color` field has three possible values and is used for two purposes: for marking, we use `BLACK` for a marked object and `WHITE` for an unmarked one; and for allocation, we use `BLUE` for an unallocated object which will neither be traced nor be reclaimed, but can be allocated later. New objects are created `BLACK`, and when reclaiming an object, we just set its color to `BLUE`. The `dirty` field is the card bit whose value can be 0 (not dirty) or 1 (dirty). We also assume the total number of mutator threads is `N` (their IDs are $1, 2, \ldots,$ `N`) and the heap domain is [1..`M`].

To make the GC code more readable, we divide it into several methods in Figures 4.1 and 4.2, which should be viewed as macros. The GC thread executes `Collection()` and repeats the collection cycle (the loop body in the method) forever. In each collection cycle, it first clears the `dirty` cards and resets the `color`s of all the objects (the method call of `Initialize()`). After the initialization, the GC enters the mark phase by calling `Trace()`. The command `rt` := **get_root**(`t`) (line 27 in Figure 4.1) allows the GC to read the values of all the pointer variables in the thread `t`'s store at once to a set `rt`, and **foreach** `i` **in** `rt` **do** C (line 28) allows to execute C for every value `i` in `rt`. Our atomic **get_root** tries to reflect the real-world GC implementation [5], where the GC stops a mutator thread to scan its roots. Then a *mark stack* `mstk` is used to do the depth-first tracing in the method `TraceStack()`. For simplicity, we assume there are primitive commands **push**(x, `mstk`) and x := **pop**(`mstk`) to manipulate `mstk`. The stop-the-world phase (line 10) is implemented by an atomic block **atomic**$\{C\}$ that executes C without interference from mutator threads. Here the roots are re-scanned in `ScanRoot()`, because the write barrier is not applied to the roots and we should assume conservatively that they have been modified. Finally in the sweep phase (the call of `Sweep()` at line 11), the GC can use **free**(x) to reclaim the object x. Usually in practice, there is also a concurrent card-cleaning phase (the call of `CleanCard()` at line 9) before the stop-the-world card-cleaning (at line 10) to reduce the pause time of the latter.

The write barrier is shown in Figure 4.3, where the `dirty` field is set after modifying the object's pointer field. Here we use a write-only auxiliary variable `aux`

$$
\begin{array}{rrl}
(\mathit{HExpr}) & \mathbb{E} & ::= x \mid n \mid \mathbf{nil} \mid \mathbb{E}{+}\mathbb{E} \mid \mathbb{E}{-}\mathbb{E} \mid \ldots \\
(\mathit{HBExp}) & \mathbb{B} & ::= \mathbf{true} \mid \mathbf{false} \mid \mathbb{E}{=}\mathbb{E} \mid !\mathbb{B} \mid \ldots \\
(\mathit{HInstr}) & \mathbb{c} & ::= \boldsymbol{print}(\mathbb{E}) \mid x{:=}\mathbb{E} \mid x := y.\mathrm{fd} \mid x.\mathrm{fd} := \mathbb{E} \mid x := \boldsymbol{new}() \\
(\mathit{HStmts}) & \mathbb{C} & ::= \boldsymbol{skip} \mid \mathbb{c} \mid \mathbb{C}_1;;\mathbb{C}_2 \mid \boldsymbol{if}\ \mathbb{B}\ \boldsymbol{then}\ \mathbb{C}_1\ \boldsymbol{else}\ \mathbb{C}_2 \mid \boldsymbol{while}\ \mathbb{B}\ \boldsymbol{do}\ \mathbb{C} \\
(\mathit{HProg}) & \mathbb{W} & ::= \mathtt{t}_{\mathrm{gc}}.\mathit{AbsGC} \,|||\, \mathtt{t}_1.\mathbb{C}_1 \,|||\, \ldots \,|||\, \mathtt{t}_n.\mathbb{C}_n \\
(\mathit{HField}) & \mathrm{fd} & \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m, \mathtt{data}\} \qquad (\mathit{MutID})\ \mathtt{t} \in [1..N]
\end{array}
$$

(a) the language

$$
\begin{array}{rrlrrl}
(\mathit{Loc}) & l & \in \{L_1, \ldots, L_M, \mathbf{nil}\} & (\mathit{HVal}) & \mathrm{V} & \in \mathit{Int} \cup \mathit{Loc} \\
(\mathit{HStore}) & \mathbb{s} & \in \mathit{PVar} \rightharpoonup \mathit{HVal} & (\mathit{HObj}) & \mathrm{O} & \in \mathit{HField} \rightharpoonup \mathit{HVal} \\
(\mathit{HHeap}) & \mathbb{h} & \in \mathit{Loc} \rightharpoonup \mathit{HObj} & (\mathit{HThrds}) & \Pi & \in \mathit{MutID} \rightharpoonup \mathit{HStore} \\
(\mathit{HState}) & \Sigma & \in \mathit{HThrds} \times \mathit{HHeap} & & &
\end{array}
$$

(b) program states

Figure 4.4 High-level language and state model.

for each mutator thread to record the current object that the mutator is updating. We add `aux` for the purpose of verification only, which can be safely deleted after the proof is completed. It helps specify some fine-grained and temporal property of the write barrier in the guarantees. For instance, a mutator should ensure that after it sets a pointer field of an object x to another object y, it must first set x's `dirty` field before updating other pointers (in particular, those pointing to y). Otherwise, the GC may not know that y is newly reachable from x and may finally reclaim y. In Figure 4.3, we set `aux` to the object x when its pointer field is updated, and specify in the mutator's guarantee $\mathcal{G}$ that when $\mathtt{aux} = x$, it must set x's `dirty` field (see $\mathcal{G}^{\mathtt{t}}_{\mathrm{set\_dirty}}$ in Figure 4.12(b)). The GC does not use read barriers nor allocation barriers. Allocation can be implemented using a standard concurrent list algorithm. To be more focused on verifying the GC algorithm itself, we model allocation as an abstract instruction $x :=$ **new**() which can magically find an unallocated (`BLUE`) object in the heap.

4.3.2 The Transformation

We first present the detailed high-level and low-level languages and state models in Figures 4.4 and 4.5 respectively, which are instantiations of the generic languages in Figure 2.2.

- An object has m pointer fields and a data field from the high-level view, whereas a concrete object also has two auxiliary fields `color` and `dirty` for

$$
\begin{array}{rrl}
(\mathit{LExpr}) & E & ::= x \mid n \mid E{+}E \mid E{-}E \mid \ldots \\
(\mathit{LBExp}) & B & ::= \textbf{true} \mid \textbf{false} \mid E{=}E \mid !B \mid \texttt{is\_empty}(x) \mid \ldots \\
(\mathit{LInstr}) & c & ::= \textbf{print}(E) \mid x{:=}E \mid x := y.\mathit{fd} \mid x.\mathit{fd} := E \mid x := \textbf{new}() \\
 & & \quad \mid x := \textbf{get\_root}(y) \mid \textbf{free}(x) \mid \textbf{push}(x, y) \mid x := \textbf{pop}(y) \\
(\mathit{LStmts}) & C & ::= \textbf{skip} \mid c \mid C_1; C_2 \mid \textbf{if } (B)\ C_1 \textbf{ else } C_2 \mid \textbf{while } (B)\ C \\
 & & \quad \mid \textbf{atomic}\{C\} \mid \textbf{foreach } x \textbf{ in } y \textbf{ do } C \\
(\mathit{LProg}) & W & ::= \mathtt{t}_{\mathrm{gc}}.C_{\mathrm{gc}} \,\|\, \mathtt{t}_1.C_1 \,\|\, \ldots \,\|\, \mathtt{t}_n.C_n \\
(\mathit{LField}) & \mathit{fd} & \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m, \mathtt{data}, \mathtt{color}, \mathtt{dirty}\}
\end{array}
$$

(a) the language

$$
\begin{array}{rrlrrl}
(\mathit{LVal}) & v & \in \mathit{Int} \cup \mathscr{P}(\mathit{Int}) \cup \mathit{Seq}(\mathit{Int}) & (\mathit{LStore}) & s & \in \mathit{PVar} \rightharpoonup \mathit{LVal} \times \{0,1\} \\
(\mathit{LObj}) & o & \in \mathit{LField} \rightharpoonup \mathit{LVal} & (\mathit{LHeap}) & h & \in [1..M] \rightharpoonup \mathit{LObj} \\
(\mathit{LThrds}) & \pi & \in (\mathit{MutID} \cup \{\mathtt{t}_{\mathrm{gc}}\}) \rightharpoonup \mathit{LStore} & (\mathit{LState}) & \sigma & \in \mathit{LThrds} \times \mathit{LHeap}
\end{array}
$$

(b) program states

Figure 4.5 Low-level language and state model.

the collection.

- The set AbsGCStep of behaviors of the high-level abstract GC thread is defined in Figure 4.6(a), saying that the mutator stores and the reachable objects in the heap remain unmodified. Here $\mathsf{Reachable}(l)(\Pi, \mathbb{h})$ means the object at the location l is reachable in $\mathbb{h}$ from the roots in Π.

- The low-level concrete GC thread could use privileged commands, such as $x := \textbf{get\_root}(y)$ and $\textbf{free}(x)$, to control the mutator threads and manage the heap.

- High-level mutators can use $x := y.\mathrm{fd}$ to read a field of an object, $x.\mathrm{fd} := \mathbb{E}$ to write the value of $\mathbb{E}$ to a field of an object and $x := \textbf{\textit{new}}()$ to allocate a new object. If the instruction $x.\mathrm{fd} := \mathbb{E}$ updates a pointer field (i.e., $\mathrm{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}$), then it will be transformed to the write barrier in Figure 4.3. Note here $\mathbb{E}$ is restricted to be either **nil** (null pointers) or pointer variables.

- The high-level language is typed in the sense that heap locations and integers are regarded as distinct kinds (or types) of values. We present the high-level operational semantics in Figure 4.6(b). Here we use $\mathsf{sameType}(\mathrm{V}, \mathrm{V}')$ to mean that the two values V and V′ are of the same type.

$$
\begin{array}{lcl}
\mathsf{Root}(\mathsf{t}, S) & \stackrel{\text{def}}{=} & \lambda\Sigma.\ \Sigma = (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}_\mathsf{t}\}, \mathbb{h}) \wedge S = \{l \mid \exists x.\mathbb{s}_\mathsf{t}(x) = l\} \\
\mathsf{Edge}(l_1, l_2) & \stackrel{\text{def}}{=} & \lambda\Sigma.\ \Sigma = (\Pi, \mathbb{h}) \wedge \exists \mathrm{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}.\ \mathbb{h}(l_1)(\mathrm{fd}) = l_2 \\
\mathsf{Path}_k(l_1, l_2) & \stackrel{\text{def}}{=} & \begin{cases} l_1 = l_2 & \text{if } k = 0 \\ \exists l_3.\ \mathsf{Edge}(l_1, l_3) \wedge \mathsf{Path}_{k-1}(l_3, l_2) & \text{if } k > 0 \end{cases} \\
\mathsf{Path}(l_1, l_2) & \stackrel{\text{def}}{=} & \exists k.\ \mathsf{Path}_k(l_1, l_2) \\
\mathsf{Reachable}(\mathsf{t}, l) & \stackrel{\text{def}}{=} & \exists S, l'.\ \mathsf{Root}(\mathsf{t}, S) \wedge l' \in S \wedge \mathsf{Path}(l', l) \wedge l \neq \mathbf{nil} \\
\mathsf{Reachable}(l) & \stackrel{\text{def}}{=} & \exists \mathsf{t} \in [1..N].\ \mathsf{Reachable}(\mathsf{t}, l) \\
\mathsf{AbsGCStep} & \stackrel{\text{def}}{=} & \{((\Pi, \mathbb{h}), (\Pi, \mathbb{h}')) \mid \forall l.\ \mathsf{Reachable}(l)(\Pi, \mathbb{h}) \Longrightarrow \mathbb{h}(l) = \mathbb{h}'(l)\}
\end{array}
$$

(a) definition of AbsGCStep

$$
\frac{\mathbb{s}(x) = l \quad \mathbb{h}(l) = \mathrm{O} \quad [\![\mathbb{E}]\!]_\mathbb{s} = \mathrm{V} \quad \mathrm{O}(\mathrm{fd}) = \mathrm{V}' \quad \mathsf{sameType}(\mathrm{V}, \mathrm{V}')}{(x.\mathrm{fd} := \mathbb{E}, (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h})) \longrightarrow_\mathsf{t} (\textbf{\textit{skip}}, (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h}\{l \rightsquigarrow \mathrm{O}\{\mathrm{fd} \rightsquigarrow \mathrm{V}\}\}))}
$$

$$
\frac{x \notin \mathit{dom}(\mathbb{s}) \quad \text{or} \quad \mathbb{s}(x) \notin \mathit{dom}(\mathbb{h}) \quad \text{or} \quad [\![\mathbb{E}]\!]_\mathbb{s} = \bot \quad \text{or} \quad \neg\mathsf{sameType}(\mathbb{h}(\mathbb{s}(x))(\mathrm{fd}), [\![\mathbb{E}]\!]_\mathbb{s})}{(x.\mathrm{fd} := \mathbb{E}, (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h})) \longrightarrow_\mathsf{t} \mathbf{abort}}
$$

$$
\frac{\begin{array}{c} l \notin \mathit{dom}(\mathbb{h}) \qquad l \neq \mathbf{nil} \qquad \mathbb{s}(x) = l' \qquad \mathbb{s}' = \mathbb{s}\{x \rightsquigarrow l\} \\ \mathbb{h}' = \mathbb{h} \uplus \{l \rightsquigarrow \{\mathrm{pt}_1 \rightsquigarrow \mathbf{nil}, \ldots, \mathrm{pt}_m \rightsquigarrow \mathbf{nil}, \mathtt{data} \rightsquigarrow 0\}\} \end{array}}{(x := \textbf{\textit{new}}(), (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h})) \longrightarrow_\mathsf{t} (\textbf{\textit{skip}}, (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}'\}, \mathbb{h}'))}
$$

$$
\frac{\neg(\exists l.l \notin \mathit{dom}(\mathbb{h}) \wedge l \neq \mathbf{nil}) \quad \mathbb{s}(x) = l' \quad \mathbb{s}' = \mathbb{s}\{x \rightsquigarrow \mathbf{nil}\}}{(x := \textbf{\textit{new}}(), (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h})) \longrightarrow_\mathsf{t} (\textbf{\textit{skip}}, (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}'\}, \mathbb{h}))}
$$

$$
\frac{x \notin \mathit{dom}(\mathbb{s}) \quad \text{or} \quad \neg\exists l.\mathbb{s}(x) = l}{(x := \textbf{\textit{new}}(), (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h})) \longrightarrow_\mathsf{t} \mathbf{abort}}
$$

$$
\frac{(\mathbb{C}_i, \Sigma) \longrightarrow_{\mathsf{t}_i} (\mathbb{C}_i', \Sigma') \quad \text{or} \quad (\Sigma, \Sigma') \in \mathsf{AbsGCStep} \wedge \mathbb{C}_i' = \mathbb{C}_i}{(\mathsf{t}_{\mathrm{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots\, \mathsf{t}_i.\mathbb{C}_i \ldots |||\, \mathsf{t}_n.\mathbb{C}_n, \Sigma) \longrightarrow (\mathsf{t}_{\mathrm{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots\, \mathsf{t}_i.\mathbb{C}_i' \ldots |||\, \mathsf{t}_n.\mathbb{C}_n, \Sigma')}
$$

$$
\frac{(\mathbb{C}_i, \Sigma) \longrightarrow_{\mathsf{t}_i} \mathbf{abort}}{(\mathsf{t}_{\mathrm{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots\, \mathsf{t}_i.\mathbb{C}_i \ldots |||\, \mathsf{t}_n.\mathbb{C}_n, \Sigma) \longrightarrow \mathbf{abort}}
$$

(b) selected operational semantics rules

Figure 4.6 High-level garbage-collected machine.

$$
\begin{aligned}
[\![n]\!]_{(s,tag)} &= \begin{cases} n & \text{if } tag = 0 \text{ or } tag = 2 \\ 0 & \text{if } tag = 1 \text{ and } n = 0 \\ \bot & \text{otherwise} \end{cases} \\
[\![x]\!]_{(s,tag)} &= \begin{cases} n & \text{if } s(x) = (n, b) \text{ and } (tag = b \vee tag = 2) \\ \bot & \text{otherwise} \end{cases} \\
[\![E_1 + E_2]\!]_{(s,tag)} &= \begin{cases} n_1 + n_2 & \text{if } [\![E_1]\!]_{(s,tag)} = n_1 \text{ and } [\![E_2]\!]_{(s,tag)} = n_2 \\ & \quad \text{and } (tag = 0 \vee tag = 2) \\ \bot & \text{otherwise} \end{cases} \\
[\![\texttt{is\_empty}(x)]\!]_{(s,tag)} &= \begin{cases} \textbf{true} & \text{if } tag = 0 \text{ and } s(x) = (\epsilon, 0) \\ \textbf{false} & \text{if } tag = 0 \text{ and } s(x) = (n :: A, 0) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.7 Expression evaluation on the low-level machine.

- On the low-level machine, we allow the GC to perform pointer arithmetic, so we do not distinguish locations and integers. A low-level value v can be an integer, a set or a sequence of integers. We use $\mathscr{P}(\_)$ for the power set and $Seq(\_)$ for the set of sequences. Every low-level variable is given an extra bit to preserve its high-level type information (0 for non-pointers and 1 for pointers), so that the GC can easily get the roots. The low-level mutators are still prohibited from pointer arithmetic. An expression E is evaluated (shown in Figure 4.7) under the store s with an extra tag tag to indicate whether it is used as an object location in the heap ($tag = 1$ if E is used as a heap location; and $tag = 0$ otherwise). When $tag = 2$, we do not care about the usage of the expression, and such an expression will be used in the GC code since the GC has the privilege to use an integer as an address and vice versa. We present part of the low-level operational semantics rules in Figure 4.8. To formulate the semantics of **foreach** x **in** y **do** C, we assume x and y are temporary variables and not updated by C. At the beginning of each iteration, we set x to an arbitrary item in the set y, and after executing C we remove that item from y. The **foreach** loop terminates when y becomes empty.

- We do not provide infinite heaps; instead there are only M valid high-level locations and the low-level heap domain is $[1..M]$. High-level mutators can use **nil** for null pointers and it will be translated to 0 on the low-level machine. We assume there is a bijective function Loc2Int from high-level locations to low-level integers.

$$\mathsf{Loc2Int} : \ Loc \leftrightarrow [0..M]$$

$$\frac{\mathtt{t} \in [1..N] \quad s(x) = (\_, b) \quad [\![E]\!]_{(s,b)} = n \quad s' = s\{x \rightsquigarrow (n, b)\}}{(x := E, (\pi \uplus \{\mathtt{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t} \rightsquigarrow s'\}, h))}$$

$$\frac{s(x) = (\_, b) \quad [\![E]\!]_{(s,2)} = n \quad s' = s\{x \rightsquigarrow (n, b)\}}{(x := E, (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}_{\mathrm{gc}}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s'\}, h))}$$

$$\frac{\begin{array}{c} s(y) = (n_y, 1) \quad h(n_y)(fd) = n \quad s(x) = (\_, b) \\ fd \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \Longrightarrow b = 1 \quad fd \in \{\mathtt{data}\} \Longrightarrow b = 0 \quad s' = s\{x \rightsquigarrow (n, b)\} \end{array}}{(x := y.fd, (\pi \uplus \{\mathtt{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t} \rightsquigarrow s'\}, h))}$$

$$\frac{\begin{array}{c} s(x) = (n, 1) \quad h(n) = o \quad fd \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \Longrightarrow [\![E]\!]_{(s,1)} = n' \\ fd \in \{\mathtt{data}\} \Longrightarrow [\![E]\!]_{(s,0)} = n' \quad fd \in \{\mathtt{color}, \mathtt{dirty}\} \Longrightarrow [\![E]\!]_{(s,2)} = n' \end{array}}{(x.fd := E, (\pi \uplus \{\mathtt{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t} \rightsquigarrow s\}, h\{n \rightsquigarrow o\{fd \rightsquigarrow n'\}\}))}$$

$$\frac{s(y) = (\mathtt{t}, 0) \quad s(x) = (\_, 0) \quad \pi(\mathtt{t}) = s_{\mathtt{t}} \quad S = \{n \mid \exists x. s_{\mathtt{t}}(x) = (n, 1)\}}{(x := \mathbf{get\_root}(y), (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}_{\mathrm{gc}}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s\{x \rightsquigarrow (S, 0)\}\}, h))}$$

$$\frac{x \in dom(s) \quad s(y) = (\emptyset, 0)}{(\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}_{\mathrm{gc}}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s\}, h))}$$

$$\frac{s(x) = (\_, b) \quad s(y) = (\{n_1, \ldots, n_k\}, 0) \quad s' = s\{x \rightsquigarrow (n_1, b)\}}{\begin{array}{l}(\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C,\ (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}_{\mathrm{gc}}} \\ \qquad (C; y := y \backslash \{x\}; \mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C,\ (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s'\}, h))\end{array}}$$

$$\frac{(C, (\pi \uplus \{\mathtt{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}}^{*} (\mathbf{skip}, (\pi \uplus \{\mathtt{t} \rightsquigarrow s'\}, h'))}{(\mathbf{atomic}\{C\}, (\pi \uplus \{\mathtt{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t} \rightsquigarrow s'\}, h'))}$$

$$\frac{\begin{array}{c} \mathtt{t} \in [1..N] \quad s(x) = (\_, 1) \quad h(n)(\mathtt{color}) = \mathtt{BLUE} \quad s' = s\{x \rightsquigarrow (n, 1)\} \\ h' = h\{n \rightsquigarrow \{\mathrm{pt}_1 \rightsquigarrow 0, \ldots, \mathrm{pt}_m \rightsquigarrow 0, \mathtt{data} \rightsquigarrow 0, \mathtt{color} \rightsquigarrow \mathtt{BLACK}, \mathtt{dirty} \rightsquigarrow 0\}\} \end{array}}{(x := \mathbf{new}(), (\pi \uplus \{\mathtt{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t} \rightsquigarrow s'\}, h'))}$$

$$\frac{\mathtt{t} \in [1..N] \quad s(x) = (\_, 1) \quad \neg(\exists n.\ h(n)(\mathtt{color}) = \mathtt{BLUE}) \quad s' = s\{x \rightsquigarrow (0, 1)\}}{(x := \mathbf{new}(), (\pi \uplus \{\mathtt{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t} \rightsquigarrow s'\}, h))}$$

$$\frac{s(x) = (n, 1) \quad h(n) = o}{(\mathbf{free}(x), (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}_{\mathrm{gc}}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s\}, h\{n \rightsquigarrow o\{\mathtt{color} \rightsquigarrow \mathtt{BLUE}\}\}))}$$

$$\frac{s(x) = (n', b) \quad s(y) = (A, 0) \quad s' = s\{y \rightsquigarrow (n' :: A, 0)\}}{(\mathbf{push}(x, y), (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}_{\mathrm{gc}}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s'\}, h))}$$

$$\frac{s(x) = (\_, b) \quad s(y) = (n :: A, 0) \quad s' = s\{x \rightsquigarrow (n, b), y \rightsquigarrow (A, 0)\}}{(x := \mathbf{pop}(y), (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathtt{t}_{\mathrm{gc}}} (\mathbf{skip}, (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \rightsquigarrow s'\}, h))}$$

Figure 4.8 Selected operational semantics rules on the low-level machine.

$$\mathbf{T}(\Sigma) \stackrel{\text{def}}{=} (\{\mathsf{t} \rightsquigarrow \mathbf{T}(\mathbb{s}) \mid (\mathsf{t} \rightsquigarrow \mathbb{s}) \in \Pi\} \uplus \{\mathsf{t}_{\text{gc}} \rightsquigarrow s_{\text{gc\_init}}\}, \mathbf{T}(\mathbb{h}))\,, \text{ if } \Sigma = (\Pi, \mathbb{h}) \wedge \mathsf{WfState}(\Sigma)$$

where $\mathsf{WfState}(\Pi, \mathbb{h}) \stackrel{\text{def}}{=} \forall l.\ \mathsf{Reachable}(l)(\Pi, \mathbb{h}) \Longrightarrow l \in \mathit{dom}(\mathbb{h})$

$$s_{\text{gc\_init}} \stackrel{\text{def}}{=} \{\mathtt{mstk} \rightsquigarrow \epsilon^{\text{np}}, \mathtt{rt} \rightsquigarrow \emptyset^{\text{np}}, \mathtt{i} \rightsquigarrow 0^{\text{p}}, \mathtt{j} \rightsquigarrow 0^{\text{p}}, \mathtt{c} \rightsquigarrow 0^{\text{np}}, \mathtt{d} \rightsquigarrow 0^{\text{np}}, \mathtt{t} \rightsquigarrow 0^{\text{np}}\}$$

$$\mathbf{T}(\mathbb{s})(x) \stackrel{\text{def}}{=} \begin{cases} n^{\text{np}} & \text{if } \mathbb{s}(x) = n \\ n^{\text{p}} & \text{if } \mathbb{s}(x) = l \wedge \mathsf{Loc2Int}(l) = n \\ 0^{\text{p}} & \text{if } x = \mathtt{aux} \end{cases}$$

$$\mathbf{T}(\mathbb{h})(i) \stackrel{\text{def}}{=} \begin{cases} \{\mathrm{pt}_1 \rightsquigarrow n_1, \ldots, \mathrm{pt}_m \rightsquigarrow n_m, \mathtt{data} \rightsquigarrow n, \mathtt{color} \rightsquigarrow \mathtt{WHITE}, \mathtt{dirty} \rightsquigarrow 0\} \\ \qquad\qquad \text{if } \exists l.\ l \in \mathit{dom}(\mathbb{h}) \wedge \mathsf{Loc2Int}(l) = i \wedge 1 \le i \le M \\ \qquad\qquad \wedge\ \mathbb{h}(l) = \{\mathrm{pt}_1 \rightsquigarrow l_1, \ldots, \mathrm{pt}_m \rightsquigarrow l_m, \mathtt{data} \rightsquigarrow n\} \\ \qquad\qquad \wedge\ \mathsf{Loc2Int}(l_1) = n_1 \wedge \ldots \wedge \mathsf{Loc2Int}(l_m) = n_m \\ \{\mathrm{pt}_1 \rightsquigarrow 0, \ldots, \mathrm{pt}_m \rightsquigarrow 0, \mathtt{data} \rightsquigarrow 0, \mathtt{color} \rightsquigarrow \mathtt{BLUE}, \mathtt{dirty} \rightsquigarrow 0\} \\ \qquad\qquad \text{if } \exists l.\ l \notin \mathit{dom}(\mathbb{h}) \wedge \mathsf{Loc2Int}(l) = i \wedge 1 \le i \le M \end{cases}$$

Figure 4.9 Transformation $\mathbf{T}$ on initial states for Boehm et al. GC.

It satisfies $\mathsf{Loc2Int}(\mathbf{nil}) = 0$.

The transformation $\mathbf{T}$ is defined as follows. For *code*, the high-level abstract GC thread is transformed to the GC thread shown in Figures 4.1 and 4.2. Each instruction $x.\mathrm{fd} := \mathbb{E}$ in mutators is transformed to the write barrier $\mathtt{update}(x, \mathrm{fd}, \mathbf{T}(\mathbb{E}))$, where fd is a pointer field of x. $\mathbf{T}$ over expressions $\mathbb{E}$ returns 0 if $\mathbb{E}$ is **nil**, and keeps the syntax otherwise. Other instructions and the program structures of mutators are unchanged.

We also need to transform the initial high-level state to the low level. The transformation $\mathbf{T}(\Sigma)$ is defined in Figure 4.9.

- First we require the high-level initial state to be *well-formed* ($\mathsf{WfState}(\Sigma)$), i.e., reachable locations cannot be dangling pointers.
- High-level locations are transformed to integers by the bijective function $\mathsf{Loc2Int}$.
- Variables are transformed to the low level using an extra bit to preserve the high-level type information (0 for non-pointers and 1 for pointers). Usually we use v^{np} and v^{p} short for $(v, 0)$ and $(v, 1)$ respectively.
- High-level objects are transformed to the low level by adding the `color` and `dirty` fields with initial values `WHITE` and 0 respectively. Other addresses in the low-level heap domain $[1..M]$ are filled out using unallocated objects whose `color`s are `BLUE` and all the other fields are initialized by 0.

$$\mathsf{store\_map}(s, \mathbb{s}) \stackrel{\text{def}}{=} \forall x \neq \mathsf{aux}.\ (\forall n.\ s(x) = n^{\text{np}} \iff \mathbb{s}(x) = n) \\ \wedge\ (\forall n.\ s(x) = n^{\text{p}} \iff \exists l.\ \mathsf{Loc2Int}(l) = n \wedge \mathbb{s}(x) = l)$$

$$\mathsf{obj\_map}(o, \mathrm{O}) \stackrel{\text{def}}{=} \exists n_1, \ldots, n_m, n, c, l_1, \ldots, l_m.\ \mathsf{Loc2Int}(l_1) = n_1 \wedge \ldots \wedge \mathsf{Loc2Int}(l_m) = n_m \\ \wedge\ o = \{\mathrm{pt}_1 \rightsquigarrow n_1, \ldots, \mathrm{pt}_m \rightsquigarrow n_m, \mathtt{data} \rightsquigarrow n, \mathtt{color} \rightsquigarrow c, \mathtt{dirty} \rightsquigarrow \_\} \\ \wedge\ c \neq \mathtt{BLUE} \wedge \mathrm{O} = \{\mathrm{pt}_1 \rightsquigarrow l_1, \ldots, \mathrm{pt}_m \rightsquigarrow l_m, \mathtt{data} \rightsquigarrow n\})$$

$$\mathsf{unalloc}(o, \mathbb{h}, l) \stackrel{\text{def}}{=} (o = \{\mathrm{pt}_1 \rightsquigarrow \_, \ldots, \mathrm{pt}_m \rightsquigarrow \_, \mathtt{data} \rightsquigarrow \_, \mathtt{color} \rightsquigarrow \mathtt{BLUE}, \mathtt{dirty} \rightsquigarrow \_\}) \\ \wedge\ l \notin \mathit{dom}(\mathbb{h})$$

$$\mathsf{heap\_map}(h, \mathbb{h}) \stackrel{\text{def}}{=} \forall i, l.\ 1 \leq i \leq M \wedge \mathsf{Loc2Int}(l) = i \\ \implies \mathsf{obj\_map}(h(i), \mathbb{h}(l)) \vee \mathsf{unalloc}(h(i), \mathbb{h}, l)$$

$$\alpha \stackrel{\text{def}}{=} \{((\pi \uplus \{\mathsf{t}_{\text{gc}} \rightsquigarrow \_\}, h), (\Pi, \mathbb{h})) \mid \\ \forall \mathsf{t}.\ \mathsf{store\_map}(\pi(\mathsf{t}), \Pi(\mathsf{t})) \wedge \mathsf{heap\_map}(h, \mathbb{h}) \wedge \mathsf{WfState}(\Pi, \mathbb{h})\}$$

Figure 4.10 Relation α for Boehm et al. GC.

- The concrete GC thread is given an initial store $s_{\text{gc\_init}}$ where its local variables are initialized by 0 (for integer and pointer variables), ϵ (for the mark stack `mstk`) or $\emptyset$ (for the root set `rt`).

To prove $\mathsf{Correct}(\mathbf{T})$ in our framework, we apply Theorem 4.1, prove the refinement between low-level and high-level mutators, and verify the GC code using a unary Rely-Guarantee-based logic.

4.3.3 Refinement Proofs for Mutator Instructions

We first define the α and $\zeta(\mathsf{t})$ relations. In α (see Figure 4.10), the relations between low-level and high-level stores and heaps are enforced by store_map and heap_map respectively. Their definitions reflect the state transformations in Figure 4.9. We ignore the values of those high-level-invisible structures, including the GC's local variables, the `color` and `dirty` fields for non-blue objects and all the fields of blue objects. The relation α also requires the well-formedness of high-level states. Here we still use Loc2Int to relate integers and locations.

For each mutator thread t, the $\zeta(\mathsf{t})$ relation enforced at the beginning and the end of each transformation unit (each high-level instruction) is stronger than α. It requires that the value of the auxiliary variable `aux` (see Figure 4.3) be a null pointer (0^{p}):

$$\zeta(\mathsf{t}) \stackrel{\text{def}}{=} \alpha \cap \{((\pi, h), (\Pi, \mathbb{h})) \mid \pi(\mathsf{t})(\mathtt{aux}) = 0^{\text{p}}\}.$$

To define the guarantees of the mutator instructions, we first introduce some separation logic assertions in Figure 4.11 to describe states. Following Parkinson

$$
\begin{array}{rrcl}
(\mathit{PVarList}) & O & ::= & \bullet \mid x, O \\
(\mathit{StateAssert}) & p, q & \in & \mathit{LThrds} \times \mathit{LStore} \times \mathit{LHeap} \to \mathit{Prop}
\end{array}
$$

(a) state assertions

$$
\begin{array}{ll}
B & \stackrel{\text{def}}{=} \lambda(\pi, s, h).\ [\![B]\!]_{(s,2)} = \mathbf{true} \\
\mathsf{emp}_h & \stackrel{\text{def}}{=} \lambda(\pi, s, h).\ \mathit{dom}(h) = \emptyset \\
\mathsf{own}_{\mathrm{np}}(x) & \stackrel{\text{def}}{=} \lambda(\pi, s, h).\ \mathit{dom}(s) = \{x\} \wedge s(x) = (\_, 0) \\
\mathsf{own}_{\mathrm{p}}(x) & \stackrel{\text{def}}{=} \lambda(\pi, s, h).\ \mathit{dom}(s) = \{x\} \wedge s(x) = (\_, 1) \\
\mathsf{own}(x) & \stackrel{\text{def}}{=} \lambda(\pi, s, h).\ \mathit{dom}(s) = \{x\} \\
p * q & \stackrel{\text{def}}{=} \lambda(\pi, s, h).\ \exists \pi_1, s_1, h_1, \pi_2, s_2, h_2.\ p(\pi_1, s_1, h_1) \wedge q(\pi_2, s_2, h_2) \\
 & \qquad \wedge \pi = \pi_1 \oplus \pi_2 \wedge s = s_1 \uplus s_2 \wedge h = h_1 \oplus h_2 \\
\mathsf{t}.x = E & \stackrel{\text{def}}{=} \lambda(\pi, s, h).\ \exists n, b.\ \pi(\mathsf{t})(x) = (n, b) \wedge [\![E]\!]_{(s,2)} = n \\
E_1.\mathit{fd} \mapsto E_2 & \stackrel{\text{def}}{=} \lambda(\pi, s, h).\ \exists n, n'.\ [\![E_1]\!]_{(s,2)} = n' \wedge \mathit{dom}(h) = \{n'\} \\
 & \qquad \wedge h(n')(\mathit{fd}) = n \wedge \mathit{dom}(h(n')) = \{\mathit{fd}\} \wedge [\![E_2]\!]_{(s,2)} = n \\
E_1.\mathit{fd} \hookrightarrow E_2 & \stackrel{\text{def}}{=} (E_1.\mathit{fd} \mapsto E_2) * \mathbf{true} \\
O_{\mathrm{np}}; O_{\mathrm{p}} \Vdash p & \stackrel{\text{def}}{=} (\mathsf{own}_{\mathrm{np}}(x_1) * \ldots * \mathsf{own}_{\mathrm{np}}(x_i) * \mathsf{own}_{\mathrm{p}}(y_1) * \ldots * \mathsf{own}_{\mathrm{p}}(y_j)) \wedge p \\
 & \qquad \text{where } O_{\mathrm{np}} = x_1, \ldots, x_i, \bullet \text{ and } O_{\mathrm{p}} = y_1, \ldots, y_j, \bullet \\
x \in S & \stackrel{\text{def}}{=} \exists X.\, S = X \uplus \{x\} \\
\circledast_{x \in S}.p(x) & \stackrel{\text{def}}{=} (S = \phi \wedge \mathsf{emp}) \vee (\exists z, S'.\ (S = \{z\} \uplus S') \wedge (\circledast_{x \in S'}.p(x)) * p(z))
\end{array}
$$

(b) shorthand notations for some state assertions ($\uplus$ and $\oplus$ defined below)

$$
\begin{array}{ll}
f_1 \perp f_2 & \stackrel{\text{def}}{=} (\mathit{dom}(f_1) \cap \mathit{dom}(f_2) = \emptyset) \\
f_1 \uplus f_2 & \stackrel{\text{def}}{=} f_1 \cup f_2\,, \quad \text{if } f_1 \perp f_2 \\
h_1 \oplus h_2 & \stackrel{\text{def}}{=} \mathsf{curry}(\mathsf{uncurry}(h_1) \cup \mathsf{uncurry}(h_2))\,, \quad \text{if } \mathsf{uncurry}(h_1) \perp \mathsf{uncurry}(h_2) \\
\pi_1 \oplus \pi_2 & \stackrel{\text{def}}{=} \{\mathsf{t} \rightsquigarrow (\pi_1(\mathsf{t}) \uplus \pi_2(\mathsf{t})) \mid \mathsf{t} \in \mathit{dom}(\pi_1)\} \\
 & \qquad \text{if } \mathit{dom}(\pi_1) = \mathit{dom}(\pi_2) \wedge\ \forall \mathsf{t} \in \mathit{dom}(\pi_1).\ \pi_1(\mathsf{t}) \perp \pi_2(\mathsf{t}) \\
\sigma_1 \oplus \sigma_2 & \stackrel{\text{def}}{=} (\pi, h)\,, \quad \text{if } \sigma_1 = (\pi_1, h_1) \wedge \sigma_2 = (\pi_2, h_2) \wedge \pi_1 \oplus \pi_2 = \pi \wedge h_1 \oplus h_2 = h
\end{array}
$$

(c) disjoint unions

$$
\begin{array}{ll}
p \ltimes_{\mathsf{t}} q & \stackrel{\text{def}}{=} \{((\pi \uplus \{\mathsf{t} \rightsquigarrow s\}, h), (\pi \uplus \{\mathsf{t} \rightsquigarrow s'\}, h'))\ \mid \\
 & \qquad \exists s_1, h_1, s_2, h_2, s_1', h_1'.\ p(\pi, s_1, h_1) \wedge q(\pi, s_1', h_1') \\
 & \qquad \wedge (s = s_1 \uplus s_2) \wedge (h = h_1 \uplus h_2) \wedge (s' = s_1' \uplus s_2) \wedge (h' = h_1' \uplus h_2)\} \\
p \ltimes_{\mathsf{t}} q \text{ provided } p' & \stackrel{\text{def}}{=} (p \ltimes_{\mathsf{t}} q) \cap ((p * p') \ltimes_{\mathsf{t}} (q * p'))
\end{array}
$$

(d) actions

Figure 4.11 Semantics of basic assertions.

et al. [55], we treat program variables as resource, and use $\mathsf{own}_{\mathrm{p}}(x)$ and $\mathsf{own}_{\mathrm{np}}(x)$ for the current thread's ownership of the variable x when it is a pointer and a non-pointer respectively. Assertions are interpreted under (π, s, h), where s is the store of the current thread, π consists of the stores of all the other threads and h is the shared heap. We use $E_1.fd \mapsto E_2$ to specify a single-object single-field heap with E_2 stored in the field fd of the object E_1. The separating conjunction $p * q$ means p and q hold on disjoint states. We define the disjoint union of states in Figure 4.11(c). We use $f_1 \uplus f_2$ as usual to denote the union of two partial functions when their domains are disjoint. Since heaps are curried functions that first map locations to objects, which then map field names to values, they can be transformed to an uncurried form by the $\mathsf{uncurry}$ operator. We then use $h_1 \oplus h_2$ to denote the union when their domains of $\mathsf{uncurry}(h_1)$ and $\mathsf{uncurry}(h_2)$ are disjoint. The disjoint union of states is defined based on the disjoint unions of the shared heaps and the stores for each thread. We use $E_1.fd \hookrightarrow E_2$ for $(E_1.fd \mapsto E_2) * \mathbf{true}$ and $\circledast_{x \in S}.p(x)$ for iterated separating conjunction over the set S. We overload the notations to the high-level machine and use $\mathbb{E}_1.\mathrm{fd} \Mapsto \mathbb{E}_2$ for a single-object single-field heap at the high level.

Following RGSep [66], we define two forms of actions in Figure 4.11(d). $p \ltimes_{\mathtt{t}} q$ represents the update of some state satisfying p to some state satisfying q, where only the current thread $\mathtt{t}$'s store and the shared heap could be modified. The stores π of other threads should remain unchanged, and there may be parts of the current thread's store and the shared heap which are not affected by the action. $p \ltimes_{\mathtt{t}} q$ $\mathsf{provided}$ p' ensures that the context state satisfying p' is not changed by the action.

In Figure 4.12, we give the guarantees of the high-level mutator instructions and the transformed code, which are defined following their operational semantics. We use $(x^{\mathrm{p}} = n)$ short for $(x = n) \wedge \mathsf{own}_{\mathrm{p}}(x)$ and $(x^{\mathrm{np}} = n)$ for $(x = n) \wedge \mathsf{own}_{\mathrm{np}}(x)$. When the context is clear, we omit the superscript. The predicates blueO and newO denote a blue object and a newly allocated object, which are defined in Figure 4.14. Each action just accesses the local store of the mutator and will not touch the GC store.

The refinement between the write barrier at the low level and the pointer update instruction at the high level is formulated as

$$(\mathtt{update}(x, \mathrm{fd}, E), \mathcal{R}(\mathsf{t}), \mathcal{G}^{\mathsf{t}}_{\mathrm{write\_barrier}}) \preceq_{\alpha;\zeta(\mathsf{t})\ltimes\zeta(\mathsf{t})} (x.\mathrm{fd} := \mathbb{E}, \mathbb{R}(\mathsf{t}), \mathbb{G}^{\mathsf{t}}_{\mathrm{write\_pt}}),$$

where $\mathcal{G}^{\mathsf{t}}_{\mathrm{write\_barrier}} \stackrel{\mathrm{def}}{=} \mathcal{G}^{\mathsf{t}}_{\mathrm{write\_pt}} \cup \mathcal{G}^{\mathsf{t}}_{\mathrm{set\_dirty}}$, i.e., the guarantee of the low-level two-step write barrier. $\mathbb{G}^{\mathsf{t}}_{\mathrm{write\_pt}}$ is the guarantee of the high-level atomic write operation.

$$
\begin{array}{lcl}
\mathbb{G}^{\mathsf{t}}_{\text{assgn\_int}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x = n \wedge \mathsf{emp}_h) \ltimes_{\mathsf{t}} (x = n' \wedge \mathsf{emp}_h) \\
\mathbb{G}^{\mathsf{t}}_{\text{assgn\_pt}} & \stackrel{\text{def}}{=} & \exists x, l, l'.\ (x = l \wedge \mathsf{emp}_h) \ltimes_{\mathsf{t}} (x = l' \wedge \mathsf{emp}_h) \\
 & & \quad \mathsf{provided}\ (l' = \mathbf{nil} \vee \exists y.\ y = l' \vee \exists y, \mathrm{fd}.\ y.\mathrm{fd} \Rightarrow l') \\
\mathbb{G}^{\mathsf{t}}_{\text{write\_data}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x.\mathtt{data} \Rightarrow n) \ltimes_{\mathsf{t}} (x.\mathtt{data} \Rightarrow n') \\
\mathbb{G}^{\mathsf{t}}_{\text{write\_pt}} & \stackrel{\text{def}}{=} & \exists x, \mathrm{fd}, l, l'.\ (x.\mathrm{fd} \Rightarrow l) \ltimes_{\mathsf{t}} (x.\mathrm{fd} \Rightarrow l')\ \ \mathsf{provided}\ \ (l' = \mathbf{nil} \vee \exists y.\ y = l') \\
\mathbb{G}^{\mathsf{t}}_{\text{new}} & \stackrel{\text{def}}{=} & \exists x.\ (x = \_ \wedge \mathsf{emp}_h) \ltimes_{\mathsf{t}} (x = l \wedge l.\mathrm{pt}_1 \Rightarrow \mathbf{nil} * \ldots * l.\mathrm{pt}_m \Rightarrow \mathbf{nil} * l.\mathtt{data} \Rightarrow 0) \\
 & & \\
\mathbb{G}(\mathsf{t}) & \stackrel{\text{def}}{=} & \mathbb{G}^{\mathsf{t}}_{\text{assgn\_int}} \cup \mathbb{G}^{\mathsf{t}}_{\text{assgn\_pt}} \cup \mathbb{G}^{\mathsf{t}}_{\text{write\_data}} \cup \mathbb{G}^{\mathsf{t}}_{\text{write\_pt}} \cup \mathbb{G}^{\mathsf{t}}_{\text{new}}
\end{array}
$$

(a) high-level guarantees

$$
\begin{array}{lcl}
\mathcal{G}^{\mathsf{t}}_{\text{assgn\_int}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x^{\mathrm{np}} = n \wedge \mathsf{emp}_h) \ltimes_{\mathsf{t}} (x^{\mathrm{np}} = n' \wedge \mathsf{emp}_h)\ \ \mathsf{provided}\ \ (\mathtt{aux}^{\mathrm{p}} = 0) \\
\mathcal{G}^{\mathsf{t}}_{\text{assgn\_pt}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x^{\mathrm{p}} = n \wedge \mathsf{emp}_h) \ltimes_{\mathsf{t}} (x^{\mathrm{p}} = n' \wedge \mathsf{emp}_h) \\
 & & \quad \mathsf{provided}\ (\mathtt{aux}^{\mathrm{p}} = 0 * (n' = 0 \vee \exists y.\ y^{\mathrm{p}} = n' \\
 & & \qquad \vee \exists y, fd.\ fd \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \wedge y.fd \mapsto n' \vee n = n')) \\
\mathcal{G}^{\mathsf{t}}_{\text{write\_data}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x.\mathtt{data} \mapsto n) \ltimes_{\mathsf{t}} (x.\mathtt{data} \mapsto n')\ \mathsf{provided}\ (\mathtt{aux}^{\mathrm{p}} = 0) \\
\mathcal{G}^{\mathsf{t}}_{\text{write\_pt}} & \stackrel{\text{def}}{=} & \exists x, fd, n, n'.\ (\mathtt{aux}^{\mathrm{p}} = 0 * x.fd \mapsto n) \ltimes_{\mathsf{t}} (\mathtt{aux}^{\mathrm{p}} = x * x.fd \mapsto n') \\
 & & \quad \mathsf{provided}\ ((n' = 0 \vee \exists y.\ y^{\mathrm{p}} = n') \wedge fd \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}) \\
\mathcal{G}^{\mathsf{t}}_{\text{set\_dirty}} & \stackrel{\text{def}}{=} & \exists n.\ (\mathtt{aux}^{\mathrm{p}} = n * n.\mathtt{dirty} \mapsto \_) \ltimes_{\mathsf{t}} (\mathtt{aux}^{\mathrm{p}} = 0 * n.\mathtt{dirty} \mapsto 1) \\
\mathcal{G}^{\mathsf{t}}_{\text{new}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x^{\mathrm{p}} = n * \mathsf{blueO}(n')) \ltimes_{\mathsf{t}} (x^{\mathrm{p}} = n' * \mathsf{newO}(n'))\ \mathsf{provided}\ (\mathtt{aux}^{\mathrm{p}} = 0) \\
 & & \\
\mathcal{G}(\mathsf{t}) & \stackrel{\text{def}}{=} & \mathcal{G}^{\mathsf{t}}_{\text{assgn\_int}} \cup \mathcal{G}^{\mathsf{t}}_{\text{assgn\_pt}} \cup \mathcal{G}^{\mathsf{t}}_{\text{write\_data}} \cup \mathcal{G}^{\mathsf{t}}_{\text{write\_pt}} \cup \mathcal{G}^{\mathsf{t}}_{\text{set\_dirty}} \cup \mathcal{G}^{\mathsf{t}}_{\text{new}}
\end{array}
$$

(b) low-level guarantees

Figure 4.12 Guarantees of mutator instructions.

Recall $\mathcal{R}(\mathsf{t})$ and $\mathbb{R}(\mathsf{t})$ are defined in Eq. (4.2) in Section 4.2. Since the transformation of other high-level instructions is identity, the corresponding refinement proofs are simple. For example, we can prove

$$(x := \mathbf{new}(), \mathcal{R}(\mathsf{t}), \mathcal{G}^{\mathsf{t}}_{\text{new}} \cup \mathcal{G}^{\mathsf{t}}_{\text{assgn\_pt}}) \preceq_{\alpha;\zeta(\mathsf{t})\ltimes\zeta(\mathsf{t})} (x := \mathbf{new}(), \mathbb{R}(\mathsf{t}), \mathbb{G}^{\mathsf{t}}_{\text{new}} \cup \mathbb{G}^{\mathsf{t}}_{\text{assgn\_pt}})\,.$$

4.3.4 Rely-Guarantee Reasoning about the GC Code

We use a unary logic to verify the GC thread. The proof details here are orthogonal to our simulation-based proof (but it is RGSim that allows us to derive Theorem 4.1, which then links proofs in the unary logic with relational proofs). Thus below we only give a sketch of the assertion language, the unary logic, the precondition and the guarantee of the GC thread, the key invariants and the proof structure.

The unary program logic we use to verify the GC thread is a standard Rely-Guarantee logic adapted to the target language. The assertions are defined in

$$\overline{\{(x^{\mathrm{np}} = X') * (1 \leq y^{\mathrm{np}} \leq N)\} x := \mathbf{get\_root}(y) \{(x^{\mathrm{np}} = X) * (1 \leq y^{\mathrm{np}} \leq N \wedge \mathsf{root}(y, X))\}}$$

$$\overline{\{x.\mathtt{color} \mapsto \_\} \mathbf{free}(x) \{x.\mathtt{color} \mapsto \mathtt{BLUE}\}}$$

$$\overline{\{y, O_{\mathrm{np}}; O_{\mathrm{p}} \Vdash x = X \wedge y = Y\} \mathbf{push}(x, y) \{y, O_{\mathrm{np}}; O_{\mathrm{p}} \Vdash x = X \wedge y = X :: Y\}}$$

$$\overline{\{y, O_{\mathrm{np}}; O_{\mathrm{p}} \Vdash x = X \wedge y = X' :: Y\} x := \mathbf{pop}(y) \{y, O_{\mathrm{np}}; O_{\mathrm{p}} \Vdash x = X' \wedge y = Y\}}$$

$$\frac{\{p\} C \{q\} \quad (p \ltimes q) \Rightarrow \mathcal{G}}{\mathsf{Id}; \mathcal{G} \vdash \{p\} \mathbf{atomic}\{C\} \{q\}} \qquad \frac{\mathsf{Id}; \mathcal{G} \vdash \{p\} \mathbf{atomic}\{C\} \{q\} \quad \mathsf{Sta}(\{p, q\}, \mathcal{R})}{\mathcal{R}; \mathcal{G} \vdash \{p\} \mathbf{atomic}\{C\} \{q\}}$$

$$\frac{p \Rightarrow \mathsf{own}_{\mathrm{np}}(y) * \mathbf{true} \quad \mathcal{R}; \mathcal{G} \vdash \{p * \mathsf{own}(x) \wedge x \in y\}\, C; y := y \backslash \{x\}\, \{p * \mathsf{own}(x)\}}{\mathcal{R}; \mathcal{G} \vdash \{p * \mathsf{own}(x)\} \mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C \{p * \mathsf{own}(x) \wedge y = \emptyset\}}$$

Figure 4.13 Selected inference rules for GC verification.

Figure 4.11 and discussed before. We show the inference rules in Figure 4.13. Rules on the top half are for sequential reasoning. Most are exactly the same as separation logic [59] and omitted here. The figure only shows some rules we added for the GC-specific commands (e.g., $x := \mathbf{get\_root}(y)$) and some particular heap manipulation rules adapted to our low-level machine model (e.g., $\mathbf{free}(x)$ just sets the object's `color` to `BLUE`). The concurrency rules in the bottom half follow standard rely-guarantee reasoning. Since primitive instructions should be viewed as atomic code, we could apply the two rules for $\mathbf{atomic}\{C\}$ to verify them. The soundness of the logic with respect to the operational semantics is straightforward and we omit the proofs here.

To verify the GC code, we first give the precondition p_{gc} and the guarantee $\mathcal{G}_{\mathrm{gc}}$ of the GC. The GC starts its executions from a low-level *well-formed* state, i.e., $p_{\mathrm{gc}} \stackrel{\mathrm{def}}{=} \mathsf{wfstate}$. Just corresponding to the high-level WfState definition (see Figure 4.9), the low-level wfstate predicate says that none of the reachable objects are `BLUE`, as follows.

$$\mathsf{wfstate} \stackrel{\mathrm{def}}{=} \circledast_{x \in [1..M]}.\mathsf{obj}(x) \wedge (\forall x.\ \mathsf{reachable}(x) \Longrightarrow \mathsf{notBlue}(x))\,.$$

Here $\mathsf{obj}(x)$, $\mathsf{reachable}(x)$ and $\mathsf{notBlue}(x)$ are all defined in Figure 4.14. $\mathsf{obj}(x)$ means x is a low-level heap location with the $\mathrm{pt}_1, \ldots, \mathrm{pt}_m$, `data`, `color` and `dirty` fields, and $\mathsf{reachable}(x)$ is defined similarly to the high-level definition $\mathsf{Reachable}(l)$ in Figure 4.6. It is easy to see that any low-level initial state transformed from the high level (see Figure 4.9) is well-formed. We define $\mathcal{G}_{\mathrm{gc}}$ as follows.

$$
\begin{array}{lcl}
\mathsf{obj}(x) & \stackrel{\text{def}}{=} & x.\mathrm{pt}_1 \mapsto \_ * \ldots * x.\mathrm{pt}_m \mapsto \_ * x.\mathtt{data} \mapsto \_ * x.\mathtt{color} \mapsto \_ * x.\mathtt{dirty} \mapsto \_ * \mathbf{true} \\
\mathsf{blueO}(x) & \stackrel{\text{def}}{=} & x.\mathrm{pt}_1 \mapsto \_ * \ldots * x.\mathrm{pt}_m \mapsto \_ * x.\mathtt{data} \mapsto \_ * x.\mathtt{color} \mapsto \mathtt{BLUE} * x.\mathtt{dirty} \mapsto \_ \\
\mathsf{newO}(x) & \stackrel{\text{def}}{=} & x.\mathrm{pt}_1 \mapsto 0 * \ldots * x.\mathrm{pt}_m \mapsto 0 * x.\mathtt{data} \mapsto 0 * x.\mathtt{color} \mapsto \mathtt{BLACK} * x.\mathtt{dirty} \mapsto 0 \\
\mathsf{black}(x) & \stackrel{\text{def}}{=} & x.\mathtt{color} \hookrightarrow \mathtt{BLACK} \\
\mathsf{white}(x) & \stackrel{\text{def}}{=} & x.\mathtt{color} \hookrightarrow \mathtt{WHITE} \\
\mathsf{dirty}(x) & \stackrel{\text{def}}{=} & x.\mathtt{dirty} \hookrightarrow 1 \\
\mathsf{notBlue}(x) & \stackrel{\text{def}}{=} & \exists c.\ (x.\mathtt{color} \hookrightarrow c \wedge c \neq \mathtt{BLUE}) \\
\mathsf{notWhite}(x) & \stackrel{\text{def}}{=} & \exists c.\ (x.\mathtt{color} \hookrightarrow c \wedge c \neq \mathtt{WHITE}) \\
\mathsf{notDirty}(x) & \stackrel{\text{def}}{=} & x.\mathtt{dirty} \hookrightarrow 0 \\
\mathsf{root}(\mathsf{t}, S) & \stackrel{\text{def}}{=} & \lambda(\pi, s, h).\ \exists s_{\mathsf{t}}.\ s_{\mathsf{t}} = \pi(\mathsf{t}) \wedge S = \{n \mid \exists x.\ s_{\mathsf{t}}(x) = (n, 1) \wedge x \neq \mathtt{aux}\} \\
\mathsf{edge}(x, y) & \stackrel{\text{def}}{=} & \exists fd \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}.\ (x.fd \hookrightarrow y) \\
\mathsf{path}_k(x, y) & \stackrel{\text{def}}{=} & \begin{cases} x = y & \text{if } k = 0 \\ \exists z.\ \mathsf{edge}(x, z) \wedge \mathsf{path}_{k-1}(z, y) & \text{if } k > 0 \end{cases} \\
\mathsf{path}(x, y) & \stackrel{\text{def}}{=} & \exists k.\ \mathsf{path}_k(x, y) \\
\mathsf{reachable}(\mathsf{t}, x) & \stackrel{\text{def}}{=} & \exists S, y.\ \mathsf{root}(\mathsf{t}, S) \wedge y \in S \wedge \mathsf{path}(y, x) \wedge x \neq 0 \\
\mathsf{reachable}(x) & \stackrel{\text{def}}{=} & \exists t \in [1..N].\ \mathsf{reachable}(t, x) \\
\mathsf{wfstate} & \stackrel{\text{def}}{=} & \circledast_{x \in [1..M]}.\mathsf{obj}(x) \wedge (\forall x.\ \mathsf{reachable}(x) \implies \mathsf{notBlue}(x)) \\
\mathsf{whiteEdge}(x, fd, y) & \stackrel{\text{def}}{=} & (x.fd \hookrightarrow y) \wedge \mathsf{white}(y) \wedge fd \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \\
\mathsf{whiteEdge}(x, y) & \stackrel{\text{def}}{=} & \exists fd.\ \mathsf{whiteEdge}(x, fd, y) \\
\mathsf{todirty}(x, n) & \stackrel{\text{def}}{=} & \exists \mathsf{t}, S.\ (\mathsf{t}.\mathtt{aux} = x \wedge \mathsf{root}(\mathsf{t}, S) \wedge n \in S) \\
\mathsf{instk}(n, A) & \stackrel{\text{def}}{=} & \exists n', A'.\ A = n' :: A' \wedge (n = n' \vee \mathsf{instk}(n, A')) \\
\mathsf{stkBlack}(A) & \stackrel{\text{def}}{=} & \forall x.\ \mathsf{instk}(x, A) \implies \mathsf{black}(x) \\
\mathsf{reachBlack} & \stackrel{\text{def}}{=} & \forall x.\ \mathsf{reachable}(x) \implies \mathsf{black}(x) \\
\mathsf{ptfdSta}(x.fd, y) & \stackrel{\text{def}}{=} & \exists n.\ (x.fd \hookrightarrow n) \wedge (y = n \vee \mathsf{dirty}(x) \vee n = 0 \vee \mathsf{todirty}(x, n)) \\
\mathsf{newOSta}(x) & \stackrel{\text{def}}{=} & \mathsf{obj}(x) \wedge \mathsf{black}(x) \wedge \forall fd \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}.\ \mathsf{ptfdSta}(x.fd, 0) \\
\mathsf{rtBlack}(t) & \stackrel{\text{def}}{=} & \exists S.\ \mathsf{root}(t, S) \wedge \forall n \in S.\ \mathsf{black}(n) \\
\mathsf{rtBlack} & \stackrel{\text{def}}{=} & \forall t \in [1..N].\ \mathsf{rtBlack}(t) \\
\mathsf{markRt}(n) & \stackrel{\text{def}}{=} & \forall t \in [1..n].\ \mathsf{rtBlack}(t) \\
\mathsf{clearColor}(n) & \stackrel{\text{def}}{=} & \forall x \in [1..n].\ (x.\mathtt{color} \hookrightarrow \mathtt{BLACK} \implies \mathsf{newOSta}(x)) \\
\mathsf{clearDirty}(n) & \stackrel{\text{def}}{=} & \forall x \in [1..n].\ \mathsf{notDirty}(x) \\
\mathsf{reclaim}(n) & \stackrel{\text{def}}{=} & \forall x \in [1..n].\ \mathsf{notWhite}(x) \\
\mathsf{reachInv} & \stackrel{\text{def}}{=} & \forall x, y.\ \mathsf{reachable}(x) \wedge \mathsf{black}(x) \wedge \mathsf{whiteEdge}(x, y) \\
 & & \implies \mathsf{dirty}(x) \vee \mathsf{todirty}(x, y) \\
\mathsf{reachStk}(A) & \stackrel{\text{def}}{=} & \forall x, y.\ \mathsf{reachable}(x) \wedge \mathsf{black}(x) \wedge \mathsf{whiteEdge}(x, y) \\
 & & \implies \mathsf{dirty}(x) \vee \mathsf{todirty}(x, y) \vee \mathsf{instk}(x, A) \\
\multicolumn{3}{l}{\mathsf{reachTomk}(A, x_p, S_f, x_n) \stackrel{\text{def}}{=}} \\
\multicolumn{3}{l}{\quad \forall x, fd, y.\ \mathsf{reachable}(x) \wedge \mathsf{black}(x) \wedge \mathsf{whiteEdge}(x, fd, y)} \\
\multicolumn{3}{l}{\quad \implies \mathsf{dirty}(x) \vee \mathsf{todirty}(x, y) \vee \mathsf{instk}(x, A) \vee (x = x_p \wedge fd \in S_f) \vee (y = x_n)}
\end{array}
$$

Figure 4.14 Useful assertions for verifying Boehm et al. GC.

```
{wfstate}
Collection() {
  local mstk: Seq(Int);
  Loop Invariant:  {wfstate ∗ (own_np(mstk) ∧ mstk = ϵ)}
  while (true) {
    Initialize();
    {(wfstate ∧ reachInv) ∗ (own_np(mstk) ∧ mstk = ϵ)}
    Trace();
    {(wfstate ∧ reachInv) ∗ (own_np(mstk) ∧ mstk = ϵ)}
    CleanCard();
    {(wfstate ∧ reachInv) ∗ (own_np(mstk) ∧ mstk = ϵ)}
    atomic{
      ScanRoot();
      { ∃X. (wfstate ∧ reachStk(X) ∧ stkBlack(X) ∧ rtBlack)
      { ∗ (own_np(mstk) ∧ mstk = X)
      CleanCard();
    }
    {(wfstate ∧ reachBlack) ∗ (own_np(mstk) ∧ mstk = ϵ)}
    Sweep();
  }
}
{false}
```

Figure 4.15 Proof outline of `Collection()`.

$$\mathcal{G}_{\text{gc}} \stackrel{\text{def}}{=} \{((\pi \uplus \{\mathsf{t}_{\text{gc}} \rightsquigarrow s\}, h), (\pi \uplus \{\mathsf{t}_{\text{gc}} \rightsquigarrow s'\}, h')) \mid \forall n.\ \mathsf{reachable}(n)(\pi, h) \implies \lfloor h(n) \rfloor = \lfloor h'(n) \rfloor \wedge h(n).\mathtt{color} \neq \mathtt{BLUE} \wedge h'(n).\mathtt{color} \neq \mathtt{BLUE}\}\,.$$

The GC guarantees not modifying the mutator stores. For any mutator-reachable object, the GC does not update its fields coming from the high-level mutator (i.e., the pointer fields and the `data` field), nor does it reclaim the object. Here $\lfloor \_ \rfloor$ lifts a low-level object to a new one that contains mutator data only.

$$\lfloor o \rfloor \stackrel{\text{def}}{=} \{\mathrm{pt}_1 \rightsquigarrow o(\mathrm{pt}_1), \ldots, \mathrm{pt}_m \rightsquigarrow o(\mathrm{pt}_m), \mathtt{data} \rightsquigarrow o(\mathtt{data})\}\,.$$

We could prove that $\mathcal{G}_{\text{gc}}$ does not contain more behaviors than $\mathsf{AbsGCStep}$.

$$\mathcal{G}_{\text{gc}} \circ \alpha^{-1} \subseteq \alpha^{-1} \circ \mathsf{AbsGCStep}\,.$$

We present the proof of the top-level collection cycle in Figure 4.15. One of the key invariants used in the proofs is $\mathsf{reachInv}$ (defined in Figure 4.14). It says, if a reachable `BLACK` object x points to a `WHITE` object y, then either x is dirty or a mutator is going to dirty x (the predicate $\mathsf{todirty}(x, y)$ holds). The latter occurs when the mutator thread `t` has done the first step of its write barrier $\mathtt{update}(x, fd, y)$. We have $\mathtt{t.aux} = x$ and from the mutator's guarantees (Figure 4.12(b)), we know `t` must be going to dirty x.

Each instruction in the GC code is atomic and can be verified by applying the rules for **atomic**$\{C\}$ in Figure 4.13. As required by the second rule for **atomic**$\{C\}$, we need to stabilize the pre- and post-conditions in the verification. For example, when reading a pointer field of an object to a local variable, the postcondition should be stabilized as follows since the parallel mutators might update the field.

$$\mathcal{R}_{\text{gc}}; \mathcal{G}_{\text{gc}} \vdash \begin{array}{l} \left\{ \ \exists X, Y.\ (\texttt{j} = Y) * (\texttt{i}.\text{pt}_1 \hookrightarrow X) \ \right\} \\ \quad \texttt{j := i.pt1;} \\ \left\{ \ \exists X.\ (\texttt{j} = X) * \mathsf{ptfdSta}(\texttt{i}.\text{pt}_1, X) \ \right\} \end{array} \tag{4.5}$$

Here $\mathsf{ptfdSta}(\texttt{i}.\text{pt}_1, X)$ is defined in Figure 4.14. It says either the pt_1 field of `i` is X, or `i` is (or is going to be) marked as dirty. Similarly, when reading the `color` of an object, the postcondition should take into account the mutators' possible update of the `color` field via allocation.

$$\mathcal{R}_{\text{gc}}; \mathcal{G}_{\text{gc}} \vdash \begin{array}{l} \left\{ \ \exists X, Y.\ (\texttt{c} = X) * (\texttt{i.color} \hookrightarrow Y) \ \right\} \\ \quad \texttt{c := i.color;} \\ \left\{ \begin{array}{l} \exists X, Y.\ (\texttt{c} = X) * (\texttt{i.color} \hookrightarrow Y) \\ \wedge\ (X = Y \vee (X = \texttt{BLUE} \wedge \mathsf{newOSta}(\texttt{i}))) \end{array} \right\} \end{array} \tag{4.6}$$

As defined in Figure 4.14, $\mathsf{newOSta}(\texttt{i})$ says `i` points to a new object whose color field is `BLACK`, and each pointer field is either 0 or updated (in this case, the object is dirty).

The module `MarkAndPush(i)` will be called several times in the GC code, so we first give its general specification here. When the object `i` is `WHITE`, `MarkAndPush(i)` colors it `BLACK` and pushes it onto the mark stack.

$$\mathcal{R}_{\text{gc}}; \mathcal{G}_{\text{gc}} \vdash \begin{array}{l} \left\{ \begin{array}{l} \exists A.\ \mathsf{wfstate} \wedge \mathsf{reachTomk}(A, x_p, S_f, \texttt{i}) \\ \wedge\ \mathsf{stkBlack}(A) \wedge (\texttt{i} = 0 \vee \mathsf{obj}(\texttt{i})) \end{array} \right\} \\ \quad \texttt{MarkAndPush(i);} \\ \left\{ \begin{array}{l} \exists A.\ \mathsf{wfstate} \wedge \mathsf{reachTomk}(A, x_p, S_f, 0) \\ \wedge\ \mathsf{stkBlack}(A) \wedge (\texttt{i} = 0 \vee \mathsf{notWhite}(\texttt{i})) \end{array} \right\} \end{array} \tag{4.7}$$

As defined in Figure 4.14, $\mathsf{reachTomk}(A, x_p, S_f, x_n)$ means, if a reachable `BLACK` object x points to a `WHITE` object y via the field fd, then one of the following cases holds.

(1) $\texttt{dirty}(x) \vee \mathsf{todirty}(x)$: x is (or is going to be) marked as dirty, as required in

reachInv.

(2) instk(x, A): x is on the stack A.

(3) $x = x_p \wedge fd \in S_f$: x is x_p, and fd is a field in S_f.

(4) $y = x_n$: y is x_n.

The case (2) will be useful during tracing when some objects have been colored `BLACK` and pushed onto the stack. We define reachStk to express that only cases (1) and (2) are satisfied. We will discuss the uses of the last two cases later.

Every collection cycle in Figure 4.15 starts from a well-formed state (wfstate) with an empty mark stack `mstk` in the GC's local store. Then the GC does the following in order.

1. Concurrent initialization (`Initialize()`, shown in Figure 4.16). We use clearColor(n) to mean that the GC has done color-clearing from locations 1 to n in the heap, but there might still be `BLACK` objects since the mutators could allocate an `BLACK` object after the GC's clearing. We could prove reachInv holds when the GC has cleared the colors of all the objects in the heap, as shown in the following lemma.

   Lemma 4.2. wfstate $\wedge$ clearColor$(M) \Longrightarrow$ reachInv.

   That is, after initialization, if a `BLACK` reachable object x points to a `WHITE` object y, then x must be a newly-allocated object whose pointer field is updated and dirty bit is (or is going to be) set to 1.

2. Concurrent mark-phase (`Trace()`, shown in Figure 4.17).

   (a) The GC first calls `MarkAndPush(i)` to mark and push every root object. We need the following two lemmas to relate the unified pre- and post-conditions of `MarkAndPush(i)` in judgment (4.7) and the actual pre- and post-conditions when calling the module.

   Lemma 4.3. reachStk$(X) \Longrightarrow$ reachTomk$(X, 0, \emptyset, i)$.

   Lemma 4.4. reachTomk$(X, 0, \emptyset, 0) \Longrightarrow$ reachStk(X).

   Then by the consequence rule, we can reuse the proof of `MarkAndPush(i)`.

   (b) Then the GC calls the module `TraceStack()` (Figure 4.18) to perform the depth-first traversal. The loop invariant reachStk holds at each time before the GC pops an object from the mark stack. Suppose the top object `i` on the mark stack points to a `WHITE` object x. The GC does the following in order:

$\{\mathsf{wfstate}\}$
```
Initialize() {
  local i: [1..M], c: {BLACK, WHITE, BLUE};
  i := 1;
```
Loop Invariant: $\{(\mathsf{wfstate} \wedge \mathsf{clearColor}(\mathtt{i}-1) \wedge 1 \le \mathtt{i} \le M+1) * \mathsf{own}_{\mathrm{np}}(\mathtt{c})\}$
```
  while (i <= M) { ... } // See Figure 4.1 for the full code
}
```
$\{\mathsf{wfstate} \wedge \mathsf{reachInv}\}$ *// using Lemma 4.2*

Figure 4.16 Proof outline of `Initialize()`.

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$
```
Trace() {
  local t: [1..N], rt: Set(Int), i: [0..M];
  t := 1;
```
Loop Invariant: $\left\{ \begin{array}{l} (\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon) \\ * (\mathsf{own}_{\mathrm{np}}(\mathtt{t}) \wedge 1 \le \mathtt{t} \le N+1) * \mathsf{own}_{\mathrm{np}}(\mathtt{rt}) * \mathsf{own}_{\mathrm{p}}(\mathtt{i}) \end{array} \right\}$
```
  while (t <= N) {
    rt := get_root(t);
```
Foreach Invariant: $\{FInv\}$
```
    foreach i in rt do {
```
$\{FInv \wedge \mathtt{i} \in \mathtt{rt}\}$ *// using Lemma 4.3*
```
      MarkAndPush(i);
```
$\{FInv \wedge \mathtt{i} \in \mathtt{rt}\}$ *// using Lemma 4.4*
```
    }
    t := t + 1;
```
$\left\{ \begin{array}{l} \exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X)) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) \\ * (\mathsf{own}_{\mathrm{np}}(\mathtt{t}) \wedge 1 \le \mathtt{t} \le N+1) * \mathsf{own}_{\mathrm{np}}(\mathtt{rt}) * \mathsf{own}_{\mathrm{p}}(\mathtt{i}) \end{array} \right\}$
```
    TraceStack();
```
$\left\{ \begin{array}{l} (\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon) \\ * (\mathsf{own}_{\mathrm{np}}(\mathtt{t}) \wedge 1 \le \mathtt{t} \le N+1) * \mathsf{own}_{\mathrm{np}}(\mathtt{rt}) * \mathsf{own}_{\mathrm{p}}(\mathtt{i}) \end{array} \right\}$
```
  }
}
```
$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

where $FInv \stackrel{\text{def}}{=}$
$\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X)) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)$
$* (\mathsf{own}_{\mathrm{np}}(\mathtt{t}) \wedge 1 \le \mathtt{t} \le N) * (\mathsf{own}_{\mathrm{np}}(\mathtt{rt}) \wedge \forall n \in \mathtt{rt}.\ 0 \le n \le M) * \mathsf{own}_{\mathrm{p}}(\mathtt{i})$

Figure 4.17 Proof outline of `Trace()`.

i. Pop `i`. Then the `BLACK` object `i` that points to x is not on the stack now.

ii. Read the pt_1 field of `i` to a local variable `j`. Then $\mathsf{ptfdSta}(\mathtt{i}.\mathrm{pt}_1, \mathtt{j})$ holds as we explained in judgment (4.5). Also we know x must be either `j`, or pointed to by `i` via fields $\mathrm{pt}_2, \ldots, \mathrm{pt}_m$. Thus we have $\mathsf{reachTomk}(\mathtt{mstk}, \mathtt{i}, \{\mathrm{pt}_2, \ldots, \mathrm{pt}_m\}, \mathtt{j})$ holds. Formally, the following lemma holds:

Lemma 4.5.

A. $\mathsf{reachStk}(i :: X) \iff \mathsf{reachTomk}(X, i, \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}, 0)$;

B. $\mathsf{reachTomk}(X, i, S_f, 0) \implies \mathsf{reachTomk}(X, i, S_f, j)$;

C. $\mathsf{reachTomk}(X, i, S_f, j) \wedge \mathsf{ptfdSta}(i.fd, j) \wedge fd \in S_f$
$\implies \mathsf{reachTomk}(X, i, S_f \backslash \{fd\}, j)$.

iii. `MarkAndPush(j)`. We can reuse the proof of this module again.

iv. Mark and push other children. The proof is similar to the above two steps, so we omit the discussions. Finally, reachStk holds because no reachable `WHITE` object needs to rely on the reachability from `i` (it could be reachable from a child of `i` which is on the stack now).

After tracing, we can ensure reachInv still holds. That is, if a `BLACK` object x points to a `WHITE` object, then x must be (or is going to be) dirty and its pointer field is updated by the mutators.

3. Concurrent card-cleaning (`CleanCard()`, as shown in Figure 4.19). We reuse the proof of `TraceStack()` via the frame rule. We can conclude reachInv is maintained at the end of this phase.

4. Stop-the-world card-cleaning.

   (a) The GC first re-scans the roots (`ScanRoot()`, shown in Figure 4.20) as if they were dirty. Then reachStk and rtBlack hold. rtBlack says all the root objects are `BLACK`. Moreover, all the objects on the stack are `BLACK` (denoted by stkBlack). The atomic `MarkAndPush(i)` is proved similarly to the concurrent one in (4.7) with the same pre- and post-conditions.

   (b) Then the GC cleans the cards (the atomic `CleanCard()`, shown in Figure 4.21) without the interference from the mutators. At the end, the mark stack is empty and all the reachable objects are `BLACK` (denoted by reachBlack). The proof for the atomic `TraceStack()` is similar to the proof of the concurrent one and omitted here.

$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X)) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)\}$

```
TraceStack() {
  local i: [1..M], j: [0..M];
```

Loop Invariant: $\left\{\begin{array}{l}\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X)) \\ \quad * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own}_{\mathrm{p}}(\mathtt{i}) * \mathsf{own}_{\mathrm{p}}(\mathtt{j})\end{array}\right\}$

```
  while (!is_empty(mstk)) {
    i := pop(mstk);
```

$\left\{\begin{array}{l}\exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(\mathtt{i} :: X') \wedge \mathsf{stkBlack}(X') \wedge \mathsf{obj}(\mathtt{i})) \\ \quad * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X') * \mathsf{own}_{\mathrm{p}}(\mathtt{j})\end{array}\right\}$

```
    j := i.pt1;
```

$\left\{\begin{array}{l}\exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(\mathtt{i} :: X') \wedge \mathsf{stkBlack}(X') \wedge \mathsf{obj}(\mathtt{i}) \\ \qquad \wedge \mathsf{ptfdSta}(\mathtt{i}.\mathrm{pt}_1, \mathtt{j}) \wedge (\mathtt{j} = 0 \vee \mathsf{obj}(\mathtt{j}))) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X')\end{array}\right\}$

$\left\{\begin{array}{l}\exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reachTomk}(X', \mathtt{i}, \{\mathrm{pt}_2, \ldots, \mathrm{pt}_m\}, \mathtt{j}) \wedge \mathsf{stkBlack}(X') \\ \qquad \wedge (\mathtt{j} = 0 \vee \mathsf{obj}(\mathtt{j})) \wedge 1 \leq \mathtt{i} \leq M) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X')\end{array}\right\}$

// using Lemma 4.5

```
    MarkAndPush(j);
```

$\left\{\begin{array}{l}\exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reachTomk}(X', \mathtt{i}, \{\mathrm{pt}_2, \ldots, \mathrm{pt}_m\}, 0) \wedge \mathsf{stkBlack}(X') \\ \qquad \wedge (\mathtt{j} = 0 \vee \mathsf{notWhite}(\mathtt{j})) \wedge 1 \leq \mathtt{i} \leq M) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X')\end{array}\right\}$

```
    ...
    j := i.ptm; MarkAndPush(j);
```

$\left\{\begin{array}{l}\exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reachTomk}(X', \mathtt{i}, \emptyset, 0) \wedge \mathsf{stkBlack}(X') \\ \qquad \wedge (\mathtt{j} = 0 \vee \mathsf{notWhite}(\mathtt{j}))) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X')\end{array}\right\}$

```
  }
}
```

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

Figure 4.18 Proof outline of `TraceStack()`.

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

```
CleanCard() {
  local i: [1..M], c: {BLACK, WHITE, BLUE}, d: {1, 0};
  i := 1;
```

Loop Invariant: $\{LInv\}$

```
  while (i <= M) { ... } // See Figure 4.1 for the full code
```

$\{LInv\}$

```
  TraceStack();
```

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon) * \mathsf{own}_{\mathrm{p}}(\mathtt{i}) * \mathsf{own}_{\mathrm{np}}(\mathtt{c}) * \mathsf{own}_{\mathrm{np}}(\mathtt{d})\}$

```
}
```

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

where $LInv \stackrel{\text{def}}{=}$

$\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X)) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)$
$* \ (\mathsf{own}_{\mathrm{p}}(\mathtt{i}) \wedge 1 \leq \mathtt{i} \leq M + 1) * \mathsf{own}_{\mathrm{np}}(\mathtt{c}) * \mathsf{own}_{\mathrm{np}}(\mathtt{d})$

Figure 4.19 Proof outline of `CleanCard()`.

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$
```
ScanRoot() {
  local t: [1..N], rt: Set(Int), i: [0..M];
  t := 1;
```
Loop Invariant: $\left\{ \begin{array}{l} \exists X.\ (\mathit{Inv} \wedge 1 \leq \mathtt{t} \leq N+1) \\ \quad * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own}_{\mathrm{p}}(\mathtt{i}) * \mathsf{own}_{\mathrm{np}}(\mathtt{rt}) \end{array} \right\}$
```
  while (t <= N) {
    rt := get_root(t);
```
Foreach Invariant:
$\left\{ \begin{array}{l} \exists X, Y.\ (\mathit{Inv} \wedge 1 \leq \mathtt{t} \leq N \wedge \mathsf{root}(\mathtt{t}, Y) \wedge \forall n \in (Y \backslash \mathtt{rt}).\ \mathsf{black}(n) \wedge \mathtt{rt} \subseteq Y) \\ \quad * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own}_{\mathrm{p}}(\mathtt{i}) \end{array} \right\}$
```
    foreach i in rt do { MarkAndPush(i); }
    t := t + 1;
  }
}
```
$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X) \wedge \mathsf{rtBlack}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)\}$

where $\mathit{Inv} \stackrel{\text{def}}{=} \mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X) \wedge \mathsf{markRt}(\mathtt{t} - 1)$

Figure 4.20 Proof outline of `ScanRoot()` in an atomic block.

$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X) \wedge \mathsf{rtBlack}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)\}$
```
CleanCard() {
  local i: [1..M], c: {BLACK, WHITE, BLUE}, d: {1, 0};
  i := 1;
```
Loop Invariant:
$\left\{ \begin{array}{l} \exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X) \wedge \mathsf{rtBlack} \wedge \mathsf{clearDirty}(\mathtt{i} - 1) \\ \qquad \wedge 1 \leq \mathtt{i} \leq M+1) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own}_{\mathrm{np}}(\mathtt{c}) * \mathsf{own}_{\mathrm{np}}(\mathtt{d}) \end{array} \right\}$
```
  while (i <= M) { ... } // See Figure 4.1 for the full code
```
$\left\{ \begin{array}{l} \exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X) \wedge \mathsf{rtBlack} \wedge \mathsf{clearDirty}(M)) \\ \quad * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own}_{\mathrm{np}}(\mathtt{c}) * \mathsf{own}_{\mathrm{np}}(\mathtt{d}) \end{array} \right\}$
```
  TraceStack();
```
$\left\{ \begin{array}{l} (\mathsf{wfstate} \wedge \mathsf{reachInv} \wedge \mathsf{rtBlack} \wedge \mathsf{clearDirty}(M)) \\ \quad * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon) * \mathsf{own}_{\mathrm{np}}(\mathtt{c}) * \mathsf{own}_{\mathrm{np}}(\mathtt{d}) \end{array} \right\}$
```
}
```
$\{(\mathsf{wfstate} \wedge \mathsf{reachBlack}) * (\mathsf{own}_{\mathrm{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

Figure 4.21 Proof outline of `CleanCard()` in an atomic block.

$\{\mathsf{wfstate} \wedge \mathsf{reachBlack}\}$
```
Sweep() {
  local i: [1..M], c: {BLACK, WHITE, BLUE};
  i := 1;
```
Loop Invariant:
$\{(\mathsf{wfstate} \wedge \mathsf{reachBlack} \wedge \mathsf{reclaim}(\mathtt{i} - 1) \wedge 1 \leq \mathtt{i} \leq M+1) * \mathsf{own}_{\mathrm{np}}(\mathtt{c})\}$
```
  while (i <= M) { ... } // See Figure 4.1 for the full code
}
```
$\{\mathsf{wfstate} \wedge \mathsf{reachBlack} \wedge \mathsf{reclaim}(M)\}$

Figure 4.22 Proof outline of `Sweep()`.

5. Concurrent sweep-phase (`Sweep()`, shown in Figure 4.22). No matter how the mutators interleave with the GC, all the white objects remain unreachable. Thus the reclamation is safe that guarantees $\mathcal{G}_{\text{gc}}$. After sweep, the state is still well-formed.

4.4 Related Work

Vechev et al. [69] define transformations to generate concurrent GCs from an abstract collector. Afterwards, Pavlovic et al. [56] present refinements to derive concrete concurrent GCs from specifications. These methods focus on describing the behaviors of variants (or instantiations) of a correct abstract collector (or a specification) in a single framework, assuming all the mutator operations are atomic. By comparison, we provide a general correctness notion and a proof method for verifying concurrent GCs and the interactions with mutators (where the barriers could be fine-grained). Furthermore, the correctness of their transformations or refinements is expressed in a GC-oriented way (e.g., the target GC should mark no less objects than the source), which cannot be used to justify other refinement applications.

Kapoor et al. [39] verify Dijkstra's GC using concurrent separation logic. To validate the GC specifications, they also verify a representative mutator in the same system. In contrast, we reduce the problem of verifying a concurrent GC to verifying a transformation, ensuring semantics preservation for *all* mutators.

Our GC verification framework is inspired by McCreight et al. [49], who propose a framework for separate verification of stop-the-world and incremental GCs and their mutators, but their framework does not handle concurrency.

Chapter 5

Verifying Linearizability

In this chapter we verify linearizability [35] of concurrent objects. As we mentioned in Section 1.1.2, the most intuitive approach is to find the Linearization Point (LP) where the object method takes effect. However, for objects with helping mechanism and/or future-dependent LPs, we cannot statically locate their LPs in the implementation code. Verifying linearizability of those objects is a well-known challenging problem. In this chapter, we propose a Hoare-style program logic for modular verification of linearizability with non-fixed LPs. We design a new simulation as the meta-theory of the logic, which extends the RGSim relation with the support for non-fixed LPs. Both the logic and the simulation ensure a contextual refinement, which is equivalent to linearizaiblity.

We first analyze the challenges in the logic design and informally explain our approach in Section 5.1. Then we give the basic technical setting in Section 5.2, including a formal operational definition of linearizability and its equivalence to a contextual refinement. We present our program logic in Section 5.3 and the new simulation in Section 5.4. Finally in Section 5.5 we apply our logic to verify 12 classic algorithms, two of which are used in the `java.util.concurrent` package.

5.1 Challenges and Our Approach

Below we start from a simple program logic for linearizability with fixed LPs, and extend it to support algorithms with non-fixed LPs. We also discuss the problems with the underlying meta-theory, which establishes the soundness of the logic with respect to linearizability.

5.1.1 Basic Logic for Fixed LPs

We first show a simple and intuitive logic which follows the LP approach. As a working example, Figure 5.1(a) shows the implementation of push in Treiber stack [62] (let's first ignore the blue code at line `7'`). The stack object is implemented as a linked list pointed to by `S`, and `push(v)` repeatedly tries to update `S` to point to the new node using compare-and-swap (`cas`) until it succeeds. The atomic instruction `cas(&S, t, x)` reads the value from the location of `S`, compares it with an expected value `t`, writes out a new value `x` if the two match, and returns whether the update succeeds.

To verify linearizability, we first locate the LP in the code. The LP of `push(v)` is at the `cas` statement when it succeeds (line `7`). That is, the successful `cas` can correspond to the abstract atomic `PUSH(v)` operation: `Stk := v::Stk`; and all the other concrete steps cannot. Here we simply represent the abstract stack `Stk` as a sequence of values with "`::`" for concatenation. Then `push(v)` can be linearized at the successful `cas` since it is the single point where the operation takes effect.

We can encode the above reasoning in an existing (unary) concurrent program logic, such as Rely-Guarantee reasoning [38] and CSL [52]. Inspired by Vafeiadis [66], we embed the abstract operation γ and the abstract state σ_a as auxiliary states on the concrete side, so the program state now becomes $(\sigma, (\gamma, \sigma_a))$, where σ is the original concrete state. Then we instrument the concrete implementation with an auxiliary command **linself** (shorthand for "linearize self") at the LP to update the auxiliary state. Intuitively, **linself** will execute the abstract operation γ over the abstract state σ_a, as described in the following operational semantics rule.

$$\frac{(\gamma, \sigma_a) \rightsquigarrow (\mathbf{end}, \sigma_a')}{(\mathbf{linself}, (\sigma, (\gamma, \sigma_a))) \longrightarrow (\mathbf{skip}, (\sigma, (\mathbf{end}, \sigma_a')))}$$

Here $\rightsquigarrow$ encodes the transition of γ at the abstract level, and **end** is a termination marker. We insert **linself** into the same atomic block with the concrete statement at the LP, such as line `7'` in Figure 5.1(a), so that the concrete and abstract sides are executed simultaneously. Here the atomic block $\langle C \rangle$ means C is executed atomically. Then we reason about the instrumented code using a traditional concurrent logic extended with a new inference rule for **linself**.

The idea is intuitive, but it cannot handle more advanced algorithms with non-fixed LPs, including the algorithms with the helping mechanism and those whose locations of LPs depend on the future interleavings. Below we analyze the two challenges in detail and explain our solutions using two representative algorithms, the HSY stack and the pair snapshot.

```
1 push(int v) {
2  local x, t, b;
3  x := new node(v);
4  do {
5    t := S;
6    x.next := t;
7    <b := cas(&S,t,x);
7'    if(b) linself;>
8  } while(!b);
9 }
```

(a) Treiber stack

```
1 readPair(int i, j) {
2  local a, b, v, w;
3  while(true) {
4    <a := m[i].d; v := m[i].v;>
5    <b := m[j].d; w := m[j].v;
5'    trylinself;>
6    if(v = m[i].v) {
6'     commit(cid ↣ (end, (a, b)));
7      return (a, b); }
8  } }
9 write(int i, d) {
10  <m[i].d := d; m[i].v++;> }
```

(c) pair snapshot

```
 1  push(int v) {
 2   local p, him, q;
 3   p := new threadDescriptor(cid, PUSH, v);
 4   while(true) {
 5     if (tryPush(v)) return;
 6     loc[cid] := p;
 7     him := rand(); q := loc[him];
 8     if (q != null && q.id = him && q.op = POP)
 9       if (cas(&loc[cid], p, null)) {
10         <b := cas(&loc[him], q, p);
10'         if(b) {lin(cid); lin(him);}>
11         if (b) return; }
12     ...
13   } }
```

(b) HSY elimination-based stack

Figure 5.1 LPs and instrumented auxiliary commands.

5.1.2 Support Helping Mechanism with Pending Thread Pool

HSY elimination-based stack [30] is a typical example using the helping mechanism. Figure 5.1(b) shows part of its push method implementation. The basic idea behind the algorithm is to let a push and a pop cancel out each other.

At the beginning of the method in Figure 5.1(b), the thread allocates its *thread descriptor* (line 3), which contains the thread ID, the name of the operation to be performed, and the argument. The current thread `cid` first tries to perform Treiber stack's push (line 5). It returns if succeeds. Otherwise, it writes its descriptor in the global `loc` array (line 6) to allow other threads to eliminate its push. The elimination array `loc`$[1..n]$ has one slot for each thread, which holds the pointer

to a thread descriptor. The current thread `cid` randomly reads a slot `him` in `loc` (line 7). If the descriptor `q` says `him` is doing pop, `cid` tries to eliminate itself with `him` by two `cas` instructions. The first clears `cid`'s entry in `loc` so that no other thread could eliminate with `cid` (line 9). The second attempts to mark the entry of `him` in `loc` as "eliminated with `cid`" (line 10). If successful, it should be the LPs of *both* the push of `cid` and the pop of `him`, with the push happening immediately before the pop.

The helping mechanism may cause the current thread to linearize the operations of other threads, which cannot be expressed in the basic logic in Section 5.1.1. It also breaks modularity and makes thread-local verification difficult. For the thread `cid`, its concrete step could correspond to the steps of both `cid` and `him` at the abstract level. For `him`, a step from its environment could fulfill its abstract operation. We must ensure in the thread-local verification that the two threads `cid` and `him` always take consistent views on whether and how the abstract operation of `him` is done. For example, if we let a concrete step in `cid` fulfill the abstract pop of `him`, we must know `him` is indeed doing pop and its pop has not been done before. Otherwise, we will not be able to compose `cid` and `him` in parallel.

We extend the basic logic to express the helping mechanism. First we introduce a new auxiliary command **lin**(`t`) to linearize a specific thread `t`. For instance, in Figure 5.1(b) we insert line `10'` at the LP to execute both the push of `cid` and the pop of `him` at the abstract level. We also extend the auxiliary state to record both abstract operations of `cid` and `him`. More generally, we embed a pending thread pool U, which maps threads to their abstract operations. It specifies a set of threads whose operations might be helped by others. Then under the new state $(\sigma, (U, \sigma_a))$, the semantics of **lin**(`t`) just executes the thread `t`'s abstract operation in U, similarly to the semantics of **linself** discussed before.

The shared pending thread pool U allows us to recover the thread modularity when verifying the helping mechanism. A concrete step of `cid` could fulfill the operation of `him` in U as well as its own abstract operation; and conversely, the thread `him` running in parallel could check U to know if its operation has been finished by others (such as `cid`) or not. We gain consistent abstract information of other threads in the thread-local verification. Note that the need of U itself does not break modularity because the required information of other threads' abstract operations can be inferred from the concrete state. In the HSY stack example, we know `him` is doing pop by looking at its thread descriptor in the elimination array. In this case U can be viewed as an abstract representation of the elimination array.

5.1.3 Try-Commit Commands for Future-Dependent LPs

Another challenge is to reason about optimistic algorithms whose LPs depend on the future interleavings.

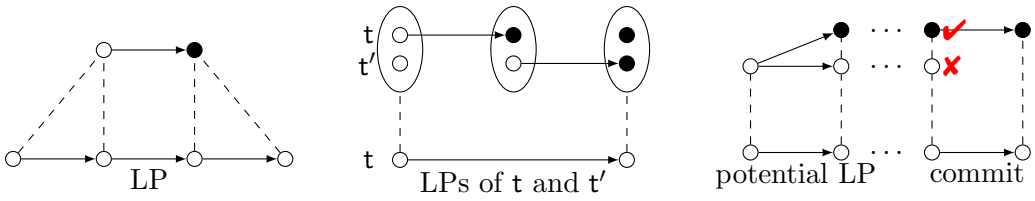
We give a toy example, pair snapshot [58], in Figure 5.1(c). The object is an array `m`, each slot of which contains two fields: `d` for the data and `v` for the version number. The `write(i,d)` method (line `9`) updates the data stored at address `i` and increments the version number instantaneously. The `readPair(i,j)` method intends to perform an atomic read of two slots `i` and `j` in the presence of concurrent writes. It reads the data at slots `i` and `j` separately at lines `4` and `5`, and validate the first read at line `6`. If `i`'s version number has not been increased, the thread knows that when it read `j`'s data at line `5`, `i`'s data had not been updated. This means the two reads were at a consistent state, thus the thread can return. We can see that the LP of `readPair` should be at line `5` when the thread reads `j`'s data, but only if the validation at line `6` succeeds. That is, whether we should linearize the operation at line `5` depends on the future unpredictable behavior of line `6`.

As discussed a lot in previous work (e.g., [1, 66]), the future-dependent LPs cannot be handled by introducing history variables, which are auxiliary variables storing values or events in the past executions. We have to refer to events coming from the unpredictable future. Thus people propose prophecy variables [1, 66] as the dual of history variables to store future behaviors. But as far as we know, there is no semantics of prophecy variables suitable for Hoare-style local and compositional reasoning.

Instead of resorting to prophecy variables, we follow the speculation idea [65]. For the concrete step at a potential LP (e.g., line `5` of `readPair`), we execute the abstract operation speculatively and keep both the result and the original abstract configuration. Later based on the result of the validation (e.g., line `6` in `readPair`), we keep the appropriate branch and discard the other.

For the logic, we introduce two new auxiliary commands: **trylinself** is to do speculation, and **commit**(p) will commit to the appropriate branch satisfying the assertion p. In Figure 5.1(c), we insert lines `5'` and `6'`, where $\texttt{cid} \rightarrowtail (\textbf{end}, (\texttt{a}, \texttt{b}))$ means that the current thread `cid` should have done its abstract operation and would return $(\texttt{a}, \texttt{b})$. We also extend the auxiliary state to record the multiple possibilities of abstract operations and abstract states after speculation.

Furthermore, we can combine the speculation idea with the pending thread pool. We allow the abstract operations in the pending thread pool as well as the current thread to speculate. Then we could handle some trickier algorithms such



(a) simple simulation (b) pending thread pool (c) speculation

Figure 5.2 Simulation diagrams for linearizability verification.

as RDCSS [27], in which the location of LP for thread $\mathtt{t}$ may be in the code of some other thread $\mathtt{t}'$ and also depend on the future behaviors of $\mathtt{t}'$. Please see Section 5.5 for one such example.

5.1.4 Simulation as Meta-Theory

The LP proof method can be understood as building simulations between the concrete implementations and the abstract atomic operations, such as the simple weak simulation in Figure 5.2(a). The lower-level and higher-level arrows are the steps of the implementation and of the abstract operation respectively, and the dashed lines denote the simulation relation. We use dark nodes and white nodes at the abstract level to distinguish whether the operation has been finished or not. The only step at the concrete side corresponding to the single abstract step should be the LP of the implementation (labeled "LP" in the diagram). Since our program logic is based on the LP method, we can expect simulations to justify its soundness. In particular, we want a *thread-local* simulation which can handle both the helping mechanism and future-dependent LPs and can ensure linearizability.

First, to ensure linearizability, the thread-local simulation has to be *compositional*. To this end, we adopt the ideas in our RGSim relation and parameterize the simple simulation with the interference from the environments in the form of rely/guarantee conditions. We can prove that the compositional RGSim relation ensures a contextual refinement, which is equivalent to linearizability.

To support helping in the simulation, we should allow the LP step at the concrete level to correspond to an abstract step made by a thread other than the one being verified. This requires information about code and thread-local states of *environment* threads at the abstract side, which unfortunately is not provided in traditional thread-local simulations (including RGSim). To address the problem, we introduce the pending thread pool at the abstract level of the simulation, just

as in the development of our logic in Section 5.1.2. The new simulation is shown in Figure 5.2(b). We can see that a concrete step of thread $\mathtt{t}$ could help linearize the operation of $\mathtt{t}'$ in the pending thread pool as well as its own operation. Thus the new simulation intuitively supports the helping mechanism.

As forward simulations, neither of the simulations in Figure 5.2(a) and (b) supports future-dependent LPs. For each step along the concrete execution in those simulations, we need to decide immediately whether the step is at the LP, and cannot postpone the decision to the future. As discussed a lot in previous work (e.g., [1, 12, 17, 48]), we have to introduce backward simulations or hybrid simulations to support future-dependent LPs. Here we exploit the speculation idea and develop a forward-backward simulation [48]. As shown in Figure 5.2(c), we keep both speculations after the potential LP, where the higher black nodes result from executing the abstract operation and the lower white nodes record the original abstract configuration. Then at the validation step we commit to the correct branch.

We combine the above ideas and develop a new compositional simulation with the support of non-fixed LPs as the meta-theory of our logic. We will discuss our simulation formally in Section 5.4.

5.2 Basic Technical Settings and Linearizability

In this section, we formalize linearizability of an object implementation with respect to its abstract operations, and show that linearizability is equivalent to a contextual refinement. We first define a programming language that supports concurrent objects. It is an instantiation of the generic language in Figure 2.2 introduced for refinement verification.

5.2.1 Language and Semantics

As shown in Figure 5.3, a program W contains several client threads in parallel, each of which could call the methods declared in the object Π. A method is defined as a pair (x, C), where x is the formal argument and C is the method body. For simplicity, we assume there is only one object in W and each method takes one argument only, but it is easy to extend our work with multiple objects and arguments.

Each method returns a value to the client using the command **return** E, unless it does not terminate. We use a runtime command **noret** to abort methods that terminate but do not execute a **return** statement. It is automatically appended to

$$
\begin{array}{rrcl}
(\mathit{MName}) & f & \in & \mathit{String} \\
(\mathit{Expr}) & E & ::= & x \mid n \mid E + E \mid \ldots \\
(\mathit{BExp}) & B & ::= & \mathsf{true} \mid \mathsf{false} \mid E = E \mid !B \mid \ldots \\
(\mathit{Instr}) & c & ::= & x := E \mid x := [E] \mid [E] := E \mid \mathbf{print}(E) \\
 & & \mid & x := \mathbf{cons}(E, \ldots, E) \mid \mathbf{dispose}(E) \mid \ldots \\
(\mathit{Stmt}) & C & ::= & \mathbf{skip} \mid c \mid x := f(E) \mid \mathbf{return}\ E \mid \mathbf{fret}(n) \mid \mathbf{noret} \\
 & & \mid & \langle C \rangle \mid C; C \mid \mathbf{if}\ (B)\ C\ \mathbf{else}\ C \mid \mathbf{while}\ (B)\{C\} \\
(\mathit{Prog}) & W & ::= & \mathbf{skip} \mid \mathbf{let}\ \Pi\ \mathbf{in}\ C \,\|\, \ldots \,\|\, C \\
(\mathit{ODecl}) & \Pi & ::= & \{f_1 \rightsquigarrow (x_1, C_1), \ldots, f_n \rightsquigarrow (x_n, C_n)\}
\end{array}
$$

Figure 5.3 Syntax of the programming language.

$$
\begin{array}{rrcl}
(\mathit{ThrdID}) & \mathsf{t} & \in & \mathit{Nat} \\
(\mathit{Store}) & s & \in & \mathit{PVar} \rightharpoonup \mathit{Int} \qquad (\mathit{Heap})\ h \in \mathit{Nat} \rightharpoonup \mathit{Int} \\
(\mathit{Mem}) & \sigma & ::= & (s, h) \\
(\mathit{CallStk}) & \kappa & ::= & (s_l, x, C) \mid \circ \\
(\mathit{ThrdPool}) & \mathcal{K} & ::= & \{\mathsf{t}_1 \rightsquigarrow \kappa_1, \ldots, \mathsf{t}_n \rightsquigarrow \kappa_n\} \\
(\mathit{PState}) & \mathcal{S} & ::= & (\sigma_c, \sigma_o, \mathcal{K}) \\
(\mathit{LState}) & \varsigma & ::= & (\sigma_c, \sigma_o, \kappa) \\
(\mathit{Evt}) & e & ::= & (\mathsf{t}, f, n) \mid (\mathsf{t}, \mathbf{ret}, n) \mid (\mathsf{t}, \mathbf{obj}) \mid (\mathsf{t}, \mathbf{obj}, \mathbf{abort}) \\
 & & \mid & (\mathsf{t}, \mathbf{out}, n) \mid (\mathsf{t}, \mathbf{clt}) \mid (\mathsf{t}, \mathbf{clt}, \mathbf{abort}) \\
(\mathit{ETrace}) & T & ::= & \epsilon \mid e :: T
\end{array}
$$

Figure 5.4 States and event traces.

the method code and is not supposed to be used by programmers. The command **return** E will first calculate the return value n and reduce to $\mathbf{fret}(n)$, another runtime command automatically generated during executions. We separate the evaluation of E from returning its value n to the client, to allow interference between the two steps.

Other commands are mostly standard (see [20, 59]). Commands $x := [E]$ and $[E] := E'$ do memory load and store. Memory allocation and free are done by $x := \mathbf{cons}(E_1, \ldots, E_n)$ and $\mathbf{dispose}(E)$. The atomic block $\langle C \rangle$ executes C atomically. Clients can also use $\mathbf{print}(E)$ to produce externally observable events. We do not allow the object's methods to produce external events. To simplify the semantics, we also assume there are no nested method calls.

Figure 5.4 gives the model of program states. Here we partition a global state $\mathcal{S}$ into the client memory σ_c, the object memory σ_o and a thread pool $\mathcal{K}$. Memory

σ consists of a store s which maps variables to integers and a heap h which maps locations (i.e., natural numbers) to integers. A client can only access the client memory σ_c, unless it calls object methods. The thread pool maps each thread ID t to its local call stack frame. A call stack κ could be either empty ($\circ$) when the thread is not executing a method, or a triple (s_l, x, C), where s_l maps the method's formal argument and local variables (if any) to their values, x is the caller's variable to receive the return value, and C is the caller's remaining code to be executed after the method returns. We define $\circledcirc$ as a special thread pool where the call stacks of all threads are $\circ$. To give a thread-local semantics (see Figure 5.5(b)), we also define the thread local view ς of the state.

The set of events e defined in Figure 5.4 includes not only externally observable events, but also unobservable events which can help define linearizability. A method invocation event (t, f, n) is produced when thread t executes $x := f(E)$, where n is the value of the argument E. Each step of the method body produces $(\mathsf{t}, \mathbf{obj})$. When the method returns, a return event $(\mathsf{t}, \mathbf{ret}, n)$ is produced with the return value n. $\mathbf{print}(E)$ generates an output $(\mathsf{t}, \mathbf{out}, n)$, and any other client step generates $(\mathsf{t}, \mathbf{clt})$. For steps that abort (e.g., invalid memory access), fault events are produced: $(\mathsf{t}, \mathbf{obj}, \mathbf{abort})$ by the object code and $(\mathsf{t}, \mathbf{clt}, \mathbf{abort})$ by the client code. Note that here we explicitly distinguish the faults caused by the methods and by the clients. This allows us to clearly know where to place the blame when the program aborts, and then to discuss the safety of the object. Only outputs and the two kinds of faults are externally observable. Method invocations, returns and object faults are called *history* events, which will be used to define linearizability below. We write $\mathsf{tid}(e)$ for the thread ID in the event e. We also define several predicates to specify the kind of e, as summarized below.

- e is an invocation event: $\mathsf{is\_inv}(e)$ holds iff there exist t, f and n such that $e = (\mathsf{t}, f, n)$.
- e is a return event: $\mathsf{is\_ret}(e)$ holds iff there exist t and n such that $e = (\mathsf{t}, \mathbf{ret}, n)$.
- e is an object fault: $\mathsf{is\_obj\_abt}(e)$ holds iff there exists t such that $e = (\mathsf{t}, \mathbf{obj}, \mathbf{abort})$.
- e is a response event: $\mathsf{is\_res}(e)$ holds iff either $\mathsf{is\_ret}(e)$ or $\mathsf{is\_obj\_abt}(e)$ holds.
- e is an object event: $\mathsf{is\_obj}(e)$ holds iff there exists t such that $e = (\mathsf{t}, \mathbf{obj})$, or $\mathsf{is\_inv}(e)$, or $\mathsf{is\_res}(e)$ holds.
- e is a client fault: $\mathsf{is\_clt\_abt}(e)$ holds iff there exists t such that $e = (\mathsf{t}, \mathbf{clt}, \mathbf{abort})$.

$$\frac{}{(\textbf{let } \Pi \textbf{ in } \textbf{skip} \,\|\, \ldots \,\|\, \textbf{skip}, \mathcal{S}) \longmapsto (\textbf{skip}, \mathcal{S})}$$

$$\frac{(C_i, (\sigma_c, \sigma_o, \mathcal{K}(i))) \xrightarrow{e}_{i,\Pi} (C_i', (\sigma_c', \sigma_o', \kappa'))) \qquad \mathcal{K}' = \mathcal{K}\{i \rightsquigarrow \kappa'\}}{(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots C_i \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \mathcal{K})) \overset{e}{\longmapsto} (\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots C_i' \ldots \,\|\, C_n, (\sigma_c', \sigma_o', \mathcal{K}'))}$$

$$\frac{(C_i, (\sigma_c, \sigma_o, \mathcal{K}(i))) \xrightarrow{e}_{i,\Pi} \textbf{abort}}{(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots C_i \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \mathcal{K})) \overset{e}{\longmapsto} \textbf{abort}}$$

(a) program transitions

$$\frac{\Pi(f) = (y, C) \qquad [\![E]\!]_{s_c} = n \qquad x \in \mathit{dom}(s_c) \qquad \kappa = (\{y \rightsquigarrow n\}, x, \mathbf{E}[\,\textbf{skip}\,])}{(\mathbf{E}[\,x := f(E)\,], ((s_c, h_c), \sigma_o, \circ)) \xrightarrow{(\mathsf{t}, f, n)}_{\mathsf{t},\Pi} (C; \textbf{noret}, ((s_c, h_c), \sigma_o, \kappa))}$$

$$\frac{f \notin \mathit{dom}(\Pi) \quad \text{or} \quad [\![E]\!]_{s_c} \textit{ undefined} \quad \text{or} \quad x \notin \mathit{dom}(s_c)}{(\mathbf{E}[\,x := f(E)\,], ((s_c, h_c), \sigma_o, \circ)) \xrightarrow{(\mathsf{t}, \textbf{clt}, \textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}}$$

$$\frac{\kappa = (s_l, x, C) \qquad s_c' = s_c\{x \rightsquigarrow n\}}{(\textbf{fret}(n), ((s_c, h_c), \sigma_o, \kappa)) \xrightarrow{(\mathsf{t}, \textbf{ret}, n)}_{\mathsf{t},\Pi} (C, ((s_c', h_c), \sigma_o, \circ))}$$

$$\frac{[\![E]\!]_{s_c} = n}{(\mathbf{E}[\,\textbf{print}(E)\,], ((s_c, h_c), \sigma_o, \circ)) \xrightarrow{(\mathsf{t}, \textbf{out}, n)}_{\mathsf{t},\Pi} (\mathbf{E}[\,\textbf{skip}\,], ((s_c, h_c), \sigma_o, \circ))}$$

$$\frac{(C, (s_o \uplus s_l, h_o)) \longrightarrow_{\mathsf{t}} (C', (s_o' \uplus s_l', h_o')) \qquad \mathit{dom}(s_l) = \mathit{dom}(s_l')}{(C, (\sigma_c, (s_o, h_o), (s_l, x, C_c))) \xrightarrow{(\mathsf{t}, \textbf{obj})}_{\mathsf{t},\Pi} (C', (\sigma_c, (s_o', h_o'), (s_l', x, C_c)))}$$

$$\frac{\sigma_o = (s_o, h_o) \quad (C, (s_o \uplus s_l, h_o)) \longrightarrow_{\mathsf{t}} \textbf{abort}}{(C, (\sigma_c, \sigma_o, (s_l, x, C_c))) \xrightarrow{(\mathsf{t}, \textbf{obj}, \textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}} \qquad \frac{(C, \sigma_c) \longrightarrow_{\mathsf{t}} (C', \sigma_c')}{(C, (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathsf{t}, \textbf{clt})}_{\mathsf{t},\Pi} (C', (\sigma_c', \sigma_o, \circ))}$$

(b) thread transitions

$$\frac{[\![E]\!]_s = n}{(\mathbf{E}[\,\textbf{return } E\,], (s, h)) \longrightarrow_{\mathsf{t}} (\textbf{fret}(n), (s, h))} \qquad \frac{}{(\textbf{noret}, \sigma) \longrightarrow_{\mathsf{t}} \textbf{abort}}$$

$$\frac{(C, \sigma) \longrightarrow_{\mathsf{t}}^{*} (\textbf{skip}, \sigma')}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \longrightarrow_{\mathsf{t}} (\mathbf{E}[\,\textbf{skip}\,], \sigma')} \qquad \frac{(C, \sigma) \longrightarrow_{\mathsf{t}}^{*} (\textbf{fret}(n), \sigma')}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \longrightarrow_{\mathsf{t}} (\textbf{fret}(n), \sigma')} \qquad \frac{(C, \sigma) \longrightarrow_{\mathsf{t}}^{*} \textbf{abort}}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \longrightarrow_{\mathsf{t}} \textbf{abort}}$$

(c) local thread transitions

Figure 5.5 Selected operational semantics rules.

$$\begin{aligned}
&\mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \\
&\quad \{T \mid \exists W',\mathcal{S}'.\ (W,(\sigma_c,\sigma_o,\circledcirc)) \stackrel{T}{\longmapsto}{}^{*}\ (W',\mathcal{S}') \vee (W,(\sigma_c,\sigma_o,\circledcirc)) \stackrel{T}{\longmapsto}{}^{*}\ \mathbf{abort}\} \\
&\mathcal{H}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get_hist}(T) \mid T \in \mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!]\} \\
&\mathcal{O}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get_obsv}(T) \mid T \in \mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!]\}
\end{aligned}$$

Figure 5.6 Generation of finite event traces.

- e is a fault: $\mathsf{is\_abt}(e)$ holds iff either $\mathsf{is\_obj\_abt}(e)$ or $\mathsf{is\_clt\_abt}(e)$ holds.
- e is a client event: $\mathsf{is\_clt}(e)$ holds iff there exist t and n such that $e = (\mathsf{t}, \mathbf{clt})$, or $e = (\mathsf{t}, \mathbf{out}, n)$, or $e = (\mathsf{t}, \mathbf{clt}, \mathbf{abort})$ holds.

An event trace T is a finite sequence of events. We write $T(i)$ for the i-th event of T, and $|T|$ for the length of the trace. $T|_{\mathsf{t}}$ represents the sub-trace of T consisting of all events whose thread ID is t. We use $\mathsf{get\_hist}(T)$ to project T to the sub-trace consisting of all the history events, and $\mathsf{get\_obsv}(T)$ for the sub-trace of all the observable events. A traces of history events is called a *history*.

Figure 5.5 gives selected rules of the operational semantics. We have three kinds of transitions. We write $(W,\mathcal{S}) \stackrel{e}{\longmapsto} (W',\mathcal{S}')$ for the top-level program transitions and $(C,\varsigma) \stackrel{e}{\longrightarrow}_{\mathsf{t},\Pi} (C',\varsigma')$ for the transitions of thread t with the object Π. We also introduce the local transitions $(C,\sigma) \longrightarrow\!\triangleright_{\mathsf{t}} (C',\sigma')$ to describe steps inside or outside method calls of thread t. It accesses only the object memory and method local variables (for the case inside method calls), or only client memory (for the other case). We then lift a local transition to a thread transition that produces an event $(\mathsf{t}, \mathbf{obj})$ or $(\mathsf{t}, \mathbf{clt})$. All three transitions also include steps that lead to the error state **abort**. To describe the operational semantics for threads, we use an execution context $\mathbf{E}$, where

$$(\mathit{ExecContext}) \quad \mathbf{E} ::= [\,] \mid \mathbf{E}; C$$

The hole $[\,]$ shows the place where the execution of code occurs. $\mathbf{E}[\,C\,]$ represents the code that results from placing C into the hole. The semantics defined in Figure 5.5 is mostly straightforward. Note that we append **noret** at the end of method body when the thread invokes the method. Since **noret** aborts the program, a safe method implementation must end with a **return** statement.

In Figure 5.6, we define $\mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!]$ for the prefix-closed set of finite event traces produced by the executions of W with the initial state $(\sigma_c,\sigma_o,\circledcirc)$. We use $(W,\mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} (W',\mathcal{S}')$ for zero or multiple-step program transitions that generate

the trace T. We also define $\mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$ and $\mathcal{O}[\![W, (\sigma_c, \sigma_o)]\!]$ to get histories and finite observable traces produced by the executions.

5.2.2 Object Specification and Linearizability Definition

Before formalizing linearizability, we first define the abstract operations Π_A of an object, which we consider as the specification. Π_A is of the same type as the concrete implementation Π (see Figure 5.3), but requires that each method body should be an atomic operation of the form $\langle C \rangle$ and it should be always safe and deterministic to execute it. To simplify the presentation, we write γ for the method declaration in Π_A. That is,

$$\begin{array}{rcl} \gamma & ::= & (x, \langle C \rangle) \\ \Pi_A & ::= & \{f_1 \rightsquigarrow \gamma_1, \ldots, f_n \rightsquigarrow \gamma_n\} \end{array}$$

Note that the atomic block $\langle C \rangle$ may contain the command **return** E. In this case, $\langle C \rangle$ would reduce to $\mathbf{fret}(n)$, as shown in the operational semantics in Figure 5.5(c). We write σ_a for the object memory at the abstract level, and assume the program variables used in the abstract operations are different from those in the concrete implementations. The semantics of the client program with Π_A is the same as the one in Figure 5.5, but replaces the concrete object memory σ_o by the abstract object memory σ_a.

Linearizability [35] is defined using the notion of histories, which are special event traces T consisting of only history events (i.e., method invocations and responses).

We say a response e_2 *matches* an invocation e_1, written as $\mathsf{match}(e_1, e_2)$, iff they have the same thread ID.

$$\mathsf{match}(e_1, e_2) \stackrel{\text{def}}{=} \mathsf{is\_inv}(e_1) \wedge \mathsf{is\_res}(e_2) \wedge (\mathsf{tid}(e_1) = \mathsf{tid}(e_2))$$

A history T is *sequential*, i.e., $\mathsf{seq}(T)$ holds, iff the first event of T is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response. It is inductively defined as follows.

$$\frac{}{\mathsf{seq}(\epsilon)} \qquad \frac{\mathsf{is\_inv}(e)}{\mathsf{seq}(e :: \epsilon)} \qquad \frac{\mathsf{match}(e_1, e_2) \qquad \mathsf{seq}(T)}{\mathsf{seq}(e_1 :: e_2 :: T)}$$

Then T is *well-formed* iff the sub-history $T|_{\mathsf{t}}$ for every thread t is sequential.

$$\mathsf{well\_formed}(T) \stackrel{\text{def}}{=} \forall \mathsf{t}.\ \mathsf{seq}(T|_{\mathsf{t}})\,.$$

T is *complete* iff it is well-formed and every invocation has a matching response. An invocation is *pending* in T if no matching response follows it. We handle pending invocations in an incomplete history T following the standard linearizability definition [35]. We append zero or more return events to T, and drop the remaining pending invocations. Then we get a set of complete histories, which is denoted by $\mathsf{completions}(T)$. Formally, we need to first define $\mathsf{extensions}(T)$ and $\mathsf{truncate}(T)$.

Definition 5.1 (Extensions of a History)**.** $\mathsf{extensions}(T)$ is a set of well-formed histories where we extend T by appending return events. It is inductively defined as follows.

$$\frac{\mathsf{well\_formed}(T)}{T \in \mathsf{extensions}(T)} \qquad \frac{T' \in \mathsf{extensions}(T) \quad \mathsf{is\_ret}(e) \quad \mathsf{well\_formed}(T' :: e)}{T' :: e \in \mathsf{extensions}(T)}$$

Definition 5.2 (Completions of a History)**.** $\mathsf{truncate}(T)$ is the maximal complete sub-history of T. It is inductively defined by dropping the pending invocations in T.

$$\mathsf{truncate}(\epsilon) \stackrel{\text{def}}{=} \epsilon$$

$$\mathsf{truncate}(e :: T) \stackrel{\text{def}}{=} \begin{cases} e :: \mathsf{truncate}(T) & \text{if } \mathsf{is\_res}(e) \text{ or } \exists i.\ \mathsf{match}(e, T(i)) \\ \mathsf{truncate}(T) & \text{otherwise} \end{cases}$$

Then $\mathsf{completions}(T) \stackrel{\text{def}}{=} \{\mathsf{truncate}(T') \mid T' \in \mathsf{extensions}(T)\}$.

Then we can formulate the linearizability relation between two well-formed histories, which is a core notion used in the linearizability definition of an object.

Definition 5.3 (Linearizable Histories)**.** $T \preceq_{\mathsf{lin}} T'$ iff both the following hold.

1. $\forall \mathsf{t}.\ T|_{\mathsf{t}} = T'|_{\mathsf{t}}$.
2. There exists a bijection $\pi : \{1, \ldots, |T|\} \to \{1, \ldots, |T'|\}$ such that $\forall i.\ T(i) = T'(\pi(i))$ and

$$\forall i, j.\ i < j \wedge \mathsf{is\_res}(T(i)) \wedge \mathsf{is\_inv}(T(j)) \implies \pi(i) < \pi(j)\,.$$

That is, T is linearizable with respect to T' if the latter is a permutation of the former, preserving the order of events in the same threads and the order of the non-overlapping method calls.

An *object* Π is linearizable iff each of its concurrent histories after completions is linearizable with respect to some *legal sequential* history. We generate the concurrent histories of Π by all the possible clients that may use the object, according to the operational semantics in Figure 5.5. Figure 5.6 defines $\mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$ to get

the set of histories from the executions of W. Similarly, $\mathcal{H}[\![W, (\sigma_c, \sigma_a)]\!]$ generates histories when using the abstract object. Then, a legal sequential history T is a sequential history generated by some client using the abstract object Π_A with some initial memory σ_a. It satisfies the following.

$$\Pi_A \rhd (\sigma_a, T) \stackrel{\text{def}}{=}$$
$$\exists n, C_1, \ldots, C_n, \sigma_c.\ T \in \mathcal{H}[\![(\textbf{let } \Pi_A \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a)]\!] \wedge \mathsf{seq}(T)$$

Definition 5.4 (Linearizability of Objects)**.** The object implementation Π is linearizable with respect to Π_A under a refinement mapping φ, denoted by $\Pi \preceq_\varphi \Pi_A$, iff

$$\begin{array}{l}\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \sigma_a, T. \\ T \in \mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o)]\!] \wedge (\varphi(\sigma_o) = \sigma_a) \\ \Longrightarrow\ \exists T_c, T'.\ T_c \in \mathsf{completions}(T) \wedge \Pi_A \rhd (\sigma_a, T') \wedge T_c \preceq_{\mathsf{lin}} T'\end{array}$$

Here the partial mapping $\varphi : \mathit{Mem} \rightharpoonup \mathit{Mem}$ relates the concrete object memory to the abstract one.

The side condition $\varphi(\sigma_o) = \sigma_a$ in the above definition requires the initial concrete object memory σ_o to be well-formed in that it represents some valid abstract object memory σ_a. For instance, φ may need σ_o to contain a linked list and relate it to an abstract mathematical set in σ_a for a set object.

5.2.3 Contextual Refinement and Linearizability

Next we define a contextual refinement between the concrete object and its specification, and prove its equivalence to linearizability. This equivalence will serve as the basis of our logic soundness with respect to linearizability.

Informally, the contextual refinement says, substituting the concrete object Π for its specification Π_A in any context (i.e., in a client program) does not add observable behaviors. Below we use $\mathcal{O}[\![W, (\sigma_c, \sigma_o)]\!]$ to represent the set of observable event traces generated during the executions of W. It is defined in Figure 5.6 similarly as $\mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$, but now the traces consist of observable events only (output events, client faults or object faults).

Definition 5.5 (Basic Contextual Refinement)**.** $\Pi \sqsubseteq_\varphi \Pi_A$ iff

$$\begin{array}{l}\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \sigma_a.\ (\varphi(\sigma_o) = \sigma_a) \implies \\ \mathcal{O}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o)]\!] \subseteq \mathcal{O}[\![(\textbf{let } \Pi_A \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a)]\!]\,.\end{array}$$

Theorem 5.6 (Basic Equivalence)**.** $\Pi \preceq_\varphi \Pi_A \iff \Pi \sqsubseteq_\varphi \Pi_A$.

The proof of Theorem 5.6 follows Filipović et al. [21] and Gotsman and Yang [25], and is sketched in Appendix B.

The theorem gives us another point of view to understand linearizability. Since linearizability implies the contextual refinement, we can modularly verify a client by replacing the linearizable object implementation with its specification. On the other hand, since contextual refinement also implies linearizability, we can use proof methods (such as RGSim) for the former to verify the latter.

5.3 Logic for Linearizability

To prove object linearizability, we first instrument the object implementation by introducing auxiliary states and auxiliary commands, which relate the concrete code with the abstract object. Our program logic extends LRG [20] with a relational interpretation of assertions and new rules for auxiliary commands. Although our logic is based on LRG [20], this approach is mostly independent with the base logic. Similar extensions can also be made over other logics, such as RGSep [66].

Our logic is proposed to verify object methods only. Verified object methods are guaranteed to be a contextual refinement of their abstract atomic operations, which ensures linearizability of the object. We will discuss verification of whole programs consisting of both client code and object code at the end of Section 5.3.3.

5.3.1 Instrumented Code and States

In Figure 5.7, we show the syntax of the instrumented code and its state model. As explained in Section 5.1, the new program state Σ for the instrumented method code consists of two parts, the physical object state σ and the auxiliary data Δ. Δ is a *nonempty* set of (U, σ_a) pairs, each pair representing a speculation of the situation at the abstract level. Here σ_a is the abstract object memory at the specific speculation. U is a pending thread pool recording the remaining operation to be fulfilled by each thread. It maps a thread ID to its remaining abstract operation, which is either (γ, n) (the operation γ needs to be executed with argument n) or $(\mathbf{end}, n)$ (the operation has been finished with the return value n).

Below we informally explain the effects over Δ of the newly introduced commands. We leave their formal semantics to Section 5.3.4. The auxiliary command **linself** executes the unfinished abstract operation of the current thread in every U in Δ, and changes the abstract object memory σ_a correspondingly. $\mathbf{lin}(E)$ executes the abstract operation of the thread with ID E. **linself** or $\mathbf{lin}(E)$ is executed when we know for sure that a step is the linearization point. The **trylinself** command

$$
\begin{array}{rrcl}
(\mathit{InsStmt}) & \widetilde{C} & ::= & \textbf{skip} \mid c \mid \textbf{return}\ E \mid \textbf{noret} \mid \textbf{linself} \mid \textbf{lin}(E) \\
& & \mid & \textbf{trylinself} \mid \textbf{trylin}(E) \mid \textbf{commit}(p) \mid \langle \widetilde{C} \rangle \\
& & \mid & \widetilde{C}; \widetilde{C} \mid \textbf{if}\ (B)\ \widetilde{C}\ \textbf{else}\ \widetilde{C} \mid \textbf{while}\ (B)\{\widetilde{C}\} \\
(\mathit{RelState}) & \Sigma & ::= & (\sigma, \Delta) \\
(\mathit{SpecSet}) & \Delta & ::= & \{(U_1, \sigma_1), \ldots, (U_n, \sigma_n)\} \\
(\mathit{PendThrds}) & U & ::= & \{\mathsf{t}_1 \rightsquigarrow \Upsilon_1, \ldots, \mathsf{t}_n \rightsquigarrow \Upsilon_n\} \\
(\mathit{AbsOp}) & \Upsilon & ::= & (\gamma, n) \mid (\textbf{end}, n) \\
(\mathit{RelAss}) & p, q, I & ::= & B \mid \mathit{own}(x) \mid \boldsymbol{own}(x) \mid \mathsf{emp} \\
& & \mid & E \mapsto E \mid E \Mapsto E \mid E \rightarrowtail (\gamma, E) \mid E \rightarrowtail (\textbf{end}, E) \\
& & \mid & p * q \mid p \oplus q \mid p \vee q \mid \ldots \\
(\mathit{RelAct}) & R, G & ::= & [p] \mid p \ltimes q \mid p \propto q \mid R * R \mid \ldots
\end{array}
$$

Figure 5.7 Instrumented code, relational state model and assertion language.

introduces uncertainty. Since we do not know if the abstract operation of the current thread is fulfilled or not at the current point, we consider both possibilities. For each (U, σ_a) pair in Δ that contains unfinished abstract operation of the current thread, we add in Δ a new speculation (U', σ'_a) where the abstract operation is done and σ'_a is the resulting abstract object memory. Since the original (U, σ_a) is also kept, we have both speculations in Δ. Similarly, the $\textbf{trylin}(E)$ command introduces speculations about the thread E. For simplicity, we assume the thread ID E in $\textbf{lin}(E)$ or $\textbf{trylin}(E)$ will not use variables in the abstract object memory. When we have enough knowledge p about the situation of the abstract operations and memory, the $\textbf{commit}(p)$ step keeps only the subset of speculations consistent with p and drops the rest. Here p is a logical assertion about the state Σ, which is explained below.

5.3.2 Assertions

Syntax of assertions is shown in Figure 5.7. Following rely-guarantee style reasoning [38], assertions are either single state assertions p and q or binary rely/guarantee conditions R and G. Note here states refer to the relational states Σ defined in Figure 5.7.

Figure 5.8(b) shows the semantics of the single state assertions. We use standard separation logic assertions to describe the concrete object memory σ. Following Parkinson et al. [55], we treat program variables as resource (just as in Section 4.3) and use $\mathit{own}(x)$ for the ownership of the variable x in the concrete code. $E_1 \mapsto E_2$ specifies a singleton heap with E_2 stored at the location E_1.

$$f \perp g \quad \text{iff} \quad dom(f) \cap dom(g) = \emptyset$$

$$(s_1, h_1) \perp (s_2, h_2) \quad \text{iff} \quad s_1 \perp s_2 \wedge h_1 \perp h_2 \qquad (s_1, h_1) \uplus (s_2, h_2) \stackrel{\text{def}}{=} (s_1 \uplus s_2, h_1 \uplus h_2)$$

$$\Delta_1 \perp \Delta_2 \quad \text{iff} \quad \forall U_1, \sigma_1, U_2, \sigma_2.\ (U_1, \sigma_1) \in \Delta_1 \wedge (U_2, \sigma_2) \in \Delta_2 \implies U_1 \perp U_2 \wedge \sigma_1 \perp \sigma_2$$

$$\Delta_1 * \Delta_2 \stackrel{\text{def}}{=} \{(U_1 \uplus U_2, \sigma_1 \uplus \sigma_2) \mid (U_1, \sigma_1) \in \Delta_1 \wedge (U_2, \sigma_2) \in \Delta_2\}$$

$$\Sigma_1 * \Sigma_2 \stackrel{\text{def}}{=} (\sigma_1 \uplus \sigma_2, \Delta_1 * \Delta_2)\,, \ \text{if } \Sigma_1 = (\sigma_1, \Delta_1), \Sigma_2 = (\sigma_2, \Delta_2), \sigma_1 \perp \sigma_2 \text{ and } \Delta_1 \perp \Delta_2$$

$$\Sigma_1 \oplus \Sigma_2 \stackrel{\text{def}}{=} (\sigma, \Delta_1 \cup \Delta_2)\,, \ \text{if } \Sigma_1 = (\sigma, \Delta_1) \text{ and } \Sigma_2 = (\sigma, \Delta_2)$$

(a) auxiliary definitions

$$((s,h), \Delta) \models B \quad \text{iff} \quad \forall U, s_a, h_a.\ (U, (s_a, h_a)) \in \Delta \implies [\![B]\!]_{s \uplus s_a} = \mathbf{true}$$

$$((s,h), \Delta) \models own(x) \quad \text{iff} \quad (dom(s) = \{x\}) \wedge (\forall U, s_a, h_a.\ (U, (s_a, h_a)) \in \Delta \Longrightarrow (s_a = \emptyset))$$

$$((s,h), \Delta) \models \boldsymbol{own}(x) \quad \text{iff} \quad (s = \emptyset) \wedge (\forall U, s_a, h_a.\ (U, (s_a, h_a)) \in \Delta \Longrightarrow (dom(s_a) = \{x\}))$$

$$((s,h), \Delta) \models \mathsf{emp} \quad \text{iff} \quad (s = \emptyset) \wedge (h = \emptyset) \wedge (\Delta = \{(\emptyset, (\emptyset, \emptyset))\})$$

$$((s,h), \Delta) \models E_1 \mapsto E_2 \quad \text{iff} \quad \exists l, n.\ [\![E_1]\!]_s = l \wedge [\![E_2]\!]_s = n \wedge h = \{l \rightsquigarrow n\} \wedge \Delta = \{(\emptyset, (\emptyset, \emptyset))\}$$

$$((s,h), \Delta) \models E_1 \Mapsto E_2 \quad \text{iff} \quad \exists s_a, h_a, l, n.\ \Delta = \{(\emptyset, (s_a, h_a))\} \wedge s = \emptyset \wedge h = \emptyset \wedge [\![E_1]\!]_{s_a} = l \wedge [\![E_2]\!]_{s_a} = n \wedge h_a = \{l \rightsquigarrow n\}$$

$$((s,h), \Delta) \models E_1 \rightarrowtail (\gamma, E_2) \quad \text{iff} \quad \exists U, s_a, t, n.\ \Delta = \{(U, (s_a, \emptyset))\} \wedge h = \emptyset \wedge [\![E_1]\!]_{s \uplus s_a} = t \wedge [\![E_2]\!]_{s \uplus s_a} = n \wedge U = \{t \rightsquigarrow (\gamma, n)\}$$

$$((s,h), \Delta) \models E_1 \rightarrowtail (\mathbf{end}, E_2) \quad \text{iff} \quad \exists U, s_a, t, n.\ \Delta = \{(U, (s_a, \emptyset))\} \wedge h = \emptyset \wedge [\![E_1]\!]_{s \uplus s_a} = t \wedge [\![E_2]\!]_{s \uplus s_a} = n \wedge U = \{t \rightsquigarrow (\mathbf{end}, n)\}$$

$$\Sigma \models p * q \quad \text{iff} \quad \exists \Sigma_1, \Sigma_2.\ \Sigma = \Sigma_1 * \Sigma_2 \wedge \Sigma_1 \models p \wedge \Sigma_2 \models q$$

$$\Sigma \models p \oplus q \quad \text{iff} \quad \exists \Sigma_1, \Sigma_2.\ \Sigma = \Sigma_1 \oplus \Sigma_2 \wedge \Sigma_1 \models p \wedge \Sigma_2 \models q$$

(b) semantics of state assertions

$$(\Sigma, \Sigma') \models [p] \quad \text{iff} \quad \Sigma \models p \wedge \Sigma = \Sigma'$$

$$(\Sigma, \Sigma') \models p \ltimes q \quad \text{iff} \quad \Sigma \models p \wedge \Sigma' \models q$$

$$(\Sigma, \Sigma') \models p \propto q \quad \text{iff} \quad \exists \Sigma_1, \Sigma_1'.\ (\Sigma = \Sigma_1 \oplus \Sigma) \wedge (\Sigma' = \Sigma_1' \oplus \Sigma) \wedge \Sigma_1 \models p \wedge \Sigma_1' \models q$$

$$(\Sigma, \Sigma') \models R_1 * R_2 \quad \text{iff} \quad \exists \Sigma_1, \Sigma_2, \Sigma_1', \Sigma_2'.\ (\Sigma = \Sigma_1 * \Sigma_2) \wedge (\Sigma' = \Sigma_1' * \Sigma_2') \wedge (\Sigma_1, \Sigma_1') \models R_1 \wedge (\Sigma_2, \Sigma_2') \models R_2$$

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

$$[G]_I \stackrel{\text{def}}{=} (G \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$$

(c) semantics of actions

Figure 5.8 Semantics of assertions.

New assertions are introduced to specify Δ. $\boldsymbol{own}(x)$ means the ownership of the abstract variable x at every speculation in Δ. We assume the store for the abstract operations is always disjoint with the concrete store. We generalize the boolean expression B to specify both stores, so that we can describe the relationships between concrete variables and abstract ones. $E_1 \Mapsto E_2$ specifies the heap of the sole speculation in Δ, with an empty pending thread pool U, while $E_1 \rightarrowtail (\gamma, E_2)$ and $E_1 \rightarrowtail (\mathbf{end}, E_2)$ specify the single thread E_1 in U.

The semantics of separating conjunction $p * q$ is similar as in separation logic, except that it is now lifted to assertions over the relational states Σ. Note that the underlying "disjoint union" over Δ for separating conjunction should not be confused with the normal disjoint union operator over sets. The former (denoted as $\Delta_1 * \Delta_2$ in Figure 5.8(a)) describes the split of pending thread pools and/or abstract object memory. For example, the left side Δ in the following equation specifies two speculations of threads t_1 and t_2 (we assume the abstract object memory is empty and omitted here), and it can be split into two sets Δ_1 and Δ_2 on the right side, each of which describes the speculations of a single thread.

$$\left\{ \begin{array}{|c|c|}\hline \mathsf{t}_1 & \Upsilon_1 \\ \hline \mathsf{t}_2 & \Upsilon_2 \\ \hline \end{array}\ , \quad \begin{array}{|c|c|}\hline \mathsf{t}_1 & \Upsilon_1 \\ \hline \mathsf{t}_2 & \Upsilon_2' \\ \hline \end{array}\ \right\} \quad = \quad \begin{array}{c} \left\{ \begin{array}{|c|c|}\hline \mathsf{t}_1 & \Upsilon_1 \\ \hline \end{array}\ \right\} \\ * \\ \left\{ \begin{array}{|c|c|}\hline \mathsf{t}_2 & \Upsilon_2 \\ \hline \end{array}\ , \quad \begin{array}{|c|c|}\hline \mathsf{t}_2 & \Upsilon_2' \\ \hline \end{array}\ \right\} \end{array}$$

The most interesting new assertion is $p \oplus q$, where p and q specify two different speculations. It is this assertion that reflects uncertainty about the abstract level. However, the readers should not confuse $\oplus$ with disjunction. It is more like conjunction since it says Δ contains both speculations satisfying p *and* those satisfying q. As an example, the above equation could be formulated at the assertion level using $*$ and $\oplus$.

$$\begin{array}{rl} & (\mathsf{t}_1 \rightarrowtail \Upsilon_1 * \mathsf{t}_2 \rightarrowtail \Upsilon_2) \oplus (\mathsf{t}_1 \rightarrowtail \Upsilon_1 * \mathsf{t}_2 \rightarrowtail \Upsilon_2') \\ \Leftrightarrow & \mathsf{t}_1 \rightarrowtail \Upsilon_1 * (\mathsf{t}_2 \rightarrowtail \Upsilon_2 \oplus \mathsf{t}_2 \rightarrowtail \Upsilon_2') \end{array}$$

Rely and guarantee assertions specify transitions over Σ. Their semantics is given in Fig. 5.8(c). Following LRG [20], we use $[p]$ for identity transitions with the states satisfying p. The action $p \ltimes q$ says that the initial states satisfy p and the resulting states satisfy q. Besides, we introduce a new action $p \propto q$ to specify the speculative behaviors. It requires the initial state to contain the speculations p. Then the action adds new speculations q to the state, and keep all the original speculations. For instance, $(\mathsf{t} \rightarrowtail \Upsilon) \propto (\mathsf{t} \rightarrowtail \Upsilon')$ represents the speculative execution of the operation Υ by thread t. After the action, we have

$$\frac{p \Rrightarrow_{\mathsf{t}} q}{\vdash_{\mathsf{t}} \{p\}\mathbf{linself}\{q\}}\ (\textsc{linself}) \qquad \frac{p \Rightarrow (E = E) \qquad p \Rrightarrow_E q}{\vdash_{\mathsf{t}} \{p\}\mathbf{lin}(E)\{q\}}\ (\textsc{lin})$$

$$\frac{p \Rrightarrow_{\mathsf{t}} q}{\vdash_{\mathsf{t}} \{p\}\mathbf{trylinself}\{p \oplus q\}}\ (\textsc{tryself}) \qquad \frac{p \Rightarrow (E = E) \qquad p \Rrightarrow_E q}{\vdash_{\mathsf{t}} \{p\}\mathbf{trylin}(E)\{p \oplus q\}}\ (\textsc{try})$$

$$\frac{\mathsf{SpecExact}(p) \qquad q_1|_p \Rrightarrow q_2}{\vdash_{\mathsf{t}} \{q_1\}\mathbf{commit}(p)\{q_2\}}\ (\textsc{commit})$$

$$\frac{}{\vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}\mathbf{E}[\,\mathbf{return}\ E\,]\{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}}\ (\textsc{ret})$$

$$\frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}}{\vdash_{\mathsf{t}} \{p * r\}\widetilde{C}\{q * r\}}\ (\textsc{frame}) \qquad \frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\} \qquad \vdash_{\mathsf{t}} \{p'\}\widetilde{C}\{q'\}}{\vdash_{\mathsf{t}} \{p \oplus p'\}\widetilde{C}\{q \oplus q'\}}\ (\textsc{Spec-Conj})$$

$$\frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}}{\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}}\ (\textsc{env})$$

$$\frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\} \qquad (p \ltimes q) \Rightarrow G * \mathsf{True} \qquad p \vee q \Rightarrow I * \mathsf{true} \qquad I \triangleright G}{[I], G, I \vdash_{\mathsf{t}} \{p\}\langle\widetilde{C}\rangle\{q\}}\ (\textsc{atom})$$

$$\frac{[I], G, I \vdash_{\mathsf{t}} \{p\}\langle\widetilde{C}\rangle\{q\} \qquad \mathsf{Sta}(\{p, q\}, R * \mathsf{Id}) \qquad I \triangleright R}{R, G, I \vdash_{\mathsf{t}} \{p\}\langle\widetilde{C}\rangle\{q\}}\ (\textsc{atom-r})$$

$$\frac{R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}_1\{q\} \qquad R, G, I \vdash_{\mathsf{t}} \{q\}\widetilde{C}_2\{r\}}{R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}_1; \widetilde{C}_2\{r\}}\ (\textsc{p-seq})$$

$$\frac{R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\} \qquad \mathsf{Sta}(r, R' * \mathsf{Id}) \qquad I' \triangleright \{R', G'\} \qquad r \Rightarrow I' * \mathsf{true}}{R * R', G * G', I * I' \vdash_{\mathsf{t}} \{p * r\}\widetilde{C}\{q * r\}}\ (\textsc{p-frame})$$

Figure 5.9 Selected inference rules.

both the original operation Υ and the result Υ' as speculations. We will show more use of the assertions in the examples of Section 5.5.

5.3.3 Inference Rules

The rules of our logic are shown in Figure 5.9. Each judgment is parameterized with the ID t of the current thread. Rules on the top half are for sequential Hoare-style reasoning. They are proposed to verify code $\widetilde{C}$ in the atomic block $\langle\widetilde{C}\rangle$.

In the LINSELF rule, we use $p \Rrightarrow_{\mathsf{t}} q$ to execute the abstract operation of the

$$\frac{U(\mathsf{t}) = (\mathbf{end}, n)}{(U, \sigma) \dashrightarrow_{\mathsf{t}} (U, \sigma)} \qquad \frac{U(\mathsf{t}) = ((x, \langle C \rangle), n) \quad (\langle C \rangle, (s \uplus \{x \rightsquigarrow n\}, h)) \longrightarrow_{\mathsf{t}} (\mathbf{fret}(n'), (s', h'))}{(U, (s, h)) \dashrightarrow_{\mathsf{t}} (U\{\mathsf{t} \rightsquigarrow (\mathbf{end}, n')\}, (s' \backslash \{x\}, h'))}$$

$$\frac{}{\emptyset \rightarrow_{\mathsf{t}} \emptyset} \qquad \frac{(U, \sigma) \dashrightarrow_{\mathsf{t}} (U', \sigma') \qquad \Delta \rightarrow_{\mathsf{t}} \Delta'}{\{(U, \sigma)\} \uplus \Delta \ \rightarrow_{\mathsf{t}} \ \{(U', \sigma')\} \cup \Delta'}$$

$p \Rrightarrow_E q$ iff
$\forall s, h, \Delta, \mathsf{t}.\ (((s, h), \Delta) \models p) \wedge (\llbracket E \rrbracket_s = \mathsf{t}) \implies \exists \Delta'.\ (\Delta \rightarrow_{\mathsf{t}} \Delta') \wedge ((s, h), \Delta') \models q$

$\Delta|_{\Delta_p} \stackrel{\text{def}}{=}$
$\{(U, \sigma) \mid (U, \sigma) \in \Delta \wedge \exists U_p, \sigma_p, U_1, \sigma_1.\ (U_p, \sigma_p) \in \Delta_p \wedge U = U_p \uplus U_1 \wedge \sigma = \sigma_p \uplus \sigma_1\}$

$(\sigma, \Delta)|_p = \Delta'$ iff $\exists \sigma_p, \Delta_p, \sigma_1.\ ((\sigma_p, \Delta_p) \models p) \wedge (\sigma = \sigma_p \uplus \sigma_1) \wedge (\Delta|_{\Delta_p} = \Delta')$

$q_1|_p \Rrightarrow q_2$ iff $\forall \sigma, \Delta.\ (\sigma, \Delta) \models q_1 \implies \exists \Delta'.\ ((\sigma, \Delta)|_p = \Delta') \wedge (\sigma, \Delta') \models q_2$

$\mathsf{DomExact}(\Delta)$ iff $\forall U, \sigma, U', \sigma'.\ (U, \sigma) \in \Delta \wedge (U', \sigma') \in \Delta \implies (dom(U) = dom(U')) \wedge (dom(\sigma) = dom(\sigma'))$

$\mathsf{SpecExact}(p)$ iff $\forall \Delta, \Delta'.\ ((\_, \Delta) \models p) \wedge ((\_, \Delta') \models p) \implies (\Delta = \Delta') \wedge \mathsf{DomExact}(\Delta)$

$\mathsf{Precise}(p)$ iff $\forall \sigma_1, \Delta_1, \sigma_2, \Delta_2, \sigma_1', \Delta_1', \sigma_2', \Delta_2'.$
$((\sigma_1 \uplus \sigma_2 = \sigma_1' \uplus \sigma_2') \wedge ((\sigma_1, \_) \models p) \wedge ((\sigma_2, \_) \models p) \implies (\sigma_1 = \sigma_2)) \wedge$
$((\Delta_1 * \Delta_2 = \Delta_1' * \Delta_2') \wedge ((\_, \Delta_1) \models p) \wedge ((\_, \Delta_1) \models p) \implies (\Delta_1 = \Delta_2))$

$\mathsf{Sta}(p, R)$ iff $\forall \Sigma, \Sigma'.\ (\Sigma \models p) \wedge ((\Sigma, \Sigma') \models R) \implies \Sigma' \models p$

$I \triangleright R$ iff $([I] \Rightarrow R) \wedge (R \Rightarrow I \ltimes I) \wedge \mathsf{Precise}(I)$

Figure 5.10 Auxiliary definitions for the inference rules.

current thread t, which transforms the abstract states satisfying p to new ones satisfying q. It is formally defined in Figure 5.10, where the transitions over Δ is written as $\Delta \rightarrow_{\mathsf{t}} \Delta'$. For each (U, σ) pair in Δ, if the abstract operation of thread t has not been done, we will execute the atomic code and update the (U, σ) pair correspondingly. The LIN rule is similar. Its side condition $p \Rightarrow (E = E)$ requires that the initial state contain the resource used to evaluate E (with variables as resource, $E = E$ is no longer a tautology). The TRYSELF rule allows the current thread t to speculate. The resulting state contains both the case q where the abstract operation is done and the original case p. The TRY rule is similar.

The COMMIT rule allows us to commit to specific speculations and drop the rest. $\mathbf{commit}(p)$ keeps only the speculations satisfying p. We require p to satisfy $\mathsf{SpecExact}$, which is defined in Figure 5.10. Informally, p should specify an exact set Δ of speculations, all of which should describe the same set of threads and the same domain of abstract object memory (i.e., $\mathsf{DomExact}(\Delta)$ holds). For instance,

the following p_1 satisfies $\mathsf{SpecExact}$, while neither p_2 or p_3 does.

$$\begin{array}{rcl} p_1 & \stackrel{\text{def}}{=} & \mathsf{t} \rightarrowtail (\gamma, n) \oplus \mathsf{t} \rightarrowtail (\mathbf{end}, n') \\ p_2 & \stackrel{\text{def}}{=} & \mathsf{t} \rightarrowtail (\gamma, n) \vee \mathsf{t} \rightarrowtail (\mathbf{end}, n') \\ p_3 & \stackrel{\text{def}}{=} & \mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) \oplus \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2) \end{array}$$

In all of our examples in Section 5.5, the assertion p in $\mathbf{commit}(p)$ describes a singleton speculation, so $\mathsf{SpecExact}(p)$ trivially holds. In the COMMIT rule, we use $q_1|_p \Rrightarrow q_2$ to filter out the wrong speculations which are not consistent with p in the initial state q_1. The result state of the remaining speculations satisfies q_2. The key of its formal definition in Figure 5.10 is the filter $\Delta|_{\Delta_p}$. Here Δ is the initial set of speculations satisfying q_1, and Δ_p is the set of speculations satisfying p. Since $\mathsf{SpecExact}(p)$ holds, we know Δ_p describes an exact domain of threads and abstract memory. This filter allows Δ to contain some extra resource of threads and abstract memory other than those described in p. For instance, the following Δ talks about two threads t_1 and t_2, while p (and Δ_p) could mention t_1 only.

$$\Delta : \left\{ \begin{array}{|c|c|}\hline \mathsf{t}_1 & (\gamma_1, n_1) \\ \hline \mathsf{t}_2 & (\gamma_2, n_2) \\ \hline \end{array}, \quad \begin{array}{|c|c|}\hline \mathsf{t}_1 & (\mathbf{end}, n_1') \\ \hline \mathsf{t}_2 & (\mathbf{end}, n_2') \\ \hline \end{array} \right\}$$

Let p be $\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1)$. Then the filter $\Delta|_{\Delta_p}$ will keep only the left speculation and discard the other. Thus the following judgment holds.

$$\begin{array}{ll} \vdash_{\mathsf{t}} & \{(\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)) \oplus (\mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1') * \mathsf{t}_2 \rightarrowtail (\mathbf{end}, n_2'))\} \\ & \mathbf{commit}(\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1)) \\ & \{\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)\} \end{array}$$

If p is $\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) \oplus \mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1')$, the filter $\Delta|_{\Delta_p}$ will keep both speculations and the following judgment holds.

$$\begin{array}{ll} \vdash_{\mathsf{t}} & \{(\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)) \oplus (\mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1') * \mathsf{t}_2 \rightarrowtail (\mathbf{end}, n_2'))\} \\ & \mathbf{commit}(\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) \oplus \mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1')) \\ & \{(\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)) \oplus (\mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1') * \mathsf{t}_2 \rightarrowtail (\mathbf{end}, n_2'))\} \end{array}$$

Before the current thread returns, it must know its abstract operation has been done, as required in the RET rule. We also have a standard FRAME rule as in separation logic for local reasoning. Besides, the SPEC-CONJ rule allows us to separately consider different speculations. It is like the conjunction rule in traditional Hoare logic.

Rules in the bottom half of Figure 5.9 show how to do rely-guarantee style concurrency reasoning, which are very similar to those in LRG [20]. As in LRG, we use a precise invariant I to specify the boundary of the well-formed shared

resource. When there is no resource sharing (i.e., $\widetilde{C}$ only accesses its private resource), we can apply the ENV rule and verify $\widetilde{C}$ as if it was sequential code. Here the rely/guarantee conditions are Emp and the invariant is emp (defined in Figure 5.8), showing the shared resource is empty. The ATOM rule says we could reason sequentially about code in the atomic block. Then we can lift it to the concurrent setting as long as its effects over the shared resource satisfy the guarantee G, which is fenced by the invariant I. The fence $I \triangleright G$ is defined in Figure 5.10. In this step we assume the environment does not update shared resource, thus using $[I]$ as the rely condition. To allow general environment behaviors R, we should apply the ATOM-R rule later. The rule requires that R be fenced by I and the pre- and post-conditions be stable with respect to $R * \mathsf{Id}$, R for the shared resource and Id for the private (i.e., the environment does not touch the private resource of the current thread). The stability Sta is defined in Figure 5.10, and here $\mathsf{Sta}(\{p, q\}, R * \mathsf{Id})$ requires both p and q be stable with respect to $R * \mathsf{Id}$. We also show the P-SEQ rule for sequential compositions and the P-FRAME rule for local reasoning in concurrent settings. They have the same forms as in LRG, but note the assertions here are interpreted over the relational states Σ.

Linking with client program verification. Our relational logic is introduced for object verification, but it can also be used to verify client code, since it is just an extension over the general-purpose concurrent logic LRG (which includes the rule for parallel composition). Moreover, as we will see in Section 5.4, our logic ensures contextual refinement. Therefore, to verify a program W, we could replace the object implementation with the abstract operations and verify the corresponding abstract program W' instead. Since W' abstracts away the concrete object representation and method implementation details, this approach provides us with "separation and information hiding" [54] over the object, but still keeps enough information (i.e., the abstract operations) about the method calls in concurrent client verification.

5.3.4 Semantics and Partial Correctness

We first show some key operational semantics rules for the instrumented code $\widetilde{C}$ under the relational state model Σ in Figure 5.11.

A single step execution of the instrumented code by thread t is represented as $(\widetilde{C}, \Sigma) \longhookrightarrow_{\mathsf{t}} (\widetilde{C}', \Sigma')$. When we reach the **return** E command (the second rule in Figure 5.11), we must know for sure that the abstract operation of thread t has been done. That is, in every speculation U in Δ, we always know $U(\mathsf{t})$ is **end**

$$\frac{(C,\sigma) \longrightarrow_{\mathsf{t}} (C',\sigma') \qquad C \neq \mathbf{E}[\,\mathbf{return}\ \_\,]}{(C,(\sigma,\Delta)) \hookrightarrow_{\mathsf{t}} (C',(\sigma',\Delta))}$$

$$\frac{[\![E]\!]_s = n \qquad \forall U.\ (U,\_) \in \Delta \implies U(\mathsf{t}) = (\mathbf{end},n)}{(\mathbf{E}[\,\mathbf{return}\ E\,],((s,h),\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{skip},((s,h),\Delta))}$$

$$\frac{\Delta \rightarrow_{\mathsf{t}} \Delta'}{(\mathbf{E}[\,\mathbf{linself}\,],(\sigma,\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,],(\sigma,\Delta'))}$$

$$\frac{[\![E]\!]_s = \mathsf{t}' \qquad \Delta \rightarrow_{\mathsf{t}'} \Delta'}{(\mathbf{E}[\,\mathbf{lin}(E)\,],((s,h),\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,],((s,h),\Delta'))}$$

$$\frac{\Delta \rightarrow_{\mathsf{t}} \Delta'}{(\mathbf{E}[\,\mathbf{trylinself}\,],(\sigma,\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,],(\sigma,\Delta \cup \Delta'))}$$

$$\frac{[\![E]\!]_s = \mathsf{t}' \qquad \Delta \rightarrow_{\mathsf{t}'} \Delta'}{(\mathbf{E}[\,\mathbf{trylin}(E)\,],((s,h),\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,],((s,h),\Delta \cup \Delta'))}$$

$$\frac{\mathsf{SpecExact}(p) \qquad (\sigma,\Delta)|_p = \Delta'}{(\mathbf{E}[\,\mathbf{commit}(p)\,],(\sigma,\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,],(\sigma,\Delta'))}$$

$$\frac{(\widetilde{C},\Sigma) \hookrightarrow_{\mathsf{t}} (\widetilde{C}',\Sigma')}{(\widetilde{C},\Sigma) \overset{R}{\hookrightarrow}_{\mathsf{t}} (\widetilde{C}',\Sigma')} \qquad \frac{(\Sigma,\Sigma') \models R}{(\widetilde{C},\Sigma) \overset{R}{\hookrightarrow}_{\mathsf{t}} (\widetilde{C},\Sigma')}$$

Figure 5.11 Operational semantics in the relational state model.

with the same return value E. Meanings of the auxiliary commands have been explained before. Here we use $\Delta \rightarrow_{\mathsf{t}} \Delta'$ to characterize the changes over Δ made by executing thread t's abstract operations. The semantics of $\mathbf{commit}(p)$ requires p to satisfy $\mathsf{SpecExact}$ and uses $(\sigma,\Delta)|_p = \Delta'$ to filter out the wrong speculations. These auxiliary definitions are given in Figure 5.10.

Based on the thread-local semantics, we could next define the transitions $(\widetilde{C},\Sigma) \overset{R}{\hookrightarrow}_{\mathsf{t}} (\widetilde{C},\Sigma)$, which describe the behaviors of thread t with interference R from the environment.

Semantics preservation by the instrumentation. It is easy to see that the newly introduced auxiliary commands do not change the physical state σ, nor do they affect the program control flow. Thus the instrumentation does not change program behaviors, unless the auxiliary commands are inserted into the wrong places and they get stuck, but this can be prevented by our program logic.

Soundness with respect to partial correctness. Following LRG [20], we could give the semantics of the logic judgment as $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$, which encodes the partial correctness of $\widetilde{C}$ with respect to the pre- and post-conditions. We first define $\models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$, the semantics of the judgment for sequential reasoning.

Definition 5.7 (Sequential Judgment Semantics)**.** $\models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$ holds iff for any Σ such that $\Sigma \models p$, both the following are true.

1. For any Σ', if $(\widetilde{C}, \Sigma) \hookrightarrow^*_{\mathsf{t}} (\mathbf{skip}, \Sigma')$, then $\Sigma' \models q$.
2. $(\widetilde{C}, \Sigma) \hookrightarrow\!\!\!\!\!\!\rightarrow^*_{\mathsf{t}} \mathbf{abort}$.

Definition 5.8 (Concurrent Judgment Semantics)**.** $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$ holds iff for any Σ such that $\Sigma \models p$, both the following are true.

1. For any Σ', if $(\widetilde{C}, \Sigma) \xhookrightarrow{R*\mathsf{Id}}{}^*_{\mathsf{t}} (\mathbf{skip}, \Sigma')$, then $\Sigma' \models q$.
2. For any n, $(\widetilde{C}, \Sigma, R * \mathsf{Id})\ \mathsf{guarantees}^n_{\mathsf{t}}\ (G * \mathsf{True})$.

Here the property $(\widetilde{C}, \Sigma, R)\ \mathsf{guarantees}^n_{\mathsf{t}}\ G$ is inductively defined as follows.

1. $(\widetilde{C}, \Sigma, R)\ \mathsf{guarantees}^0_{\mathsf{t}}\ G$ always holds.
2. $(\widetilde{C}, \Sigma, R)\ \mathsf{guarantees}^{k+1}_{\mathsf{t}}\ G$ iff all of the following hold.
   (a) $(\widetilde{C}, \Sigma) \hookrightarrow\!\!\!\!\!\!\rightarrow_{\mathsf{t}} \mathbf{abort}$.
   (b) For any Σ', if $(\Sigma, \Sigma') \models R$, then $(\widetilde{C}, \Sigma', R)\ \mathsf{guarantees}^k_{\mathsf{t}}\ G$.
   (c) For any $\widetilde{C}'$ and Σ', if $(\widetilde{C}, \Sigma) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', \Sigma')$, then $(\Sigma, \Sigma') \models G$ and $(\widetilde{C}', \Sigma', R)\ \mathsf{guarantees}^k_{\mathsf{t}}\ G$.

Informally, $(\widetilde{C}, \Sigma, R)\ \mathsf{guarantees}^n_{\mathsf{t}}\ G$ says, executing in an initial state Σ and an environment R, the thread t's code $\widetilde{C}$ does not abort within n steps and each state transition made by $\widetilde{C}$ satisfies G. Then $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$ requires all executions that satisfy the assumptions about the initial state and the environment interference also satisfy the guarantee and the postcondition. That is, given an initial state satisfying p and the environment R, the executions of $\widetilde{C}$ do not abort, all its steps satisfy G and the final states satisfy q if it terminates. We can prove that our logic ensures partial correctness, i.e., the judgment implies its semantics.

Theorem 5.9 (Partial Correctness)**.** *If* $R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$*, then* $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$.

Theorem 5.9 is proved by induction over the derivation of $R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$. In the next section, we give a stronger soundness of the logic, i.e., soundness with respect to linearizability.

5.4 Soundness via Simulation

Our logic intuitively relates the concrete object code with its abstract level specification. In this section we formalize the intuition and prove that the logic indeed ensures object linearizability. The proof is constructed in the following steps. We propose a new rely-guarantee-based forward-backward simulation between the concrete code and the abstract operation. We prove the simulation is compositional and implies contextual refinement between the two sides, and our logic indeed establishes such a simulation. Thus the logic establishes contextual refinement. Finally we get linearizability following Theorem 5.6.

Below we first define a rely-guarantee-based forward-backward simulation. It extends RGSim with the support of the helping mechanism and speculations.

Definition 5.10 (New Simulation for Method)**.** $R, G, I \models_{\mathsf{t}} (x, C) \preceq_p \gamma$ iff

$$\forall \sigma, \Delta.\ \ (\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\gamma, x) * p) \implies R, G, I \models_{\mathsf{t}} (C; \mathbf{noret}, \sigma) \preceq_p \Delta\,.$$

Whenever $R, G, I \models_{\mathsf{t}} (C, \sigma) \preceq_p \Delta$, then $(\sigma, \Delta) \models I * \mathsf{true}$ and the following are true.

1. If $C \neq \mathbf{E}[\,\mathbf{return}\ \_\,]$, then both the following hold.
   (a) For any C', σ'', σ_F and Δ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow\!\triangleright_{\mathsf{t}} (C', \sigma'')$ and $\Delta \perp \Delta_F$, then there exist σ' and Δ' such that $\sigma'' = \sigma' \uplus \sigma_F$, $(\Delta * \Delta_F) \rightrightarrows (\Delta' * \Delta_F)$, $((\sigma, \Delta), (\sigma', \Delta')) \models (G * \mathsf{True})$ and $R, G, I \models_{\mathsf{t}} (C', \sigma') \preceq_p \Delta'$.
   (b) For any σ_F, $(C, \sigma \uplus \sigma_F) \not\longrightarrow\!\triangleright_{\mathsf{t}} \mathbf{abort}$.
2. For any σ' and Δ', if $((\sigma, \Delta), (\sigma', \Delta')) \models (R * \mathsf{Id})$, then $R, G, I \models_{\mathsf{t}} (C, \sigma') \preceq_p \Delta'$.
3. If $C = \mathbf{E}[\,\mathbf{return}\ E\,]$ and $\sigma = (s, h)$, then there exists n' such that $[\![E]\!]_s = n'$ and $(\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\mathbf{end}, n') * \mathit{own}(x) * p)$.

As in RGSim, $R, G, I \models_{\mathsf{t}} (x, C) \preceq_p \gamma$ says, the concrete implementation C is simulated by the abstract operation γ under the interference with the environment, which is specified by R and G. The new simulation holds if the executions of the concrete code C are related to the *speculative* executions of some Δ. The Δ could specify abstract operations of other threads that might be helped, as well as the current thread t. Initially, the abstract operation of t is γ, with the same argument as the concrete side x. The abstract operations of other threads can be known from the precondition p.

For each step of the concrete code C, we require it to be safe, and correspond to some steps of Δ, as shown in the first condition in Definition 5.10. We define the transition $\Delta \rightrightarrows \Delta'$ as follows.

$$\begin{array}{l}\Delta \rightrightarrows \Delta' \quad \text{iff} \\ \quad \forall U', \sigma'.\ (U', \sigma') \in \Delta' \implies \exists U, \sigma.\ (U, \sigma) \in \Delta \wedge (U, \sigma) \dashrightarrow^* (U', \sigma'), \\ \text{where } (U, \sigma) \dashrightarrow (U', \sigma') \stackrel{\text{def}}{=} \exists \mathsf{t}.\ (U, \sigma) \dashrightarrow_{\mathsf{t}} (U', \sigma') \\ \quad \text{and } (U, \sigma) \dashrightarrow_{\mathsf{t}} (U', \sigma') \text{ has been defined in Figure 5.10.}\end{array}$$

It says, any (U', σ') pair in Δ' should be "reachable" from Δ. This allows us to execute the abstract operation of some thread t' (which could be the current thread t or some others), or drop some (U, σ) pair in Δ. The former is like a step of $\mathbf{trylin}(\mathsf{t}')$ or $\mathbf{lin}(\mathsf{t}')$, depending on whether or not we keep the original abstract operation of t'. The latter can be viewed as a **commit** step, in which we discard the wrong speculations.

Inspired by Vafeiadis [68], we directly embed the framing aspect of separation logic in Definition 5.10. We introduce the explicit frame states σ_F and Δ_F at the concrete and abstract levels to represent the remaining parts of the states owned by other threads running in parallel. The concrete code and the abstract operation of the current thread should not change the frame states during their executions.

We also require the related steps at the two levels to satisfy the guarantee $G * \mathsf{True}$, G for the shared part and True (arbitrary transitions) for the local part. Symmetrically, the second condition in Definition 5.10 says, the simulation should be preserved under the environment interference $R * \mathsf{Id}$, R for the shared part and Id (identity transitions) for the local part.

Finally, when the method returns (the last condition in Definition 5.10), we require the current thread t has finished its abstract operation, and the return values match at the two levels.

Like RGSim, our new simulation is *compositional*, thus can ensure a contextual refinement between the implementation and the abstract operation, as shown in the following lemma.

Lemma 5.11 (Simulation Implies Contextual Refinement)**.** *For any Π, Π_A and φ, suppose there exist R, G, I and p such that the following hold for all t.*

1. *For any f such that $\Pi(f) = (x, C)$, we have $R_{\mathsf{t}}, G_{\mathsf{t}}, I \models_{\mathsf{t}} \Pi(f) \preceq_{p_{\mathsf{t}}} \Pi_A(f)$ and $x \notin dom(I)$.*
2. *$R_{\mathsf{t}} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$, $I \triangleright \{R_{\mathsf{t}}, G_{\mathsf{t}}\}$, $p_{\mathsf{t}} \Rightarrow I$, and $\mathsf{Sta}(p_{\mathsf{t}}, R_{\mathsf{t}})$.*
3. *$\lfloor \varphi \rfloor \Rightarrow \bigwedge_{\mathsf{t}} p_{\mathsf{t}}$.*

$$\mathsf{Er}(\mathbf{linself}) \stackrel{\text{def}}{=} \mathbf{skip} \qquad \mathsf{Er}(\mathbf{trylinself}) \stackrel{\text{def}}{=} \mathbf{skip} \qquad \mathsf{Er}(\mathbf{lin}(E)) \stackrel{\text{def}}{=} \mathbf{skip}$$
$$\mathsf{Er}(\mathbf{trylin}(E)) \stackrel{\text{def}}{=} \mathbf{skip} \qquad \mathsf{Er}(\mathbf{commit}(p)) \stackrel{\text{def}}{=} \mathbf{skip} \qquad \mathsf{Er}(C) \stackrel{\text{def}}{=} C$$
$$\mathsf{Er}(\langle \widetilde{C} \rangle) \stackrel{\text{def}}{=} \langle\, \mathsf{Er}(\widetilde{C}) \,\rangle \qquad \mathsf{Er}(\widetilde{C}_1; \widetilde{C}_2) \stackrel{\text{def}}{=} \mathsf{Er}(\widetilde{C}_1); \mathsf{Er}(\widetilde{C}_2)$$
$$\mathsf{Er}(\mathbf{if}\ (B)\ \widetilde{C}_1\ \mathbf{else}\ \widetilde{C}_2) \stackrel{\text{def}}{=} \mathbf{if}\ (B)\ \mathsf{Er}(\widetilde{C}_1)\ \mathbf{else}\ \mathsf{Er}(\widetilde{C}_2)$$
$$\mathsf{Er}(\mathbf{while}\ (B)\{\widetilde{C}\}) \stackrel{\text{def}}{=} \mathbf{while}\ (B)\{\,\mathsf{Er}(\widetilde{C})\,\}$$

Figure 5.12 Erasure of auxiliary commands.

Then $\Pi \sqsubseteq_\varphi \Pi_A$.

Here $x \notin dom(I)$ means the formal argument x is always in the local state, and $\lfloor \varphi \rfloor$ lifts φ to a state assertion: $\lfloor \varphi \rfloor \stackrel{\text{def}}{=} \{(\sigma, \{(\emptyset, \sigma_a)\}) \mid \varphi(\sigma) = \sigma_a\}$.

Lemma 5.11 allows us to prove the contextual refinement $\Pi \sqsubseteq_\varphi \Pi_A$ by showing the simulation $R_\mathsf{t}, G_\mathsf{t}, I \models_\mathsf{t} \Pi(f) \preceq_{p_\mathsf{t}} \Pi_A(f)$ for each method f, where R, G and p are defined over the shared states fenced by the invariant I, and the interference constraint $R_\mathsf{t} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$ holds following Rely-Guarantee reasoning [38]. Its proof is similar to the compositionality proofs of RGSim, but now we need to be careful with the helping between threads and the speculations.

Lemma 5.12 (Logic Ensures Simulation for Method)**.** *For any* t*,* x*,* C*,* γ*,* R*,* G*,* I *and* p*, if there exists* $\widetilde{C}$ *such that* $\mathsf{Er}(\widetilde{C}) = (C; \mathbf{noret})$ *and*

$$R, G, I \vdash_\mathsf{t} \{\mathsf{t} \rightarrowtail (\gamma, x) * p\}\ \widetilde{C}\ \{\mathsf{t} \rightarrowtail (\mathbf{end}, \_) * own(x) * p\}\,,$$

then $R, G, I \models_\mathsf{t} (x, C) \preceq_p \gamma$.

Here we use $\mathsf{Er}(\widetilde{C})$ to erase the instrumented auxiliary commands in $\widetilde{C}$. It is defined in Figure 5.12.

Lemma 5.12 shows that, verifying $\widetilde{C}$ in our logic establishes the simulation between the original code and the abstract operation. Its proof is based on Theorem 5.9, i.e., our logic ensures the standard rely-guarantee-style partial correctness of the instrumented code. Then we build the simulation by projecting the instrumented semantics (Figure 5.11) to the concrete semantics of C (Figure 5.5) and the speculative steps $\rightrightarrows$ of Δ.

Finally, from Lemmas 5.11 and 5.12, we get the soundness theorem of our logic, which says our logic can verify linearizability.

Theorem 5.13 (Logic Soundness)**.** *For any* Π*,* Π_A *and* φ*, suppose there exist* R*,* G*,* I *and* p *such that the following hold for all* t*.*

1. *For any* f*, if* $\Pi(f) = (x, C)$*, there exists* $\widetilde{C}$ *such that*

Objects	Helping	Fut. LP	Java pkg	HS book
Treiber stack [62]				√
HSY stack [30]	√			√
MS two-lock queue [51]				√
MS lock-free queue [51]		√	√	√
DGLM queue [17]		√		
Lock-coupling list [33]				√
Optimistic list [33]				√
Heller et al. lazy list [29]	√	√		√
Harris-Michael lock-free list [26, 50]	√	√	√	√
Pair snapshot [58]		√		
CCAS [65]	√	√		
RDCSS [27]	√	√		

Table 5.1 Verified algorithms using our logic.

$$R_\mathsf{t}, G_\mathsf{t}, I \vdash_\mathsf{t} \{\mathsf{t} \rightarrowtail (\Pi_A(f), x) * p_\mathsf{t}\}\, \widetilde{C}\, \{\mathsf{t} \rightarrowtail (\mathbf{end}, \_) * own(x) * p_\mathsf{t}\}\,,$$

$\mathsf{Er}(\widetilde{C}) = (C; \mathbf{noret})$, *and* $x \notin dom(I)$.

2. $R_\mathsf{t} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$, $p_\mathsf{t} \Rightarrow I$, *and* $\mathsf{Sta}(p_\mathsf{t}, R_\mathsf{t})$.

3. $\lfloor \varphi \rfloor \Rightarrow \bigwedge_\mathsf{t} p_\mathsf{t}$.

Then $\Pi \sqsubseteq_\varphi \Pi_A$, *and thus* $\Pi \preceq_\varphi \Pi_A$.

5.5 Examples

Our logic gives us an effective approach to verify linearizability. As shown in Table 5.1, we have verified 12 algorithms, including two stacks, three queues, four lists and three algorithms on atomic memory reads or writes. Table 5.1 summarizes their features, including the helping mechanism (**Helping**) and future-dependent LPs (**Fut. LP**). Some of them are used in the `java.util.concurrent` package (**Java pkg**). The last column (**HS book**) shows whether it occurs in Herlihy and Shavit's classic textbook on concurrent algorithms [33]. We have almost covered all the fine-grained stacks, queues and lists in the book. We can see that our logic supports various objects ranging from simple ones with static LPs to sophisticated ones with non-fixed LPs. Although many of the examples can be verified using other approaches, we provide the first program logic which is proved sound and useful enough to verify all of these algorithms.

In general we verify linearizability in the following steps. First we instrument the code with the auxiliary commands such as **linself**, **trylin**(E) and **commit**(p)

```
 1 readPair(int i, j) {
     {I * (cid ⤳ (γ, (i, j)))}
 2   local a, b, v, w, done := false;
     {((¬done) * I * (cid ⤳ (γ, (i, j)) ⊕ true)) ∨ (done * I * (cid ⤳ (end, (a, b))))}
 3   while(!done) {
       {(¬done) * I * (cid ⤳ (γ, (i, j)) ⊕ true)}
 4     < a := m[i].d; v := m[i].v; >
       {∃v'. (¬done) * (I ∧ readCell(i, a, v; v')) * (cid ⤳ (γ, (i, j)) ⊕ true)}
 5     < b := m[j].d; w := m[j].v;  trylinself; >
       {∃v'. (¬done) * (I ∧ readCell(i, a, v; v') ∧ readCell(j, b, w; _)) * afterTry}
 6     if (v = m[i].v) {
         {I * (cid ⤳ (end, (a, b)) ⊕ true)}
 7       commit(cid ⤳ (end, (a, b)));
         {I * (cid ⤳ (end, (a, b)))}
 8       done := true;
 9     }
10   }
     {I * (cid ⤳ (end, (a, b)))}
11   return (a, b);
     {I * (cid ⤳ (end, (a, b)))}
12 }
```

Auxiliary definitions:

$\mathsf{readCell}(i, d, v; v') \stackrel{\text{def}}{=} (\mathsf{cell}(i, d, v) \vee (\mathsf{cell}(i, \_, v') \wedge v < v')) * \mathsf{true}$

$\mathsf{absRes}(a, b; v, v') \stackrel{\text{def}}{=} (\mathtt{cid} \rightarrowtail (\mathbf{end}, (a, b)) \wedge v \leq v') \vee (\mathtt{cid} \rightarrowtail (\mathbf{end}, (\_, b)) \wedge v < v')$

$\mathsf{afterTry} \stackrel{\text{def}}{=} \mathtt{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})) \oplus \mathsf{absRes}(\mathtt{a}, \mathtt{b}; \mathtt{v}, v') \oplus \mathsf{true}$

Figure 5.13 Proof outline of `readPair` in pair snapshot.

at proper program points. The instrumentation should not be difficult based on the intuition of the algorithm. Then, we specify the assertions (as in Theorem 5.13) and reason about the instrumented code by applying our inference rules, just like the usual partial correctness verification in LRG. In our experience, handling the auxiliary commands usually would not introduce much difficulty over the plain verification with LRG. Below we sketch the proofs of three representative examples: the pair snapshot, MS lock-free queue and the CCAS algorithm. The complete proofs of all the 12 algorithms we have verified can be found in the technical report [43].

5.5.1 Pair Snapshot

As discussed in Section 5.1.3, the pair snapshot algorithm has a future-dependent LP. In Fig. 5.13, we show the proof of `readPair` for the current thread `cid`. We will use γ for its abstract operation, which atomically reads the cells `i` and `j` at the abstract level.

First, we insert **trylinself** and **commit** as highlighted in Figure 5.13. The **commit** command says, when the validation at line 6 succeeds, we must have $\texttt{cid} \rightarrowtail (\textbf{end}, (\texttt{a}, \texttt{b}))$ as a possible speculation. This actually requires a correct instrumentation of **trylinself**. In Figure 5.13, we insert it at line 5 because line 5 is the only place where executing γ could return $(\texttt{a}, \texttt{b})$ at the abstract level. We cannot replace it by a **linself**, because if the later validation at line 6 fails, we have to restart to do the original abstract operation γ.

After the instrumentation, we can define the precise invariant I, the rely R and the guarantee G. The invariant I simply maps every memory cell (d, v) at the concrete level to a cell with data d at the abstract level, as shown below.

$$\begin{array}{ll} I & \stackrel{\text{def}}{=} \circledast_{i \in [1..\texttt{size}]}.(\exists d, v.\ \mathsf{cell}(i, d, v)) \\ \text{where} & \mathsf{cell}(i, d, v) \stackrel{\text{def}}{=} (\texttt{m}[i] \mapsto (d, v)) * (\texttt{m}[i] \Rrightarrow d)) \end{array}$$

Every thread guarantees that when writing a cell, it also increases the version number. Thus G is defined as follows.

$$G \stackrel{\text{def}}{=} [\mathit{Write}]_I \qquad \mathit{Write} \stackrel{\text{def}}{=} \exists i, v.\ \mathsf{cell}(i, \_, v) \ltimes \mathsf{cell}(i, \_, v+1)$$

Here we use $[G]_I$ short for $(G \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$ (as defined in Figure 5.8(c)). The rely condition R is the same as the guarantee G.

Then we specify the pre- and post-conditions, and reason about the instrumented code using our inference rules. The proof in Figure 5.13 follows the intuition of the algorithm. Note that we relax $\texttt{cid} \rightarrowtail (\gamma, (\texttt{i}, \texttt{j}))$ in the precondition of the method to $\texttt{cid} \rightarrowtail (\gamma, (\texttt{i}, \texttt{j})) \oplus \mathsf{true}$ to ensure the loop invariant (the assertion above line 3). The latter says, `cid` may not have finished its abstract operation, or `cid` has speculatively finished it but needs to redo it.

The `readPair` method in the pair snapshot algorithm is "read-only" in the sense that the abstract operation does not update the abstract object memory. This perhaps means that it does not matter to linearize the method multiple times. In Section 5.5.3 we will verify an algorithm, CCAS, which has future-dependent LPs and is not "read-only". We *can* still "linearize" a method with side effects multiple times.

5.5.2 MS Lock-Free Queue

The widely-used MS lock-free queue [51] also has future-dependent LPs. We show its code in Figure 5.14.

The queue is implemented as a linked list with `Head` and `Tail` pointers. `Head` always points to the first node (a sentinel) in the list, and `Tail` points to either

```
 1 enq(v) {                          16 deq() {
 2  local x, t, s, b;                17  local h, t, s, v, b;
 3  x := cons(v, null);              18  while (true) {
 4  while (true) {                   19   h := Head; t := Tail;
 5   t := Tail; s := t.next;         20   s := h.next;
 6   if (t = Tail) {                 21   if (h = Head)
 7    if (s = null) {                22    if (h = t) {
 8      b:=cas(&(t.next),s,x);       23      if (s = null)
 9      if (b) {                     24        return EMPTY;
10        cas(&Tail, t, x);          25      cas(&Tail, t, s);
11        return; }                  26    }else {
12    }else cas(&Tail, t, s);        27      v := s.val;
13   }                               28      b:=cas(&Head,h,s);
14  }                                29      if(b) return v; }
15 }                                30  } }
```

Figure 5.14 MS lock-free queue code.

the last or second to last node. The `enq` method appends a new node at the tail of the list and advances `Tail`. The `deq` method replaces the sentinel node by its next node and returns the value in the new sentinel. If the list contains only the sentinel node, meaning the queue is empty, then `deq` returns `EMPTY`.

The algorithm employs the helping mechanism for the `enq` method to swing the `Tail` pointer when it lags behind the end of the list. A thread should first try to help the half-finished `enq` by advancing `Tail` (lines `12` and `25` in Fig. 5.14) before doing its own operation. But this helping mechanism would not affect the LP of `enq` which is statically located at line `8` when the `cas` succeeds, since the new node already becomes visible in the queue after being appended to the list, and updating `Tail` will not affect the abstract queue. We simply instrument line `8` as follows to verify `enq`.

```
< b := cas(&(t.next), s, x); if (b) linself; >
```

On the other hand, the original queue algorithm [51] checks `Head` or `Tail` (line `6` or `21` in Figure 5.14) to make sure that its value has not been changed since its local copy was read (at line `5` or `19`), and if it fails, the operation will restart. This check can improve efficiency of the algorithm, but it makes the LP of the `deq` method for the empty queue case depend on future executions. That LP should be at line `20`, if the method returns `EMPTY` at the end of the *same* iteration. In fact, when we read `null` from `h.next` at line `20` (indicating the abstract queue must be empty there), we do not know how the iteration would terminate at that time. If the later check over `Head` at line `21` fails, the operation

$$I \stackrel{\text{def}}{=} \exists h, t, A.\ (\texttt{Q} = A) * (\texttt{Head} = h) * (\texttt{Tail} = t) * \mathsf{lsq}(h, t, A) * \mathsf{garb}(h)$$

$$\mathsf{lsq}(h, t, A) \stackrel{\text{def}}{=} \exists v, A', A''.\ (v :: A = A' :: A'') \wedge \mathsf{ls}(h, A', t) * \mathsf{tls}(t, \_, A'')$$

$$\mathsf{ls}(x, A, y) \stackrel{\text{def}}{=} (x = y \wedge A = \epsilon \wedge emp) \vee (x \neq y \wedge \exists z, v, A'.\ A = v :: A' \wedge \mathsf{node}(x, v, z) * \mathsf{ls}(z, A', y))$$

$$\mathsf{ls}(x, y) \stackrel{\text{def}}{=} \exists A.\ \mathsf{ls}(x, A, y) \qquad \mathsf{garb}(h) \stackrel{\text{def}}{=} \exists g.\ (\texttt{GH} = g) * \mathsf{ls}(g, h)$$

$$\mathsf{tls}(t, x, A) \stackrel{\text{def}}{=} \exists v, v'.\ (A = v \wedge \mathsf{node}(t, v, x) \wedge x = \texttt{null}) \vee (A = v :: v' \wedge \mathsf{last2}(t, v, x, v'))$$

$$\mathsf{tls}(t, x) \stackrel{\text{def}}{=} \exists A.\ \mathsf{tls}(t, x, A) \qquad \mathsf{tls}(t) \stackrel{\text{def}}{=} \mathsf{tls}(t, \_)$$

$$\mathsf{last2}(t, v, x, v') \stackrel{\text{def}}{=} \mathsf{node}(t, v, x) * \mathsf{node}(x, v', \texttt{null})$$

$$\mathsf{last2}(t, x) \stackrel{\text{def}}{=} \mathsf{last2}(t, \_, x, \_) \qquad \mathsf{last2}(t) \stackrel{\text{def}}{=} \mathsf{last2}(t, \_)$$

$$\mathsf{node}(x, v, y) \stackrel{\text{def}}{=} x \mapsto (v, y) \qquad \mathsf{node}(x, y) \stackrel{\text{def}}{=} \mathsf{node}(x, \_, y)$$

$$R = G \stackrel{\text{def}}{=} [\mathit{Enq} \vee \mathit{Deq} \vee \mathit{Swing}]_I$$

$$\mathit{Enq} \stackrel{\text{def}}{=} \exists v, v', A, t, x.\ ((\texttt{Q} = A) * (\texttt{Tail} = t) * \mathsf{node}(t, v, \texttt{null})) \ltimes ((\texttt{Q} = A :: v') * (\texttt{Tail} = t) * \mathsf{last2}(t, v, x, v'))$$

$$\mathit{Deq} \stackrel{\text{def}}{=} \exists v, A, h, t, x, y.\ ((\texttt{Q} = v :: A) * (\texttt{Head} = h) * \mathsf{node}(h, x) * \mathsf{node}(x, v, y) * (\texttt{Tail} = t) \wedge h \neq t) \ltimes ((\texttt{Q} = A) * (\texttt{Head} = x) * \mathsf{node}(h, x) * \mathsf{node}(x, v, y) * (\texttt{Tail} = t))$$

$$\mathit{Swing} \stackrel{\text{def}}{=} \exists v, v', t, x.\ ((\texttt{Tail} = t) * \mathsf{last2}(t, v, x, v')) \ltimes ((\texttt{Tail} = x) * \mathsf{last2}(t, v, x, v'))$$

Figure 5.15 Invariant and rely/guarantee conditions for MS lock-free queue.

would restart and line 20 (which we just executed) may not be the LP. We use our try-commit instrumentation to handle this future-dependent LP. We insert **trylinself** at line 20, as follows.

```
< s := h.next;  if (h = t && s = null) trylinself; >
```

Before the method returns `EMPTY`, we commit to the finished abstract operation, i.e., we insert `commit(cid` $\rightarrowtail$ `(end, EMPTY))` just before line 24. Also, when we know we have to do another iteration, we can commit to the original `DEQ` operation, i.e., we insert `commit(cid` $\rightarrowtail$ `DEQ)` at the end of the loop body (just before line 30).

For the case of nonempty queues, the LP of the `deq` method is statically at line 28 when the `cas` succeeds. Thus we can instrument **linself** there, as shown below.

```
< b := cas(&Head, h, s);  if (b) linself; >
```

After the instrumentation, we define I, R and G and verify the code using our logic rules. As shown in Figure 5.15, the invariant I relates the concrete linked list to the abstract queue. We represent the abstract queue by a value sequence `Q`.

$$\mathsf{readTail}(t,n) \stackrel{\text{def}}{=} (\mathtt{Tail}=t) * \mathsf{tls}(t,n) \vee \mathsf{readTailEnvAdv}(t,n)$$

$$\mathsf{readTailEnvAdv}(t,n) \stackrel{\text{def}}{=} \exists x.\ \mathsf{node}(t,n) * \mathsf{ls}(n,x) * (\mathtt{Tail}=x) * \mathsf{tls}(x)$$

$$\mathsf{readTail}(t) \stackrel{\text{def}}{=} \mathsf{readTail}(t,\_) \qquad \mathsf{readTailEnvAdv}(t) \stackrel{\text{def}}{=} \mathsf{readTailEnvAdv}(t,\_)$$

$$\mathsf{readTailNext}(t,n) \stackrel{\text{def}}{=} \mathsf{readTail}(t,n) \vee \mathsf{readTailNextNullEnv}(t,n)$$

$$\mathsf{readTailNextNullEnv}(t,n) \stackrel{\text{def}}{=} (n=\mathtt{null}) \wedge ((\mathtt{Tail}=t) * \mathsf{last2}(t)) \vee \mathsf{readTailEnvAdv}(t))$$

$$\mathsf{readTailNextNull}(t,n) \stackrel{\text{def}}{=} ((\mathtt{Tail}=t) * \mathsf{node}(t,n) \wedge (n=\mathtt{null})) \vee \mathsf{readTailNextNullEnv}(t,n)$$

$$\mathsf{readTailNextNonnull}(t,n) \stackrel{\text{def}}{=} ((\mathtt{Tail}=t) * \mathsf{last2}(t,n)) \vee \mathsf{readTailEnvAdv}(t,n)$$

$$\mathsf{readHead}(h,x) \stackrel{\text{def}}{=} ((h=x) \wedge (\mathtt{Head}=x)) \vee \mathsf{readHeadEnv}(h,x)$$

$$\mathsf{readHeadEnv}(h,n,x) \stackrel{\text{def}}{=} (h\neq x) \wedge \mathsf{node}(h,n) * \mathsf{ls}(n,x) * (\mathtt{Head}=x)$$

$$\mathsf{readHead}(h) \stackrel{\text{def}}{=} \mathsf{readHead}(h,\_) \qquad \mathsf{readHeadEnv}(h,x) \stackrel{\text{def}}{=} \mathsf{readHeadEnv}(h,n,x)$$

$$\mathsf{readHeadTail}(h,t) \stackrel{\text{def}}{=} (\exists x.\ \mathsf{readHead}(h,x) * \mathsf{ls}(x,t) * \mathsf{readTail}(t)) \vee \mathsf{readHeadTailEnv}(h,t)$$

$$\mathsf{readHeadTailEnv}(h,t) \stackrel{\text{def}}{=} \exists x,y,z.\ \mathsf{ls}(h,t) * \mathsf{node}(t,x) * \mathsf{ls}(x,y) * (\mathtt{Head}=y) * \mathsf{ls}(y,z) * (\mathtt{Tail}=z) * \mathsf{tls}(z)$$

$$\mathsf{readHeadNextAfterTail}(h,n,t) \stackrel{\text{def}}{=} (\mathtt{Head}=h) * (((h=t) \wedge \mathsf{readTailNext}(t,n)) \vee (\mathsf{node}(h,n) * \mathsf{ls}(n,t) * \mathsf{readTail}(t))) \vee (\exists x.\ \mathsf{readHeadEnv}(h,n,x) * \mathsf{ls}(x,t) * \mathsf{readTail}(t)) \vee \mathsf{readHeadNextEnv}(h,n,t)$$

$$\mathsf{readHeadNextEnv}(h,n,t) \stackrel{\text{def}}{=} \exists x,y,z.\ (((h=t) \wedge \mathsf{node}(t,x) \wedge ((x=n) \vee (n=\mathtt{null})) \vee (\mathsf{node}(h,n) * \mathsf{ls}(n,t) * \mathsf{node}(t,x))) * \mathsf{ls}(x,y) * (\mathtt{Head}=y) * \mathsf{ls}(y,z) * (\mathtt{Tail}=z)$$

$$\mathsf{readHeadNextVal}(h,n,v) \stackrel{\text{def}}{=} ((\mathtt{Head}=h) * \mathsf{node}(h,n) * \mathsf{node}(n,v,\_) * (\mathtt{Tail}=n)) \vee (\exists x,t.\ (\mathtt{Head}=h) * \mathsf{node}(h,n) * \mathsf{node}(n,v,x) * \mathsf{ls}(x,t) * (\mathtt{Tail}=t)) \vee (\exists x,t.\ \mathsf{readHeadEnv}(h,n,x) * \mathsf{ls}(x,t) * (\mathtt{Tail}=t))$$

Figure 5.16 Auxiliary definitions in the proofs of MS lock-free queue.

The abstract `ENQ(v)` operation is an atomic command `<Q := Q::v>`. The abstract atomic `DEQ()` returns `EMPTY` if `Q` is empty and takes out the first node otherwise, as shown below.

```
DEQ() { local v;
  < if (Q = ϵ) { v := EMPTY; }
    else { v := head(Q); Q := tail(Q); }
    return v;
  >
}
```

In the definition of I, the predicate lsq requires that the values of the nodes in the concrete list (except the sentinel node) form the sequence `Q`. Recall that `Tail`

```
1 enq(v) {
2  local x, t, s, b;
   {I * (cid ↣ (ENQ, v))}
3  b := false;  x := cons(v, null);
   {((¬b) * I * toEnq) ∨ (b * I * (cid ↣ end))}
4  while (!b) {
     {(¬b) * I * toEnq}
5    < t := Tail; >
     {(¬b) * (I ∧ readTail(t) * true) * toEnq}
6    s := t.next;
     {(¬b) * (I ∧ readTailNext(t, s) * true) * toEnq}
7    if (t = Tail) {
       {(¬b) * (I ∧ readTailNext(t, s) * true) * toEnq}
8      if (s = null) {
         {(¬b) * (I ∧ readTailNextNull(t, s) * true) * toEnq}
9        < b := cas(&(t.next), s, x);  if (b) linself; >
         { (b * (I ∧ readTailNextNonnull(t, x) * true) * (cid ↣ end))
         { ∨ ((¬b) * (I ∧ readTailNextNullEnv(t, s) * true) * toEnq)
10       if (b) {
           {b * (I ∧ readTailNextNonnull(t, x) * true) * (cid ↣ end)}
11         cas(&Tail, t, x);
           {b * I * (cid ↣ end)}
12       }
         {(b * I * (cid ↣ end)) ∨ ((¬b) * I * toEnq)}
13     } else {
         {(¬b) * (I ∧ readTailNextNonnull(t, s) * true) * toEnq}
14       cas(&Tail, t, s);
         {(¬b) * I * toEnq}
15     }
16   }
17 }
   {I * (cid ↣ end)}
18 }
```

Here $\mathsf{toEnq} \stackrel{\text{def}}{=} \mathsf{node}(\mathtt{x}, \mathtt{v}, \mathtt{null}) * (\mathtt{cid} \rightarrowtail (\mathtt{ENQ}, \mathtt{v}))$.

Figure 5.17 Proof outline of `enq` in MS lock-free queue.

points to either the last node, or the second to last node (see the predicate tls used in the definition of lsq). Besides, to formulate the shared resource in the *precise* invariant, we introduce an auxiliary variable `GH` to help collect the "garbage" nodes that have been removed from the list (the predicate garb). The auxiliary variable `GH` is set to `Head` during the initialization of the object and is no longer modified afterwards. Then all the dequeued nodes form a list segment from `GH` to the current `Head`, since the `deq` operation does not update the `next` pointer of a dequeued node.

The rely/guarantee condition R and G specify the related transitions at the

```
1  deq() {
2   local v, s, h, t, b;
    {I * (cid ⤳ DEQ)}
3   b := false;
    {((¬b) * I * (cid ⤳ DEQ)) ∨ (b * I * (cid ⤳ (end, v)))}
4   while (!b) {
      {(¬b) * I * (cid ⤳ DEQ)}
5     < h := Head; >
      {(¬b) * (I ∧ readHead(h) * true) * (cid ⤳ DEQ)}
6     < t := Tail; >
      {(¬b) * (I ∧ readHeadTail(h, t) * true) * (cid ⤳ DEQ)}
7     < s := h.next;  if (h = t && s = null) trylinself; >
      { (¬b) * (I ∧ readHeadNextAfterTail(h, s, t) * true)
        * ((h = t ∧ s = null ∧ (cid ⤳ DEQ ⊕ cid ⤳ (end, EMPTY)))
          ∨ ((h ≠ t ∨ s ≠ null) ∧ (cid ⤳ DEQ))) }
8     if (h = Head) {
9       if (h = t) {
10        if (s = null) {
            {(¬b) * I * (h = t ∧ s = null ∧ (cid ⤳ DEQ ⊕ cid ⤳ (end, EMPTY)))}
11          commit(cid ⤳ (end, EMPTY));
            {(¬b) * I * (cid ⤳ (end, EMPTY))}
12          v := EMPTY;  b := true;
            {b * I * (cid ⤳ (end, v))}
13        } else {
            {(¬b) * (I ∧ readTailNextNonnull(t, s) * true) * (cid ⤳ DEQ)}
14          cas(&Tail, t, s);
            {(¬b) * I * (cid ⤳ DEQ)}
15        }
16      } else {
          {(¬b) * (I ∧ readHeadNextAfterTail(h, s, t) * true) * (cid ⤳ DEQ) ∧ (h ≠ t)}
17        v := s.val;
          {(¬b) * (I ∧ readHeadNextVal(h, s, v) * true) * (cid ⤳ DEQ)}
18        < b := cas(&Head, h, s);  if (b) linself; >
          {((¬b) * I * (cid ⤳ DEQ)) ∨ (b * I * (cid ⤳ (end, v)))}
19      }
20    } else {
        {(¬b) * I * ((cid ⤳ DEQ ⊕ cid ⤳ (end, EMPTY)) ∨ (cid ⤳ DEQ))}
21      commit(cid ⤳ DEQ);
        {(¬b) * I * (cid ⤳ DEQ)}
22    }
23  }
    {I * (cid ⤳ (end, v))}
24  return v;
    {I * (cid ⤳ (end, v))}
25 }
```

Figure 5.18 Proof outline of `deq` in MS lock-free queue.

```
 1 CCAS(o, n) {
 2   local r, d;
 3   d := cons(cid, o, n);
 4   r := cas1(&a, o, d);
 5   while(IsDesc(r)) {
 6     Complete(r);
 7     r := cas1(&a, o, d);
 8   }
 9   if(r = o) Complete(d);
10   return r;
11 }
```

```
12 Complete(d) {
13   local b;
14   b := flag;
15   if (b)
16     cas1(&a, d, d.n);
17   else
18     cas1(&a, d, d.o);
19 }
20 SetFlag(b) {
21   flag := b;
22 }
```

Figure 5.19 CCAS code.

concrete and the abstract levels, which simply include all the actions over the shared states in the algorithm. As defined in Figure 5.15, the actions *Enq* and *Deq* correspond to the LPs at line 8 of the `enq` method and at line 28 of `deq` respectively. They modify both the concrete list and the abstract queue. The *Enq* action transfers the new node from the thread local memory to the shared part, while *Deq* keeps the removed node in the shared memory to allow concurrent accesses. Note *Deq* requires that `Head` and `Tail` should not point to the same node before the action. This ensures `Head` and `Tail` will not cross (e.g., `Head` points to the last node but `Tail` points to the second to last node in the list) after the *Deq* action, which preserves the invariant I. The action *Swing* advances `Tail` when it lags behind the last node of the list. It does not update the abstract queue.

Figures 5.17 and 5.18 show the proof outlines of `enq` and `deq` respectively. The auxiliary assertions used in the proofs are defined in Figure 5.16. The proofs follow the intuition of the algorithm, and are similar to the partial correctness proofs in LRG, but now we need to specify the abstract queue and abstract operations in assertions and reason about instrumented commands.

5.5.3 Conditional CAS

Conditional Compare-And-Swap (CCAS) [65] is a simplified version of the RDCSS algorithm [27]. It involves both the helping mechanism and future-dependent LPs. We show its code in Figure 5.19.

The object contains an integer variable `a` and a boolean bit `flag`. The method `SetFlag` (line 20 in Figure 5.19) sets the bit directly. The method `CCAS` takes two arguments: an expected current value `o` of the variable `a` and a new value `n`. It atomically updates `a` with the new value `n` if `flag` is **true** and `a` indeed has the

value o; and does nothing otherwise. CCAS always returns the (actual) old value of a.

The implementation in Figure 5.19 uses a variant of the cas instruction, which we write as cas1. Instead of a boolean value indicating whether it succeeds, cas1(&a,o,n) returns the old value stored in a. When starting a CCAS, a thread first allocates its descriptor (line 3), which contains the thread ID and the arguments for CCAS. It then tries to put its descriptor in a using cas1 (line 4). If successful, we know a's actual old value r equals the expected value o, so the if-condition at line 9 will succeed. Then the thread calls the auxiliary Complete function, which restores a to the new value n (line 16) or to the original value o (line 18), depending on whether flag is true. If the cas1 instruction at line 4 finds a contains a descriptor (i.e., IsDesc holds), the current thread will try to help complete the operation in the descriptor (line 6) before doing its own. Since we disallow nested function calls to simplify the language, the auxiliary Complete function should be viewed as a macro.

The LPs of the algorithm are at lines 4, 7 and 14. If a contains a different *value* from o at lines 4 and 7, then CCAS fails and they are LPs of the current thread. We can instrument these lines as follows.

```
< r := cas1(&a, o, d);  if(r!=o && !IsDesc(r)) linself; >
```

If the descriptor d gets placed in a, then the LP should be in the Complete function. Since any thread can call Complete to help the operation, the LP should be at line 14 of the thread which will succeed at line 16 or 18. It is a future-dependent LP which may be in other threads' code. We instrument line 14 using **trylin**(d.id) to speculatively execute the abstract operation for the thread in d, which may not be the current thread. That is, line 14 becomes

```
< b := flag;  if (a = d) trylin(d.id); >
```

The condition a=d requires that the abstract operation in the descriptor has not been finished. Then at lines 16 and 18, we commit to the correct guess. We show the instrumentation at line 16 below (where s is a local variable).

```
< s := cas1(&a, d, d.n);
  if(s = d) commit(d.id ↣(end, d.o) * (a_a = d.n)); >
```

That is, when the cas1 instruction succeeds, it should be possible that the thread in d has finished its operation, and the current abstract a (denoted by a_a to be distinguished from the concrete variable a) contains the new value n. Line 18 is instrumented similarly, as shown below.

$$I \stackrel{\text{def}}{=} (\texttt{flag} = \texttt{flag}_a) * (\mathsf{aVal} \vee \mathsf{aDesc}) * \mathsf{garb}$$

$$\mathsf{aVal} \stackrel{\text{def}}{=} (\mathsf{a} = \mathsf{a}_a) \wedge \neg \texttt{IsDesc}(a) \qquad \mathsf{aDesc} \stackrel{\text{def}}{=} \exists d, t, o, n.\ \mathsf{aDesc}(d, t, o, n)$$

$$\mathsf{aDesc}(d, t, o, n) \stackrel{\text{def}}{=} (\mathsf{a} = d) * d \mapsto (t, o, n) * (\mathsf{notDone}(t, o, n) \vee \mathsf{trySucc}(t, o, n) \vee \mathsf{tryFail}(t, o, n) \vee \mathsf{tryBoth}(t, o, n))$$

$$\mathsf{notDone}(t, o, n) \stackrel{\text{def}}{=} t \rightarrowtail (\texttt{CCAS}, o, n) * (\mathsf{a}_a = o)$$

$$\mathsf{endSucc}(t, o, n) \stackrel{\text{def}}{=} t \rightarrowtail (\mathbf{end}, o) * (\mathsf{a}_a = n) \qquad \mathsf{endFail}(t, o) \stackrel{\text{def}}{=} t \rightarrowtail (\mathbf{end}, o) * (\mathsf{a}_a = o)$$

$$\mathsf{trySucc}(t, o, n) \stackrel{\text{def}}{=} \mathsf{notDone}(t, o, n) \oplus \mathsf{endSucc}(t, o, n)$$

$$\mathsf{tryFail}(t, o, n) \stackrel{\text{def}}{=} \mathsf{notDone}(t, o, n) \oplus \mathsf{endFail}(t, o)$$

$$\mathsf{tryBoth}(t, o, n) \stackrel{\text{def}}{=} \mathsf{notDone}(t, o, n) \oplus \mathsf{endSucc}(t, o, n) \oplus \mathsf{endFail}(t, o)$$

Figure 5.20 Invariant for CCAS.

```
< s := cas1(&a, d, d.o);
  if(s = d) commit(d.id ↣(end, d.o) * (a_a = d.o)); >
```

Then we define I, R and G, and verify the code by applying our inference rules. As shown in Figure 5.20, the invariant I describes `flag` and `a` at the concrete and the abstract levels (here the abstract variables take the subscript a to differ from the concrete ones). It requires `flag` at the two levels to be the same, and `a` to be the same when it is a normal value at the concrete side (see the predicate aVal). When the concrete `a` contains thread t's descriptor d (see the predicate aDesc), the abstract `CCAS` operation of t is also shared, which either has not been done (notDone) or is speculatively finished with different results (trySucc, tryFail and tryBoth). We use the predicate garb to remember the garbage thread descriptors and abstract operations, which have been done but are still shared between threads. The detailed definition of garb is omitted here, which is similar to the one in the previous example and can be found in the technical report [43].

The rely R and the guarantee G include the action over the shared states at each line. For instance, the action at the successful `cas1` of lines `4` and `7` in Figure 5.19 is defined as *PlaceD*.

$$\mathit{PlaceD}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists v, d, o, n.\ ((\mathsf{a} = v) \wedge \neg \texttt{IsDesc}(v)) \ltimes ((\mathsf{a} = d) * d \mapsto (\mathsf{t}, o, n) * \mathsf{t} \rightarrowtail (\texttt{CCAS}, o, n))$$

It transfers both the descriptor d and the corresponding abstract operation of the current thread `t` from the thread local memory to the shared memory. This puts the abstract operation in the pending thread pool and enables other threads to help execute it.

The action at line 14 (after instrumentation) guarantees $\mathit{TrylinSucc} \vee \mathit{TrylinFail}$, which demonstrates the use of our logic for both helping and speculation.

$$\mathit{TrylinSucc} \stackrel{\text{def}}{=} \texttt{flag} * (\exists t, o, n.\ \mathsf{notDone}(t, o, n) \propto \mathsf{endSucc}(t, o, n))$$
$$\mathit{TrylinFail} \stackrel{\text{def}}{=} (\neg\texttt{flag}) * (\exists t, o, n.\ \mathsf{notDone}(t, o, n) \propto \mathsf{endFail}(t, o))$$

Here we use $p \propto q$ (defined in Figure 5.8) to describe the action of **trylin**. Before the action of *TrylinSucc*, we must have notDone as one of the speculations. The action adds a new speculation endSucc and also keeps all the original speculations. Note that *TrylinSucc* and *TrylinFail* allow the current thread to help execute the abstract operation of some other thread t.

The actions *RmvDSucc* and *RmvDFail* describe the successful cas1 at lines 16 and 18 respectively. They restore a to values (i.e., remove the descriptor from a) and commit to correct speculations. We define *RmvDSucc* for the action at line 16 below, and *RmvDFail* is defined similarly.

$$\begin{aligned}\mathit{RmvDSucc} \stackrel{\text{def}}{=}\ & \\ & \exists d, t, o, n.\ ((\mathsf{a} = d) * d \mapsto (t, o, n) * (\mathsf{endSucc}(t, o, n) \oplus \mathsf{true}) * \mathsf{garb}) \\ & \ltimes\ ((\mathsf{a} = n) * d \mapsto (t, o, n) * \mathsf{endSucc}(t, o, n) * \mathsf{garb'})\end{aligned}$$

Here garb' informally means, the thread t's descriptor d and the result $t \rightarrowtail (\mathbf{end}, o)$ after executing its abstract operation have been collected by garb.

The detailed proofs are in the technical report [43]. The subtle part in the proof is to ensure that, no thread could cheat by imagining another thread's help. In any program point of CCAS, the environment may have done **trylin** and helped the operation. But whether the environment has helped it or not, the **commit** at line 16 or 18 cannot fail. This means, we should not confuse the two kinds of nondeterminism caused by speculation and by environment interference. The former allows us to discard wrong guesses, while for the latter, we should consider *all* possible environments (including none).

5.6 Summary and Related Work

In this chapter, we propose a new program logic to verify linearizability of algorithms with non-fixed LPs. The logic extends LRG [20] with new rules for the auxiliary commands introduced specifically for linearizability proofs. We also give a relational interpretation of asssertions and rely/guarantee conditions to relate concrete implementations with the corresponding abstract operations. Underlying the logic is a new thread-local simulation, which gives us contextual refinement. Linearizability is derived based on its equivalence to refinement. Both the logic

and the simulation support reasoning about the helping mechanism and future-dependent LPs. As shown in Table 5.1, we have applied the logic to verify various classic algorithms.

There is a large body of work on linearizability verification. However, most existing work supports only simple objects with static LPs in the implementation code (e.g., [3, 14, 64]). Below we mainly discuss the closely related work that can handle non-fixed LPs.

Our logic is similar to Vafeiadis' extension of RGSep to prove linearizability [66]. He also uses abstract objects and abstract atomic operations as auxiliary variables and code. There are two key differences between the logics. First he uses prophecy variables to handle future-dependent LPs, but there has been no satisfactory semantics given for prophecy variables so far. We use the simple try-commit mechanism, whose semantics is straightforward. Second the soundness of his logic with respect to linearizability is not specified and proved. We address this problem by defining a new thread-local simulation as the meta-theory of our logic. As we explained in Section 5.1, defining such a simulation to support non-fixed LPs is one of the most challenging issues we have to solve. Although recently Vafeiadis develops an automatic verification tool [67] with formal soundness for linearizability, his new work can handle non-fixed LPs for *read-only* methods only, and cannot verify algorithms like HSY stack, CCAS and RDCSS in our work.

Recently, Turon et al. [65] propose logical relations to verify fine-grained concurrency, which establish contextual refinement between the library and the specification. Underlying the model a similar simulation is defined. Our pending thread pool is proposed concurrently with their "spec thread pool", while the speculation idea in our simulation is borrowed from their work, which traces back to forward-backward simulation [48]. What is new here is a new program logic and the way we instrument code to do relational reasoning. The set of syntactic rules, including the try-commit mechanism to handle uncertainty, is much easier to use than the semantic logical relations to construct proofs. On the other hand, they support higher-order features, recursive types and polymorphism, while we focus on concurrency reasoning and use only a simple first-order language.

O'Hearn et al. [53] prove linearizability of an optimistic variant of the lazy set algorithm by identifying the "Hindsight" property of the algorithm. Their Hindsight Lemma provides a *non-constructive* evidence for linearizability. Although Hindsight can capture the insights of the set algorithm, it remains an open problem whether the Hindsight-like lemmas exist for other concurrent algorithms.

Colvin et al. [12] formally verify the lazy set algorithm using a combination

of forward and backward simulations between automata. Their simulations are not thread-local, where they need to know the program counters of all threads. Besides, their simulations are specifically constructed for the lazy set only, while ours is more general in that it can be satisfied by various algorithms.

The simulations defined by Derrick et al. [15] are thread-local and general, but they require the operations with non-fixed LPs to be read-only, thus cannot handle the CCAS example. They also propose a backward simulation to verify linearizability [60]. Although the method is proved to be complete, it does not support thread-local verification and there is no program logic given.

Elmas et al. [19] prove linearizability by incrementally *rewriting* the fine-grained code to the atomic operation. They do not need to locate LPs. Their rules are based on left/right movers and program refinements, but not for Hoare-style reasoning as in our work.

There are also lots of model-checking based tools (e.g., [46, 70]) for *checking* linearizability. For example, Vechev et al. [70] check linearizability with user-specified non-fixed LPs. Their method is not thread modular. To handle non-fixed LPs, they need users to instrument the code with enough information about the actions of other threads, which usually demands a priori knowledge about the number of threads running in parallel, as shown in their example. Besides, although their checker can detect un-linearizable code, it will not terminate for linearizable methods in general.

Chapter 6

Observing Progress

Chapter 5 discussed linearizability of concurrent objects. We proved that linearizability is equivalent to a contextual refinement between the concrete object implementations and the abstract operations (Theorem 5.6). This chapter relates *progress* properties of concurrent objects to contextual refinements. We study the five most common progress properties: wait-freedom, lock-freedom and obstruction-freedom for non-blocking implementations, and starvation-freedom and deadlock-freedom for lock-based implementations. We propose a unified framework in which each progress property together with linearizability is equivalent to a certain type of termination-sensitive contextual refinement.

Below we first give an overview of the background and explain our new equivalence results informally in Section 6.1. The five progress properties are formalized in Section 6.2 and the formal framework is presented in Section 6.3.

6.1 Background and Our Results

6.1.1 Contextual Refinement for Linearizability Fails to Preserve Progress

Informally, the object implementation Π is a contextual refinement of abstract operations Π_A, written as $\Pi \sqsubseteq \Pi_A$, if substituting Π for Π_A in any context (i.e., in a client program) does not add observable behaviors. Then external observers cannot tell that Π_A has been replaced by Π from monitoring the behaviors of a client program.

To obtain equivalence to linearizability, we define the observable behaviors in $\Pi \sqsubseteq \Pi_A$ as a prefixed-closed set of finite I/O event traces at both concrete and abstract sides (see Definition 5.5). This basic contextual refinement can be

used to distinguish linearizable objects from non-linearizable ones, but it cannot characterize progress properties of objects. For the following example, $\Pi \sqsubseteq \Pi_A$ holds although no concrete method call of `f` could finish (we assume this object contains a method `f` only).

Π(`f`) : `while(true) skip;` $\quad$ Π_A(`f`) : `skip;`
C : `print(1); f(); print(1);`

For instance, if the above client C uses the object Π, the set of its observable behaviors is $\{\epsilon, (\mathbf{out}, 1) :: \epsilon\}$. When C uses Π_A instead, the observable behavior set becomes $\{\epsilon, (\mathbf{out}, 1) :: \epsilon, (\mathbf{out}, 1) :: (\mathbf{out}, 1) :: \epsilon\}$, which indeed contains all the finite event traces at the concrete side. The key reason is that $\Pi \sqsubseteq \Pi_A$ considers a *prefix-closed* set of event traces at the abstract side.

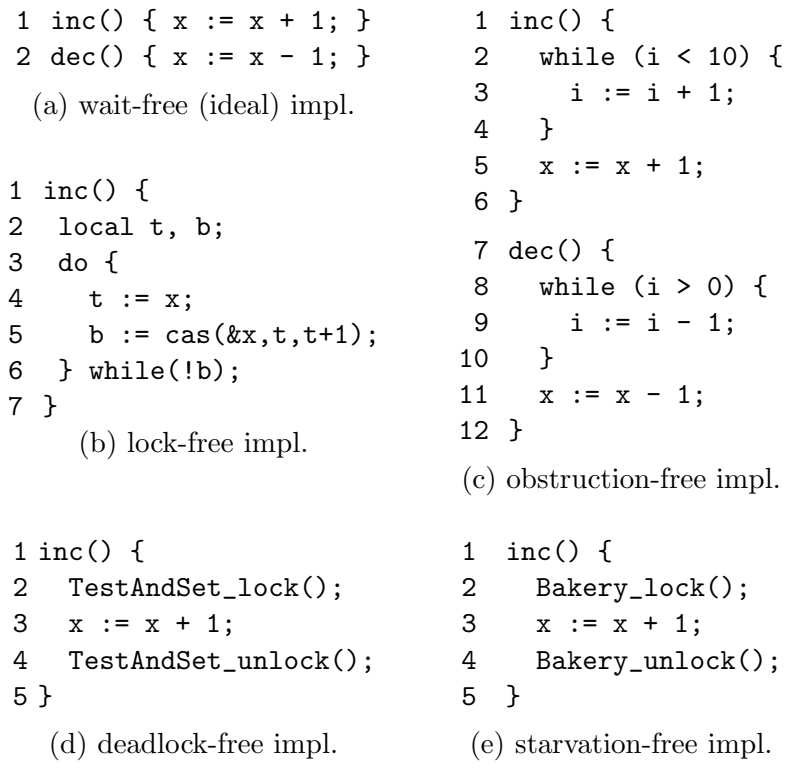
6.1.2 Overview of Progress Properties

Figure 6.1 shows several implementations of a counter with different progress guarantees that we study in this paper. A counter object provides the two methods `inc` and `dec` for incrementing and decrementing a shared variable `x`. The implementations given here are not intended to be practical but merely to demonstrate the meanings of the progress properties.

Informally, an object implementation is *wait-free*, if it guarantees that every thread can complete any started operation of the data structure in a finite number of steps [31]. Figure 6.1(a) shows an ideal wait-free implementation in which the increment and the decrement are done atomically. This implementation is obviously wait-free since it guarantees termination of every method call regardless of interference from other threads. Note that realistic implementations of wait-free counters are more complex and involve arrays and atomic snapshots [4].

Lock-freedom is similar to wait-freedom but only guarantees that *some* thread will complete an operation in a finite number of steps [31]. Typical lock-free implementations (such as Treiber stack [62], HSY elimination-backoff stack [30] and MS lock-free queue [51]) use the `cas` instruction in a loop to repeatedly attempt an update until it succeeds. Figure 6.1(b) shows such an implementation of the counter object. It is lock-free, because whenever `inc` and `dec` operations are executed concurrently, there always exists some successful update. Note that this object is not wait-free. For the following program,

```
inc();   ||   while(true) inc();        (6.1)
```



(a) wait-free (ideal) impl.

(b) lock-free impl.

(c) obstruction-free impl.

(d) deadlock-free impl.

(e) starvation-free impl.

Figure 6.1 Counter objects with methods `inc` and `dec`.

the `cas` instruction in the method called by the left thread may continuously fail due to the continuous updates of `x` made by the right thread.

Herlihy et al. [32] propose *obstruction-freedom* which "guarantees progress for any thread that eventually executes in isolation" (i.e., without other active threads in the system). They present two double-ended queues as examples. In Figure 6.1(c) we show an obstruction-free counter that may look contrived but nevertheless illustrates the idea of the progress property.

The implementation introduces a shared-variable variable `i`, and lets `inc` perform the atomic increment after increasing `i` to 10 and `dec` do the atomic decrement after decreasing `i` to 0. Whenever a method is executed in isolation (i.e., without interference from other threads), it will complete. Thus the object is obstruction-free. It is not lock-free, because for the client

$$\texttt{inc();} \quad \| \quad \texttt{dec();} \tag{6.2}$$

which executes an increment and a decrement concurrently, it is possible that neither of the method calls returns. For instance, under a specific schedule, every increment over `i` made by the left thread is immediately followed by a decrement from the right thread.

Wait-freedom, lock-freedom, and obstruction-freedom are progress properties for non-blocking implementations, where a delay of a thread cannot prevent other threads from making progress. In contrast, deadlock-freedom and starvation-freedom are progress properties for lock-based implementations. A delay of a thread holding a lock will block other threads which request the lock.

Deadlock-freedom and starvation-freedom are often defined in terms of locks and critical sections. Deadlock-freedom guarantees that some thread will succeed in acquiring the lock, and starvation-freedom states that every thread attempting to acquire the lock will eventually succeed [33]. For example, a test-and-set spin lock [33] is deadlock-free but not starvation-free. In a concurrent access, some thread will successfully set the bit and get the lock, but there might be a thread that is continuously failing to get the lock. Lamport's bakery lock [41] is starvation-free. It ensures that threads can acquire locks in the order of their requests.

However, as noted by Herlihy and Shavit [34], the above definitions based on locks are unsatisfactory, because it is often difficult to identify a particular field in the object as a lock. Instead, they suggest defining them in terms of method calls. They also notice that the above definitions implicitly assume that every thread acquiring the lock will eventually release it. This assumption requires *fair* scheduling, i.e., every thread gets eventually executed.

Following Herlihy and Shavit [34], we say an object is *deadlock-free*, if in each *fair* execution there always exists some method call that can finish. As an example in Figure 6.1(d), we use a test-and-set lock to synchronize the increments of the counter. Since some thread is guaranteed to acquire the test-and-set lock, the method call of that thread is guaranteed to finish. Thus the object is deadlock-free. Similarly, a *starvation-free* object guarantees that every method call can finish in fair executions. Figure 6.1(e) shows a counter implemented with Lamport's bakery lock. It is starvation-free because the bakery lock ensures that every thread can acquire the lock and hence every method call can eventually complete.

6.1.3 Our Results

None of the above definitions of the five progress properties describes their guarantees regarding the behaviors of client code. In this chapter, we define several contextual refinements to characterize the effects over client behaviors when the client uses objects with some progress properties. We show that linearizability together with a progress property is equivalent to a certain termination-sensitive contextual refinement. Table 6.1 summarizes our results.

For each progress property, the new contextual refinement $\Pi \sqsubseteq \Pi_A$ is defined

	Wait-free	Lock-free	Obstruction-free	Deadlock-free	Starvation-free
Π_A	(t, Div.)	Div.	Div.	Div.	(t, Div.)
Π	(t, Div.)	Div.	Div. if isolating	Div. if fair	(t, Div.) if fair

Table 6.1 Characterizing progress properties via contextual refinements $\Pi \sqsubseteq \Pi_A$.

with respect to a *divergence* behavior and/or a specific scheduling at the implementation level (the third row in Table 6.1) and at the abstract side (the second row), in addition to the I/O events in the basic contextual refinement for linearizability. Here "divergence" can be roughly viewed as non-termination, which will be explained in detail in Section 6.3.

- For wait-freedom, we need to observe the divergence of each individual thread t, represented by "(t, Div.)" in Table 6.1, at both the concrete and the abstract levels. We show that, if the thread t of a client program diverges when the client uses a linearizable and wait-free object Π, then thread t must also diverge when using Π_A instead.

- The case for lock-freedom is similar, except that we now consider the divergence behaviors of the *whole* client program rather than individual threads (denoted by "Div." in Table 6.1). If a client diverges when using a linearizable and lock-free object Π, it must also diverge when it uses Π_A instead.

- For obstruction-freedom, we consider the behaviors of *isolating* executions at the concrete side (denoted by "Div. if isolating" in Table 6.1). In those executions, eventually only one thread is running. We show that, if a client diverges in an isolating execution when it uses a linearizable and obstruction-free object Π, it must also diverge in some abstract execution.

- For deadlock-freedom, we only care about *fair* executions at the concrete level (denoted by "Div. if fair" in Table 6.1).

- For starvation-freedom, we observe the divergence of each individual thread at both levels and restrict our considerations to fair executions for the concrete side ("(t, Div.) if fair" in Table 6.1). Any thread using Π can diverge in a fair execution, only if it also diverges in some abstract execution.

We will formalize the results and give examples in Section 6.3. These new contextual refinements form a unified framework that characterizes progress properties as well as linearizability. The framework can serve as a new alternative definition for the full correctness properties of concurrent objects. The contextual refinement implied by linearizability and a progress guarantee precisely characterizes

the properties at the abstract level that are preserved by the object implementation. When proving these properties of a client of the object, we can soundly replace the concrete method implementations by its abstract operations. On the other hand, since the contextual refinement also implies linearizability and the progress property, we can potentially borrow ideas from existing proof methods for contextual refinements (e.g., RGSim) to verify linearizability and the progress guarantee together.

6.2 Formalizing Progress Properties

As shown in Figure 6.2, we define progress properties over event traces T as well as over object implementations Π. Different from the finite event traces defined in Figure 5.4, an event trace T in this chapter could be an infinite sequence of events. It is co-inductively defined below. To discuss the progress properties, we also extend the definition of events e in Figure 5.4 with two new events. First, we introduce an auxiliary command **end** which can generate a termination event $(\mathsf{t}, \mathbf{term})$. The new command **end** is a special marker that will be added at the end of a thread, but is not supposed to be used directly by programmers. The second newly introduced event is $(\mathbf{spawn}, n)$, saying that n threads are spawned. Both of the two new events are unobservable. Other technical settings are almost the same as those in Section 5.2, including the program semantics and the linearizability definition.

$$
\begin{array}{rrcl}
(\mathit{Stmt}) & C & ::= & \ldots \mid \mathbf{end} \\
(\mathit{Evt}) & e & ::= & \ldots \mid (\mathsf{t}, \mathbf{term}) \mid (\mathbf{spawn}, n) \\
(\mathit{ETrace}) & T & ::= & \epsilon \mid e :: T \qquad \text{(co-inductive)}
\end{array}
$$

We follow the notations used in Chapter 5. For instance, $\mathsf{tid}(e)$ is still for the thread ID in the event e. Predicates $\mathsf{is\_inv}(e)$, $\mathsf{is\_ret}(e)$ and $\mathsf{is\_abt}(e)$ still say that e is a method invocation, a return and a fault, respectively. $\mathsf{match}(e_1, e_2)$ still requires that the invocation e_1 and the response e_2 (i.e., a return or an object fault) have the same thread ID. We still use $T(i)$ for the i-th event of T. Besides, we write $\mathsf{last}(T)$ for the last event when T is finite. The trace $T(1..i)$ is the sub-trace $T(1), \ldots, T(i)$ of T, and $|T|$ still represents the length of T ($|T| = \omega$ if T is infinite). The trace $T|_{\mathsf{t}}$ is still for the sub-trace of T consisting of all events whose thread ID is t.

We say an object implementation Π has a progress property P iff all its event traces have the property (Definition 6.1). Here we use $\mathcal{T}_\omega$ to generate the *complete*

Definition 6.1. An object Π satisfies a progress property P under a refinement mapping φ, written as $P_\varphi(\Pi)$, iff

$$\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, T.\ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o)]\!] \wedge (\sigma_o \in dom(\varphi)) \implies P(T)\,.$$

$$\mathcal{T}_\omega[\![W, (\sigma_c, \sigma_o)]\!] \stackrel{\text{def}}{=} \{(\mathbf{spawn}, |W|)::T \mid (\lfloor W \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \stackrel{T}{\longmapsto}{}^{\omega} \cdot \vee (\lfloor W \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \stackrel{T}{\longmapsto}{}^{*} (\mathbf{skip}, \_) \vee (\lfloor W \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \stackrel{T}{\longmapsto}{}^{*} \mathbf{abort}\}$$

$$\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor \stackrel{\text{def}}{=} \mathbf{let}\ \Pi\ \mathbf{in}\ (C_1; \mathbf{end}) \,\|\, \ldots \,\|\, (C_n; \mathbf{end})$$

$$|\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n| \stackrel{\text{def}}{=} n \qquad \mathsf{tnum}((\mathbf{spawn}, n)::T) \stackrel{\text{def}}{=} n$$

$$\mathsf{pend\_inv}(T) \stackrel{\text{def}}{=} \{e \mid \exists i.\ e = T(i) \wedge \mathsf{is\_inv}(e) \wedge \neg\exists j.\ (j > i \wedge \mathsf{match}(e, T(j)))\}$$

$$\mathsf{prog\text{-}t}(T) \quad \text{iff} \quad \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{match}(e, T(j))$$

$$\mathsf{prog\text{-}s}(T) \quad \text{iff} \quad \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{is\_ret}(T(j))$$

$$\mathsf{abt}(T) \quad \text{iff} \quad \exists i.\ \mathsf{is\_abt}(T(i))$$

$$\mathsf{sched}(T) \quad \text{iff} \quad |T| = \omega \wedge \mathsf{pend\_inv}(T) \neq \emptyset \implies \exists e.\ e \in \mathsf{pend\_inv}(T) \wedge |(T|_{\mathsf{tid}(e)})| = \omega$$

$$\mathsf{fair}(T) \quad \text{iff} \quad |T| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ |(T|_{\mathsf{t}})| = \omega \vee \mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \mathbf{term})$$

$$\mathsf{iso}(T) \quad \text{iff} \quad |T| = \omega \implies \exists \mathsf{t}, i.\ (\forall j.\ j \geq i \implies \mathsf{tid}(T(j)) = \mathsf{t})$$

$\mathsf{wait\text{-}free}$ iff $\mathsf{sched} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}$ $\qquad$ $\mathsf{starvation\text{-}free}$ iff $\mathsf{fair} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}$

$\mathsf{lock\text{-}free}$ iff $\mathsf{sched} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt}$ $\qquad$ $\mathsf{deadlock\text{-}free}$ iff $\mathsf{fair} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt}$

$\mathsf{obstruction\text{-}free}$ iff $\mathsf{sched} \wedge \mathsf{iso} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}$

Figure 6.2 Formalizing progress properties.

event traces of an object. Its definition in Figure 6.2 is similar to $\mathcal{T}[\![W, (\sigma_c, \sigma_o)]\!]$ of Figure 5.6, but $\mathcal{T}_\omega[\![W, (\sigma_c, \sigma_o)]\!]$ is for the set of finite or infinite event traces produced by complete executions. We use $(W, \mathcal{S}) \stackrel{T}{\longmapsto}{}^{\omega} \cdot$ to denote the existence of a T-labelled infinite execution. By using $\lfloor W \rfloor$, we append **end** at the end of each thread to explicitly mark the termination of the thread. We also insert the spawning event $(\mathbf{spawn}, n)$ at the beginning of T, where n is the number of threads in W. We assume the threads take continuous positive numbers as IDs, thus n should also be the greatest thread ID in W. Later we can use $\mathsf{tnum}(T)$ to get this number n, which is needed to define fairness, as shown below.

Before formulating each progress property over event traces, we first define some auxiliary properties in Figure 6.2. We use $\mathsf{pend\_inv}(T)$ to get the set of

$$\mathsf{lock\text{-}free} \iff \mathsf{wait\text{-}free} \vee \mathsf{prog\text{-}s} \qquad \mathsf{starvation\text{-}free} \iff \mathsf{wait\text{-}free} \vee \neg\mathsf{fair}$$
$$\mathsf{obstruction\text{-}free} \iff \mathsf{lock\text{-}free} \vee \neg\mathsf{iso} \qquad \mathsf{deadlock\text{-}free} \iff \mathsf{lock\text{-}free} \vee \neg\mathsf{fair}$$

Figure 6.3 Relationships between progress properties.

pending invocations in T. Recall that an invocation is pending if no matching return follows it. The predicate $\mathsf{prog\text{-}t}(T)$ guarantees that every method call in T eventually finishes. It says, even if the invocation e is pending in the subtrace $T(1..i)$, we must be able to find a matching response $T(j)$ later. $\mathsf{prog\text{-}s}(T)$ guarantees that *some* pending method call finishes. Different from prog-t, the return event $T(j)$ in prog-s does not have to be a matching return of the pending invocation e. As an example, in the following infinite event trace T_1,

$$T_1 : \quad (\mathsf{t}_1, f, 1) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_2, f, 2) :: (\mathsf{t}_2, \mathbf{ret}, 2) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_2, f, 2) :: (\mathsf{t}_2, \mathbf{ret}, 2) :: \ldots$$

the thread t_2 keeps producing invocations and returns while the method call from thread t_1 never finishes. Thus it satisfies prog-s but not prog-t. Besides, $\mathsf{abt}(T)$ says that T ends with a fault event.

There are three useful conditions on scheduling. The basic requirement for a good schedule is sched. If T is infinite and there exist pending calls, then at least one pending thread should be scheduled infinitely often. In fact, there are two possible reasons causing a method call of thread t to pend. Either t is no longer scheduled, or it is always scheduled but the method call never finishes. sched rules out the bad schedule where no thread with an invoked method is active. For instance, the following infinite trace T_2 does *not* satisfy sched, while T_3 and the above T_1 satisfies it.

$$T_2 : \quad (\mathsf{t}_1, f_1, n_1) :: (\mathsf{t}_2, f_2, n_2) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_3, \mathbf{clt}) :: (\mathsf{t}_3, \mathbf{clt}) :: (\mathsf{t}_3, \mathbf{clt}) :: \ldots$$
$$T_3 : \quad (\mathsf{t}_1, f_1, n_1) :: (\mathsf{t}_2, f_2, n_2) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_2, \mathbf{obj}) :: (\mathsf{t}_2, \mathbf{obj}) :: (\mathsf{t}_2, \mathbf{obj}) :: \ldots$$

Besides, if T is infinite, $\mathsf{fair}(T)$ requires every non-terminating thread be scheduled infinitely often. Note here we use $\mathsf{tnum}(T)$ to get the total number of threads of the *client*, which may not equal the number of threads mentioned in T. Then the definition of fair can rule out the case where some thread is never scheduled. We can see that a fair schedule is a good schedule satisfying sched. The isolating schedule $\mathsf{iso}(T)$ requires eventually only one thread be scheduled. For instance, the above traces T_2 and T_3 both satisfy iso.

At the bottom of Figure 6.2 we define the progress properties over event traces formally. We omit the parameter T in the formulas to simplify the presentation.

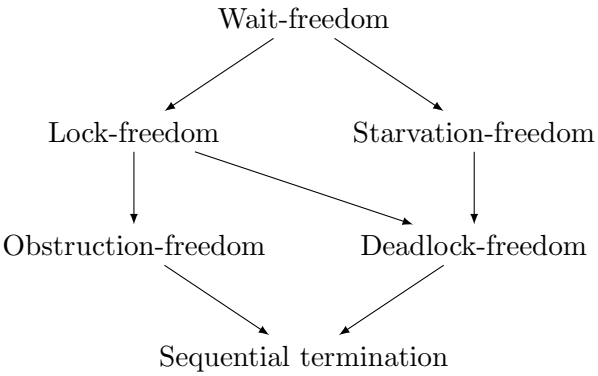


Figure 6.4 Relationship lattice of progress properties.

An event trace T is wait-free (i.e., wait-free(T) holds) if under the good schedule sched, it guarantees prog-t unless it ends with a fault. lock-free(T) is similar except that it guarantees prog-s. Starvation-freedom and deadlock-freedom guarantee prog-t and prog-s under fair scheduling. Obstruction-freedom guarantees prog-t if some pending thread is always scheduled (sched) and runs in isolation (iso). For the above examples, the event trace T_1 satisfies lock-free but not wait-free. T_2 satisfies all the five progress properties since it does not satisfy sched. Though T_3 satisfies both sched and iso, it does not ensure obstruction-free since prog-t is not guaranteed.

Figure 6.3 contains lemmas that relate progress properties. For instance, an event trace is starvation-free, iff it is wait-free or not fair. These lemmas give us the relationship lattice in Figure 6.4, where the arrows represent implications. For example, wait-freedom implies lock-freedom and starvation-freedom implies deadlock-freedom. To close the lattice, we also define a progress property in the sequential setting, which we call *sequential termination*.

Definition 6.2 (Sequential Termination)**.** $\mathsf{seq\text{-}term}_{\varphi}(\Pi)$ iff

$$\forall C_1, \sigma_c, \sigma_o, T.\ T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1), (\sigma_c, \sigma_o)]\!] \wedge (\sigma_o \in \mathit{dom}(\varphi)) \implies \mathsf{starvation\text{-}free}(T)\,.$$

It guarantees that every method call must finish in a trace produced by a sequential client. It is implied by each of the five progress properties for concurrent objects.

6.3 Equivalence to Contextual Refinements

We extend the basic contextual refinement in Definition 5.5 to observe progress as well as linearizability. For each progress property, we carefully choose the

$$\mathsf{div\_tids}(T) \stackrel{\text{def}}{=} \{\mathsf{t} \mid (\,|(T|_{\mathsf{t}})| = \omega\,)\}$$
$$\mathcal{O}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!]\}$$
$$\mathcal{O}_{i\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!] \wedge \mathsf{iso}(T)\}$$
$$\mathcal{O}_{f\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!] \wedge \mathsf{fair}(T)\}$$
$$\mathcal{O}_{t\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T),\mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!]\}$$
$$\mathcal{O}_{ft\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T),\mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!] \wedge \mathsf{fair}(T)\}$$

Figure 6.5 Generation of complete event traces.

observable behaviors at the concrete and the abstract levels.

6.3.1 Observable Behaviors

In Figure 6.5, we define various observable behaviors for the termination-sensitive contextual refinements.

We use $\mathcal{O}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!]$ to represent the set of observable event traces produced by *complete* executions. Recall that $\mathsf{get\_obsv}(T)$ gets the sub-trace of T consisting of all the observable events (i.e., outputs, client faults and object faults) only. Unlike the prefix-closed set $\mathcal{O}[\![W,(\sigma_c,\sigma_o)]\!]$ (defined in Figure 5.6), this definition utilizes $\mathcal{T}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!]$ (see Figure 6.2) whose event traces are all complete and could be infinite. Thus it allows us to observe divergence of the whole program. $\mathcal{O}_{i\omega}$ and $\mathcal{O}_{f\omega}$ take the complete observable traces of isolating and fair executions respectively. Here $\mathsf{iso}(T)$ and $\mathsf{fair}(T)$ have been defined in Figure 6.2.

We could also observe divergence of individual threads. We define $\mathsf{div\_tids}(T)$ to collect the set of threads that diverge in the trace T. Then we write $\mathcal{O}_{t\omega}[\![W,\sigma]\!]$ to get both the observable behaviors and the diverging threads in the complete executions. $\mathcal{O}_{ft\omega}[\![W,\sigma]\!]$ is defined similarly but considers fair executions only.

More on divergence. In general, divergence means non-termination. For example, we could say that the following two-threaded program (6.3) must diverge since it never terminates.

```
x := x + 1;    ||    while(true) skip;          (6.3)
```

But for individual threads, divergence is not equivalent to non-termination. A non-terminating thread may either have an infinite execution or simply be not scheduled from some point due to unfair scheduling. We view only the former

P	wait-free	lock-free	obstruction-free	deadlock-free	starvation-free
$\Pi \sqsubseteq_{\varphi}^{P} \Pi_A$	$\mathcal{O}_{t\omega} \subseteq \mathcal{O}_{t\omega}$	$\mathcal{O}_{\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{i\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{f\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{ft\omega} \subseteq \mathcal{O}_{t\omega}$

Table 6.2 Contextual refinements $\Pi \sqsubseteq_{\varphi}^{P} \Pi_A$ for progress properties P.

case as divergence. For instance, in an unfair execution, the left thread of (6.3) may never be scheduled and hence it has no chance to terminate. It does not diverge. Similarly, for the following program (6.4),

```
while(true) skip;    ||    while(true) skip;          (6.4)
```

the whole program must diverge, but it is possible that a single thread does not diverge in an execution.

6.3.2 New Contextual Refinements and Equivalence Results

In Table 6.2, we summarize the definitions of the termination-sensitive contextual refinements. Each new contextual refinement follows the basic one in Definition 5.5 but takes different observable behaviors as specified in Table 6.2. For example, the contextual refinement for wait-freedom is formally defined as follows.

$\Pi \sqsubseteq_{\varphi}^{\text{wait-free}} \Pi_A$ iff
$\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \sigma_a.\ (\varphi(\sigma_o) = \sigma_a) \Longrightarrow$
$\mathcal{O}_{t\omega}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o))]\!] \subseteq \mathcal{O}_{t\omega}[\![(\textbf{let } \Pi_A \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a)]\!]$.

Theorem 6.3 says that linearizability with a progress property P together is equivalent to the corresponding contextual refinement $\sqsubseteq_{\varphi}^{P}$.

Theorem 6.3 (Equivalence)**.** $\Pi \preceq_{\varphi} \Pi_A \wedge P_{\varphi}(\Pi) \iff \Pi \sqsubseteq_{\varphi}^{P} \Pi_A$*, where* P *is* wait-free*,* lock-free*,* obstruction-free*,* deadlock-free *or* starvation-free*.*

Here we assume the object specification Π_A is *total*, i.e., the abstract operations never block. We sketch the proofs of Theorem 6.3 in Appendix B.

The contextual refinement for wait-freedom takes $\mathcal{O}_{t\omega}$ at both the concrete and the abstract levels. The divergence of individual threads as well as I/O events are treated as observable behaviors. The intuition of the equivalence is as follows. Since a wait-free object Π guarantees that every method call finishes, we have to blame the client code itself for the divergence of a thread using Π. That is, even if the thread uses the abstract object Π_A, it must still diverge.

As an example, consider the client program (6.1). Intuitively, for any execution in which the client uses the abstract operations, only the right thread $\mathtt{t}_2$ diverges.

Thus $\mathcal{O}_{t\omega}$ of the abstract program is a singleton set $\{(\epsilon, \{\mathsf{t}_2\})\}$. When the client uses the wait-free object in Figure 6.1(a), its $\mathcal{O}_{t\omega}$ set is still $\{(\epsilon, \{\mathsf{t}_2\})\}$. It does not produce more observable behaviors. But if it uses a non-wait-free object (such as the one in Figure 6.1(b)), the left thread t_1 does not necessarily finish. The $\mathcal{O}_{t\omega}$ set becomes $\{(\epsilon, \{\mathsf{t}_2\}), (\epsilon, \{\mathsf{t}_1, \mathsf{t}_2\})\}$. It produces more observable behaviors than the abstract client, breaking the contextual refinement. Thanks to observing div_tids that collects the diverging threads, we can rule out non-wait-free objects which may cause more threads to diverge.

$\Pi \sqsubseteq_{\varphi}^{\text{lock-free}} \Pi_A$ takes coarser observable behaviors. We observe the divergence of the whole client program by using $\mathcal{O}_{\omega}$ at both the concrete and the abstract levels. Intuitively, a lock-free object Π ensures that some method call will finish, thus the client using Π diverges only if there are an infinite number of method calls. Then it must also diverge when using the abstract object Π_A.

For example, consider the client (6.1). The whole client program diverges in every execution both when it uses the lock-free object in Figure 6.1(b) and when it uses the abstract one. The $\mathcal{O}_{\omega}$ set of observable behaviors is $\{\epsilon\}$ at both levels. On the other hand, the following client (6.5) must terminate and print out both 1 and 2 in every execution. The $\mathcal{O}_{\omega}$ set is $\{1 :: 2 :: \epsilon, 2 :: 1 :: \epsilon\}$ at both levels.

```
inc(); print(1);   ||   dec(); print(2);          (6.5)
```

Instead, if the client (6.5) uses the non-lock-free object in Figure 6.1(c), it may diverge and nothing is printed out. The $\mathcal{O}_{\omega}$ set becomes $\{\epsilon, 1 :: 2 :: \epsilon, 2 :: 1 :: \epsilon\}$, which contains more behaviors than the abstract side. Thus $\Pi \sqsubseteq_{\varphi}^{\text{lock-free}} \Pi_A$ fails.

Obstruction-freedom ensures progress for isolating executions in which eventually only one thread is running. Correspondingly, $\Pi \sqsubseteq_{\varphi}^{\text{obstruction-free}} \Pi_A$ restricts our considerations to isolating executions. It takes $\mathcal{O}_{i\omega}$ at the concrete level and $\mathcal{O}_{\omega}$ at the abstract level.

To understand the equivalence, consider the client (6.5) again. For isolating executions with the obstruction-free object in Figure 6.1(c), it *must* terminate and print out both 1 and 2. The $\mathcal{O}_{i\omega}$ set at the concrete level is $\{1 :: 2 :: \epsilon, 2 :: 1 :: \epsilon\}$, the same as the set $\mathcal{O}_{\omega}$ of the abstract side. Non-obstruction-free objects in general do not guarantee progress for some isolating executions. If the client uses the object in Figure 6.1(d) or (e), the $\mathcal{O}_{i\omega}$ set is $\{\epsilon, 1 :: 2 :: \epsilon, 2 :: 1 :: \epsilon\}$, not a subset of the abstract $\mathcal{O}_{\omega}$ set. The undesired empty observable trace is produced by unfair executions, where a thread acquires the lock and gets suspended and then the other thread would keep requesting the lock forever (it is executed in isolation).

$\Pi \sqsubseteq_{\varphi}^{\text{deadlock-free}} \Pi_A$ uses $\mathcal{O}_{f\omega}$ at the concrete side, ruling out undesired divergence caused by unfair scheduling. For the client (6.5) with the object in Figure 6.1(d) or (e), its $\mathcal{O}_{f\omega}$ set is same as the set $\mathcal{O}_{\omega}$ at the abstract level.

For $\Pi \sqsubseteq_{\varphi}^{\text{starvation-free}} \Pi_A$, we still consider only fair executions at the concrete level (similar to deadlock-freedom), but observe the divergence of individual threads rather than the whole program (similar to wait-freedom). It uses $\mathcal{O}_{ft\omega}$ at the concrete side and $\mathcal{O}_{t\omega}$ at the abstract level. For the client (6.5) with the starvation-free object in Figure 6.1(e), no thread diverges in any fair execution. Then the set $\mathcal{O}_{ft\omega}$ of concrete observable behaviors is $\{(1 :: 2 :: \epsilon, \emptyset), (2 :: 1 :: \epsilon, \emptyset)\}$, which is same as the set $\mathcal{O}_{t\omega}$ at the abstract level.

Observing threaded divergence is the key to distinguishing starvation-free objects from deadlock-free objects. Consider the client (6.1). Under fair scheduling, we know only the right thread t_2 would diverge when using the starvation-free object in Figure 6.1(e). The set $\mathcal{O}_{ft\omega}$ is $\{(\epsilon, \{\mathsf{t}_2\})\}$. It coincides with the abstract behaviors $\mathcal{O}_{t\omega}$. But when using the deadlock-free object of Figure 6.1(d), the $\mathcal{O}_{ft\omega}$ set becomes $\{(\epsilon, \{\mathsf{t}_2\}), (\epsilon, \{\mathsf{t}_1, \mathsf{t}_2\})\}$, breaking the contextual refinement.

6.4 Summary and Related Work

This chapter introduced a contextual refinement framework to unify various progress properties. For linearizable objects, each progress property is equivalent to a specific termination-sensitive contextual refinement, as summarized in Table 6.1. The framework allows us to verify safety and liveness properties of client programs at a high abstraction level by replacing concrete method implementations with abstract operations. It also makes it possible to borrow ideas from existing proof methods for contextual refinements to verify linearizability and a progress property together, which we leave as future work.

There is a large body of work discussing the five progress properties and the contextual refinements individually. Our work in contrast studies their relationships, which have not been considered much before.

Gotsman and Yang [24] propose a new linearizability definition that preserves lock-freedom, and suggest a connection between lock-freedom and a termination-sensitive contextual refinement. We do not redefine linearizability here. Instead, we propose a unified framework to systematically relate all the five progress properties plus linearizability to various contextual refinements.

Herlihy and Shavit [34] informally discuss all the five progress properties. Our definitions in Section 6.2 mostly follow their explanations, but they are more

formal and close the gap between program semantics and their history-based interpretations. We also notice that their obstruction-freedom is inappropriate for some examples (see Appendix A), and propose a different definition that is closer to the common intuition [33]. In addition, we relate the progress properties to contextual refinements, which consider the extensional effects on client behaviors.

Fossati et al. [22] propose a uniform approach in the π-calculus to formulate both the standard progress properties and their observational approximations. Their technical setting is completely different from ours. Also, their observational approximations for lock-freedom and wait-freedom are strictly weaker than the standard notions. Their deadlock-freedom and starvation-freedom are not formulated, and there is no observational approximation given for obstruction-freedom. In comparison, our framework relates each of the five progress properties (plus linearizablity) to an *equivalent* contextual refinement.

There are also formulations of progress properties based on temporal logics. For example, Petrank et al. [57] formalize the three non-blocking properties and Dongol [18] formalize all the five progress properties, using linear temporal logics. Those formulations make it easier to do model checking (e.g., Petrank et al. [57] also build a tool to model check a variant of lock-freedom), while our contextual refinement framework is potentially helpful for modular Hoare-style verification.

Chapter 7

Conclusions and Future Work

Many verification problems can be reduced to verifying refinement. This dissertation studies proof techniques for refinement and its applications in a concurrent setting. It has covered four main topics.

- A novel simulation, RGSim, as a general, application-independent and compositional proof method for concurrent program refinement.
- A verification framework, based on RGSim, for proving the correctness of concurrent garbage collectors.
- A program logic, for verifying linearizability of concurrent objects with non-fixed linearization points.
- A unified framework, that characterizes progress properties of concurrent objects via contextual refinements.

RGSim parameterizes the simulation between concurrent programs with the interference from their parallel environments. It is compositional with respect to parallel compositions, allowing us to decompose refinement proofs for multi-threaded programs into proofs for individual threads.

RGSim can incorporate the assumptions about environments made by specific refinement applications, so it is flexible and practical. It makes relational reasoning about optimizations possible in parallel contexts. We present a set of relational reasoning rules to characterize and justify common optimizations in a concurrent setting, including hoisting loop invariants, strength reduction and induction variable elimination, dead code elimination, redundancy introduction.

We also reduce the problem of verifying concurrent garbage collectors to verifying transformations, and present a general GC verification framework based

on RGSim. We have verified the Boehm et al. concurrent garbage collection algorithm [9] using our framework.

Besides, RGSim gives us a refinement-based proof method to verify fine-grained implementations of abstract algorithms and concurrent objects. It can ensure a contextual refinement that is equivalent to linearizability of concurrent objects. However, RGSim does not support objects with non-fixed LPs.

We propose the first program logic that has a formal soundness proof for linearizability with non-fixed LPs. Our logic is built upon the unary program logic LRG [20], but we give a relational interpretation of assertions and rely/guarantee conditions. We also introduce new logic rules for auxiliary commands used specifically for linearizability proofs. We introduce the pending thread pool to support the helping mechanism, and the try-commit clauses for future-dependent LPs. We successfully apply our logic to verify 12 well-known algorithms (see Table 5.1). Some of them are used in the `java.util.concurrent` package, such as MS non-blocking queue [51] and Harris-Michael lock-free list [26, 50].

We also design a new simulation as the meta-theory for our logic. It generalizes RGSim with the support for non-fixed LPs. It is still compositional and ensures a contextual refinement, which is equivalent to linearizability. A program logic for contextual refinement is interesting in its own right, since contextual refinement is also a widely accepted (and probably more natural) correctness criterion for library code.

Finally, we relate progress properties of concurrent objects to contextual refinements. We formalize the definitions of the five most common progress properties: wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, and deadlock-freedom. Their relationships form a lattice shown in Figure 6.4. We develop a unified framework to characterize progress properties via contextual refinements. With linearizability, each progress property is proved equivalent to a contextual refinement which takes into account divergence of programs.

Our contextual refinement framework can serve as a new alternative definition for the *full* correctness properties of a concurrent object. It enables us to modularly verify a client of the object by replacing the concrete method implementations with the abstract operations. Also, it becomes possible to extend existing proof methods for contextual refinements (such as RGSim) to verify linearizability and the progress guarantee together.

Future work.

- *Verifying termination-preserving refinement.* Most existing proof techniques

for concurrent program refinement (including ours) do not reason about the preservation of termination, allowing a diverging program to trivially refine any programs. We would like to design a new simulation *and* a new Hoare-style program logic, both of which support compositional verification of termination-preserving refinement of concurrent programs.

- *A verification framework for linearizability and progress properties.* The contextual refinement framework in this dissertation enables us to use each termination-sensitive contextual refinement to verify linearizability and the corresponding progress property together. This dissertation also presents a program logic for a termination *insensitive* contextual refinement that ensures linearizability alone. Can we extend the logic to verify each termination-sensitive contextual refinement? Furthermore, can we have a *unified* program logic (perhaps with a few parameters and plug-in rules) that supports all those contextual refinements?

- *More applications.* We would like to verify implementations of software transactional memory, operating system kernels, practical compilers for concurrent programs and other real-world refinement applications.

- *Tool support.* We would like to develop specialized tools that take two programs as the input and prove the refinement between them automatically or semi-automatically. We also hope to mechanize the logic for linearizability in Coq and then build tools to automate the verification process.

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, May 1991.

[2] M. Abadi and G. Plotkin. A model of cooperative threads. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 29–40, New York, NY, USA, 2009. ACM Press.

[3] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, pages 477–490, Berlin, Heidelberg, 2007. Springer.

[4] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous pram model. In *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'90)*, pages 340–349, New York, NY, USA, 1990. ACM.

[5] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst.*, 27(6):1097–1146, Nov. 2005.

[6] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 14–25, New York, NY, USA, 2004. ACM Press.

[7] H.-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pages 261–268, New York, NY, USA, 2005. ACM Press.

[8] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 68–78, New York, NY, USA, 2008. ACM Press.

[9] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 157–164, New York, NY, USA, 1991. ACM Press.

[10] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

[11] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC'10/ETAPS'10)*, pages 104–123, Berlin, Heidelberg, 2010. Springer.

[12] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV*, pages 475–488. Springer, 2006.

[13] Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.3, 2010. http://coq.inria.fr.

[14] J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4:1–4:43, Jan. 2011.

[15] J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisability with potential linearisation points. In *FM*, pages 323–337. Springer, 2011.

[16] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing (DISC'06)*, pages 194–208, Berlin, Heidelberg, 2006. Springer.

[17] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE'04*, pages 97–114. Springer, 2004.

[18] B. Dongol. Formalising progress properties of non-blocking programs. In *ICFEM*, pages 284–303, 2006.

[19] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, pages 296–311, Berlin, Heidelberg, 2010. Springer.

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

[21] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, Dec. 2010.

[22] L. Fossati, K. Honda, and N. Yoshida. Intensional and extensional characterisation of global progress in the π-calculus. In *CONCUR*, pages 287–301. Springer, 2012.

[23] D. S. Gladstein and M. Wand. Compiler correctness for concurrent languages. In *Proceedings of the 1st International Conference on Coordination Languages and Models (COORDINATION'96)*, pages 231–248. Springer, April 1996.

[24] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP'11)*, volume 6756 of *Lecture Notes in Computer Science*, pages 453–465. Springer, 2011.

[25] A. Gotsman and H. Yang. Linearizability with ownership transfer. In *CONCUR*, pages 256–271. Springer, 2012.

[26] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC'01)*, pages 300–314, London, UK, UK, 2001. Springer.

[27] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing (DISC'02)*, pages 265–279, London, UK, UK, 2002. Springer.

[28] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP'86)*, pages 187–196. Springer, 1986.

[29] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS'05)*, pages 3–16, Berlin, Heidelberg, 2006. Springer.

[30] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'04)*, pages 206–215, New York, NY, USA, 2004. ACM Press.

[31] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[32] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference*

on Distributed Computing Systems (ICDCS'03), pages 522–529, Washington, DC, USA, 2003. IEEE Computer Society.

[33] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, April 2008.

[34] M. Herlihy and N. Shavit. On the nature of progress. In *Proceedings of 15th International Conference on Principles of Distributed Systems (OPODIS'11)*, pages 313–328. Springer, 2011.

[35] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[36] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1(4):271–281, 1972.

[37] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, pages 133–146, New York, NY, USA, 2011. ACM Press.

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

[39] K. Kapoor, K. Lodaya, and U. Reddy. Fine-grained concurrency with separation logic. *J. Philosophical Logic*, 40(5):583–632, 2011.

[40] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, June 2010.

[41] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, Aug. 1974.

[42] X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, Dec. 2009.

[43] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. Technical report, University of Science and Technology of China, March 2013. http://kyhcs.ustcsz.edu.cn/relconcur/lin.

[44] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. Technical report (with Coq implementation), University of Science and Technology of China, October 2011. http://kyhcs.ustcsz.edu.cn/relconcur/rgsim.

[45] H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. Technical report, University of Science and Technology of China, April 2013. http://kyhcs.ustcsz.edu.cn/relconcur/prog.

[46] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *FM*, pages 321–337. Springer, 2009.

[47] A. Lochbihler. Verifying a compiler for java threads. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP'10)*, pages 427–447, Berlin, Heidelberg, 2010. Springer.

[48] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[49] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, pages 468–479, New York, NY, USA, 2007. ACM Press.

[50] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, pages 73–82, New York, NY, USA, 2002. ACM Press.

[51] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 267–275, New York, NY, USA, 1996. ACM Press.

[52] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, Apr. 2007.

[53] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC*, pages 85–94. ACM Press, 2010.

[54] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 268–280, New York, NY, USA, 2004. ACM Press.

[55] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 137–146, Washington, DC, USA, 2006. IEEE Computer Society.

[56] D. Pavlovic, P. Pepper, and D. R. Smith. Formal derivation of concurrent garbage collectors. In *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC'10)*, pages 353–376, 2010.

[57] E. Petrank, M. Musuvathi, and B. Steesngaard. Progress guarantee for parallel programs via bounded lock-freedom. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, pages 144–154, New York, NY, USA, 2009. ACM Press.

[58] S. Qadeer, A. Sezgin, and S. Tasiran. Back and forth: Prophecy variables for static verification of concurrent programs. Technical Report MSR-TR-2009-142, Microsoft, October 2009.

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

[60] G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In *CAV*, pages 243–259. Springer, 2012.

[61] J. Ševčík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, pages 43–54, New York, NY, USA, 2011. ACM Press.

[62] R. K. Treiber. System programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[63] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 377–390, New York, NY, USA, 2013. ACM Press.

[64] A. Turon and M. Wand. A separation logic for refining concurrent objects. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, pages 247–258, New York, NY, USA, 2011. ACM Press.

[65] A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*, pages 343–356, New York, NY, USA, 2013. ACM Press.

[66] V. Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[67] V. Vafeiadis. Automatically proving linearizability. In *CAV*, pages 450–464. Springer, 2010.

[68] V. Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, pages 335–351. Elsevier Science Publishers Ltd., 2011.

[69] M. T. Vechev, E. Yahav, and D. F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 341–353, New York, NY, USA, 2006. ACM Press.

[70] M. T. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *SPIN*, pages 261–278. Springer, 2009.

[71] M. Wand. Compiler correctness for parallel languages. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 120–134, New York, NY, USA, 1995. ACM Press.

[72] H. Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3):308–334, Apr. 2007.

Appendix A

Comparisons with Herlihy and Shavit's Obstruction-Freedom

Herlihy and Shavit [34] define obstruction-freedom using the notion of *uniformly isolating* executions. A trace is uniformly isolating, if "for every $k > 0$, any thread that takes an infinite number of steps has an interval where it takes at least k concrete contiguous steps" [34]. Then, their obstruction-free object guarantees wait-freedom for every uniformly isolating execution. They also propose a new progress property, clash-freedom, which guarantees lock-freedom for uniformly-isolating executions.

Below we give an example showing that their definition is inconsistent with the common intuition of obstruction-freedom. The object implementation uses three shared variables: `x`, `a` and `b`. It provides two methods `f` and `g`.

```
f() {
  while (a <= x <= b) {
    x++;
    a--;
  }
}
```

```
g() {
  while (a <= x <= b) {
    x--;
    b++;
  }
}
```

We can see that, if `f()` or `g()` is eventually executed in isolation (i.e., we suspend all but one threads), it must returns. Thus intuitively this object should be obstruction-free. It also satisfies our formulation in Figure 6.2.

However, we could construct an execution which is uniformly isolating but is not lock-free or wait-free. Consider the client program `f()` $\|$ `g()`. It has an execution shown in Figure A.1. Starting from `x` $= 0$, `a` $= -1$ and `b` $= 1$, we alternatively let each thread execute more and more iterations. Then for any k, we could always find an interval of k iterations for each thread in this execution.

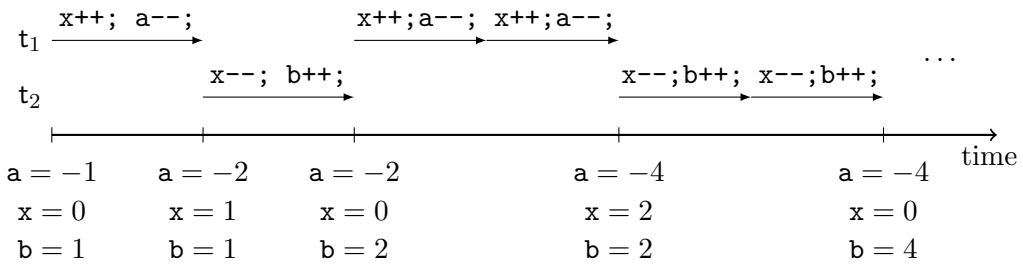


Figure A.1 Example execution.

Thus the execution is uniformly isolating. But neither method call finishes. This execution is not lock-free nor wait-free. Thus the object does not satisfy Herlihy and Shavit's obstruction-freedom or clash-freedom definitions.

Appendix B

Proofs of Equivalence Results

In this appendix, we sketch the proofs for the equivalence results between linearizablity, progress properties and contextual refinements, including Theorem 5.6 (the basic equivalence for linearizability) and Theorem 6.3 (our new equivalence results for various progress properties). More detailed proofs can be found in our technical report [45].

Below we use $\mathcal{T}[\![W, \mathcal{S}]\!]$ for the prefix-closed set of finite event traces generated by W with the initial state $\mathcal{S}$. It generalizes the definition of $\mathcal{T}[\![W, (\sigma_c, \sigma_o)]\!]$ (in Figure 5.6) to allow nonempty call stacks in the initial state. We also use $\mathcal{H}[\![W, \mathcal{S}]\!]$ and $\mathcal{O}[\![W, \mathcal{S}]\!]$ for the sets of histories and finite observable event traces respectively. Similar notations are used for the sets of complete traces as well.

B.1 Most General Client

The key in our proofs is the use of the Most General Client (MGC). Informally, an MGC is a special client which itself can produce all the possible behaviors produced by any clients. We can define the MGC versions of linearizability, progress properties, and observable refinements, and prove their relationships to the original definitions, which universally quantify over arbitrary client programs. Then we reduce the problems of proving the equivalence between original definitions to proving some relationships between the MGC versions. Since an MGC is a specific client, the latter task is usually much simpler.

In fact, we define three MGCs, which produce different "general" behaviors. We assume $dom(\Pi) = \{f_1, \ldots, f_m\}$, and introduce two instructions to get a random (nondeterministic) value. $x := \mathbf{rand}(m)$ assigns x a random integer $i \in [1..m]$, and $x := \mathbf{rand}()$ computes an arbitrarily large random integer. Then, for any n, we define MGC_n as follows.

$$
\begin{array}{rcl}
\mathsf{MGT}_\mathsf{t} & \stackrel{\text{def}}{=} & \textbf{while } (\textbf{true})\{ \\
& & \quad x_\mathsf{t} := \textbf{rand}();\ y_\mathsf{t} := \textbf{rand}(m); \\
& & \quad z_\mathsf{t} := f_{y_\mathsf{t}}(x_\mathsf{t}); \\
& & \} \\
\mathsf{MGC}_n & \stackrel{\text{def}}{=} & ||_{\mathsf{t}\in[1..n]}\, \mathsf{MGT}_\mathsf{t}
\end{array}
$$

Here x_t, y_t and z_t are all local variables for thread t. Each thread in MGC_n keeps calling a random method with a random argument. We also define MGCp_n, which print out the arguments and return values for method calls.

$$
\begin{array}{rcl}
\mathsf{MGTp}_\mathsf{t} & \stackrel{\text{def}}{=} & \textbf{while } (\textbf{true})\{ \\
& & \quad x_\mathsf{t} := \textbf{rand}();\ y_\mathsf{t} := \textbf{rand}(m);\ \textbf{print}(y_\mathsf{t}, x_\mathsf{t}); \\
& & \quad z_\mathsf{t} := f_{y_\mathsf{t}}(x_\mathsf{t});\ \textbf{print}(z_\mathsf{t}); \\
& & \} \\
\mathsf{MGCp}_n & \stackrel{\text{def}}{=} & ||_{\mathsf{t}\in[1..n]}\, \mathsf{MGTp}_\mathsf{t}
\end{array}
$$

The third MGC $\mathsf{MGCp1}_n$ always print out 1 when a method call is finished. It is useful to observe the progress of objects.

$$
\begin{array}{rcl}
\mathsf{MGTp1}_\mathsf{t} & \stackrel{\text{def}}{=} & \textbf{while } (\textbf{true})\{ \\
& & \quad x_\mathsf{t} := \textbf{rand}();\ y_\mathsf{t} := \textbf{rand}(m); \\
& & \quad z_\mathsf{t} := f_{y_\mathsf{t}}(x_\mathsf{t});\ \textbf{print}(1); \\
& & \} \\
\mathsf{MGCp1}_n & \stackrel{\text{def}}{=} & ||_{\mathsf{t}\in[1..n]}\, \mathsf{MGTp1}_\mathsf{t}
\end{array}
$$

The client memory for the above MGCs should contain the local variables for each thread.

$$
\sigma_{\mathsf{MGC}} \stackrel{\text{def}}{=} \{x_\mathsf{t} \rightsquigarrow \_, y_\mathsf{t} \rightsquigarrow \_, z_\mathsf{t} \rightsquigarrow \_ \mid 1 \le \mathsf{t} \le n\}
$$

B.2 Theorem 5.6 (Basic Equivalence)

We first define the MGC versions of "linearizability" and "refinement".

Definition B.1. $\Pi \preceq_\varphi^{\mathsf{MGC}} \Pi_A$ iff

$$
\begin{array}{l}
\forall n, \sigma_{\mathsf{MGC}}, \sigma_o, \sigma_a, T.\ \ T \in \mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!] \wedge (\varphi(\sigma_o) = \sigma_a) \\
\Longrightarrow\ \exists T_c, T'.\ T_c \in \mathsf{completions}(T) \wedge \Pi_A \rhd_n (\sigma_{\mathsf{MGC}}, \sigma_a, T') \wedge T_c \preceq_{\mathsf{lin}} T'
\end{array}
$$

where

$$
\Pi_A \rhd_n (\sigma_{\mathsf{MGC}}, \sigma_a, T) \stackrel{\text{def}}{=} T \in \mathcal{H}[\![(\textbf{let } \Pi_A \textbf{ in } \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!] \wedge \mathsf{seq}(T)\,.
$$

Definition B.2. $\Pi \subseteq_\varphi \Pi_A$ iff

$\forall n, \sigma_{\mathsf{MGC}}, \sigma_o, \sigma_a.\ (\varphi(\sigma_o) = \sigma_a)$
$\implies \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]\,.$

To prove Theorem 5.6, we prove the following lemmas.

Lemma B.3. $\Pi \preceq_\varphi \Pi_A \iff \Pi \preceq_\varphi^{\mathsf{MGC}} \Pi_A\,.$

Lemma B.4. $\Pi \sqsubseteq_\varphi \Pi_A \iff \Pi \subseteq_\varphi \Pi_A\,.$

Lemma B.5. $\Pi \subseteq_\varphi \Pi_A \iff \Pi \preceq_\varphi^{\mathsf{MGC}} \Pi_A\,.$

Proof of Lemma B.3. We can see $\Pi \preceq_\varphi^{\mathsf{MGC}} \Pi_A$ is simply defined by fixing the arbitrarily quantified client programs in $\Pi \preceq_\varphi \Pi_A$ (Definition 5.4) as MGC_n. Intuitively, the key to prove this lemma is to show that any history generated by an arbitrary client program can also be generated by MGC, as shown in the following lemma.

Lemma B.6 (MGC is the Most General)**.** *For any n, C_1, ..., C_n, σ_c, σ_{MGC} and σ_o, $\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$.*

Proof. We construct a simulation $\precsim_{\mathsf{MGC}}$ between the client program and the MGC. We need the simulation relation to satisfy the following (B.1).

For any W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$ and e_1, if $(W_1, \mathcal{S}_1) \precsim_{\mathsf{MGC}} (W_2, \mathcal{S}_2)$, then

(1) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} \mathbf{abort}$ and $\mathsf{is\_obj\_abt}(e_1)$, then
there exists T_2 such that $(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^* \mathbf{abort}$ and
$e_1 = \mathsf{get\_hist}(T_2)$;

(2) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} (W_1', \mathcal{S}_1')$, then
there exist T_2, W_2' and $\mathcal{S}_2'$ such that $(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^* (W_2', \mathcal{S}_2')$,
$\mathsf{get\_hist}(e_1) = \mathsf{get\_hist}(T_2)$ and $(W_1', \mathcal{S}_1') \precsim_{\mathsf{MGC}} (W_2', \mathcal{S}_2')$.

(B.1)

The simulation relation is constructed as follows. For each client thread t, if the left side is in some normal client code, it corresponds to MGT_t at the right side; otherwise, if the left side is inside a method call, its code is the same as the right side. Informally, the following hold for each thread t.

(1) Each normal client step of the left corresponds to zero step of MGT_t.

(2) Each method invocation corresponds to the steps executing MGT_t to the same method body, with the same argument.

(3) Each step inside the method body at the left corresponds to the same step at the right.

(4) Each return step at the left corresponds to the same return step (plus a few client steps) executing the right side code to MGT_t.

Then with (B.1), we can prove the following by induction over the number of steps generating the event trace of $\mathcal{H}[\![W_1, \mathcal{S}_1]\!]$.

$$\text{If } (W_1, \mathcal{S}_1) \precsim_{\mathsf{MGC}} (W_2, \mathcal{S}_2), \text{ then } \mathcal{H}[\![W_1, \mathcal{S}_1]\!] \subseteq \mathcal{H}[\![W_2, \mathcal{S}_2]\!].$$

Also we have

$$(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \circledcirc)) \precsim_{\mathsf{MGC}} (\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n, (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)),$$

thus we are done. □

Then Lemma B.3 is immediate by unfolding the definitions of $\Pi \preceq_\varphi^{\mathsf{MGC}} \Pi_A$ and $\Pi \preceq_\varphi \Pi_A$, and applying Lemma B.6.

Proof of Lemma B.4.

1. $\Pi \sqsubseteq_\varphi \Pi_A \implies \Pi \subseteq_\varphi \Pi_A$:

   To prove this direction, we show that any history generated by MGC_n is "equivalent" to an observable trace generated by MGCp_n, i.e., the following lemma holds.

   Lemma B.7. *For any n, σ_o and σ_{MGC}, both the following holds.*

   (a) For any T_1, if $T_1 \in \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$, then there exists T_2 such that $T_2 \in \mathcal{O}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$ and $T_1 \approx T_2$.

   (b) For any T_2, if $T_2 \in \mathcal{O}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$, then there exists T_1 such that $T_1 \in \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$ and $T_1 \approx T_2$.

   Here $T_1 \approx T_2$ is inductively defined as follows.

$$\frac{}{\epsilon \approx \epsilon} \qquad \frac{e_1 \approx e_2 \qquad T_1 \approx T_2}{e_1 :: T_1 \approx e_2 :: T_2}$$

$$\frac{}{(\mathsf{t}, f_i, n) \approx (\mathsf{t}, \mathbf{out}, (i, n))} \qquad \frac{}{(\mathsf{t}, \mathbf{ret}, n) \approx (\mathsf{t}, \mathbf{out}, n)}$$

$$\frac{}{(\mathsf{t}, \mathbf{obj}, \mathbf{abort}) \approx (\mathsf{t}, \mathbf{obj}, \mathbf{abort})}$$

Proof. By constructing simulations between MGC_n and MGCp_n. □

2. $\Pi \subseteq_\varphi \Pi_A \implies \Pi \sqsubseteq_\varphi \Pi_A$:

Proof. For any n, C_1, …, C_n, σ_c, σ_{MGC} and σ_o, by Lemma B.6, we know

$$\begin{aligned}&\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\\ &\subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]\,.\end{aligned}$$

Since $\Pi \subseteq_\varphi \Pi_A$, we know for any σ_a such that $\sigma_a = \varphi(\sigma_o)$, we have

$$\begin{aligned}&\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]\\ &\subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]\,.\end{aligned}$$

Thus, for any T such that

$$T \in \mathcal{T}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,,$$

there exists T_{MGC} such that $\mathsf{get\_hist}(T) = \mathsf{get\_hist}(T_{\mathsf{MGC}})$ and

$$T_{\mathsf{MGC}} \in \mathcal{T}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]\,.$$

Intuitively, we can then replace the object events in T with those in T_{MGC} and keep other events (and the order between them) unchanged. Thus the resulting trace T' satisfies $\mathsf{get\_obsv}(T') = \mathsf{get\_obsv}(T)$. We prove T' can be produced by the corresponding abstract client program, that is

$$T' \in \mathcal{T}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,.$$

Then we are done.

Alternatively, we can actually construct a "simulation" relation $\precsim$ between the three programs: the concrete client program, the abstract MGC and the corresponding abstract client program, and prove it satisfies the following (B.2).

> For any W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$, W_3, $\mathcal{S}_3$ and e_1,
> if $(W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, then
>
> (1) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} \mathbf{abort}$, then there exists T_3 such that $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^* \mathbf{abort}$ and $e_1 = \mathsf{get\_obsv}(T_3)$;
>
> (2) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} (W_1', \mathcal{S}_1')$ and $\mathsf{is\_clt}(e_1)$, then there exist W_3' and $\mathcal{S}_3'$ such that $(W_3, \mathcal{S}_3) \overset{e_1}{\longmapsto} (W_3', \mathcal{S}_3')$ and $(W_1', \mathcal{S}_1') \precsim (W_2, \mathcal{S}_2; W_3', \mathcal{S}_3')$.

(3) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} (W_1', \mathcal{S}_1')$ and $\mathsf{is\_obj}(e_1)$, then
there exist T_2, W_2', $\mathcal{S}_2'$, T_3, W_3' and $\mathcal{S}_3'$ such that
$(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^* (W_2', \mathcal{S}_2')$, $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^* (W_3', \mathcal{S}_3')$,
$\mathsf{get\_hist}(e_1) = \mathsf{get\_hist}(T_2)$, $\mathsf{get\_obj}(T_2) = T_3$ and
$(W_1', \mathcal{S}_1') \precsim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3')$.

(B.2)

That is, the executions of the abstract program W_3 follow the concrete program W_1 for client steps and the abstract MGC W_2 for object steps. Here we use $\mathsf{get\_obj}(T)$ to get the sub-trace of T consisting of object events only. We can prove the following from (B.2).

$$\text{If } (W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3), \text{ then } \mathcal{O}[\![W_1, \mathcal{S}_1]\!] \subseteq \mathcal{O}[\![W_3, \mathcal{S}_3]\!].$$

Initially,

$$(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \circledcirc))$$
$$\precsim (\textbf{let } \Pi_A \textbf{ in } \mathsf{MGC}_n, (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc);\ \textbf{let } \Pi_A \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \circledcirc))$$

holds, and we are done. □

Proof of Lemma B.5.

1. $\Pi \subseteq_{\varphi} \Pi_A \implies \Pi \preceq_{\varphi}^{\mathsf{MGC}} \Pi_A$:

The premise $\Pi \subseteq_{\varphi} \Pi_A$ tells us that every history generated by using the concrete object Π can also be generated with the abstract object Π_A. Thus, to prove the concrete object Π is linearizable, we only need to show the abstract object Π_A (whose methods are atomic) is linearizable with respect to itself.

Lemma B.8 (Π_A is Linearizable)**.** *For any n, σ_{MGC}, σ_a and T,*
if $T \in \mathcal{H}[\![(\textbf{let } \Pi_A \textbf{ in } \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$, then
there exist T_c and T' such that $T_c \in \mathsf{completions}(T)$, $T_c \preceq_{\mathsf{lin}} T'$, $\mathsf{seq}(T')$ and $T' \in \mathcal{H}[\![(\textbf{let } \Pi_A \textbf{ in } \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$.

Proof. From the execution that generates T, we construct another execution as follows. We postpone every invocation step and advance the latter return step to the single step of the atomic method body in between. We prove the resulting execution generates a history T' satisfying all the requirements. □

2. $\Pi \preceq_{\varphi}^{\mathsf{MGC}} \Pi_A \implies \Pi \subseteq_{\varphi} \Pi_A$:

The key is to prove that every linearizable history can be generated by the MGC with the *abstract* object Π_A.

Lemma B.9 (Rearrangement)**.** *For any n, σ_{MGC}, σ_a, T and T', if $T \preceq_{\mathsf{lin}} T'$, $\mathsf{seq}(T')$ and $T' \in \mathcal{H}[\![(\textbf{let } \Pi_A \textbf{ in } \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$, then $T \in \mathcal{H}[\![(\textbf{let } \Pi_A \textbf{ in } \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$.*

Proof. We construct the execution generating T, where the order to execute the atomic method body for concurrent method calls simply follow the order of the pairs of invocation and subsequent return events in T'. The detailed proof is by induction over the length of T. □

B.3 Theorem 6.3 (New Equivalence Results)

Below we sketch the proofs for the case of lock-freedom. By Theorem 5.6, the goal is reduced to proving

$$\Pi \sqsubseteq_{\varphi} \Pi_A \wedge \mathsf{lock\text{-}free}_{\varphi}(\Pi) \iff \Pi \sqsubseteq_{\varphi}^{\mathsf{lock\text{-}free}} \Pi_A\,.$$

We first define the MGC version of "lock-freedom".

Definition B.10. $\mathsf{lock\text{-}free}_{\varphi}^{\mathsf{MGC}}(\Pi)$, iff

$$\forall n, \sigma_{\mathsf{MGC}}, \sigma_o, T.\ \ T \in \mathcal{T}_{\omega}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!] \wedge (\sigma_o \in \mathit{dom}(\varphi)) \implies (\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j)))$$

We prove the following lemmas.

Lemma B.11. $\mathsf{lock\text{-}free}_{\varphi}(\Pi) \iff \mathsf{lock\text{-}free}_{\varphi}^{\mathsf{MGC}}(\Pi)$.

Proof. Similar to the proof of Lemma B.3. We relate every complete event trace generated by an arbitrary client program to an event trace of MGC_n. □

Lemma B.12. $\Pi \sqsubseteq_{\varphi}^{\mathsf{lock\text{-}free}} \Pi_A \implies \Pi \sqsubseteq_{\varphi} \Pi_A$.

Proof. Immediate from the definitions. □

Lemma B.13. $\Pi \sqsubseteq_{\varphi}^{\mathsf{lock\text{-}free}} \Pi_A \implies \mathsf{lock\text{-}free}_{\varphi}^{\mathsf{MGC}}(\Pi)$.

Proof. We utilize the most general client $\mathsf{MGTp1}_n$. We prove:

(1) For any n, σ_{MGC}, σ_a and T, if $T \in \mathcal{O}_\omega[\![(\textbf{let } \Pi_A \textbf{ in } \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$, then T must be an infinite trace of $(\_, \textbf{out}, 1)$. As a consequence, from $\Pi \sqsubseteq_\varphi^{\mathsf{lock\text{-}free}} \Pi_A$, we know if $T' \in \mathcal{O}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$, then T' is also an infinite trace of $(\_, \textbf{out}, 1)$.

(2) For any n, σ_{MGC}, σ_o and T, if $T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$, then there exists $T' \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$ such that

   (a) $T' \backslash (\_, \textbf{out}, 1) = T$. That is, we can get T by removing all the output events in T'.

   (b) $\forall i, \mathsf{t}.\ T'(i) = (\mathsf{t}, \textbf{ret}, \_) \Leftrightarrow T'(i{+}1) = (\mathsf{t}, \textbf{out}, 1)$. That is, every output immediately follows a return event.

   From (1), we know $\mathsf{get\_obsv}(T')$ is an infinite trace of $(\_, \textbf{out}, 1)$. Thus T' contains an infinite number of return events, and so does T.

By Definition B.10, we are done. □

Lemma B.14. $\Pi \sqsubseteq_\varphi \Pi_A \wedge \mathsf{lock\text{-}free}_\varphi(\Pi) \implies \Pi \sqsubseteq_\varphi^{\mathsf{lock\text{-}free}} \Pi_A$.

Proof. The key is to show the following (B.3).

> For any n, C_1, ..., C_n, σ_c, σ_o and σ_a such that $\varphi(\sigma_o) = \sigma_a$,
> if $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^\omega \cdot$, then
> there exists T_a such that $(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^\omega \cdot$
> and $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.
>
> (B.3)

Its proof is similar to the proof of $\Pi \subseteq_\varphi \Pi_A \implies \Pi \sqsubseteq_\varphi \Pi_A$. We can replace the object events in the concrete *infinite* trace T with those generated by MGC using Π_A. The resulting trace T_a satisfies $\mathsf{get\_obsv}(T_a) = \mathsf{get\_obsv}(T)$. From $\mathsf{lock\text{-}free}_\varphi(\Pi)$, we can show T_a must be infinite too. □

The proofs for the cases of other progress properties are similar and omitted here. Detailed proofs can be found in our technical report [45].

Acknowledgments

Thanks Xinyu Feng, my advisor. I have benefited much from his research experience and wisdom, his high standards, and his pushing me to explore various ideas. He has also given me strong emotional support, and taught me a lot about life. The most important things that I have learned from him are to "be logical", "be professional" and "be proactive", which I believe will be useful for my whole life. Without his assistance in research and friendship in life, the dissertation would be impossible. I am so grateful for his time and efforts, and still hope to learn more from him. Thank you very much, Xinyu.

Thanks Zhong Shao, my "official advisor", for his kindness and guidance. He taught me by example to find the "good" in the "bad". I really appreciate his patience in our technical discussions.

Thanks the colleagues in the Software Security Lab of USTC and the FLINT group at Yale, for their friendship and insight. In particular I would like to thank Professor Yiyun Chen, who cares much about me; Xinyu Jiang, who knows everything; Xi Wang, who reminds me to explore life and have fun; and, Yu Guo and Zhaopeng Li, who open their doors for me when I feel down.

People in our community are so nice that they always tolerate my ignorance and poor communication skills. Their recognition on my work can always cheer me up. Their comments and suggestions make me work hard to improve. Thanks Matthew Parkinson, Derek Dreyer, Hongseok Yang, Viktor Vafeiadis, and everybody who I have forgotten to mention, for their encouragement and help. Thanks anonymous referees for reading earlier versions of parts of my work.

This dissertation is dedicated to my paternal grandmother, who passed away a few weeks after I applied to enter the graduate program, and my maternal grandmother, who passed away a few days before my defense.

Special thanks go to my parents. I love you, Mom and Dad.

Hongjin Liang
May 27, 2014

Previously Published Materials

This dissertation draws heavily on the earlier work and writing in the following papers, written jointly with several collaborators.

1. H. Liang, X. Feng and M. Fu. Rely-Guarantee-Based Simulation for Compositional Verification of Concurrent Program Transformations. *ACM Trans. Program. Lang. Syst.*, 36(1):3:1–3:55, March 2014.

2. H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. In *Proceedings of 24th International Conference on Concurrency Theory (CONCUR 2013)*, Buenos Aires, Argentina, pages 227–241, August 2013.

3. H. Liang and X. Feng. Modular Verification of Linearizability with Non-Fixed Linearization Points. In *Proceedings of 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, Seattle, USA, pages 459–470, June 2013.

4. H. Liang, X. Feng and M. Fu. A Rely-Guarantee-based Simulation for Verifying Concurrent Program Transformations. In *Proceedings of 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, Philadelphia, USA, pages 455–468, January 2012.