A Rely-Guarantee-Based Simulation for Verifying Concurrent Program Transformations

Hongjin Liang  Xinyu Feng  Ming Fu
School of Computer Science and Technology, University of Science and Technology of China
Hefei, Anhui 230026, China
lhj1018@mail.ustc.edu.cn  xyfeng@ustc.edu.cn  fuming@ustc.edu.cn

Abstract
Verifying program transformations usually requires proving that the resulting program (the target) refines or is equivalent to the original one (the source). However, the refinement relation between individual sequential threads cannot be preserved in general with the presence of parallel compositions, due to instruction reordering and the different granularities of atomic operations at the source and the target. On the other hand, the refinement relation defined based on fully abstract semantics of concurrent programs assumes arbitrary parallel environments, which is too strong and cannot be satisfied by many well-known transformations.

In this paper, we propose a Rely-Guarantee-based Simulation (RGSim) to verify concurrent program transformations. The relation is parametrized with constraints of the environments that the source and the target programs may compose with. It considers the interference between threads and their environments, thus is less permissive than relations over sequential programs. It is compositional w.r.t. parallel compositions as long as the constraints are satisfied. Also, RGSim does not require semantics preservation under all environments, and can incorporate the assumptions about environments made by specific program transformations in the form of rely/guarantee conditions. We use RGSim to reason about optimizations and prove atomicity of concurrent objects. We also propose a general garbage collector verification framework based on RGSim, and verify the Boehm et al. concurrent mark-sweep GC.

1. Introduction
Many verification problems can be reduced to verifying program transformations, i.e., proving the target program of the transformation has no more observable behaviors than the source. Below we give some typical examples in concurrent settings:

• Correctness of compilation and optimizations of concurrent programs. In this most natural program transformation verification problem, every compilation phase does a program transformation \( T \), which needs to preserve the semantics of the inputs.

• Atomicity of 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 [13]. Their correctness can be reduced to the correctness of the transformation from abstract atomic operations to concrete and executable programs in a concurrent context.

• Verifying implementations of software transactional memory (STM). Many languages supporting STM provide a high-level atomic block \( \text{atomic}(\cdot) \), so that programmers can assume the atomicity of the execution of \( \cdot \). Atomic blocks are implemented using some STM protocol (e.g., TL2 [11]) 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 concurrent 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 implementation as a transformation from high-level mutators to low-level ones with a concrete GC thread, the GC safety can be reduced naturally to the semantics preservation of the transformation.

To verify the correctness of a program transformation \( T \), we follow Leroy’s approach [19] and define a refinement relation \( \subseteq \) between the target and the source programs, which says the target has no more observable behaviors than the source. Then we can formalize the correctness of the transformation as follows:

\[
\text{Correct}(T) \triangleq \forall C, C = T(C) \implies C \subseteq C. \tag{1.1}
\]

That is, for any source program \( C \) acceptable by \( T \), \( T(C) \) is a refinement of \( C \). When the source and the target are shared-state concurrent programs, the refinement \( \subseteq \) needs to satisfy the following requirements to support effective proof of \( \text{Correct}(T) \):

• Since the target \( T(C) \) may be in a different language from the source, the refinement should be general and independent of the language details.

• To verify fine-grained implementations of abstract operations, the refinement should support different views of program states and different granularities of state accesses at the source and the target levels.

• When \( T \) is syntax-directed (and it is usually the case for parallel compositions, i.e., \( T(C || C') = T(C) || T(C') \)), a compositional refinement is of particular importance for modular verification of \( T \).

However, existing refinement (or equivalence) relations cannot satisfy all these requirements at the same time. Contextual equivalence, the canonical notion for comparing program behaviors, fails to handle different languages since the contexts of the source and the target will be different. Simulations and logical relations have been used to verify compilation [4, 7, 8, 9, 21], but they are usually designed for sequential programs (except [7, 8, 21]), which we will discuss in Section 5. Since the refinement or equivalence relation between sequential threads cannot be preserved in general with parallel compositions, we cannot simply adapt existing work on sequential programs to verify transformations of concurrent programs. Refinement relations based on fully abstract semantics of concurrent programs are compositional, but they assume arbitrary program contexts, which is too strong for many practical transformations. We will explain the challenges in detail in Section 2.
In this paper, we propose a Rely-Guarantee-based Simulation (RGSim) for compositional verification of concurrent transformations. By addressing the above problems, we make the following contributions:

- RGSim parametrizes the simulation between concurrent programs with rely/guarantee conditions [17], which specify the interactions between the programs and their environments. This makes the corresponding refinement relation compositionally w.r.t. parallel compositions, allowing us to decompose refinement proofs for multi-threaded programs into proofs for individual threads. On the other hand, the rely/guarantee conditions can incorporate the assumptions about environments made by specific program transformations, so RGSim can be applied to verify many practical transformations.

- 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.

- RGSim 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, etc.

- RGSim gives us a refinement-based proof method to verify fine-grained implementations of abstract algorithms and concurrent objects. We successfully apply RGSim to verify concurrent counters, the concurrent GCD algorithm, Treiber's non-blocking stack and the lock-coupling list.

- We reduce the problem of verifying concurrent garbage collectors to verifying transformations, and present a general GC verification framework, which combines unary Rely-Guarantee-based verification [13] with relational proofs based on RGSim.

- We verify the Boehm et al. concurrent garbage collection algorithm [2] using our framework. As far as we know, it is the first time to formally prove the correctness of this algorithm.

In the rest of this paper, we first analyze the challenges for compositional verification of concurrent program transformations, and explain our approach informally in Section 2. Then we give the basic technical settings in Section 3 and present the formal definition of RGSim in Section 4. We show the use of RGSim to reason about optimizations in Section 5. Finally, we discuss related work and conclude in Section 6.

2. Challenges and Our Approach

The major challenge we face is to have a compositional refinement relation \( \subseteq \) between concurrent programs, i.e., we should be able to know \( T(C_1) \parallel T(C_2) \subseteq C_1 \parallel C_2 \) if we have \( T(C_1) \subseteq C_1 \) and \( T(C_2) \subseteq C_2 \).

2.1 Sequential Refinement Loses Parallel Compositional

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 [5]. The Dekker’s algorithm shown in Figure 1(a) has been widely used to demonstrate the problem. Re-ordering the first two 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 source and the target programs can also break the compositionality of program equivalence in a concurrent setting. In Figure 1(b), the target program at the bottom behaves differently from the source at the top (assuming each statement is executed atomically), although the individual threads at the target and the source have the same behaviors.

2.2 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. People have given fully abstract semantics to concurrent programs (e.g., [1, 8]). 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, but unfortunately is too strong to formulate the correctness of many well-known transformations, including the four classes of transformations mentioned before:

- Many concurrent languages (e.g., C++ [4]) do not give semantics to programs with data races (like the examples shown in Figure 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 operation, we can assume that all accesses to the object in the context of the target program use the same set of primitives.

- Usually the implementation of STM (e.g., TL2 [11]) ensures the atomicity of a transaction atomic\{C\} only when there are no data races. Therefore, the correctness of the transformation from high-level atomic blocks to fine-grained concurrent code assumes data-race-freedom in the source.

- Many garbage-collected languages are type-safe and prohibit operations such as pointer arithmetics. Therefore the garbage collector could make corresponding assumptions about the mutators that run in parallel.

---

Figure 1. Equivalence Lost after Parallel Composition

```
llocal rl; local r2;
x := 1;
r1 := y := x;
if (r1 = 0) then if (r2 = 0) then
```

(a) Dekker’s Mutual Exclusion Algorithm

```x := x+1; || x := x+1;
```

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

(b) Different Granularities of Atomic Operations
In all these cases, the transformations of individual threads are allowed to make various assumptions about the environments. They do not have to ensure semantics preservation within all contexts.

### 2.3 Languages at Source and Target May Be Different

The use of different languages at the source and the target levels makes the formulation of the transformation correctness more difficult. If the source and the target languages have different views of program states and different atomic primitives, we cannot directly compare the state transitions made by the source and the target 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.

### 2.4 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 source and the target 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. On the other hand, relating all state accesses of programs is too strong. Any rendering of state accesses or change of atomicity would fail the refinement.

### 2.5 Our Approach

In this paper we propose a Rely-Guarantee-based Simulation (RGSim) \( \preceq \) between the target and the source programs. It establishes a weak simulation, ensuring that for every externally observable event made by the target program there is a corresponding one in the source. 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 (R-G) verification method [17], 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 others.

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).

Parametrized with rely/guarantee conditions for the two levels, our relation \((C, R, G) \preceq (C, R, G)\) talks about not only the target \(C\) and the source \(C\), but also the interference \(R\) and \(G\) between \(C\) and its target-level environment, and \(R\) and \(G\) between \(C\) and its environment at the source level. Informally, \((C, R, G) \preceq (C, R, G)\) says the executions of \(C\) under the environment \(R\) do not exhibit more observable behaviors than the executions of \(C\) under the environment \(R\), and the state transitions of \(C\) and \(C\) satisfy \(G\) and \(G\).

### Figure 2. Generic Languages at Target and Source Levels

\[ \begin{align*}
(LState) & \quad \sigma \ ::= \ldots \\
(LExpr) & \quad E \in LState \rightarrow \mathit{Int}_\perp \\
(LBExp) & \quad B \in LState \rightarrow \{ \mathit{true}, \mathit{false} \}_\perp \\
(LInstr) & \quad c \in LState \rightarrow \mathcal{P} (\{ \mathit{Labels} \times LState \} \cup \{ \mathit{abort} \}) \\
(LStmt) & \quad C ::= \mathit{skip} | e | C_1 ; C_2 | \mathit{if} (B) C_1 \mathit{else} C_2 \\
& \quad | \mathit{while} (B) C' | \mathit{while} \not\mathit{do} C_1 | C_1 \| C_2 \\
(LStep) & \quad \rightarrow_L \in \mathcal{P}((LStmt / \{ \mathit{skip} \} \times LState) \cup \{ \mathit{abort} \}) \\
(HState) & \quad \Sigma \ ::= \ldots \\
(HExpr) & \quad E \in HState \rightarrow \mathit{Int}_\perp \\
(HBExp) & \quad B \in HState \rightarrow \{ \mathit{true}, \mathit{false} \}_\perp \\
(HInstr) & \quad c \in HState \rightarrow \mathcal{P} (\{ \mathit{Labels} \times HState \} \cup \{ \mathit{abort} \}) \\
(HStmt) & \quad C ::= \mathit{skip} | e | C_1 ; C_2 | \mathit{if} (\not\mathit{then} C_1 \mathit{else} C_2) \\
& \quad | \mathit{while} B \not\mathit{do} C_1 | C_1 \| C_2 \\
(HStep) & \quad \rightarrow_H \in \mathcal{P}((HStmt / \{ \mathit{skip} \} \times HState) \cup \{ \mathit{abort} \}) \\
\end{align*} \]

\[ \begin{align*}
\begin{array}{ll}
(Events) & e :: \ldots \\
(Labels) & o :: e | \tau
\end{array} \]

(a) Events and Transition Labels

(b) The Low-Level Language

(c) The High-Level Language

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 \((C_1, R_1, G_1) \preceq (C_1, R_1, G_1)\) and \((C_2, R_2, G_2) \preceq (C_2, R_2, G_2)\), and also the interference constraints are satisfied, i.e., \(G_2 \subseteq R_1, G_1 \subseteq R_2, G_2 \subseteq R_1\) and \(G_1 \subseteq R_2\), we could get \((C_1 \| C_2, R_1 \cap R_2, G_1 \cup G_2) \preceq (C_1 \| C_2, R_1 \cap R_2, G_1 \cup G_2)\).

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

Also different from fully abstract semantics for threads, which assumes arbitrary behaviors of environments, RGSim allows us to instantiate the interference \(R, G, R, G\) differently for different assumptions about environments, therefore it can be used to verify the aforementioned four classes of transformations. For instance, if we want to prove that a transformation preserves the behaviors of data-race-free programs, we can specify the data-race-freedom in \(R\) and \(G\). Then we are no longer concerned with the examples in Figure 1 both of which have data races.

### 3. Basic Technical Settings

In this section, we present the source and the target programming languages. Then we define a basic refinement \(\sqsubseteq\), which naturally says the target has no more externally observable event traces than the source. We use \(\sqsubseteq\) as an intuitive formulation of the correctness of transformations.

#### 3.1 The Languages

Following standard simulation techniques, we model the semantics of target and source programs as labeled transition systems. Before showing the languages, we first define events and labels in Figure 2(a). We leave the set of events unspecified here. It can be instantiated by program verifiers, depending on their interest (e.g., input/output events). A label that will be associated with a state
transition is either an event or \( \tau \), which means the corresponding transition does not generate any event (i.e., a silent step).

The target language, which we also call the low-level language, is shown in Figure 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 \( \bot \). Boolean expressions are modeled similarly. An instruction is a partial function from states to sets of labels and state pairs, describing the state transitions and the events it generates. We use \( P(\bot) \) to denote the power set. Unsafe executions lead to \texttt{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 acquiring a lock.

Statements are either primitive instructions or compositions of them. \texttt{skip} is a special statement used as a flag to show the end of executions. A single-step execution of statements is modeled as a labeled transition \( \alpha \rightarrow \bot \), which is a triple of an initial program configuration (a pair of statement and state), a label and a resulting configuration. It is undefined when the initial statement is \texttt{skip}. The step aborts if an unsafe instruction is executed.

The high-level language (source language) is defined similarly in Figure 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. \( C_1 \cdot C_2 \) and \( C_1 \parallel C_2 \) are sequential and parallel compositions of \( C_1 \) and \( C_2 \) 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 \( C_1 \| C_2 \)) make it convenient for us to discuss the compositionality later.

Figure 3 shows part of the definition of \( \alpha \rightarrow \bot \) which gives the high-level operational semantics of statements. We often omit the subscript \( H \) (or \( L \)) in \( \alpha \rightarrow \bot \) and the label on top of the arrow when it is \( \tau \). The semantics is mostly standard. Note that when a primitive instruction \( c \) is blocked at state \( \Sigma \) (i.e., \( \Sigma \notin dom(c) \)), we let the program configuration reduce to itself. For example, the instruction \texttt{lock(1)} would be blocked when \( 1 \) is not 0, making it be repeated until \( 1 \) becomes 0; whereas \texttt{unlock(1)} simply sets \( 1 \) 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. Below we use \( \alpha \rightarrow^* \tau \) for zero or multiple-step transitions with no events generated, and \( \alpha \rightarrow^+ \tau \) for multiple-step transitions with only one event \( \epsilon \) generated by the transition.

3.2 The Event Trace Refinement

Now we can formally define the refinement relation \( \sqsubseteq \) that relates the set of externally observable event traces generated by the target and the source programs. A trace is a sequence of events \( e \), and may end with a termination marker \texttt{done} or a fault marker \texttt{abort}.

\[
\text{EvtTrace} \quad E := \epsilon \mid \text{done} \mid \text{abort} \mid e ::: E
\]

**Definition 1 (Event Trace Set).** \( ETrSet_{n}(C, \sigma) \) represents a set of external event traces produced by \( C \) in \( n \) steps from the state \( \sigma \):

- \( ETrSet_{0}(C, \sigma) \equiv \{ \epsilon \} \);
- \( ETrSet_{n+1}(C, \sigma) \equiv \{ E \mid (C, \sigma) \rightarrow (C', \sigma') \land E \in ETrSet_{n}(C', \sigma') \}
  \lor (C, \sigma) \rightarrow (C', \sigma') \land \epsilon \in ETrSet_{n}(C', \sigma') \lor (C, \sigma) \rightarrow \text{abort} \land \epsilon \in ETrSet_{n}(C', \sigma') \land E = e \rightarrow \epsilon
  \lor (C, \sigma) \rightarrow \text{skip} \land \epsilon \in ETrSet_{n}(C', \sigma') \land E = \text{done} \} \).

We define \( ETrSet_{n}(C, \sigma) \) as \( \bigcup_{n} ETrSet_{n}(C, \sigma) \).

3.3 The RGSim Relation

The e-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 w.r.t. parallel compositions. In this section we propose RGSim, which can be viewed as a compositional proof technique that allows us to derive the simple e-trace refinement and then verify the corresponding transformation \( T \).

4.1 The Definition

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

\[
R, G \in P(LState \times LState) \quad \land \quad R, G \in P(HState \times HState) .
\]

The simulation also takes two additional parameters: the step invariant \( \alpha \) and the postcondition \( \gamma \), which are both relations between the low-level and the high-level states.

\[
\alpha, \gamma, \zeta \in P(LState \times HState) .
\]

Before we formally define RGSim in Definition 4, we first introduce the \( \alpha \)-related transitions as follows.

**Definition 3 (\( \alpha \)-Related Transitions).**

\[
\langle R, R \rangle_{\alpha} \equiv \{ ((\sigma, \sigma'), (\Sigma, \Sigma')) \mid (\sigma, \sigma') \in R \land (\Sigma, \Sigma') \in R \land \sigma, \Sigma \in \alpha \land (\sigma', \Sigma') \in \alpha \} .
\]
\[
\begin{align*}
(c, \Sigma) \mapsto (\text{skip}, \Sigma') & & (c, \Sigma) \mapsto (\text{abort}, \Sigma') & & \Sigma \not\in \text{dom}(c) \\
\sum : = \text{true} & & (if \ B \ then \ C_1 \ else \ C_2, \Sigma) \mapsto (C_1, \Sigma) & & \sum : = \text{false} \\
\sum : = \text{true} & & (if \ B \ then \ C_1 \ else \ C_2, \Sigma) \mapsto (C_1, \Sigma) & & \sum : = \perp \\
\sum : = \text{true} & & (while \ B \ do \ C, \Sigma) \mapsto (C; \Sigma) & & \sum : = \text{false} \\
(C; \Sigma) \mapsto (C', \Sigma') & & (C; \Sigma) \rightarrow (C', \Sigma') & & (C, \Sigma) \rightarrow \text{abort} \\
(C_1; C_2, \Sigma) \rightarrow (C_1'; C_2', \Sigma') & & (C_1; C_2', \Sigma) \rightarrow (C_1'; C_2', \Sigma') & & (C_1, \Sigma) \rightarrow \text{abort} \\
(C_1 \parallel \text{skip}, \Sigma) \rightarrow (\text{skip}, \Sigma) & & (C_1 \parallel \text{C}_2, \Sigma) \rightarrow (C_1' \parallel \text{C}_2', \Sigma') & & (C_1\parallel \text{C}_2, \Sigma) \rightarrow \text{abort} \\
\end{align*}
\]

Figure 3. Operational Semantics of the High-Level Language

\[
\begin{align*}
(C, \sigma) \xrightarrow{\alpha} (C, \Sigma) & & (C, \sigma) \xrightarrow{\alpha} (C, \Sigma) \\
\mathcal{G} \xrightarrow{e} \mathcal{G} & & \mathcal{R} \xrightarrow{e} \mathcal{R} \\
(C', \sigma') \xrightarrow{\alpha} (C', \Sigma') & & (C', \sigma') \xrightarrow{\alpha} (C', \Sigma') \\
\end{align*}
\]

Figure 5. Simulation Diagrams of RGSim

\[
\langle \mathcal{R}, \mathcal{R} \rangle_\alpha \text{ represents a set of the } \alpha\text{-related transitions in } \mathcal{R} \text{ and } \mathcal{R}, \\
\text{putting together the corresponding transitions in } \mathcal{R} \text{ and } \mathcal{R} \text{ that can be related by } \alpha, \text{ as illustrated in Figure 4(a).} \langle \mathcal{G}, \mathcal{G} \rangle_\alpha \text{ is defined in the same way.}
\]

Definition 4 (RGSim). Whenever \((C, \sigma, \mathcal{R}, \mathcal{G}) \leq_{\alpha, \gamma} (C, \Sigma, \mathcal{R}, \mathcal{G})\), then \((\sigma, \Sigma) \in \alpha\) and the following are true:

1. if \((C, \sigma) \rightarrow (C', \sigma')\), then there exist \(C'\) and \(\Sigma'\) such that \((C, \Sigma) \rightarrow^* (C', \Sigma')\), \((\sigma, \sigma'), (\Sigma, \Sigma') \in \langle \mathcal{G}, \mathcal{G}^* \rangle_\alpha\) and \((C', \sigma', \mathcal{R}, \mathcal{G}) \leq_{\gamma, \alpha} (C', \Sigma', \mathcal{R}, \mathcal{G})\);

2. if \((C, \sigma) \xrightarrow{c} (C', \sigma')\), then there exist \(C'\) and \(\Sigma'\) such that \((C, \Sigma) \xrightarrow{c}^* (C', \Sigma')\), \((\sigma, \sigma'), (\Sigma, \Sigma') \in \langle \mathcal{G}, \mathcal{G}^* \rangle_\alpha\) and \((C', \sigma', \mathcal{R}, \mathcal{G}) \leq_{\gamma, \alpha} (C', \Sigma', \mathcal{R}, \mathcal{G})\);

3. if \(C = \text{skip}\), then there exists \(\Sigma'\) such that \((C, \Sigma) \rightarrow^* (\text{skip}, \Sigma')\), \((\sigma, \sigma'), (\Sigma, \Sigma') \in \langle \mathcal{G}, \mathcal{G}^* \rangle_\alpha\), \((\sigma, \Sigma') \in \gamma\) and \(\gamma \subseteq \alpha\);

4. if \((C, \sigma) \rightarrow \text{abort}\), then \((C, \Sigma) \rightarrow^* \text{abort}\); 

5. if \((\sigma, \sigma'), (\Sigma, \Sigma') \in \langle \mathcal{R}, \mathcal{R} \rangle_\alpha\), then \((C, \sigma', \mathcal{R}, \mathcal{G}) \leq_{\gamma, \alpha} (C, \Sigma', \mathcal{R}, \mathcal{G})\).

Then, \((C, \sigma, \mathcal{R}, \mathcal{G}) \leq_{\alpha, \gamma, \alpha} (C, \sigma, \mathcal{R}, \mathcal{G})\) iff for all \(\sigma\) and \(\Sigma\), if \((\sigma, \Sigma) \in \alpha\), then \((C, \sigma, \mathcal{R}, \mathcal{G}) \leq_{\alpha, \gamma} (C, \Sigma, \mathcal{R}, \mathcal{G})\).

The precondition \(\gamma\) is used to relate the initial states \(\sigma\) and \(\Sigma\).

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

- Starting from \(\alpha\)-related states, each step of \(C\) corresponds to zero or multiple steps of \(\mathcal{C}\), and the resulting states are \(\alpha\)-related too. If an external event is produced in the step of \(C\), the same event should be produced by \(\mathcal{C}\). We show the simulation diagram with events generated by the program steps in Figure 5(a), where solid lines denote hypotheses and dashed lines denote conclusions, following Leroy’s notations [19].

- The \(\alpha\) 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 \(\alpha\) 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 \(\mathcal{C}\) need to be in \(\langle \mathcal{G}, \mathcal{G}^* \rangle_\alpha\). That is, for each step of \(\mathcal{C}\), its state transition should satisfy the guarantee \(\mathcal{G}\), and the corresponding transition made by the multiple steps of \(\mathcal{C}\) should be in the transitive closure of \(\mathcal{G}\). The guarantees are abstractions of the programs’ behaviors. As we will show later in the PAR rule in Figure 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 \(\mathcal{C}\) to be in \(\mathcal{G}\), although we could do so. This is because we only care about the coarse-grained behaviors (with mumbling) of the source that are used to simulate the target. We will explain more by the example 4.1 in Section 4.2.

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

- \(C\) is not safe only if \(\mathcal{C}\) is not safe either. This means the transformation should not make a safe high-level program unsafe at the low level.

- Whatever the low-level environment \(\mathcal{R}\) and the high-level one \(\mathcal{R}\) do, as long as the state transitions are \(\alpha\)-related, they should not affect the simulation between \(C\) and \(\mathcal{C}\), as shown in Figure 5(b). Here a step in \(\mathcal{R}\) may correspond to zero or multiple steps of \(\mathcal{R}\). Note that different from the program steps, for the environment steps we do not require each step of \(\mathcal{R}\) to correspond to zero or multiple steps of \(\mathcal{R}\). On the other hand, only requiring that \(\mathcal{R}\) be simulated by \(\mathcal{R}\) is not sufficient for parallel compositionality, which we will explain later in Section 4.2.
Definition 7 (Stability). \( \text{Sta}(\zeta, \Lambda) \) holds iff for all \( \sigma, \sigma', \Sigma \) and \( \Sigma' \), if \( (\sigma, \Sigma) \in \zeta \) and \( (\sigma', \Sigma') \in \Lambda \), then \( (\sigma', \Sigma') \in \zeta \).

Usually we need \( \text{Sta}(\zeta, (\mathcal{R}, \mathcal{R}^+)_{\alpha}) \), which says whenever \( \zeta \) holds initially and \( \mathcal{R} \) and \( \mathcal{R}^+ \) perform related actions, the resulting states still satisfy \( \zeta \). By unfolding \( \langle \mathcal{R}, \mathcal{R}^+ \rangle_{\alpha} \), we could see that \( \alpha \) itself is stable w.r.t. any \( \alpha \)-related transitions, i.e., \( \text{Sta}(\alpha, (\mathcal{R}, \mathcal{R}^+)_{\alpha}) \).

Another simple example is given below, where both environments could increment \( x \) and the unary unstable assertion \( \{x \geq 0\} \) is lifted to the relation \( \zeta \):

\[
\zeta \equiv \{ (\sigma, \Sigma) \mid \sigma(x) = \Sigma(x) \geq 0 \} \quad \alpha \equiv \{ (\sigma, \Sigma) \mid \sigma(x) = \Sigma(x) \}
\]

We can prove \( \text{Sta}(\zeta, (\mathcal{R}, \mathcal{R}^+)_{\alpha}) \). Stability of the pre- and post-conditions under the environments' interference is assumed to be an implicit side-condition at every proof rule, e.g., we assume \( \text{Sta}(\zeta, (\mathcal{R}, \mathcal{R}^+)_{\alpha}) \) in the \textsc{skip} rule. We also require implicitly that the reifies and guarantees are closed over identity transitions, since stuttering steps will not affect observable event traces.

In Figure 7, the rules \textsc{skip}, \textsc{seq}, \textsc{if} and \textsc{while} reveal a high degree of similarity to the corresponding inference rules in Hoare logic. In the \textsc{seq} rule, \( \gamma \) serves as the postcondition of \( C_1 \) and \( C_2 \) and the precondition of \( C_2 \) and \( C_3 \) at the same time. The \textsc{if} rule requires the boolean conditions of both sides to be evaluated to the same value under the precondition \( \zeta \). We give the definitions of the sets \( \mathcal{B} \equiv \mathcal{B} \land \mathcal{B} \) in Figure 8. The rule also requires the precondition \( \zeta \) to imply the step invariant \( \alpha \). In the \textsc{while} rule, the \( \gamma \) relation is viewed as a loop invariant preserved at the loop entry point, which needs to ensure \( \mathcal{B} \equiv \mathcal{B} \).

Parallel compositionality. The \textsc{par} rule shows parallel compositionality of \textsc{RgSim}. The interference constraints say that two threads can be composed in parallel if one thread’s guarantee implies the reify 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 \textsc{RgSim} does not require every step of the high-level program to be in its guarantee (see the first two conditions in Definition 4), this relaxation does not affect the parallel compositionality. This is because the target could have less behaviors than the source. To let \( C_1 \parallel C_2 \) simulate \( C_1 \parallel C_2 \), we only need a subset of the interleavings of \( C_1 \) and \( C_2 \) to simulate those of \( C_1 \) and \( C_2 \). Thus the high-level relays 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

\[
(x: x+2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha, \zeta, \gamma} (x: x+1; x: x+1, \mathcal{R}, \mathcal{G}), \tag{4.1}
\]

where the relays and the guarantees say \( x \) can be increased by 2 and \( \alpha, \zeta \) and \( \gamma \) relate \( x \) of the two sides:

\[
\begin{align*}
\mathcal{R} & \equiv \{ (\sigma, \sigma') \mid \sigma' = \sigma + \sigma = \{x \wedge \sigma(x) + 1\} \} \setminus \mathcal{R} \setminus \{ (x: x+1; x: x+1, \mathcal{R}, \mathcal{G}) \}; \\
\mathcal{R} & \equiv \{ (\Sigma, \Sigma') \mid \Sigma' = \Sigma \vee \Sigma' = \{x \wedge \Sigma(x) + 1\} \}; \\
\mathcal{G} & \equiv \{ (\sigma, \Sigma) \mid |\sigma(x) = \Sigma(x)| \}.
\end{align*}
\]

Note that the high-level program is actually finer-grained than its guarantee, but to prove (4.1) we only need the execution in which it goes two steps to the end without interference from its environment. Also we can prove \( \text{print}(x), \mathcal{R}, \mathcal{G}) \preceq_{\alpha, \zeta, \gamma} (\text{print}(x), \mathcal{R}, \mathcal{G}) \) by the \textsc{par} rule, we get

\[
(x: x+2 \mid \text{print}(x), \mathcal{R}, \mathcal{G}) \preceq_{\alpha, \zeta, \gamma} (x: x+1; x: x+1 \mid \text{print}(x), \mathcal{R}, \mathcal{G}),
\]

which does not violate the natural meaning of refinements. That is, 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 \( \cal R \) be simulated by \( \cal R' \).

If \( (\sigma, \sigma') \in \cal R \), then there exists \( \Sigma' \) such that \((\Sigma, \Sigma') \in \cal R' \) and \((C, \sigma', \cal R, \cal G) \preceq_{\alpha, \zeta, (\gamma, \eta)} (C, \cal R, \cal G) \). (4.2)

We refer to this modified simulation definition as \( \preceq' \). Unfortunately, \( \preceq' \) does not have parallel compositionality. As a counterexample, if the invariant \( \alpha \) says the left-side \( x \) is not greater than the right-side \( x \) and the precondition \( \zeta \) requires \( x \) of the two sides are equal, i.e.,

\[
\alpha \triangleq \{ (\sigma, \Sigma) \mid \sigma(x) \leq \Sigma(x) \},
\]

we could prove the following:

\[
(x:x + 1, \text{Id}, \text{True}) \preceq' \alpha \leq (x:x + 2, \text{Id}, \text{True}) \\
(\text{print}(x), \text{True}, \text{Id}) \preceq' \alpha \leq (\text{print}(x), \text{True}, \text{Id})
\]

Here we use Id and True (defined in Figure 3) 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:

\[
(x:x + 1 \mid \text{print}(x), \text{True}, \text{Id}) \preceq_{\alpha, \zeta, \alpha} (x:x + 2 \mid \text{print}(x), \text{True}, \text{Id})
\]

This is because the rely \( \cal R \) (or \( \cal R' \)) is an abstraction of all the permitted behaviors in the environment of a thread. But a concrete sibling thread that runs in parallel may produce less transitions than \( \cal R \) (or \( \cal R' \)). To obtain parallel compositionality, we need to ensure that the simulation holds for all concrete sibling threads. With our definition \( \preceq \), the refinement \((\text{print}(x), \text{True}, \text{Id}) \preceq_{\alpha, \zeta, \alpha} (\text{print}(x), \text{True}, \text{Id})\) is not provable because, after the environments’ \( \alpha \)-related transitions, the target may print a value smaller than the one printed by the source.

Other rules. We also develop some other useful rules about RGSim. For example, the \( \text{STREN-}\alpha \) rule allows us to replace the invariant \( \alpha \) by a stronger invariant \( \alpha' \). We need to check that \( \alpha' \) is indeed an invariant preserved by the related program steps, i.e., \( \text{Sta}(\alpha', (G, G')) \). Holistically, the \( \text{WEAKEN-}\alpha \) rule requires \( \alpha \) to be preserved by environment steps related by the weaker invariant \( \alpha' \). As usual, the pre/post conditions, the relies and the guarantees can be strengthened or weakened by the \( \text{CONSEQ} \) rule.

The \( \text{FRAME} \) rule allows us to use local specifications. When verifying the simulation between \( C \) and \( \cal C \), we need to only talk about the locally-used resource in \( \alpha \), \( \zeta \), and \( \gamma \), and the local relies and guarantees \( R, G, \cal R \), and \( \cal G \). Then the proof can be reused in contexts where some extra resource \( \eta \) is used, and the accesses of it respect the invariant \( \beta \) and \( R_1, G_1, \cal R_1 \). We give the auxiliary definitions in Figure 3. The disjoint union \( \mathcal{U} \) between states is lifted to state pairs. An intuitionistic state relation is monotone \( w.r.t. \) the extension of states. The disjointness \( \eta \neq \alpha \) says that any state pair satisfying both \( \eta \) and \( \alpha \) can be split into two disjoint state pairs satisfying \( \eta \) and \( \alpha \) respectively. For example, let \( \eta \triangleq \{ (\sigma, \Sigma) \mid \sigma(\gamma) = \Sigma(\gamma) \} \) and \( \eta \neq (\{ (\sigma, \Sigma) \mid \sigma(\gamma) = \Sigma(\gamma) \} \) then both \( \eta \) and \( \alpha \) are intuitionistic and \( \eta \neq \alpha \) holds. We also require \( \eta \) to be stable under interference from the programs (i.e., the programs do not change the extra resource) and the extra environments. We use \( \eta \neq (\zeta, \gamma, \alpha) \) as a shorthand for \( \eta \neq \zeta \land \eta \neq \gamma \lor \eta \neq \alpha \). Similar representations are used in this rule.

Finally, the transitivity rule \( \text{TRANS} \) allows us to verify a transformation by using an intermediate level as a bridge. The intermediate environment \( R_M \) should be chosen with caution so that the \( (\beta \circ \alpha) \)-related transitions can be decomposed into \( \beta \)-related and
\(\alpha\)-related transitions, as illustrated in Figure \[2\](b). Here \(\alpha\) defines the composition of two relations and `isMldOf` defines the side condition over the environments, as shown in Figure \[6\]. We use \(\theta\) for a middle-level state.

We give all the soundness proofs in Appendix \[A\] and \[B\]. The proofs \[22\] are also mechanized in the Coq proof assistant \[10\].

**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 refinement \[4.3\] over closed programs assumes identity environments, making the interference constraints in the PAR rule unsatisfiable. This confirms the observation in Section \[2.1\] that the sequential refinement loses parallel compositionality.

\[
(C, \text{Id}, \text{True}) \preceq_{\alpha, \zeta \kappa} (C, \text{Id}, \text{True}) \tag{4.3}
\]

The refinement \[4.3\] assumes arbitrary environments, which makes the interference constraints in the PAR rule trivially true. But this assumption is too strong: usually \(4.3\) cannot be satisfied in practice.

\[
(C, \text{True}, \text{True}) \preceq_{\alpha, \zeta \kappa} (C, \text{True}, \text{True}) \tag{4.4}
\]

### 4.3 A Simple Example

Below we give a simple example to illustrate the use of RGSim and its parallel compositionality in verifying concurrent program transformations. The high-level program \(C_1 \parallel C_2\) is transformed to \(C_1 \parallel C_2\), using a lock to synchronize the accesses of the shared variable \(x\). We aim to prove \(C_1 \parallel C_2 \preceq_T C_1 \parallel C_2\). 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.

\[
\text{print}(x); \parallel x := x + 2; \\
\text{lock}(1); \parallel \text{lock}(1);
\]

\[
\text{print}(x); \parallel x := x + 1; x := x + 1;
\]

\[
\text{unlock}(1); \parallel \langle \text{unlock}(1); x := x \rangle
\]

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.

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

\[
\alpha \triangleq \{ (\varSigma, \varTheta) | \varTheta(X) = \varSigma(x) \land (\varTheta(1) = 0 \implies \varTheta(x) = \varSigma(x)) \}
\]

We let the pre- and post-conditions be \(\alpha\) as well.

The high-level threads can be executed in arbitrary environments with arbitrary guarantees: \(\mathcal{R} = \mathcal{G} \triangleq \text{True}\). The transformation uses the lock to protect every access of \(x\), thus the low-level relies and guarantees are not arbitrary:

\[
\mathcal{R} \triangleq \{ (\varSigma, \varTheta) | \varTheta(1) = \text{cid} \implies \varTheta(x) = \varTheta'(x) \land \varSigma(x) = \varSigma'(x) \land \varTheta(1) = \varTheta'(1) \};
\]

\[
\mathcal{G} \triangleq \{ (\varSigma, \varTheta) | \varTheta' = \varSigma \lor \varTheta(1) = 0 \land \varTheta = \varSigma(1 \leadsto \text{cid}) \lor \varTheta(1) = \text{cid} \land \varTheta' = \varSigma(1 \leadsto \text{cid}) \lor \varTheta(1) = \text{cid} \land \varTheta = \varSigma(1 \leadsto \text{cid}) \}.
\]

Every low-level thread guarantees that it updates \(x\) only when the lock is acquired. Its environment cannot update \(x\) or 1 if the current thread holds the lock. Here \(\text{cid}\) is the identifier of the current thread. When acquired, the lock holds the id of the owner thread.

Following the definition, we can prove \((C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha, \zeta \kappa} (C_1, \mathcal{R}, \mathcal{G})\) and \((C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha, \zeta \kappa} (C_2, \mathcal{R}, \mathcal{G})\). By applying the PAR rule and from the soundness of RGSim (Corollary \[6\]), we know \(C_1 \parallel C_2 \preceq_T C_1 \parallel C_2\) holds for any \(T\) that respects \(\alpha\).

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 \(C_1 \parallel C_2\). This does not indicate the unsoundness of our PAR rule (which is sound!). The reason is that \(x\) might have different values on the two levels after the environments’ \(\alpha\)-related transitions, so that we cannot have \((\text{print}(x), \mathcal{R}, \mathcal{G}) \preceq_{\alpha, \zeta \kappa} (\text{print}(x), \mathcal{R}, \mathcal{G})\) with the current definitions of \(\alpha, \mathcal{R}\) and \(\mathcal{G}\), even though the code of the two sides are syntactically identical.

**More discussions.** RGSim ensures that the target program preserves safety properties (including the partial correctness) of the source, but allows a terminating source program to be transformed to a target having infinite silent steps. In the above example, this allows the low-level programs 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.

In the next three sections, we show more serious examples to demonstrate the applicability of RGSim.

### 5. Relational Reasoning about Optimizations

As a general correctness notion of concurrent program transformations, RGSim establishes a relational approach to justify compiler optimizations on concurrent programs. Below we adapt Benton’s work \[3\] on sequential optimizations to the concurrent setting.

#### 5.1 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.

**Reflexivity**

\[
\mathcal{R}, \mathcal{G} \vdash \{ p \} \mathcal{C}(q) \\
\{ (C, \mathcal{R}, \mathcal{G}) \} \preceq_{\text{id} \times [p \times [q]]} \{ (C, \mathcal{R}, \mathcal{G}) \}
\]

For the code which is unchanged after optimizations, we can prove the simulation by the judgment in the Rely-Guarantee logic. Here we use \([p]\) to mean the states of the two sides are the same and satisfy the predicate \(p\). That is, \([p] \triangleq \{ (\varSigma, \varTheta) | p \varTheta \} \).

**Sequential skip Law**
(C_1, R_1, G_1) \preceq_{\alpha \lor \gamma} (C_2, R_2, G_2)

(skip; C_1, R_1, G_1) \preceq_{\alpha \lor \gamma} (C_2, R_2, G_2)

(C_1; \text{skip}, R_1, G_1) \preceq_{\alpha \lor \gamma} (C_2, R_2, G_2)

(C_1, R_1, G_1) \preceq_{\alpha \lor \gamma} (C_2, R_2, G_2)

That is, skips could be arbitrarily introduced and eliminated.

**Common Branch**
\[
\begin{align*}
\forall \sigma_1, \sigma_2. & \quad (\sigma_1, \sigma_2) \in \zeta \implies B \sigma_2 \neq \perp \\
(C, R, G) & \preceq_{\alpha \lor \gamma} (C_1, R', G') \quad \zeta_1 = (\zeta \cap (\text{true} \land B)) \\
(C, R, G) & \preceq_{\alpha \lor \gamma} (C_2, R', G') \quad \zeta_2 = (\zeta \cap (\text{true} \land \neg B)) \\
(C, R, G) & \preceq_{\alpha \lor \gamma} \text{if} (B) \text{ else } C_2, R', G'
\end{align*}
\]

This rule says that, 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**
\[
\begin{align*}
(C, R, G) & \preceq_{\alpha \lor \gamma} (C_1, R', G') \\
(C, R, G) & \preceq_{\alpha \lor \gamma} (C_2, R', G') \\
(C, R, G) & \preceq_{\alpha \lor \gamma} \text{if} (B) \text{ else } C_2, R', G'
\end{align*}
\]

This rule can be derived from the Common-Branch rule.

**Dead While**
\[
\begin{align*}
\zeta & = (\zeta \cap (\text{true} \land \neg B)) \\
\zeta & \subseteq \alpha \\
\text{Sta}(\zeta, (R_1, R_2)) & \preceq_{\alpha \lor \gamma} \text{while} (B) \text{ else } C_2, R', G'
\end{align*}
\]

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 Unrolling**
\[
\begin{align*}
\text{while} (B) \{ C \} & \preceq_{\alpha \lor \gamma} \text{while} (B) \{ C \}, R_2, G_2 \\
\text{if} (B) \{ C \} & \text{else} \text{skip}, R_1, G_1 \preceq_{\alpha \lor \gamma} \text{while} (B) \{ C \}, R_2, G_2 \\
\text{while} (B) \{ C \} & \preceq_{\alpha \lor \gamma} \text{while} (B) \{ C \}, R_2, G_2 \\
\text{while} (B) \{ C \} & \text{else} \text{skip}, R_1, G_1 \preceq_{\alpha \lor \gamma} \text{while} (B) \{ C \}, R_2, G_2
\end{align*}
\]

We show two ways to unroll the while-loop, ensuring semantics preservation in the concurrent setting.

**Dead Code Elimination**
\[
\begin{align*}
\text{Sta}(\zeta, (R_1, R_2)) & \preceq_{\alpha \lor \gamma} (C, \text{Id}, G) \\
\text{Sta}(\zeta, (R_1, R_2)) & \preceq_{\alpha \lor \gamma} (C, \text{Id}, G)
\end{align*}
\]

Intuitively (\text{skip}, \text{ld}, \text{id}) \preceq_{\alpha \lor \gamma} (C, \text{ld}, G) says that the code \( C \) can be eliminated in a sequential context where the initial and the final states satisfy \( \zeta \) and \( \gamma \) respectively. If both \( \zeta \) and \( \gamma \) are stable \( \text{w.r.t.} \) the interference from the environments \( R_1 \) and \( R_2 \), then the code \( C \) can be eliminated in such a parallel context as well.

**Redundancy Introduction**
\[
\begin{align*}
(c, \text{ld}, G) & \preceq_{\alpha \lor \gamma} (\text{skip}, \text{ld}, \text{id}) \\
(c, R_1, G) & \preceq_{\alpha \lor \gamma} (\text{skip}, R_2, G)
\end{align*}
\]

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 \( \text{w.r.t.} \) the environments. Note that here \( c \) is a single instruction, because we should consider the interference from the environments at every intermediate state when introducing a sequence of redundant instructions.

### 5.2 An Example of Invariant Hoisting

With these rules, we can prove the correctness of many traditional compiler optimizations performed on concurrent programs in appropriate contexts. Here we only give a small example of hoisting loop invariants. More optimization examples (e.g., strength reduction and induction variable elimination) can be found in Appendix D.

**Target Code (C)**
\[
\begin{align*}
\text{local } t; \\
\text{while}\{ i < n \} \{ \\
\quad t := x + 1; \\
\quad i := i + t; \\
\}
\end{align*}
\]

**Source Code (C)**
\[
\begin{align*}
\text{local } t; \\
\text{while}\{ i < n \} \{ \\
\quad t := x + 1; \\
\quad i := i + t; \\
\}
\end{align*}
\]

When we do not care about the final value of \( t \), it’s not difficult to prove that the optimized code \( C_1 \) preserves the sequential behaviors of the source \( C \). But in a concurrent setting, safely hoisting the invariant code \( t := x + 1 \) also requires that the environment should not update \( x \) nor \( t \).

\[
R \triangleq \{ (\sigma, \sigma') \mid \sigma(x) = \sigma'(x) \land \sigma(t) = \sigma'(t) \}
\]

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 \( \alpha \) can be defined as:

\[
\alpha \triangleq \{ (\sigma, \sigma') \mid \sigma(i) = \sigma'(i) \land \sigma(n) = \sigma'(n) \land \sigma(x) = \sigma'(x) \}
\]

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

\[
(C_1, R, \text{True}) \preceq_{\alpha \lor \gamma \lor \delta} (C, R, \text{True})
\]

We could prove [5.1] 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 Dead-Code-Elimination and Redundancy-Introduction rules:

\[
\begin{align*}
\text{while} \{ x := 1, R, \text{True} \} & \preceq_{\alpha \lor \gamma \lor \delta} (\text{skip}, R, \text{True}) \\
\text{skip}, R, \text{True} & \preceq_{\alpha \lor \gamma \lor \delta} (\text{skip}, \text{skip}, R, \text{True})
\end{align*}
\]

where \( \gamma \) and \( \eta \) specify the states at the specific program points:

\[
\begin{align*}
\gamma & \triangleq \alpha \cap \{ \sigma \mid \sigma(t) = \sigma(x) + 1 \} \\
\eta & \triangleq \gamma \cap \{ \sigma \mid \sigma(t) = \sigma(x) + 1 \}
\end{align*}
\]

After adding \( \text{skip} \) to \( C_1 \) and \( C \) to make them the same “shape”, we can prove the simulation by the compositionality rules \( \text{SEQ} \) and \( \text{WHILE} \). Finally, we remove all the \( \text{skip} \)s and conclude [5.1], i.e., the correctness of the optimization in appropriate contexts. Since the relies 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.

### 6. Proving Atomicity of 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. RGSim gives us a refinement-based proof method to verify the atomicity of implementations of the object: we can define abstract atomic operations in a high-level language as specifications, and prove the concrete fine-grained implementations refine the corresponding atomic operations when executed in appropriate environments.
For instance, in Figure 8(a) we define two atomic set operations, `ADD(e)` and `RMV(e)`. Figure 8(b) gives a concrete implementation of the set object using a lock-coupling list. Partial correctness and atomicity of the algorithm has been verified before [28, 29]. Here we show that its atomicity can also be verified using our RGSim by proving the low-level methods refine the corresponding abstract operations. We will discuss the key difference between the previous proofs and ours in Section 8.

We first take the generic languages in Figure 3 and instantiate the high-level program states below.

\[
\begin{align*}
(HMem) & \quad M_s, M_l \in (\text{Loc} \cup \text{PVar}) \rightarrow \text{HVal} \\
(HThds) & \quad \Pi \in \text{ThrdID} \rightarrow \text{HMemb} \\
(HSstate) & \quad \Sigma \in HThds \times \text{HMemb}
\end{align*}
\]

The state consists of shared memory \( M_s \) (where the object resides) and a thread pool \( \Pi \), which is a mapping from thread identifiers (\( t \in \text{ThrdID} \)) to their memory \( M_l \). The low-level state \( \sigma \) is defined similarly. We use \( m_s, m_l \) and \( \pi \) to represent the low-level shared memory, thread-local memory and the thread pool respectively.

To allow ownership transfer between the shared memory and thread-local memory, we use `atom` (\( C \) or \( C_s \) at the low level) to convert the shared memory to local and then execute \( C \) (or \( C_s \)) atomically. Following RGSep [29], an abstract transition \( \alpha \in \mathcal{T}(\text{HMemb} \times \text{HMemb}) \) or \( \alpha \in \mathcal{T}(\text{LMem} \times \text{LMem}) \) is used to specify the effects of the atomic operation over the shared memory, which allows us to split the resulting state back into shared and local when we exit the atomic blocks. The atomic blocks are instantiations of the generic primitive operations \( c \) (or \( c_s \)) in Figure 2. We omit the annotations \( \alpha \) and \( A \) in Figure 8 which are the same as the corresponding guarantees in Figure 11 as we will explain below. Formal presentations of the high-level and low-level languages and the operational semantics are given in Figures 9 and 10 respectively.

\[\text{(HStmts)}\]
\[
\begin{align*}
C & \ ::= \text{skip} \mid c \mid \text{atom}(C)_A \mid C_1; C_2 \\
& \quad \mid \text{if}(B) \ C_1 \text{else } C_2 \\
& \quad \mid \text{while}(B) \{C\}
\end{align*}
\]

\[\text{(VModel)}\]
\[
\begin{align*}
W & \ ::= t_1; C_1 \cdots | t_n; C_n \\
\text{HMem} & \quad M_s, M_l \in (\text{Loc} \cup \text{PVar}) \rightarrow \text{HVal} \\
\text{HThds} & \quad \Pi \in \text{ThrdID} \rightarrow \text{HMem} \\
\text{HSstate} & \quad \Sigma \in HThds \times \text{HMem}
\end{align*}
\]

\[\text{HAtomG}\]
\[
\alpha \in \mathcal{T}(\text{HMemb} \times \text{HMem})
\]

\[\text{(HAtomG)}\]
\[
\alpha \in \mathcal{T}(\text{HMem} \times \text{HMem})
\]

\[\text{(HAtomG)}\]
\[
\alpha \in \mathcal{T}(\text{LMem} \times \text{LMem})
\]

\[\text{(HAtomG)}\]
\[
\alpha \in \mathcal{T}(\text{HMemb} \times \text{HMem})
\]

\[\text{Figure 9. The Languages for Concurrent Objects}\]

In Figure 8 the abstract set is implemented by an ordered singly-linked list pointed to by a shared variable `Head`, with two sentinel nodes at the two ends of the list containing the values `MIN_VAL` and `MAX_VAL` respectively. Each list node is associated with a lock. To verify the list uses "hand-over-hand" locking: the lock on one node is not released until its successor is locked. `add(e)` inserts a new node with value \( e \) in the appropriate position while holding the lock of its predecessor. `rmv(e)` redirects the predecessor's pointer while both the node to be removed and its predecessor are locked.

We define the \( \alpha \) relation, the guarantees and the relies in Figure 11. The predicate \( m_s \equiv \text{list}(x, A) \) represents a singly-linked list in the shared memory \( m_s \) at the location \( x \), whose values form the sequence \( A \). Then the mapping `shared_map` between the low-level and the high-level shared memory is defined by only concerning about the value sequence on the list: the concrete list should be sorted and its elements constitute the abstract set. For a thread \( t \)’s local memory of the two levels, we require that the values of \( e \) are the same and enough local space is provided for `add(e)` and `rmv(e)`, as defined in the mapping `local_map`. Then \( \alpha \) relates the shared memory by `shared_map` and the local memory of each thread \( t \) by `local_map`.

The atomic actions of the algorithm are specified by `Glock`, `Gunlock`, `Gadd`, `Grmv` and `glocal` respectively, which are all parameterized with a thread identifier \( t \). For example, `Gadd(t)` says that when holding the locks of the node \( y \) and its predecessor \( x \), we can transfer the node \( y \) from the shared memory to the thread’s local memory. This corresponds to the action performed by the code of line 13 in `rmv(e)`. Every thread \( t \) is executed in the environment that any other thread \( t’ \) can only perform those five actions, as defined in `R(t)`. Similarly, the high-level `G(t)` and `R(t)` are defined according to the abstract `ADD(e)` and `RMV(e)`. The relies and guarantees are almost the same as those in the proofs in RGSep [28].

We can prove that for any thread \( t \), the following hold:

\[
(\text{t.add}(e), R(t), G(t)) \preceq \alpha \cdot G\cdot (t, A, G(t))\]

\[
(\text{t.rmv}(e), R(t), G(t)) \preceq \alpha \cdot G\cdot (t, B, G(t))
\]

Detailed proofs are given in Appendix D.
common divisors).

To prove the atomicity of other fine-grained algorithms, includ-

Figure 10. Selected Operational Semantics Rules for the High-Level Language of Concurrent Objects

By the compositionality and the soundness of RGSim, we know that the fine-grained operations (under the parallel environment $R$) are simulated by the corresponding atomic operations (under the high-level environment $\mathbb{R}$), while $R$ and $\mathbb{R}$ say all accesses to the set must be done through the add and remove operations. This gives us the atomicity of the concurrent implementation of the set object.

More examples. In Appendix we also show the use of RGSim to prove the atomicity of other fine-grained algorithms, including the non-blocking concurrent counter [27], Treiber’s stack algorithm [28], and a concurrent GCD algorithm (calculating greatest common divisors).

7. Verifying Concurrent Garbage Collectors

In this section, we explain in detail how to reduce the problem of verifying concurrent garbage collectors to transformation verification, and use RGSim to develop a general GC verification framework. We apply the framework to prove the correctness of the Boehm et al. concurrent GC algorithm [7].

7.1 Correctness of Concurrent GCs

A concurrent GC is executed by a dedicate 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, write barriers and allocation barriers).

The GC thread and the barriers constitute a concurrent garbage collecting system, which 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 make the mutators happy 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 garbage collecting system as a transformation \( T \) 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,

\[
T(t_{gc} \cdot AbsGC \parallel t_1 \cdot C_1 \parallel \ldots \parallel t_n \cdot C_n) \triangleq t_{gc} \cdot C_{gc} \parallel t_1 \cdot T(C_1) \parallel \ldots \parallel t_n \cdot T(C_n),
\]

where \( T(C) \) simply translates some memory access instructions in \( C \) into the corresponding barriers, and leaves the rest unchanged.

Then we reduce the correctness of the concurrent garbage collecting system to \( \text{Correct}(T) \), saying that any mutator program will not have unexpected behaviors when executed using this system.

### 7.2 A General Framework

The compositionality of RGSim allows us to develop a general framework to prove \( \text{Correct}(T) \), which cannot be done by monolithic proof methods. By the parallel compositionality of RGSim (the PAR rule in Figure 7), we can decompose the refinement proofs into proofs for the GC thread and each mutator thread.

**Verifying the GC.** The semantics of the abstract GC thread can be defined by a binary state predicate \( \text{AbsGCStep} \):

\[
(\Sigma, \Sigma') \in \text{AbsGCStep} \Rightarrow (t_{gc} \cdot \text{AbsGC}, \Sigma) \rightarrow (t_{gc} \cdot \text{AbsGC}, \Sigma')
\]

That is, the abstract GC thread always makes \( \text{AbsGCStep} \) to change the high-level state. We can choose different \( \text{AbsGCStep} \) for different GCs, but usually \( \text{AbsGCStep} \) 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 \( \mathcal{R}_{ge} \cdot \mathcal{G}_{gc} \vdash \{ p_{gc} \} C_{gc}[\text{false}] \) in a standard Rely-Guarantee logic with proper \( \mathcal{R}_{gc}, \mathcal{G}_{gc}, p_{gc} \) and \( q_{gc} \), as long as \( q_{gc} \) is a concrete representation of \( \text{AbsGCStep} \). The judgment says given an initial state satisfying the precondition \( p_{gc} \), if the environment’s behaviors satisfy \( \mathcal{R}_{gc} \), then each step of \( C_{gc} \) satisfies \( \mathcal{G}_{gc} \), 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 \( \text{false} \). \( \mathcal{G}_{gc} \) and \( p_{gc} \) should be provided by the verifier, where \( p_{gc} \) needs to be general enough that can be satisfied by any possible low-level initial state. \( \mathcal{R}_{ge} \) encodes the possible behaviors of mutators, which can be derived, as we will show below.

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

\[
\mathcal{G}(t) \triangleq \bigcup \mathcal{G}(t.c) \quad \mathcal{G}(t) \triangleq \bigcup \mathcal{G}(t,T(c)).
\]

Its relies should include all the possible guarantees made by other threads, and the GC’s abstract and concrete behaviors respectively:

\[
\begin{align*}
\mathcal{R}(t) & \triangleq \text{AbsGCStep} \cup \bigcup_{t' \neq t} \mathcal{G}(t') \quad \mathcal{R}(t) \triangleq \mathcal{G}_{gc} \cup \bigcup_{t' \neq t} \mathcal{G}(t').
\end{align*}
\]

The \( \mathcal{R}_{ge} \) used to verify the GC code can now be defined below:

\[
\mathcal{R}_{ge} \triangleq \bigcup \mathcal{G}(t).
\]

The refinement proof also needs definitions of binary \( \alpha, \zeta \) and \( \gamma \) relations. The invariant \( \alpha \) 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 \( \Sigma \) can be mapped to a low-level state \( \sigma \) 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. For each mutator thread \( t \), the relations \( \zeta(t) \) and \( \gamma(t) \) 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 \( \gamma(t) \) be the same as \( \zeta(t) \) to support sequential compositions. We require \( \text{InitRel}(\zeta(t)) \) (see Figure 9, i.e., \( \zeta(t) \) holds over the initial states. In addition, the target and the source boolean expressions should be evaluated to the same value under related states, as required in the \text{IF} and \text{WHILE} rules in Figure 7.

\[
\text{Good}(t) \triangleq \text{InitRel}(\zeta(t)) \land \forall B. \zeta(t) \subseteq (\mathcal{G}(B) \equiv \mathcal{B}(B)).
\]

**Theorem 8 (Verifying Concurrent Garbage Collecting Systems).** If there exist \( \mathcal{R}_{ge}, \mathcal{G}_{gc}, \mathcal{R}(t), \mathcal{R}(\mathcal{G}), \zeta(t) \) and \( \alpha \) such that (7.1), (7.2), (7.3) and the following hold:

1. (Verification of the GC code)
   \[ \mathcal{R}_{ge} ; \mathcal{G}_{gc} \vdash \{ p_{gc} \} C_{gc}[\text{false}] ; \]
2. (Correctness of \( T \) on mutator instructions)
   \[ \forall c. (t,T(c), \mathcal{R}(t), \mathcal{G}(t)) \preceq_{\alpha,\zeta(t),\gamma(t)} (t.c, \mathcal{R}(t), \mathcal{G}(t)) ; \]
3. (Side Conditions)
   \[ \mathcal{G}_{gc} \circ \alpha^{-1} \subseteq \alpha \circ \gamma \circ (\text{AbsGCStep})^* ; \quad \forall \Sigma. \sigma \in T(\Sigma) \implies p_{gc} \sigma ; \]

then \( \text{Correct}(T) \).

That is, to verify a concurrent garbage collecting system, we need to do the following:

- Define the \( \alpha \) and \( \zeta(t) \) relations, and prove the correctness of \( T \) on high-level primitive instructions. Since \( T \) 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 the barriers that they implement both the source instructions (by RGSim) and the interaction mechanism (shown in their guarantees).
- Find some proper \( \mathcal{G}_{gc} \) and \( p_{gc} \), and verify the GC code by R-G reasoning. We require the GC’s guarantee \( \mathcal{G}_{gc} \) should not contain more behaviors than \( \text{AbsGCStep} \) (the first side condition), and \( \mathcal{G}_{gc} \) can start its execution from any state \( \sigma \) transformed from a high-level one (the second side condition).

The proof of Theorem 8 is given in Appendix C.

### 7.3 Application: Boehm et al. Concurrent GC Algorithm

We illustrate the applications of the framework (Theorem 8) by proving the correctness of a mostly-concurrent mark-sweep garbage collector proposed by Boehm et al. [7]. Variants of the algorithm have been used in practice (e.g., by IBM [2]). Due to the space limit, we only describe the proof sketch here. Details are presented in Appendix D.
{wfstate}

0 Collection() {
1  local stk: Seq(Int);
2
3  while (true) {
4    {wfstate ∨ (ownp(stk) ∧ stk = e)}
5      Initialize();
6      {wfstate ∧ reach_inv) ∨ (ownp(stk) ∧ stk = e)}
7      Trace();
8      {wfstate ∧ reach_inv) ∨ (ownp(stk) ∧ stk = e)}
9      CleanCard();
10     {wfstate ∧ reach_inv) ∨ (ownp(stk) ∧ stk = e)}
11    atomic{
12      ScanRoot();
13      {∃X. (wfstate ∧ reach_trrw_stk(X) ∧ stk_black(X))
14        ∨ (ownp(stk) ∧ stk = X)}
15    }
16  }
17}

{false}

Figure 12. Outline of the GC Code and Proof Sketch

update(x.id, E) { // id ∈ {p1, ..., ptm}
  atomic{ x.id := E; aux := x; }
  atomic{ x.dirty := 1; aux := 0; }
}

Figure 13. The Write Barrier for Boehm et al. GC

Overview of the GC algorithm. The top-level code of the GC thread is shown in Figure 12. In each collection cycle, after an initialization process, the GC enters the concurrent mark-phase (line 4) and traces the objects reachable from the roots (i.e., the mutators’ local pointer variables that may contain references to the heap objects). A mark stack (stk) is used to do a depth-first tracing. 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 by dirtying the objects’ tags (called cards). When the tracing is done, the GC suspends all the mutators and re-traces from the dirty objects that have been marked (called card-cleaning, line 6 and 7). The stop-the-world phase is implemented by atomic{C}. Finally, all the reachable objects are ensured marked and the GC performs the concurrent sweep-phase (line 8), in which unmarked objects are reclaimed. Usually in practice, there is also a concurrent card-cleaning phase (line 5) before the stop-the-world card-cleaning to reduce the pause time. The full GC code CGC is given in Appendix E.2. CGC can use privilege commands to control the mutator threads and manage the heap, e.g., use x := get_root(y) to read all the pointer variables in the thread y’s store and use free(x) to reclaim an object.

The write barrier is shown in Figure 13 where the dirty field is set after modifying the object’s pointer field. Here we use a write-only auxiliary variable aux for each mutator thread to record the current object that the mutator is updating. We add aux only for the purpose of verification, so that we can easily specify the fine-grained property of the write barrier in the guarantees that immediately after updating the pointer field, the thread would do nothing else except setting the corresponding dirty field. The GC does not use read barriers nor allocation barriers.

We first present the high-level and low-level languages and state models in Figures 14 and 15 respectively. See Appendix E.1 for full descriptions of the machine models. The behaviors of the high-level abstract GC thread are defined as follows:

AbsGC

\[
\text{AbsGC} = \{ ((\Pi, H), (\Pi, H')) | \forall l. \text{reachable}(l)(\Pi, H) \Rightarrow H(l) = H'(l) \},
\]

sayng that, the mutator stores and the reachable objects in the heap are remained unmodified. Here reachable(l)(\Pi, H) means the object at the location l is reachable in H from the roots in \Pi.

The transformation. The transformation T is defined in Figure 16. For code, the high-level abstract GC thread is transformed to the GC thread shown in Figure 12. Each instruction x.id := E in mutators is transformed to the write barrier, where id is a pointer field of x. Other instructions and the program structures of mutators are unchanged.
The following transformations are made over initial states.

- First we require the high-level initial state to be well-formed:
  \[ \text{wfstate}(\Pi, H) \triangleq \forall t. \text{reachable}(t)(\Pi, H) \implies \exists l \in \text{dom}(H). \]
  That is, reachable locations cannot be dangling pointers.

- High-level locations are transformed to integers by a bijective function \( \text{Loc2Int} : \text{Loc} \leftrightarrow [0..M] \) such that \( \text{Loc2Int}(\text{nil}) = 0 \).

- 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 \( v^\alpha \) and \( v^\beta \) 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 \( \text{WHITE} \) and \( \text{dirty} \) respectively. Other addresses in the low-level heap domain \([1..M]\) are filled out using unallocated objects whose colors are \( \text{BLUE} \) and all the other fields are initialized by \( 0 \). Here we use \( \text{BLACK} \) and \( \text{WHITE} \) for marked and unmarked objects respectively, and \( \text{BLUE} \) for unallocated memory.

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

To prove Correct(G) in our framework, we apply Theorem 9 to prove the refinement between low-level and high-level mutators, and verify the GC code using a unary Rely-Guarantee-based logic.

**Refinement proofs for mutator instructions.** We first define the \( \alpha \) and \( \zeta \) (t) relations.

\[
\alpha \triangleq \{(\pi \triangleq \{t_{\text{gc}} \to \}, h), (\Pi, H)\} \quad \forall t \in \text{dom}(\Pi), \text{store_map}(\pi(t), \Pi(t)) \land \text{heap_map}(h, H) \land \text{wfstate}(H, H) .
\]

In \( \alpha \), the relation between low-level and high-level stores and heaps are enforced by \text{store_map} and \text{heap_map} respectively. Their definitions reflect the state transformations we describe above, where we consider well-formed states only and use \text{Loc2Int} to relate integers and locations. The difference between \( \alpha \) and \( T \) only lies in that, in \( \alpha \) we do not care about the values of the extra structures which are invisible on the high-level machine (e.g., the GC’s local variables, the color and dirty fields for non-blue objects and all the fields of blue objects) as long as they are valid. We present the formal definition of \( \alpha \) in Figure 17.

For each mutator thread \( t \), the \( \zeta(t) \) relation enforced at the beginning and the end of each transformation unit (each high-level instruction) is stronger than \( \alpha \). It requires that the value of the auxiliary variable \( \text{aux} \) (see Figure 15) be a null pointer \( 0^\phi \):

\[
\zeta(t) \triangleq \alpha \cap \{(\pi, h), (\Pi, H)\} \land \pi(t)(\text{aux}) = 0^\phi .
\]

As shown in Figure 18, the guarantees of the high-level mutator instructions and the transformed code are defined following their operational semantics. We can prove correctness of the write barrier:

\[
(t.\text{update}(x, i.d, E), \mathcal{R}(t), G^t_{\text{write_barrier}}) \preceq_{\alpha, \zeta(t)} (t.\text{update}(x, i.d, E), \mathcal{R}(t), G^t_{\text{write_barrier}}) \quad \text{(t.}(x := \text{new}), \mathcal{R}(t), G_{\text{new}}^t \cup G_{\text{async,pt}}^t) \preceq_{\alpha, \zeta(t)} (t.\text{update}(x, i.d, E), \mathcal{R}(t), G^t_{\text{write_barrier}})
\]

so we omit them in this paper.

**Rely-Guarantee reasoning about the GC code.** The program logic is designed by extending the traditional R-G Logic with rules for the GC-specific commands (e.g., \( x \leftarrow \text{get_root}(y) \)) and adapting some heap manipulation rules to our low-level machine model (e.g., \( \text{free}(x) \) just sets the object’s color to \( \text{BLUE} \)). We give the inference rules and the soundness proofs in Appendix E.5.

We describe states using separation logic assertions, as shown below:

\[
p, q : \mathcal{B} \mid t.\text{own}_n(x) \mid t.\text{own}_m(x) \mid E_1.\text{id} \rightarrow E_2 \mid p + q \mid \ldots
\]

Following Parkinson et al. [23], we treat program variables as resource and use \( t.\text{own}_n(x) \) and \( t.\text{own}_m(x) \) for the thread \( t \)'s ownerships of pointers and non-pointers respectively. Also in \( B \) we can use \( t.x \) to denote the thread \( t \)'s local variable \( x \). We omit the thread identifiers if these predicates hold for the current thread. We use \( E_1.\text{id} \rightarrow E_2 \) to specify a single-object single-field heap with \( E_2 \) stored in the field \( i.d \) of the object \( E_1 \). The separating conjuction \( p + q \) means \( p \) and \( q \) hold on disjoint states. We use \( E_1.\text{id} \rightarrow E_2 \sqcap \text{true} \) for \( E_1.\text{id} \rightarrow E_2 \) with \( \neg \text{false} \) for iterated separating conjunction over the set \( S \).

We first give the precondition and the guarantee of the GC. The GC starts its executions from a low-level well-formed state, i.e., \( p_{\text{gc}} \triangleq \text{wfstate} \). Just corresponding to the high-level wfstate definition, the low-level wfstate predicate says that the heap contains \( M \) objects and none of the reachable objects are \( \text{BLUE} \). We define the low-level wfstate predicate in Figure 19. It's easy to see that any low-level initial state is well-formed. We define \( p_{\text{gc}} \) as follows:

\[
p_{\text{gc}} \triangleq \{(\pi \triangleq \{t_{\text{gc}} \to \}, h), (\Pi, H)\} \land \forall n. \text{reachable}(n)(\pi, h) \implies h(n) = [h(n)] \land h(n).\text{color} = \text{BLUE} \land h'(n).\text{color} = \text{BLUE} .
\]

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, nor does it reclaim the object. Here \( \downarrow \) lifts a low-level object to a new one that contains mutator data only.

\[
[o] \triangleq \{\text{pt}_1 \to o(\text{pt}_1), \ldots, \text{pt}_m \to o(\text{pt}_m), \text{data} \to o(\text{data})\}
\]

As shown in Figure 12, every collection cycle begins from a well-formed state with an empty mark stack in the GC’s local store. Then the GC does the following things in order:

1. Concurrent Initializing: The GC scans the heap and clears the dirty card and the mark bit of each object. At the same time, the mutators can dirty the cards and allocate black objects. Thus after initialization, a white reachable object, if it cannot be traced from a root object in a white path, must be reachable from a newly-allocated object (i.e., a black object) whose pointer field was updated and dirty bit was set to 1. This property is denoted by \( \text{reach_inv} \).

2. Concurrent mark-phase: The GC reads the local store of each mutator to get the roots and then performs a depth-first traversal of the heap using the mark stack \( \text{mark} \). After tracing, we can ensure that if a white object is only reachable from a black object, then that black object must be dirty whose pointer field was updated by the mutators. In other words, \( \text{reach_inv} \) still holds after this phase.

3. Concurrent card-cleaning: The GC goes through the heap, and for every dirty object, first clear its dirty card and if it is black but points to an object which has not been marked, then the
4. Stop-the-world card-cleaning:

(a) Root-scanning: Due to possible updates during the previous concurrent phases, the pointer variables in mutators’ local stores must be re-scanned as if they were dirty. The GC marks those white root objects and pushes them onto the mark stack for future tracing. Thus after root-scanning, reach_inv is maintained at the end of the phase.

(b) Card-cleaning: The GC performs the same operations as in the concurrent card-cleaning phase. But this time the mutators cannot update the heap. Thus at the end, the mark stack is empty and all the reachable objects are black (denoted by reach_black).

5. Concurrent sweep-phase: The GC scans the heap and frees white objects. No matter how the mutators interleaves with the GC, all the white objects are remained unreachable. Thus the reclamation is safe that guarantees $G_{gc}$. After sweep, the state is still well-formed.

The predicates reach_inv, reach_rtnw_stk(X) and reach_black are defined in Figure 19 and the complete formal proofs are given in Appendix E.4.

8. Related Work and Conclusion

There is a large body of work on refinements and verification of program transformations. Here we only focus on the work most closely related to the typical applications discussed in this paper.
Verifying compilation and optimizations of concurrent programs.

Compiler verification for concurrent programming languages can date back to work by Wand [31] and Gladstein et al. [14], which is about functional languages using message-passing mechanisms. Recently, Lischbieler [23] 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 2.2 this is a too strong requirement in general for many transformations, including the examples in this paper.

Burckhardt et al. [9] 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 [19], Sevčík et al. [25] 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. Here we provide a general, compiler-independent, compositional proof technique to verify concurrent transformations.

We apply RGSim to justify concurrent optimizations, following Benton [3] who presents a declarative set of rules for sequential

Figure 18. Guarantees of Mutator Instructions

Figure 19. Boehm et al. GC Predicates
optimizations. Also the proof rules of RGSim for sequential compositions, conditional statements and loops coincide with those in relational Hoare logic \cite{26} and relational separation logic \cite{17}.

Proving linearizability or atomicity of concurrent objects. Filippovic et al. \cite{13} show linearizability can be characterized in terms of an observational refinement, where the latter is defined similarly to our Correct(T). There is no proof method given to verify the linearizability of fine-grained object implementations.

Turon and Wand \cite{27} propose a refinement-based proof method to verify concurrent objects. They first propose a simple refinement based on Brookes' fully abstract trace semantics \cite{8}, which is compositional but cannot handle complex algorithms (as discussed in Section 2.2). Their fenced refinement then uses rely conditions to filter out illegal environment transitions. The basic idea is similar to ours, and the refinement can also be used to verify Treiber's stack algorithm. However, it is “not a congruence for parallel composition”. In their settings, both the concrete (fine-grained) and the abstract (atomic) versions of object operations need to be expressed in the same language. They also require that the fine-grained implementation has only one update action over the shared state to correspond to the high-level atomic operation. These requirements and the lack of parallel compositionality limit the applicability of their method. It is unclear if the method can be used for general verification of transformations, such as concurrent GCs.

Elmas et al. \cite{12} prove linearizability by incrementally rewriting the fine-grained implementation to the atomic abstract specification. Their behavioral simulation used to characterize linearizability is an event-trace subset relation with requirements on the orders of method invocations and returns. Their rules heavily rely on movers (i.e., operations that can commute over any operation of other threads) and always rewrite programs to instructions, thus are designed specifically for atomicity verification.

In his thesis \cite{28}, Vafeiadis proves linearizability of concurrent objects in RGSep logic by introducing abstract objects and abstract atomic operations as auxiliary variables and code. The refinement between the concrete implementation and the abstract operation is implicitly embodied in the unary verification process, but is not spelled out formally in the meta-theory (e.g., the soundness).

Verifying concurrent GCs. Vecchev et al. \cite{30} define transformations to generate concurrent GCs from an abstract collector. Afterwards, Pavlovic et al. \cite{24} 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 transformations.

Kapoor et al. \cite{18} 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. \cite{22}, who propose a framework for separate verification of stop-the-world and incremental GCs and their mutators, but their framework does not handle concurrency.

Conclusion and Future Work. We propose RGSim to verify concurrent program transformations. By describing explicitly the interference with environments, RGSim is compositional, and can support many widely-used transformations. We have applied RGSim to reason about optimizations, prove atomicity of fine-grained concurrent algorithms and verify concurrent garbage collectors. In the future, we would like to further test its applicability with more applications, such as verifying STM implementations and compilers. It is also interesting to explore the possibility of building tools to automate the verification process.

Acknowledgments

We would like to thank Matthew Parkinson and anonymous referees for their suggestions and comments on earlier versions of this paper. This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant No. 60928004, 61073040 and 61103023. Xinyu Feng is also supported in part by NSF under Grant No. 90818019, by Program for New Century Excellent Talents in Universities (NCET), and by the Fundamental Research Funds for the Central Universities.

References

\begin{thebibliography}{10}
\bibitem{14} D. S. Gladstein and M. Wand. Compiler correctness for concurrent languages. In \textit{Proc. 1st Int’l Conf. on Coordination Languages
A. Soundness of RGSim (Theorem 5)

We first prove the following useful lemmas.

Lemma 9. For all \( k \geq 0 \) and \( m \geq 0 \), for all \( C, C', \sigma \) and \( \sigma' \), if \( (C, \sigma) \xrightarrow{n} (C', \sigma') \), then \( \text{ETrSet}_{m+n}(C', \sigma') \subseteq \text{ETrSet}_{m+n+1}(C, \sigma) \).

Proof: By induction over \( k \).

Base Case: \( k = 0 \), then \( C' = C \) and \( \sigma' = \sigma \), trivial.

Inductive Step: \( k = n + 1 \)

By unfolding \( (C, \sigma) \xrightarrow{n} (C', \sigma') \), we know there exists \( C'' \) and \( \sigma'' \) such that

\[
(C, \sigma) \xrightarrow{1} (C', \sigma') \quad \text{(A.1)}
\]

and

\[
(C'', \sigma'') \xrightarrow{n} (C', \sigma'). \quad \text{(A.2)}
\]

From (A.1) and Definition [1] we know

\[
\text{ETrSet}_{m+n}(C'', \sigma'') \subseteq \text{ETrSet}_{m+n+1}(C, \sigma). \quad \text{(A.3)}
\]

From (A.2) and the induction hypothesis, we know

\[
\text{ETrSet}_{m}(C', \sigma') \subseteq \text{ETrSet}_{m+1}(C'', \sigma''). \quad \text{(A.4)}
\]

From (A.3) and (A.4), we get the conclusion. \( \square \)

Lemma 10. For all \( k \geq 0 \) and \( m \geq 0 \), for all \( C, C', \sigma' \) and \( \sigma' \), if \( (C, \sigma) \xrightarrow{k} (C', \sigma') \) and \( \varepsilon \in \text{ETrSet}_{m}(C', \sigma') \), then \( \varepsilon \vdash \varepsilon \in \text{ETrSet}_{m+1}(C, \sigma) \).

Proof: By induction over \( k \).

Base Case: \( k = 0 \), trivial.

Inductive Step: \( k = n + 1 \)

By unfolding \( (C, \sigma) \xrightarrow{n} (C', \sigma') \), one of the following two cases holds:

1. there exists \( C'' \) and \( \sigma'' \) such that

\[
(C, \sigma) \xrightarrow{1} (C'', \sigma'') \quad \text{(A.1)}
\]

and

\[
(C'', \sigma'') \xrightarrow{n} (C', \sigma'). \quad \text{(A.2)}
\]

From (A.1) and Definition [1] we know

\[
\text{ETrSet}_{m+n}(C'', \sigma'') \subseteq \text{ETrSet}_{m+n+1}(C, \sigma). \quad \text{(A.3)}
\]

From (A.2) and the induction hypothesis, we know

\[
\varepsilon \vdash \varepsilon \in \text{ETrSet}_{m+1}(C', \sigma'). \quad \text{(A.4)}
\]

From (A.3) and (A.4), we get the conclusion. \( \square \)

In both cases, we can get the conclusion. \( \square \)

Lemma 11. For all \( k \geq 0 \), for all \( C, C', \sigma \) and \( \Sigma, R, G, \alpha \) and \( \gamma \), if \( (C, \sigma, R, G) \preceq_{\alpha, \gamma}(C, \Sigma, R, G) \), then \( \text{ETrSet}_{m}(C, \sigma) \subseteq \text{ETrSet}_{m}(C, \sigma) \).

Proof: By induction over \( k \).

Base Case: \( k = 0 \). We know \( \{ \varepsilon \} \subseteq \text{ETrSet}_{m}(C, \Sigma) \) always holds.

Inductive Step: \( k = n + 1 \)

For all \( \varepsilon \in \text{ETrSet}_{n+1}(C, \sigma) \), by Definition [1] we have four cases:

1. If \( C = \text{skip} \), then \( \varepsilon = \text{done} \). By unfolding \( \text{skip}, \sigma, R, G \) \( \preceq_{\alpha, \gamma}(C, \Sigma, R, G) \), we know there exists \( \Sigma' \) such that
From the 2nd premise, we know that if $C, \Sigma \rightarrow^* (skip, \Sigma')$, we know
\[ E \in \text{ETrSet}(C, \Sigma). \]  
\hfill (A.1)

2. If $(C, \sigma) \rightarrow (C', \sigma')$ and $E \in \text{ETrSet}(C', \sigma')$, then by
\[ \text{unfolding } (C, \sigma, R, G) \subseteq_{\alpha, \gamma} (C, \Sigma, R, G), \]
we know there exist $C'$ and $\Sigma'$ such that
\[ (C, \Sigma) \rightarrow^* (C', \Sigma'). \]  
\hfill (A.2)

and
\[ (C', \sigma', R, G) \subseteq_{\alpha, \gamma} (C', \Sigma', R, G). \]  
\hfill (A.3)

From (A.3) and the induction hypothesis, we know
\[ \text{ETrSet}(C', \sigma') \subseteq \text{ETrSet}(C', \Sigma'). \]
Thus
\[ E \in \text{ETrSet}(C', \Sigma'). \]  
\hfill (A.4)

By (A.2), Lemma 9 and (A.4), we know
\[ E \in \text{ETrSet}(C, \Sigma). \]  
\hfill (A.5)

3. If $(C, \sigma) \rightarrow (C', \sigma')$, $E = \epsilon : E'$ and $E' \in \text{ETrSet}(C', \sigma')$, then by
\[ \text{unfolding } (C, \sigma, R, G) \subseteq_{\alpha, \gamma} (C, \Sigma, R, G), \]
we know there exist $C'$ and $\Sigma'$ such that
\[ (C, \Sigma) \rightarrow^* (C', \Sigma'). \]  
\hfill (A.6)

and
\[ (C', \sigma', R, G) \subseteq_{\alpha, \gamma} (C', \Sigma', R, G). \]  
\hfill (A.7)

From (A.7) and the induction hypothesis, we know
\[ \text{ETrSet}(C', \sigma') \subseteq \text{ETrSet}(C', \Sigma'). \]
Thus
\[ E' \in \text{ETrSet}(C', \Sigma'). \]  
\hfill (A.8)

By (A.6), Lemma 10 and (A.8), we know
\[ E \in \text{ETrSet}(C, \Sigma). \]  
\hfill (A.9)

4. If $(C, \sigma) \rightarrow \text{abort}$ and $E = \text{abort}$, then by
\[ \text{unfolding } (C, \sigma, R, G) \subseteq_{\alpha, \gamma} (C, \Sigma, R, G), \]
we know $(C, \Sigma) \rightarrow^* \text{abort}$. Then we can prove
\[ E \in \text{ETrSet}(C, \Sigma). \]  
\hfill (A.10)

From (A.1), (A.5), (A.9) and (A.10), we get the conclusion. \hfill \Box

We get Theorem 5 immediately from Lemma 11.

\section{Soundness of Compositionality Rules}

\subsection{Soundness of the SEQ Rule}

\begin{lemma}
For all $C_1, C_2, C_1$ and $C_2$, for all $\sigma$ and $\Sigma$, if
\begin{enumerate}
  \item $(C_1, \sigma, R, G) \subseteq_{\alpha, \gamma} (C_1, \Sigma, R, G)$; and
  \item for all $\sigma_2$ and $\Sigma_2$, if $(\sigma_2, \Sigma_2) \in \gamma$, then $(C_2, \sigma_2, R, G) \subseteq_{\alpha, \gamma} (C_2, \Sigma_2, R, G)$;
\end{enumerate}
then
\[ (C_1; C_2, \sigma, R, G) \subseteq_{\alpha, \gamma} (C_1; C_2, \Sigma, R, G). \]
\end{lemma}

\begin{proof}
By co-induction.
Let
\[ S = \{(C_1; C_2, \sigma), (C_1; C_2, \Sigma)\} \mid \text{the premises hold} \]
\[ \cup \{(C_2, \sigma_2), (C_2, \Sigma_2)\} \mid (\sigma_2, \Sigma_2) \in \gamma. \]
We prove $S \subseteq F(S)$ where $F$ is defined by the simulation.
From the 2nd premise, we know that if $(\sigma_2, \Sigma_2) \in \gamma$, then
\[ ((C_2, \sigma_2), (C_2, \Sigma_2)) \text{ satisfies the simulation.} \]
For all $((C_1; C_2, \sigma), (C_1; C_2, \Sigma)) \in S$, we know $(\sigma, \Sigma) \in \alpha$.

1. If $(C_1; C_2, \sigma) \rightarrow (C', \sigma')$, then according to the operational semantics, we have two possible cases:
\begin{itemize}
  \item $C_1 \neq \text{skip}$. Thus $C' = C_1; C_2$ and
    \[ (C_1, \sigma) \rightarrow (C_1', \sigma'). \]
    From the 1st premise, we know $(\sigma, \sigma') \in G$ and there exist $C_1$ and $\Sigma'$ such that the followings hold:
    \[ (C_1, \Sigma) \rightarrow^* (C_1', \Sigma'), (\Sigma, \Sigma') \in G^* \]
    \[ (C_1', \sigma', R, G) \subseteq_{\alpha, \gamma} (C_1', \Sigma', R, G). \]
    Thus we know $(C_1; C_2, \sigma'), (C_1; C_2, \Sigma') \in S$.
  \item $C_1 = \text{skip}$. Thus $C' = C_2$ and $\sigma' = \sigma$.
    Since $G$ contains identity transitions, we know $(\sigma, \sigma') \in G$.
    From the 1st premise, we know there exists $\Sigma'$ such that
    \[ (C_1, \Sigma) \rightarrow^* (\text{skip}, \Sigma'), (\Sigma, \Sigma') \in G^*, (\sigma, \Sigma') \in \gamma \]
    Thus $(C_1; C_2, \Sigma) \rightarrow^* (C_2, \Sigma')$ and
    \[ ((C_2, \sigma), (C_2, \Sigma')) \in S. \]
\end{itemize}
\end{proof}

\section{Soundness of the IF Rule}

\begin{lemma}
For all $C_1, C_2, C_1$ and $C_2$, for all $\sigma$ and $\Sigma$, if
\begin{enumerate}
  \item for all $\sigma_1$ and $\Sigma_1$, if $(\sigma_1, \Sigma_1) \in G = (\zeta \cap (B \equiv B))$, then
    \[ (C_1; \sigma_1, R, G) \subseteq_{\alpha, \gamma} (C_1; \Sigma_1, R, G); \]
  \item for all $\sigma_2$ and $\Sigma_2$, if $(\sigma_2, \Sigma_2) \in G = (\zeta \cap (B \equiv B))$, then
    \[ (C_2, \sigma_2, R, G) \subseteq_{\alpha, \gamma} (C_2, \Sigma_2, R, G); \]
  \item $\zeta \subseteq (B \equiv B)$; and
  \item $(\sigma, \Sigma) \in \zeta \subseteq \alpha$,
\end{enumerate}
then
\[ \text{if } (B) \text{ then } C_1 \text{ else } C_2, \sigma, R, G \subseteq_{\alpha, \gamma} \text{ if } B \text{ then } C_1 \text{ else } C_2, \Sigma, R, G. \]
\end{lemma}

\begin{proof}
By co-induction.
Let
\[ S = \{(B) \text{ if } (B) \text{ else } C_1 \text{ else } C_2, \sigma, R, G \} \mid \text{the premises hold} \]
\[ \cup \{(C_1, \sigma_1), (C_1, \Sigma_1)\} \mid (\sigma_1, \Sigma_1) \in G \}
\[ \cup \{(C_2, \sigma_2), (C_2, \Sigma_2)\} \mid (\sigma_2, \Sigma_2) \in \gamma. \]
We prove $S \subseteq F(S)$ where $F$ is defined by the simulation.
From all $(B) \text{ if } (B) \text{ else } C_1 \text{ else } C_2, \Sigma) \in S$,
1. If $(B) \text{ if } (B) \text{ else } C_1 \text{ else } C_2, \sigma) \rightarrow (C', \sigma')$, then according to the operational semantics, we have two possible cases:
\begin{itemize}
  \item $C_1 \neq \text{skip}$. Thus $C' = C_1; C_2$ and
    \[ (C_1, \sigma) \rightarrow (C_1', \sigma'). \]
    From the 1st premise, we know $(\sigma, \sigma') \in G$ and there exist $C_1$ and $\Sigma'$ such that the followings hold:
    \[ (C_1, \Sigma) \rightarrow^* (C_1', \Sigma'), (\Sigma, \Sigma') \in G^* \]
    \[ (C_1', \sigma', R, G) \subseteq_{\alpha, \gamma} (C_1', \Sigma', R, G). \]
    Thus we know $(C_1; C_2, \sigma'), (C_1; C_2, \Sigma') \in S$.
  \item $C_1 = \text{skip}$. Thus $C' = C_2$ and $\sigma' = \sigma$.
    Since $G$ contains identity transitions, we know $(\sigma, \sigma') \in G$.
    From the 1st premise, we know there exists $\Sigma'$ such that
    \[ (C_1, \Sigma) \rightarrow^* (\text{skip}, \Sigma'), (\Sigma, \Sigma') \in G^*, (\sigma, \Sigma') \in \gamma \]
    Thus $(C_1; C_2, \Sigma) \rightarrow^* (C_2, \Sigma')$ and
    \[ ((C_2, \sigma), (C_2, \Sigma')) \in S. \]
\end{itemize}
\end{proof}
B.3 Soundness of the WHILE Rule

Lemma 14. For all C and Σ, for all σ and Σ, if
1. for all σ1 and Σ1, if (σ1, Σ1) ∈ γ1 = (γ ∩ (B ≡ B)); then (C, σ1, G, Σ) ⊑α,γ1 (C, σ1, G, Σ);
2. γ ⊑ (B ≡ B);
3. γ2 = (γ ∩ (¬B ≡ ¬B)); and
4. (σ, Σ) ∈ γ,
   then
   (while (B) C, σ, G, Σ) ⊑α,γ2 (while B do C, Σ, R, G).

Proof. By co-induction.

Let

\[ S = \{ ((\text{while } B)^* C, \sigma) \mid (\text{the premises hold}) \} \]
\[ \cup \{ ((C; \text{while } B) C, \sigma_1), (C; \text{while } B do C, \Sigma_1) \} \]
\[ \cup \{ (\text{skip}, \sigma_2), (\text{skip}, \Sigma_2) \} \mid (\sigma_2, \Sigma_2) \in \gamma_2 \} \]

We prove \( S \subseteq F(S) \) where \( F \) is defined by the simulation. From Lemma 12 we can get that if for all (σ, Σ) ∈ γ we have

((\text{while } B) C, σ, G, Σ) satisfies the simulation, then for all (σ1, Σ1) ∈ γ1,

((C; \text{while } B) C, σ1), (C; \text{while } B do C, Σ1)) satisfies the simulation.

Since Sta(γ2, (R, R + )α), we can prove that if \( (\sigma_2, \Sigma_2) \in \gamma_2 \), then

(skip, \sigma_2, R, Σ, G) \( \subseteq_\alpha,\gamma_2 \) (skip, \sigma_2, R, Σ, G)

That is, ((skip, \sigma_2), (skip, \Sigma_2)) satisfies the simulation.

For all ((\text{while } B) C, σ), (\text{while } B do C, Σ)) \( \in S \), since \( \gamma \subseteq \alpha \), we know \( (\sigma, \Sigma) \in \alpha \).

1. If (\text{while } B) C, σ) \( \rightarrow (C', \sigma') \), then according to the operational semantics, we have two possible cases:

- \( B \sigma = \text{true} \). Thus \( C' = C; \text{while } B \) and \( \sigma' = \sigma \).
  Since \( G \) contains identity transitions, we know \( (\sigma, \sigma') \in G \).
  From \( \zeta \subseteq (B \equiv B) \), we know \( \Sigma = \text{true} \).

- \( B \sigma = \text{false} \). Thus \( C' = \text{skip} \) and \( \sigma' = \sigma \).
  Since \( G \) contains identity transitions, we know \( (\sigma, \sigma') \in G \).
  From \( \zeta \subseteq (B \equiv B) \), we know \( \Sigma = \text{true} \).

2. The case for \( \text{while } B \) C, σ) \( \rightarrow (C', \sigma') \) is vacantly true.

3. If \( (\sigma, \sigma') \in \mathcal{R}, (\Sigma, \Sigma') \in \mathbb{P}^* \) and \( (\sigma', \Sigma') \in \alpha \), then from Sta(\( (R, \mathbb{P}) \alpha) \), we know \( (\sigma', \Sigma') \in \zeta \). Thus

((\text{while } B) C, \text{else } C_2, \sigma'), (\text{if } \Sigma \text{ then } C_1 \text{ else } C_2, \Sigma') \in S).

4. If (\text{while } B) C_1 \text{ else } C_2, \sigma) \( \rightarrow \text{abort} \), then \( B \sigma = \text{false} \).

Then we have ((\text{while } B) C_1 \text{ else } C_2, \sigma), (\text{if } \Sigma \text{ then } C_1 \text{ else } C_2, \Sigma)) \( \in \mathcal{F}(S) \). Thus (\text{while } B) C_1 \text{ else } C_2, \sigma) and (\text{if } \Sigma \text{ then } C_1 \text{ else } C_2, \Sigma) satisfy the largest simulation RGSim.

Then we can conclude soundness of the WHILE rule.

B.4 Soundness of the PAR Rule

Lemma 15. For all C1, C2, C3, C4, σ and Σ, if
1. (C1, σ, R1, G1) \( \subseteq_\alpha,\gamma_1 \) (C1, σ, R1, G1);
2. (C2, σ, R2, G2) \( \subseteq_\alpha,\gamma_2 \) (C2, σ, R2, G2); and
3. G1 \( \subseteq R_2 \); G2 \( \subseteq R_1 \); G1 \( \subseteq R_2 \); G2 \( \subseteq R_1 \),
   then
   (C1 || C2, σ, R1 \( \cap \) R2, G1 \( \cup \) G2) \( \subseteq_\alpha,\gamma_1,\gamma_2 \) (C1 || C2, σ, R1 \( \cap \) R2, G1 \( \cup \) G2).

Proof. By co-induction.

Let

\[ S = \{ (C_1 || C_2, \sigma) \mid (\text{the premises hold}) \} \]

We prove \( S \subseteq \mathcal{F}(S) \) where \( F \) is defined by the simulation. For all ((C || C), (C || C), Σ)) \( \in S \),

1. If (C || C) \( \rightarrow (C', \sigma') \), then according to the operational semantics, we have three possible cases:

- \( (C, \sigma) \rightarrow (C', \sigma') \) and \( C' = C' || C_2 \).

From the 1st premise, we know

\( (\sigma, \sigma') \in \gamma_1 \). \( \text{(B.1)} \)

and there exist C' \( \in \gamma_1 \) and \( \Sigma' \) such that the followings hold:

\( (C_1, \Sigma) \rightarrow^* (C_1, \Sigma') \in \gamma_1 \). \( \text{(B.2)} \)

From \( \text{B.1} \), we know \( (\sigma, \sigma') \in \gamma_1 \). \( \text{B.1} \)

From \( \text{B.2} \), we know \( (C_1 || C_2, \Sigma) \rightarrow^* (C_1 || C_2, \Sigma') \) and \( (\Sigma, \Sigma') \in \gamma_1 \). \( \text{B.3} \)

From \( \text{B.3} \), we know \( (\sigma, \sigma') \in \gamma_1 \). \( \text{B.3} \)

From \( \text{B.3} \) and \( \Sigma_1 \), we know

\( (C', \sigma', R_3, G_3) \subseteq_\alpha,\gamma_1,\gamma_2 \) (C', \sigma', R_3, G_3). \( \text{(B.4)} \)
For all

By co-induction.

From the 1st premise, we know

\[(C_1, \Sigma) \rightarrow (\text{skip}, \Sigma'), (\Sigma, \Sigma') \in G_1 \]

(\Sigma, \Sigma') ∈ \gamma_1

(\Sigma, \Sigma') ∈ \gamma_1

From (\text{[5.5]}), we know

\[(\text{skip}, \Sigma, R_2, G_2) \leq_{\gamma_2} (C_2, \Sigma', R_2, G_2) \]

Thus \((C_2, \Sigma') \rightarrow (\text{skip}, \Sigma'), (\Sigma, \Sigma') \in G_2^*\) and

\[(\Sigma, \Sigma') \in \gamma_2\]

Then from (\text{[5.7]}), we know \((C_1 || C_2, \Sigma) \rightarrow (\text{skip}, \Sigma'')\) and

On the other hand, \((\Sigma, \Sigma'') \in G_2^* \subseteq R_i^1\). From (\text{[5.6]} and \text{[5.9]}), we know

\[(\sigma, \Sigma'') \in \gamma_1\]

(\text{[5.9]} and \text{[5.10]}), we know \((\sigma, \Sigma'') \in \gamma_1 \cap \gamma_2\).

2. If \((C_1 || C_2, \Sigma) \rightarrow (C', \sigma')\), the proof is similar to the previous case.

3. If \((\sigma, \Sigma') \in \Sigma \subseteq R_i \cap R_2\), then we know \((\sigma', \Sigma') \in \alpha\) and

\[(C_1, \sigma', R_1, G_1) \leq_{\gamma_1} (C_2, \Sigma', R_1, G_1)\]

Thus \(((C_1 || C_2, \sigma'), (C_1 || C_2, \Sigma')) \in S\).

4. \(C_1 \parallel C_2 \neq \text{skip}\). This case is vacantly true.

5. If \((C_1 || C_2, \sigma) \rightarrow \text{abort}\), then \((C_1 || C_2, \Sigma) \rightarrow (C', \sigma)\) is immediate from the premises.

Then we have \(((C_1 || C_2, \sigma), (C_1 || C_2, \Sigma)) \in F(S)\). Thus \((C_1 || C_2, \sigma)\) and \((C_1 || C_2, \Sigma)\) satisfy the largest simulation RGSim. \(\square\)

Thus we can conclude soundness of the \textbf{PAR} rule.

\section{Soundness of \textbf{the WEAKEN-\(\alpha\)} Rule}

\textbf{Lemma 17. For all \(C, \sigma\) and \(\Sigma\), if}

1. \((C, \sigma, R, G) \succeq_{\alpha, \gamma} (C, \Sigma, R, G)\);
2. \(\sigma \subseteq \alpha'; \text{ and}\)
3. \(\text{Sta}(\alpha, (R, R^*)^\alpha)\).

\text{then} \((C, \sigma, R, G) \succeq_{\alpha, \gamma} (C, \Sigma, R, G)\).

\textbf{Proof.} By co-induction.

Let \(S = \{(C, \sigma, (C, \Sigma)) \mid \text{the premises hold}\}\). We prove \(S \subseteq F(S)\) where \(F\) is defined by the simulation.

For all \(((C, \sigma, (C, \Sigma)) \in S\).

1. If \((C, \sigma) \rightarrow (C', \sigma')\), then from the 1st premise, we know \((\sigma, \sigma') \in G\) and there exist \(C^*\) and \(\Sigma'\) such that the followings hold:

\[(C, \Sigma) \rightarrow (C', \Sigma'), (\Sigma, \Sigma') \in G^*\]

\[(C', \Sigma', R, G) \succeq_{\alpha, \gamma} (C', \Sigma', R, G)\]

We know \((\sigma', \Sigma') \in \alpha\). Then by applying the 4th premise, we can get \((\sigma', \Sigma') \in \alpha'\). Thus we know \(((C^*, \sigma'), (C', \Sigma')) \in S\).

2. If \((C, \sigma) \rightarrow (C', \sigma')\), the proof is similar to the previous case.

3. If \((\sigma, \Sigma') \in R \wedge (\sigma', \Sigma') \in \alpha'\), then we know \((\sigma', \Sigma') \in \alpha\). Using the 1st premise, we have

\[(C', \sigma', R, G) \succeq_{\alpha, \gamma} (C', \Sigma', R, G)\]

Thus \(((C', \sigma'), (C', \Sigma')) \in S\).

4. If \(C = \text{skip}\), then from the 1st premise, we know there exists \(\Sigma'\) such that the followings hold:

\[(C, \Sigma) \rightarrow (\text{skip}, \Sigma'), (\Sigma, \Sigma') \in G^*\]

\[(\sigma', \Sigma') \in \gamma\]

5. If \((C, \sigma) \rightarrow \text{abort}\), then \((C, \Sigma) \rightarrow (\text{abort})\) is immediate from the 1st premise.

Thus we have \(((C, \sigma), (C, \Sigma)) \in F(S)\). Thus \((C, \sigma)\) and \((C, \Sigma)\) satisfy the largest simulation RGSim. \(\square\)

Thus we can conclude soundness of the \textbf{WEAKEN-\(\alpha\)} rule.
B.7 Soundness of the FRAME Rule

Lemma 18. For all $C$, $C$, $\sigma$ and $\Sigma$, if
1. $(C, \sigma, R, G) \preceq_{\alpha \gamma} (C, \Sigma, R, G)$;
2. $(\sigma, \Sigma) \in \alpha \cup \eta$.
3. $\eta \subseteq \beta$.
4. $\eta \neq \{\gamma, \alpha\}$;
5. Intuit$(\alpha, \gamma, \beta, \eta, R, R, R, R'$);
6. $\text{Sta}(\eta, \{G, G^{*}\}_\alpha, (R', R'^{*})_\beta)$.
then $(C, \sigma, R \cup R', G \cup G') \preceq_{\alpha \beta, \gamma \eta} (C, \Sigma, R \cup R', G \cup G')$.

Proof: By co-induction.
Let $S = \{(C, \sigma, (C, \Sigma)) | \text{ the premises hold}\}$. We prove $S \subseteq F(S)$ where $F$ is defined by the simulation.
For all $((C, \sigma), (C, \Sigma)) \in S$, from the 2nd and 3rd premises, we know $(\sigma, \Sigma) \in \alpha \cup \beta$.
1. If $(C, \sigma) \rightarrow (C', \sigma')$, then from the 1st premise, we know $(\sigma, \sigma') \in G$ and there exist $C'$ and $\Sigma'$ such that the followings hold:

$\quad (C, \Sigma) \rightarrow^{*} (C', \Sigma'), (\Sigma, \Sigma') \in G^{*}$

$\quad (C', \sigma', R, G) \preceq_{\alpha \gamma} (C', \Sigma', R, G)$

Since $(\phi, \phi) \in \text{ld} \subseteq G'$, we know $G \subseteq (G \cup G')$. Thus $(\sigma, \sigma') \in G \cup G'$.
Similarly, $(\Sigma, \Sigma') \in (G \cup G')^{*}$.
Also we know $(\sigma', \Sigma') \in \alpha$. From Intuit$(\eta)$, we know $(\sigma, \Sigma) \in \alpha \cup \eta \subseteq \eta$.
From $\text{Sta}(\eta, (G, G^{*})_\alpha)$, we know $(\sigma', \Sigma') \in \eta$. Since $\eta \neq \alpha$, we know

$\quad (\sigma', \Sigma') \in \alpha \cup \eta$.

Thus we know $((C', \sigma'), (C', \Sigma')) \in S$.
2. If $(C, \sigma) \rightarrow^{*} (C', \sigma')$, the proof is similar to the previous case.
3. If $(\sigma, \sigma') \in R \cup R'$ and $(\sigma', \Sigma') \in \alpha \cup \beta$, then from Intuit$(\{R', R, \alpha\})$, we have

$\quad (\sigma, \sigma') \in R, (\Sigma, \Sigma') \in R^{*}, (\sigma', \Sigma') \in \alpha$

Then from the 1st premise, we have

$\quad (C, \sigma, R, G) \preceq_{\alpha \gamma} (C, \Sigma, R, G)$.

From Intuit$(\eta)$, we know $(\sigma, \Sigma) \in \alpha \cup \eta \subseteq \eta$. On the other hand, from Intuit$(\{R', R, \alpha\})$, we have

$\quad (\sigma, \sigma') \in R', (\Sigma, \Sigma') \in R^{*}, (\sigma', \Sigma') \in \beta$.

From $\text{Sta}(\eta, (R', R'^{*})_\beta)$, we know $(\sigma', \Sigma') \in \eta$. Since $\eta \neq \alpha$, we know

$\quad (\sigma', \Sigma') \in \alpha \cup \eta$.

Thus $((C, \sigma'), (C, \Sigma')) \in S$.
4. If $C = \text{skip}$, then from the 1st premise, we know there exists $\Sigma'$ such that the followings hold:

$\quad (C, \Sigma) \rightarrow^{*} (\text{skip}, \Sigma'), (\Sigma, \Sigma') \in G^{*}, (\sigma, \Sigma') \in \gamma$

We have $(\Sigma, \Sigma') \in (G \cup G)^{*}$ and $(\sigma, \Sigma') \in \alpha$.
From Intuit$(\eta)$, we know $(\sigma, \Sigma) \in \alpha \cup \eta \subseteq \eta$.
From $\text{Sta}(\eta, (G, G^{*})_\alpha)$, we know $(\sigma, \Sigma') \in \eta$.
Since $\eta \neq \gamma$, we know $(\sigma, \Sigma') \in \gamma \cup \eta$.
5. If $(C, \sigma) \rightarrow \text{abort}$, then $(C, \Sigma) \rightarrow \text{abort}$ is immediate from the 1st premise.
Then we have $((C, \sigma), (C, \Sigma)) \in F(S)$. Thus $(C, \sigma)$ and $(C, \Sigma)$ satisfy the largest simulation $\text{RGSim}$.

Since $(\zeta \cup \gamma) \subseteq \alpha$ and $\eta \subseteq \beta$, we have

$\quad (\zeta \cup \eta) \subseteq (\alpha \cup \beta), (\gamma \cup \eta) \subseteq (\alpha \cup \beta), (\zeta \cup \eta) \subseteq (\alpha \cup \eta)$.

Then we can conclude soundness of the FRAME rule.

B.8 Soundness of the Optimization Rules

Here we only give the proof sketch of soundness of the dead-while, the dead-code-elimination and the redundancy introduction rules. Proofs of other rules are similar.

Lemma 19 (Dead While). For all $\sigma_1$ and $\sigma_2$, if
1. $\zeta = (\zeta \cap (\text{true} \land \neg B))$;
2. $\text{Sta}(\zeta, (R, R'^{*})_\alpha)$;
3. $(\sigma_1, \sigma_2) \in \zeta \subseteq \alpha$,
then $(\text{skip}, \sigma_1, R, ld) \preceq_{\alpha \gamma} (\text{while} (B)) (C, \sigma_2, R', ld)$.

Proof: By co-induction.
Since $B \sigma_2 = \text{false}$, we know $(\text{while} (B)) (C, \sigma_2, R', ld)$.
The case for the environments’ transitions is immediate from $\text{Sta}(\zeta, (R, R'^{*})_\alpha)$.

Lemma 20 (Dead Code Elimination). For all $\sigma_1$ and $\sigma_2$, if
1. $(\text{skip}, \sigma_1, \text{ld}, G) \preceq_{\alpha \gamma} (C, \sigma_2, \text{ld}, G)$;
2. $\text{Sta}(\{\zeta, \gamma\}, (R, R'^{*})_\alpha)$;
3. $(\sigma_1, \sigma_2) \in \zeta$,
then $(\text{skip}, \sigma_1, R, ld) \preceq_{\alpha \gamma} (C, \sigma_2, R', G)$.

Proof: By co-induction.
If $C = \text{skip}$, then $(\sigma_1, \sigma_2) \in \gamma$. From $\text{Sta}(\gamma, (R, R'^{*})_\alpha)$, we can prove the conclusion.
Otherwise, there exists $\sigma_2'$ such that $(C, \sigma_2) \rightarrow^{*} (\text{skip}, \sigma_2')$, $(\sigma_2, \sigma_2') \in G'$ and $(\sigma_1, \sigma_2') \in \gamma$.
Finally, the case for the environments’ transitions is immediate from $\text{Sta}(\zeta, (R, R'^{*})_\alpha)$.

Lemma 21 (Redundancy Introduction). For all $\sigma_1$ and $\sigma_2$, if
1. $(c, \sigma_1, \text{ld}, G) \preceq_{\alpha \gamma} (\text{skip}, \sigma_2, \text{ld}, \text{ld})$;
2. $\text{Sta}(\{\zeta, \gamma\}, (R, R'^{*})_\alpha)$;
3. $(\sigma_1, \sigma_2) \in \zeta$,
then $(c, \sigma_1, R, G) \preceq_{\alpha \gamma} (\text{skip}, \sigma_2, R', \text{ld})$.

Proof: By co-induction.
Since $c$ is an instruction, by its operational semantics, we only have four cases:
1. If $(c, \sigma_1) \rightarrow (\text{skip}, \sigma_1')$, then $(\sigma_1, \sigma_1') \in G$ and $(\text{skip}, \sigma_2) \rightarrow^{0} (\text{skip}, \sigma_2), (\sigma_2, \sigma_2) \in \text{ld}$.
From $(\text{skip}, \sigma_1', \text{ld}, G) \preceq_{\alpha \gamma} (\text{skip}, \sigma_2, \text{ld}, \text{ld})$, we know $(\sigma_1', \sigma_2) \in \gamma$.
Since $\text{Sta}(\gamma, (R, R'^{*})_\alpha)$, it’s not difficult to prove

$(\text{skip}, \sigma_1', R, G) \preceq_{\alpha \gamma} (\text{skip}, \sigma_2, R', \text{ld})$.
2. $(c, \sigma_1) \rightarrow^{e} (\text{skip}, \sigma_1')$ is impossible.
3. If \((c, \sigma_1) \rightarrow (c, \sigma_1)\), trivial.
4. \((c, \sigma_1) \rightarrow \text{abort}\) is impossible.

Finally, the case for the environments’ transitions is immediate from \(\text{Sta}(\zeta; (R, R', \rho))\).

\[\square\]

C. Proof of Theorem [8]

We prove for any high-level mutator program \(W, T(W) \subseteq T\). Suppose \(W = \mathcal{E}_{gc}.AbsGC || t_1; C_1 || \ldots || t_n; C_n\).

Decompositions by RGSim

RGSim is sound w.r.t. the e-trace refinement and the programs of the two levels are all closed systems whose environments are supposed to be identity transitions, so we only need to find some \(\alpha, \zeta, \gamma\) such that:

\[\begin{align*}
(\text{gc}_{gc}, C_{gc} || t_1; T(C_1) || \ldots || t_n; T(C_n), \text{Id}, True) &\succeq_{\alpha, \zeta, \gamma} \\
(\text{gc}_{gc}.AbsGC || t_1; C_1 || \ldots || t_n; C_n, \text{Id}, True)
\end{align*}\]

We can decompose it into single threads and prove refinements on mutator threads by refinements on primitive instructions, as shown in the following lemma.

**Lemma 22.** If

1. \((C_{gc}, R_{gc}, C_{gc}) \sqsubseteq_{\alpha, \zeta, \gamma} \text{gc}_{gc} (\text{AbsGC, True, AbsGCStep})\);
2. \(\forall c, (t, T(c), R(t), G(t, T(c))) \succeq_{\alpha, \zeta, \gamma} (t, c, R(t), G(t, c))\);
3. \(G(t) = \bigcup \{G(t, T(c)) : (t, R(t), G(t, T(c))) \succeq_{\alpha, \zeta, \gamma} (t, R(t), G(t, c))\}\);
4. \(\forall t, \zeta, \zeta(t) \subseteq (T(B) \subseteq B)\).

then

\[\begin{align*}
(\text{gc}_{gc}, C_{gc} || t_1; T(C_1) || \ldots || t_n; T(C_n), \text{Id}, True) &\succeq_{\alpha, \zeta, \gamma} \\
(\text{gc}_{gc}.AbsGC || t_1; C_1 || \ldots || t_n; C_n, \text{Id}, True)
\end{align*}\]

**Proof:** By induction over the high-level program structure. \(\square\)

If \(\text{InitRel}(\zeta(t))\) then \(\text{InitRel}(\zeta)\). Thus from soundness of RGSim (Corollary [6]), we can conclude \(\text{Correct}(T)\).

**From Verification to Refinement for the GC thread** Lemma [23] allows proving refinement on the GC thread by verifying the GC code in a Rel-v-Guarantee-based logic. Let

\[\zeta_{gc} \triangleq \{(\sigma, \Sigma) \mid \sigma = T(\Sigma)\}\]

Then \(\text{InitRel}(\zeta_{gc})\). Since \(\text{InitRel}(\zeta(t))\) and \(\zeta(t) \subseteq \alpha\), we know \(\zeta_{gc} \subseteq \alpha\).

**Lemma 23.** If

1. (Verification of the GC code)
\[R_{gc}; G_{gc} \vdash \{p_{gc}\} C_{gc} \{\text{false}\};\]
2. (Side Conditions)
\[G_{gc} \circ \sigma^{-1} \subseteq \sigma^{-1} \circ \text{AbsGCStep}^*;\]
\[\forall \sigma, \Sigma, (\sigma, \Sigma) \in \zeta_{gc} \implies p_{gc} \sigma; \zeta_{gc} \subseteq \alpha,\]

then \((C_{gc}, R_{gc}, C_{gc}) \succeq_{\alpha, \zeta, \gamma} \text{gc}_{gc} (\text{AbsGC, True, AbsGCStep})\).

The semantics of \(R; G \vdash \{p\} C[\{q\}]\) is defined in a traditional way except we have an extra requirement that \(C\) does not generate external events.

**Definition 24 (Non-Interference).**
\((C, \sigma, R) \text{ guarantees } G\) always holds;
\((C, \sigma, R) \text{ guarantees } G\) holds iff
\[\neg((C, \sigma) \rightarrow \text{abort}), \neg(\exists C', \sigma', (C, \sigma) \rightarrow (C', \sigma')), \text{ and}\]
1. for all \(\sigma'\), if \((\sigma, \sigma') \in R\), then \((C, \sigma', R) \text{ guarantees } G\);
2. for all \(\sigma'\), if \((C, \sigma) \rightarrow (C', \sigma')\), then \((\sigma, \sigma') \in G\) and 
\((C', \sigma', R) \text{ guarantees } G\).

Then, \((C, \sigma, R) \text{ guarantees } G \triangleq \forall k. (C, \sigma, R) \text{ guarantees } G.\)

**Definition 25 (Semantics).** \(R; G \models \{p\} C[\{q\}]\) iff, for any \(\sigma\) such that \(p, \sigma\), the following are true:
1. if \((C, \sigma) \rightarrow (\text{skip}, \sigma')\), then \(q \sigma'\);
2. \((C, \sigma, R) \text{ guarantees } G\).

where \((C, \sigma) \rightarrow (C', \sigma')\) is defined by:

\[\begin{align*}
(C, \sigma) &\rightarrow (C', \sigma') & (\sigma, \sigma') \in R \\
(C, \sigma) &\rightarrow (C', \sigma') & (C, \sigma) \rightarrow (C', \sigma')
\end{align*}\]

Then Lemma [23] is proved immediately from soundness of the logic (i.e., if \(R; G \models \{p\} C[\{q\}]\), then \(R; G \models \{p\} C[\{q\}]\)) and the following lemma.

**Lemma 26.** For all \(C, R, G, \sigma, \Sigma, \sigma'\), if

1. \((C, \sigma, R) \text{ guarantees } G\);
2. \(G \circ \sigma^{-1} \subseteq \sigma^{-1} \circ \text{AbsGCStep}^*;\)
3. \((\sigma, \Sigma, \sigma') \in \alpha; \text{ and}\)
4. \(\neg \exists \sigma', ((C, \sigma) \rightarrow (\text{skip}, \sigma'))\),

then \((C, \sigma, R, G) \succeq_{\alpha, \gamma} (\text{AbsGC, True, AbsGCStep})\).

**Proof:** By co-induction.

Let \(S \triangleq \{(\sigma, \Sigma, \Sigma') \mid \Sigma' \in \text{True}\}. \text{ We prove } S \subseteq F(S) \text{ where } F \text{ is defined by the simulation. For all } (C, \sigma, \Sigma) \in S, \text{ we only need to consider two cases:}\)

1. if \((C, \sigma) \rightarrow (C', \sigma')\), then \((\sigma, \sigma') \in G\) and \((C', \sigma', R) \text{ guarantees } G\).
2. if \((C, \sigma, R) \subseteq (C', \sigma', \Sigma') \in \text{True}, \Sigma') \in \alpha, \text{ and}\)

then we have \((C, \sigma, R) \text{ guarantees } G\) and \((\sigma, \Sigma') \in \alpha\).

(Thus \(C', \sigma', \Sigma') \in S\).

Then we have \((C, \sigma, \Sigma) \in F(S). \text{ Thus } (C, \sigma) \text{ and } (\text{AbsGC, Sigma}) \text{ satisfy the largest simulation RGSim}\). \(\square\)

D. Examples and Their Proofs

D.1 Incrementing a Shared Variable

Some programming languages provide a single instruction to increment a variable. In a concurrent setting, such an instruction \(INC(x)\) is often understood as increasing the value of the shared variable \(x\) atomically. Compilers could have various ways to transform \(INC(x)\) to low-level machines. We present two kinds of implementations in Figure [20] \(inc(x)\) uses the compare-and-swap (CAS) instruction to obtain fine-grained atomicity, while \(inc_{\text{L}}(x)\) synchronizes reading and writing \(x\) by a global lock 1. We can view the CAS instruction \(x := \text{cas}(k, y, E_1, E_2)\) as a syntax sugar of \((y = E_1) \{y := E_2; x := 1\} \{\text{else } x := 0\}\).

To observe the value of \(x\), we use the standard \(\text{print}(E)\) operation which will produce an external event \(\text{out}(n)\) if \(E\) evaluates to \(n\). The source \(\text{PRT}(x)\) which directly prints out the value of \(x\) is transformed to two targets: \(\text{prt}(x)\) performs print in a fine-grained manner; while \(\text{prt}_{\text{L}}(x)\) uses the global lock 1 to protect accesses of \(x\).
By soundness of RGSim (Theorem 5), we come to the final result:

\( \text{inc(x)}: \text{atom} \{ x := x+1; \} \)

(a) Source Code

\[
\begin{align*}
\text{inc(x)}: & \quad \text{local d, t;} \\
0 & \quad d := 0; \\
1 & \quad \text{while (d = 0) \{ } \\
2 & \quad t := x; > \\
3 & \quad d := \text{cas} (&x, t, t+1); \\
& \quad \text{unlock(l); } \\
\end{align*}
\]

(b) Target Code: Non-Blocking and Lock-Synchronized

Figure 20. Incrementing a Shared Variable

D.1.1 Non-Blocking Implementation

The basic requirement for a fine-grained implementation is that it should not miss any increment when several threads update \( x \) concurrently.

We first define the \( \alpha \) relation between low-level and high-level states, where only the values of \( x \) are concerned:

\[ \alpha \triangleq \{ \sigma, \Sigma \mid \exists \sigma (x) = \Sigma(x) \} \]

Both \( \text{inc(x)} \) and \( \text{INC(x)} \) guarantee that they either do not update \( x \) or only increase the values of \( x \). They are executed in arbitrary environments which do not modify the thread-local variables.

\( R \triangleq \{ (\sigma, \sigma') \mid \sigma(t) = \sigma(t) \land \sigma(d) = \sigma(d) \} \)

\( G \triangleq \{ (\sigma, \sigma') \mid \sigma' = \sigma[t \leftarrow d \ldots] \}

\( R \triangleq \{ (\Sigma, \Sigma') \mid \Sigma, \Sigma' \in \text{HState} \}

\( G \triangleq \{ (\Sigma, \Sigma') \mid \Sigma' = \Sigma \lor \Sigma' = \Sigma[x \leftarrow \Sigma(x) + 1] \} \)

Then we can prove the non-blocking \( \text{inc(x)} \) does not have more behaviors than the atomic \( \text{INC(x)} \) in any environment:

\( \text{inc(x), R, G} \preceq_{\alpha, \alpha, \alpha} \text{INC(x), R, G} \)

where the pre/post conditions are the same as the invariant. On the other hand, we can also prove \( \text{INC(x)} \) refines \( \text{inc(x)} \), i.e., the latter has all the behaviors of the former. Thus \( \text{INC(x)} \) and \( \text{INC(x)} \) are actually equivalent:

\( \text{inc(x), R, G} \preceq_{\alpha, \alpha, \alpha} \text{INC(x), R, G} \) (D.1)

In other words, \( \text{inc(x)} \) and \( \text{INC(x)} \) behave just the same.

Similarly, we can prove that \( \text{prt(x)} \) and \( \text{PRT(x)} \) are equivalent:

\( \text{prt(x), R, G} \preceq_{\alpha, \alpha, \alpha} \text{PRT(x), R, G} \). (D.2)

As a simple illustration, we go on to show that the non-blocking \( \text{inc(x)} \) can be used by two threads concurrently without missing any increment, as if \( x \) was updated by the threads one after another. Formally, we prove that \( \text{(inc(x), inc(x));prt(x) and (INC(x) || INC(x));PRT(x)} \) have the same observable events when the initial values of \( x \) are the same.

By applying the rules \( \text{PAR} \) and \( \text{SEQ} \) to (D.1) and (D.2), we can get:

\( \text{inc(x) || inc(x)}; \text{PRT(x), R, G} \preceq_{\alpha, \alpha, \alpha} \text{INC(x) || INC(x)}; \text{PRT(x), R, G} \).

By soundness of RGSim (Theorem 5), we come to the final result:

\( \text{inc(x) || inc(x)}; \text{PRT(x)} \approx_{T} \text{INC(x) || INC(x)}; \text{PRT(x)} \),

for any \( T \) that respects \( \alpha \). That is, no matter how the two non-blocking threads interleave, they complete their operations with expected behaviors.

We give the complete proofs of (D.1) in Lemmas 27 and 28.

In the proofs, we find out the corresponding program points in \( \text{inc(x)} \) and \( \text{INC(x)} \) and prove their relations by co-induction. Here we use \( \text{inc(x)} \) to denote the code from line 1 to the end of the program (loops might be unrolled if needed), e.g., \( \text{inc(x)} \) is the sequence of the statement at line 3 and the whole while-loop from line 1. To simplify the proofs, we omit the cases for the stuttering state transitions made by \( \text{skip; C} \) (for some \( C \)) and even do not distinguish \( C \) and \( \text{skip; C} \) in the proofs.

Lemma 27. For all \( (\sigma, \Sigma) \in \alpha \):

1. \( (\text{inc(x)}, \sigma, R, G) \preceq_{\alpha, \alpha, \alpha} (\text{INC(x)}, \Sigma, R, G) \);

2. if \( \sigma(d) = 0 \), then \( (\text{inc(x)}, \sigma, R, G) \preceq_{\alpha, \alpha, \alpha} (\text{INC(x)}, \Sigma, R, G) \);

3. if \( \sigma(d) = 1 \), then \( (\text{inc(x)}, \sigma, R, G) \preceq_{\alpha, \alpha} (\text{skip, \Sigma, R, G}) \);

4. \( (\text{inc(x)}, \sigma, R, G) \preceq_{\alpha, \alpha} (\text{INC(x)}, \Sigma, R, G) \);

5. \( (\text{inc(x)}, \sigma, R, G) \preceq_{\alpha, \alpha} (\text{INC(x)}, \Sigma, R, G) \);

6. \( (\text{skip, \sigma, R, G}) \preceq_{\alpha, \alpha} (\text{skip, \Sigma, R, G}). \)

Proof. For each case, by co-induction.

Case: The environments are executed. The proof is trivial since \( R \) does not update \( d \).

Case: The non-blocking counter code goes one step.

1. If \( (\text{inc(x)}, \sigma) \rightarrow (\text{inc(x)}, \sigma') \), then \( \sigma'(d) = 0 \) and \( \sigma'(x) = \sigma(x) \). Correspondingly, \( \text{INC(x)} \) does not go any step:

\( (\text{INC(x)}, \Sigma) \rightarrow (\text{INC(x)}, \Sigma), (\Sigma, \Sigma) \in G^* \).

From the premise 2 we know

\( (\text{inc(x)}, \sigma', R, G) \preceq_{\alpha, \alpha} (\text{INC(x)}, \Sigma, R, G) \).

2. If \( \sigma(d) = 0 \) and \( (\text{inc(x)}, \sigma) \rightarrow (\text{inc(x)}, \sigma') \), then \( \sigma' = \sigma \).

Correspondingly:

\( (\text{INC(x)}, \Sigma) \rightarrow (\text{INC(x)}, \Sigma), (\Sigma, \Sigma) \in G^* \).

From the premise 4 we know

\( (\text{inc(x)}, \sigma', R, G) \preceq_{\alpha, \alpha} (\text{INC(x)}, \Sigma, R, G) \).

3. If \( \sigma(d) = 1 \) and \( (\text{inc(x)}, \sigma) \rightarrow (\text{skip, \sigma'}) \), then \( \sigma' = \sigma \).

Correspondingly:

\( (\text{skip, \Sigma}) \rightarrow (\text{skip, \Sigma}), (\Sigma, \Sigma) \in G^* \).

From the premise 6 we know

\( (\text{skip, \sigma', R, G}) \preceq_{\alpha, \alpha} (\text{skip, \Sigma, R, G}) \).

4. If \( (\text{inc(x)}, \sigma) \rightarrow (\text{inc(x)}, \sigma') \), then \( \sigma(x) = \sigma'(x) \).

Correspondingly:

\( (\text{INC(x)}, \Sigma) \rightarrow (\text{INC(x)}, \Sigma), (\Sigma, \Sigma) \in G^* \).

From the premise 5 we know

\( (\text{inc(x)}, \sigma', R, G) \preceq_{\alpha, \alpha} (\text{INC(x)}, \Sigma, R, G) \).

5. If \( (\text{inc(x)}, \sigma) \rightarrow (\text{inc(x)}, \sigma') \), then

(a) if \( \sigma(x) = \sigma(t) \), we have

\( \sigma' = \sigma[x \leftarrow \sigma(x) + 1, d \leftarrow 1] \).

Thus \( (\Sigma, \Sigma) \in G \). Correspondingly:

\( (\text{INC(x)}, \Sigma) \rightarrow (\text{skip, \Sigma}), (\Sigma, \Sigma) \in G + 1 \).

24
Thus we have \((\Sigma, \Sigma') \in G^* \) and \((\sigma', \Sigma') \in \alpha\). From the premise\(^2\) we know 
\[(\text{inc}(x), \sigma', R, G) \approx_{\alpha} (\text{skip}, \Sigma, R, G).\]

(b) if \(\sigma(x) \neq \sigma(t)\), then \(\sigma'(d) = 0\) and \(\sigma'(x) = \sigma(x)\).

Correspondingly:
\[
(\text{inc}(x), \Sigma) \rightarrow^0 (\text{inc}(x), \Sigma), (\Sigma, \Sigma) \in G^*.
\]

From the premise\(^2\) we know 
\[(\text{inc}(x), \sigma', R, G) \approx_{\alpha} (\text{inc}(x), \Sigma, R, G).\]

Case: Both the non-blocking and the atomic sides are skip, then they are corresponding trivially. \(\square\)

**Lemma 28.** For all \((\sigma, \Sigma) \in \alpha\):

\[1. (\text{inc}(x), \Sigma, R, G) \approx_{\alpha} (\text{inc}(x), \sigma, R, G);\]

\[2. (\text{skip}, \Sigma, R, G) \approx_{\alpha} (\text{skip}, \sigma, R, G).\]

**Proof.** By co-induction.

Case: The environments are executed. Trivial.

Case: The atomic counter code goes one step.

If \((\text{inc}(x), \Sigma) \rightarrow (\text{skip}, \Sigma')\), then \(\Sigma'(x) = \Sigma(x) + 1\). For \(\text{inc}(x)\), without the environment’s interference (i.e., \(x\) will not be modified), the statement \(d := \text{inc}(x); t := t+1\) just after \((t := x)\) will find the values of \(x\) and \(t\) are the same and succeed in updating \(x\). Thus \(\text{inc}(x)\) can be executed to \text{skip}:

\[(\text{inc}(x), \sigma) \rightarrow^\bullet (\text{skip}, \sigma'), \sigma'(x) = \sigma(x) + 1\]

Thus \((\sigma, \sigma') \in G^*\) and \((\sigma', \Sigma') \in \alpha\). Then from the premise\(^2\) we know 
\[(\text{skip}, \Sigma, R, G) \approx_{\alpha} (\text{skip}, \sigma, R, G).\]

Case: Both sides are skip, then they are corresponding trivially. \(\square\)

**D.1.2 Lock-Synchronized Implementation**

We have shown the verification of a similar transformation in Section\([4.3]\). So we omit the proof here. Also we have mechanized the proofs\([20]\) in the Coq proof assistant\([10]\).

**D.1.3 Incrementing Several Shared Variables**

We verified the transformations for \(\text{inc}(x)\) without caring about other shared resources. The FRAME rule allows us to combine several verified transformations together which work on disjoint parts of states without redoing the proofs.

For example, suppose we have another shared variable \(y\) which can be incremented as well as \(x\). It’s easy to see:

\[
(\text{inc}(y), R_1, G_1) \approx_{\alpha_1} (\text{inc}(y), R_1, G_1)
\]

where \(\alpha_1 = \{(\sigma, \Sigma) | \sigma(y) = \Sigma(y)\}\) and \(R_1, G_1\) are defined similarly as \(R, G, R\) and \(G\) except all the occurrences of \(x\) are replaced by \(y\).

By the FRAME rule and other compositionality rules, we can get:

\[
(\text{inc}(x), \Sigma, \Sigma') \approx_{\beta} (\text{inc}(y), \Sigma, y) \approx_{\beta} (\text{inc}(y), \Sigma, y)
\]

where \(\beta = \alpha \cup \alpha_1 = \{(\sigma, \Sigma) | \sigma(x) = \Sigma(x) \wedge \sigma(y) = \Sigma(y)\}\), the relies ensure that the environments cannot update any local variable used in incrementing \(x\) or \(y\), and the guarantees just say that the programs increment \(x\) or \(y\) or update local variables.

Thus we can conclude the combined transformation is correct:

\[
\text{inc}(x); \text{inc}(y); \text{prt}(x) \approx_T \text{INC}(x); \text{INC}(y); \text{PRT}(x)
\]

for any \(T\) that respects \(\beta\).

**Figure 21.** Concurrent GCD

**D.2 Concurrent GCD**

A concurrent GCD program uses two threads to compute the greatest common divisor (GCD) of the shared variables \(a\) and \(b\). One thread reads the values of \(a\) and \(b\), but only updates a if \(a > b\). Another thread does the reverse. When \(a = b\), the two threads terminate. The source program \(A_1 || A_2\) where two threads atomically update \(a\) and \(b\) respectively is transformed to a fine-grained GCD program \(C_1 || C_2\) in (Figure 21).

For the concrete fine-grained and the abstract coarse-grained GCD programs respectively, \(\text{prt}(a)\) and \(\text{PRT}(a)\) print out the results after the two threads complete their computations. Our goal is to prove that the concrete and abstract GCD programs always obtain the same result, i.e., \((C_1 || C_2); \text{prt}(a)\) and \((A_1 || A_2); \text{PRT}(a)\) have the same outputs.

By soundness 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 \(A_2\)), i.e., \(C_1^0\) is equivalent to \(A_1^0\) (and \(C_2^0\) is equivalent to \(A_2^0\)). We use \(C_1^0\) (or \(C_2^0\)) to denote the code from line 0 to line 5 in \(C_1\) (or \(C_2\)), and use \(A_1^0\) (or \(A_2^0\)) to denote the atomic block in \(A_1\) (or \(A_2\)).

It’s natural to define the \(\alpha\) relation as:

\[
\alpha = \{(\sigma, \Sigma) | \sigma(a) = \Sigma(a) \wedge \sigma(b) = \Sigma(b) \wedge (\sigma(d1) = \Sigma(d1) \wedge \sigma(d2) = \Sigma(d2))\}
\]

The threads’ guarantees and the expected environments’ behaviors can be specified as follows:

\[
R_1 = G_2 \triangleq \{(\sigma, \alpha') | \sigma(t11) = \sigma(t11) \wedge \sigma(t12) = \sigma(t12) \wedge \sigma(d1) = \sigma(d1) \wedge \sigma(a) = \sigma(a) \wedge (\sigma(a) \geq \sigma(b)) \Rightarrow \sigma(b) = \sigma(b)\}\n\]

\[
R_2 = G_1 \triangleq \{(\sigma, \alpha') | \sigma(t21) = \sigma(t21) \wedge \sigma(t22) = \sigma(t22) \wedge \sigma(d2) = \sigma(d2) \wedge \sigma(b) = \sigma(b) \wedge (\sigma(b) \geq \sigma(a)) \Rightarrow \sigma(a) = \sigma(a)\}\n\]

\[
R_1 = G_2 \triangleq \{(\Sigma, \Sigma') | \Sigma(d1) = \Sigma(d1) \wedge \Sigma(b) = \Sigma(b) \wedge \Sigma(d2) = \Sigma(d2) \wedge \Sigma(a) = \Sigma(a) \wedge (\Sigma(a) \geq \Sigma(b)) \Rightarrow \Sigma(b) = \Sigma(b)\}\n\]

\[
R_2 = G_1 \triangleq \{(\Sigma, \Sigma') | \Sigma(d2) = \Sigma(d2) \wedge \Sigma(b) = \Sigma(b) \wedge \Sigma(d1) = \Sigma(d1) \wedge \Sigma(a) = \Sigma(a) \wedge (\Sigma(b) \geq \Sigma(a)) \Rightarrow \Sigma(a) = \Sigma(a)\}\n\]
where the environment of $C_1$ (or $A_1$) is just the guarantee of $C_2$ (or $A_2$), and vice versa.

We can prove the equivalence of $C_0^1$ and $A_0^1$.

$(C_1^0, R_1, G_1) \simeq_{\alpha,\alpha} (A_0^1, R_1, G_1)$.

Then by using the rules WHILE and SEQ, we get $C_1$ and $A_2$ are equivalent:

$(C_1^0, R_1, G_1) \simeq_{\alpha,\alpha} (A_1, R_1, G_1)$.

Similarly, $C_2$ and $A_2$ are equivalent:

$(C_2, R_2, G_2) \simeq_{\alpha,\alpha} (A_2, R_2, G_2)$.

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 should be the identical transition set because the shared variables $a$ and $b$ cannot be modified when $C_1 \parallel C_2$ is computing their GCD. Its guarantee is just specified as a set of all the possible state transitions.

$R \equiv \{ (\sigma, \sigma') \mid \sigma' = \sigma \}$
$G \equiv \{ (\sigma, \sigma') \mid \sigma' \in LState \}$
$R \equiv \{ (\Sigma, \Sigma') \mid \Sigma = \Sigma' \}$
$G \equiv \{ (\Sigma, \Sigma') \mid \Sigma' \in HState \}$

It is not difficult to prove that $\text{prt}(a)$ and $\text{PRT}(a)$ are equivalent in the environments $R$ and $R$ respectively:

$(\text{prt}(a), R, G) \simeq_{\alpha,\alpha} (\text{PRT}(a), R, G)$.

By soundness of $\text{RSim}$ (Theorem 4) we obtain the result final:

$(C_1^0 \parallel C_2^0) ; \text{prt}(a) \approx_T (A_1 \parallel A_2) ; \text{PRT}(a)$,
for any $T$ that respects $\alpha$.

Thus we have proved that the concrete fine-grained and the abstract coarse-grained GCD programs can obtain the same results from the same inputs. It is not difficult to find out 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.

Below we give the detailed proofs of the similarity between $C_0^1$ and $A_0^1$ in Lemma 29 and 30.

**Lemma 29.** For all $(\Sigma, \Sigma) \in \alpha$,

1. $(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (A_0^1, \Sigma, R_1, G_1)$;
2. if $(t11) = (a)$, then $(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (A_0^1, \Sigma, R_1, G_1)$;
3. if $(t11) = (a)$ and $(t11) < \sigma(t12)$, then $(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (\text{skip}, \Sigma, R_1, G_1)$;
4. if $(t11) = (a)$, $(t12) = (b)$ and $(t11) \geq \sigma(t12)$, then $(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (A_0^1, \Sigma, R_1, G_1)$;
5. if $(t11) = (a)$, $(t12) = (b) \text{ and } (t11) = \sigma(t12)$, then $(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (A_0^1, \Sigma, R_1, G_1)$;
6. if $(t11) = (a)$, $(t11) \leq \sigma(t12)$ and $(t12) = \sigma(t12)$, then $(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (\text{skip}, \Sigma, R_1, G_1)$;
7. if $(t11) = (a)$, $(t12) = (b) \text{ and } (t11) \geq \sigma(t12)$, then $(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (A_0^1, \Sigma, R_1, G_1)$;
8. if $(t11) = (a)$, $(t12) = (b) \text{ and } (t11) > \sigma(t12)$, then $(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (A_0^1, \Sigma, R_1, G_1)$;
9. if $(t11) = (a)$, $(t11) < \sigma(t12)$ and $(t12) = \sigma(t12)$, then $(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (\text{skip}, \Sigma, R_1, G_1)$.

**Proof.** For each case, by co-induction.

**Case:** The environments are executed. The proof is trivial since the conditions for each case are just preserved under the transitions made by $R_1$ and $R_1$.

**Case:** The concrete GCD goes one step.

1. if $(C_0^1, \sigma) \rightarrow (C_1^0, \sigma')$, then $(\sigma', \sigma) \in G_1$. Correspondingly, the abstract code does not go any step:

$A_0^1 \rightarrow A_0^1, (\Sigma, \Sigma) \in G_1^*$.

From the premise we know

$(C_1^0, \sigma', R_1, G_1) \simeq_{\alpha, \alpha} (A_0^1, \Sigma, R_1, G_1)$.

2. if $\sigma(t11) = (a)$ and $(C_1^0, \sigma) \rightarrow (C_1^0, \sigma')$, then $(\sigma', \sigma) \in G_1$. Thus on the atomic side:

$A_0^1 \rightarrow (\text{skip}, \Sigma), (\Sigma, \Sigma) \in G_1^*$.

From the premise we know

$(C_1^0, \sigma', R_1, G_1) \simeq_{\alpha, \alpha} (\text{skip}, \Sigma, R_1, G_1)$.

3. if $(t11) = (a)$ and $(t11) < \sigma(t12)$, then $(C_1^0, \sigma) \rightarrow (C_1^0, \sigma)$. On the atomic side:

$A_0^1 \rightarrow (\text{skip}, \Sigma), (\Sigma, \Sigma) \in G_1^*$.

From the premise we know

$(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (\text{skip}, \Sigma, R_1, G_1)$.

4. if $\sigma(t11) = (a)$ and $(t12) = (b)$ and $(t11) = \sigma(t12)$, then $(C_1^0, \sigma) \rightarrow (C_1^0, \sigma)$. Correspondingly, the atomic code does not go any step:

From the premise we know

$(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (A_0^1, \Sigma, R_1, G_1)$.

5. if $\sigma(t11) = (a)$, $(t12) = (b)$ and $(t11) = \sigma(t12)$, then $(C_1^0, \sigma) \rightarrow (C_1^0, \sigma)$. From the $\alpha$ relation, we know $(\Sigma(a)) < (\Sigma(b))$. Thus on the atomic side:

$A_0^1 \rightarrow (\text{skip}, \Sigma), (\Sigma, \Sigma) \in G_1^*$.

Thus $(\Sigma, \Sigma') \in G_1^*$ and $(\Sigma', \Sigma') \in \alpha$.

From the premise we know

$(C_1^0, \sigma', R_1, G_1) \simeq_{\alpha, \alpha} (\text{skip}, \Sigma, R_1, G_1)$.

6. if $\sigma(t11) = (a)$, $(t11) \leq \sigma(t12)$ and $(t12) = \sigma(t12)$, then $(C_1^0, \sigma) \rightarrow (\text{skip}, \sigma)$. From the premise we know

$(C_1^0, \sigma, R_1, G_1) \simeq_{\alpha, \alpha} (\text{skip}, \Sigma, R_1, G_1)$.
We first define a predicate \( m_s \models \text{list}(x, A) \) to represent a singly-linked list at the current library state \( m_s \) whose head node’s address is \( x \) and values form a sequence \( A \). The domain of \( m_s \) is the set of all the nodes’ addresses of the list.

\[
\begin{align*}
\text{list}(x, A) & \triangleq (\text{dom}(m_s) = \phi \land x = \text{null} \land A = \epsilon) \\
& \lor (\exists \alpha. \exists y. \exists \beta. m_s(x) = (v, y) \land A = v :: B \\
& \land m_s \setminus \{x\} \models \text{list}(y, B))
\end{align*}
\]

For the high-level and low-level library states, we only consider the value sequence on the stack:

\[
\begin{align*}
\text{shared_map}(m_s, M_s) & \triangleq \exists \alpha_s. \substack{\alpha_s \models \text{list}(m_s(S), M_s(A)) \\
\land \alpha_s \subseteq m_s(S)}
\end{align*}
\]

It requires that the concrete library state \( m_s \) has a sub-state \( \alpha_s \) of a linked list as the stack, and the concrete stack has the same value sequence as the abstract one. Since \( S \) is a shared variable containing the address of the top node, it itself is not in the domain of \( \alpha_s \). On the other hand, for each thread \( t \), the value of \( v \) in the low-level local state should be the same as in the high-level local state, and the low-level local state should provide enough additional space needed by the object operations (i.e., the local variables \( d, x, t \) and \( r \)).

\[
\begin{align*}
\text{local_map}(m_t, M_t) & \triangleq \exists \alpha. \exists \beta. m_t(v) = M_t(v) \\
\land \exists m'_t, m_t = m'_t \mid (d \mapsto \_, x \mapsto t \mapsto \_, r \mapsto \_)
\end{align*}
\]

Then \( \alpha \) is defined as follows:

\[
\alpha \triangleq \{ ((\pi, m_s), (\Pi, M_s)) \mid \text{shared_map}(m_s, M_s) \\
\land \forall t \in \text{dom}(\Pi). \text{local_map}(\pi(t), \Pi(t)) \}
\]

The program guarantees and consistency can be specified as follows:

\[
\begin{align*}
G_{\text{push}}(t) & \triangleq \{ ((\pi \models t \rightarrow m_t), (\pi \models t \rightarrow m'_t), m'_t) \\
& \mid \exists x. \exists \alpha. \exists \beta. m_t(x) \notin \text{dom}(m_t) \\
& \land m'_t = m_t(S \rightarrow x) \mid \{ x \mapsto (v, m_s(S)) \}
\end{align*}
\]

\[
\begin{align*}
G_{\text{pop}}(t) & \triangleq \{ ((\pi \models t \rightarrow m_t), (\pi \models t \rightarrow m'_t), m'_t) \\
& \mid \exists x. \exists \alpha. \exists \beta. m_t(S) = x \land m_s(x)(v, y) \\
& \land m'_t = m_s(S \rightarrow y) \}
\end{align*}
\]

\[
\begin{align*}
G_{\text{local}}(t) & \triangleq \{ ((\pi \models t \rightarrow m_t), (\pi \models t \rightarrow m'_t), m'_t) \\
& \mid \exists x. \exists \alpha. \exists \beta. m_t(S) = x \land m_s(x)(v, y) \\
& \land m'_t = m_s(S \rightarrow y) \}
\end{align*}
\]

\[
\begin{align*}
G(t) & \triangleq G_{\text{push}}(t) \cup G_{\text{pop}}(t) \cup G_{\text{local}}(t) \\
R(t) & \triangleq \bigcup_{t' \neq t} G(t')
\end{align*}
\]

The ownership transfer in \( \text{push}(v) \) is reflected in the guarantee \( G_{\text{push}}(t) \), where the node \( x \) is transferred from the client state to the library state. A client thread guarantees only performing push and pop operations and local operations, and it is executed concurrently with other client threads.

We prove the non-blocking stack operations are simulated by the corresponding atomic operations in Lemmas 31 and 32:

\[
\begin{align*}
& (t.\text{push}(v), R(t), G(t)) \models \alpha_{\text{push}}(t, \text{push}(v), R(t), G(t)) \\
& (t, r := \text{pop}()), R(t), G(t)) \models \alpha_{\text{pop}}(t, \text{pop}()), R(t), G(t))
\end{align*}
\]

This gives us the atomicity of the non-blocking implementation of the stack object.

**Lemma 31.** For all \((\sigma, \Sigma) \in \alpha\) where \( \sigma = (\pi, m_s) \) and \( \Sigma = (\Pi, M_s) \),

1. \((\text{push}(v), \sigma, R(t), G(t)) \models \alpha_{\text{push}}(\text{push}(v), \Sigma, R(t), G(t)) \)
2. if there exists \( x \) such that \( \pi(t)(x) = x \) and \( \pi(t)(x) = (\_ , \_ ) \),
then \((\text{push}^1(v), \sigma, R(t), G(t)) \models \alpha_{\text{push}}^1(\text{push}(v), \Sigma, R(t), G(t)) \)

\[
\begin{align*}
\text{D.3 Treiber's Non-Blocking Stack}
\end{align*}
\]

We prove atomicity of Treiber’s non-blocking stack. As shown in Figure 22, the stack interface consists of two operations: \( \text{push}(v) \) and \( \text{POP}() \). The abstract stack \( A \) is a value sequence and the operations are executed atomically. \( \text{push}(v) \) and \( \text{POP}() \) are transformed to the non-blocking programs \( \text{push}(v) \) and \( \text{pop()} \) respectively, where the stack is implemented as a singly-linked list pointed to by a shared variable \( S \). The non-blocking stack uses the CAS instruction to obtain fine-grained atomicity.

\[
\begin{align*}
\text{push}(v) : & \quad \text{POP}() : \\
\text{local } r; & \quad \text{local } r; \\
0 \text{ atom} \{ & \quad 0 \text{ atom} \{ \\
\text{if } (A = \epsilon) \{ & \quad \text{if } (A = \epsilon) \{ \\
r := \text{EMPTY}; & \quad r := \text{EMPTY}; \\
\} \text{ else } \{ & \} \text{ else } \{ \\
A := v :: A; & \quad A := t1(A); \\
\} \} \\
\} \} \\
\}
\end{align*}
\]
3. If there exists $x$ such that $\pi(t)(x) = x$ and $\pi(t)(x) = (\pi(t)(v), \_)$, then $(\text{push}^3(v), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (\text{push}(v), \Sigma, R(t), G(t))$.
4. If $\pi(t)(d) = 1$, then $(\text{push}^3(v), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (\text{skip}, \Sigma, R(t), G(t))$.
5. If $\pi(t)(d) = 0$ and there exists $x$ such that $\pi(t)(x) = x$ and $\pi(t)(x) = (\pi(t)(v), \_)$, then $(\text{push}^3(v), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (\text{push}(v), \Sigma, R(t), G(t))$.
6. If there exists $x$ such that $\pi(t)(x) = x$ and $\pi(t)(x) = (\pi(t)(v), \_)$, then $(\text{push}^3(v), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (\text{push}(v), \Sigma, R(t), G(t))$.
7. If there exists $x$ such that $\pi(t)(x) = x$ and $\pi(t)(x) = (\pi(t)(v), \_)$, then $(\text{push}^3(v), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (\text{push}(v), \Sigma, R(t), G(t))$.
8. If there exists $x$ such that $\pi(t)(x) = x$ and $\pi(t)(x) = (\pi(t)(v), \_)$, then $(\text{push}^3(v), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (\text{push}(v), \Sigma, R(t), G(t))$.

If $(\text{push}, \sigma) \rightarrow (\text{push}^1, \sigma')$, then there exists $x$ such that
\[
\pi' = \pi \{ t \leftarrow t \{ x \mapsto x \} \} \cup \{ x \mapsto (\_ \_ \_ ) \}, m'_s = m_s.
\]

Correspondingly, the atomic code does not go any step:
\[
(\text{push}(\Sigma), \Sigma, \Sigma, \Sigma) \in G(t)^*.
\]

From the premises we know
\[
(\text{push}^1, \sigma', R(t), G(t)) \leq_{\alpha, \omega} (\text{push}(\Sigma), \Sigma, R(t), G(t)).
\]

2. Similar to the first case (but using the premises and omitted).
3. Similar to the first case (but using the premises and omitted).
4. If $\pi(t)(d) = 1$, then
\[
(\text{push}^3, \sigma) \rightarrow (\text{skip}, \sigma).
\]

Correspondingly, on the atomic side:
\[
(\text{skip}(\Sigma), \Sigma, \Sigma, \Sigma) \in G(t)^*.
\]

From the premises we know
\[
(\text{skip}, \Sigma, R(t), G(t)) \leq_{\alpha, \omega} (\text{skip}(\Sigma), \Sigma, R(t), G(t)).
\]

5. Similar to the first case (but using the premises and omitted).
6. Similar to the first case (but using the premises and omitted).
7. Similar to the first case (but using the premises and omitted).
8. If $(\text{push}^6, \sigma) \rightarrow (\text{push}^4, \sigma')$, then
   (a) if $m_s(S) = \pi(t)(t)$, then $(\sigma, \sigma') \in G(t)$ and
   \[
   \pi' = \pi \{ t \leftarrow t \{ d \mapsto 1 \} \}
   \]
   \[
   m'_s = m_s(S \times x) \cup \{ x \mapsto (\pi(t)(v), m_s(S)) \}
   \]
   where $x = \pi(t)(x)$.
   
   Correspondingly, on the atomic side:
   \[
   (\text{push}, \Sigma) \rightarrow (\text{skip}, \Sigma'), (\Sigma, \Sigma', \sigma') \in G(t)^*.
   \]

   where
   \[
   \Pi' = \Pi
   \]
   \[
   M'_s = M_s \{ A \mapsto \Pi(t)(v) :: M_s(A) \}
   \]

We know $\pi(t)(v) = \Pi(t)(v)$. And there exists a sub-state $\sigma_s$ such that
\[
\sigma_s \models \text{list}(m_s(S), M_s(A)).
\]

Let
\[
\sigma'_s = \sigma_s \cup \{ m'_s(S) \mapsto (\pi(t)(v), m_s(S)) \}.
\]

Then from (D.1) and (D.2), we can prove that
\[
\sigma'_s \models \text{list}(m'_s(S), \Pi(t)(v) :: M_s(A)).
\]

Moreover, $\sigma'_s$ is a sub-state of $m'_s(S)$. Thus shared_map($m'_s(S), M'_s(A)$).

As a result, we have $(\sigma', \Sigma') \in \alpha$. From the premises we know
\[
(\text{push}^3, \sigma', R(t), G(t)) \leq_{\alpha, \omega} (\text{skip}, \Sigma', R(t), G(t)).
\]

(b) if $m_s(S) \neq \pi(t)(t)$, then
\[
\pi' = \pi \{ t \mapsto \pi(t)(d \mapsto 0) \} \text{ and } m'_s = m_s.
\]

Correspondingly, the atomic code does not go any step. Then from the premises we know
\[
(\text{push}^3, \sigma', R(t), G(t)) \leq_{\alpha, \omega} (\text{push}, \Sigma', R(t), G(t)).
\]

Case: Both sides are skip, then they are corresponding trivially. □

For the POP operation, we assume the returned value $x$ will be assigned to a variable $r$.

Lemma 32. For all $(\sigma, \Sigma) \in \alpha$ where $\sigma = (\pi, m_\pi)$ and $\Sigma = (\Pi, M_\Sigma)$,

1. $(r := \text{pop}(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))$.
2. If $\pi(t)(d) = 0$, then
   \[
   (r := \text{pop}^1(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))
   \]
3. If $\pi(t)(d) = 1$ and $\pi(t)(x) = \Pi(t)(x)$, then
   \[
   (r := \text{pop}^2(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))
   \]
4. $(r := \text{pop}^3(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))$.
5. If there exists $x$ such that $\pi(t)(x) = \pi(t)(x)$, then
   \[
   (r := \text{pop}^3(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))
   \]
6. If $\pi(t)(v) = \text{null}$ and $\Pi(t)(v) = \text{EMPTY}$, then
   \[
   (r := \text{pop}^3(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))
   \]
7. If $\Pi(t)(x) = \text{EMPTY}$, then
   \[
   (r := \text{pop}^3(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))
   \]
8. If $\pi(t)(x) = \Pi(t)(x)$, then
   \[
   (r := \text{pop}^3(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))
   \]
9. If there exists $x$ such that $\pi(t)(x) = \pi(t)(x)$, then
   \[
   (r := \text{pop}^3(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))
   \]
10. If there exists $x$ such that $\pi(t)(x) = \pi(t)(x)$ and $m_s(S) = x \implies m_s(x) = (\pi(t)(x), \_)$, then
    \[
    (r := \text{pop}^3(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))
    \]
11. If there exists $x$ such that $\pi(t)(x) = \pi(t)(x)$ and $m_s(S) = x \implies m_s(x) = (\pi(t)(x), \pi(t)(x))$, then
    \[
    (r := \text{pop}^3(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))
    \]
12. If $\pi(t)(x) = \Pi(t)(x)$, then
    \[
    (r := \text{pop}^3(\cdot), \sigma, R(t), G(t)) \leq_{\alpha, \omega} (r := \text{pop}(\cdot), \Sigma, R(t), G(t))
    \]
Proof. By co-induction.
Case: The environments are executed. Trivial.
Case: The concrete pop operation goes one step.
1. If \( r := \text{pop}(s), \sigma \longrightarrow (r := \text{pop}^1(s), \sigma') \), then
   \[ m'_s = m_s, \pi' = \pi \{ t \leadsto \pi(t) \{ d \leadsto 0 \} \}. \]
   Thus \((\sigma, \sigma') \in \mathcal{G}(t)\). Correspondingly, the atomic code does not go any step. From the premise 2 we know
   \[ (r := \text{pop}^1(s), \sigma', \mathcal{R}(t), \mathcal{G}(t)) \preceq_{\alpha, \alpha} \]
   \[ (r := \text{pop}(s), \Sigma, \mathcal{R}(t), \mathcal{G}(t)) \]

2. Similar and omitted (using the premise 4).
3. If \( \pi(t)(d) = 1 \), then
   \[ (r := \text{pop}^1(s), \sigma) \longrightarrow (r := x, \sigma) \]
   Correspondingly, the atomic code does not go any step. From the premise 2 we know
   \[ (r := x, \sigma, \mathcal{R}(t), \mathcal{G}(t)) \preceq_{\alpha, \alpha} (r := x, \Sigma, \mathcal{R}(t), \mathcal{G}(t)) \]
   \[ \]
4. If \( r := \text{pop}^2(s), \sigma \longrightarrow (r := \text{pop}^3(s), \sigma') \), then
   \[ m'_s = m_s, \pi' = \pi \{ t \leadsto \pi(t) \{ t \leadsto m_s(S) \} \} \]
   (a) If \( m_s(S) = \text{null} \), then \( \pi'(t)(t) = \text{null} \).
   Thus there exists \( \hat{s}_s \) such that
   \[ \hat{s}_s \vdash \text{list(null, } M_s(A)) \]
   Thus \( M_s(A) = \epsilon \).
   Correspondingly, on the atomic side:
   \[ (r := \text{pop}(s), \Sigma) \longrightarrow (r := x, \Sigma'), (\Sigma, \Sigma') \in \mathcal{G}(t)^* \]
   where
   \[ M'_s = M_s, \Pi' = \Pi \{ t \leadsto \Pi(t) \{ t \leadsto \text{EMPTY} \} \} \]
   From the premise 2 we know
   \[ (r := \text{pop}^1(s), \sigma', \mathcal{R}(t), \mathcal{G}(t)) \preceq_{\alpha, \alpha} \]
   \[ (r := x, \Sigma', \mathcal{R}(t), \mathcal{G}(t)) \]
   (b) If \( m_s(S) \neq \text{null} \), then from
   \[ \hat{s}_s \vdash \text{list(m_s(S), } M_s(A)) \]
   we know there exists \( x \) such that \( \pi'(t)(t) = x \) and \( m'_s(x) = \{ x \} \).
   Correspondingly, the atomic code does not go any step. From the premise 2 we know
   \[ (r := \text{pop}^1(s), \sigma', \mathcal{R}(t), \mathcal{G}(t)) \preceq_{\alpha, \alpha} \]
   \[ (r := \text{pop}(s), \Sigma, \mathcal{R}(t), \mathcal{G}(t)) \]

5. Similar and omitted (using the premise 9).
6. Similar and omitted (using the premise 7).
7. Similar and omitted (using the premise 8).
8. Similar and omitted (using the premise 3).
9. Similar and omitted (using the premise 10).
10. Similar and omitted (using the premise 11).
11. If \( r := \text{pop}^3(s), \sigma \longrightarrow (r := \text{pop}^4(s), \sigma') \), then
   (a) if \( m_s(S) = \pi(t)(t) = x \), then \((\sigma, \sigma') \in \mathcal{G}(t)\) and
   \[ \pi' = \pi \{ t \leadsto \pi(t) \{ d \leadsto 1 \} \} \]
   \[ m'_s = m_s \{ s \leadsto \pi(t)(x) \} \]
   From \((\sigma, \Sigma) \in \alpha\), there exists \( \hat{s}_s \) such that
   \[ \hat{s}_s \vdash \text{list}(x, M_s(A)) \]
   Since \( (D.2) \) and
   \[ m_s(x) = (\pi(t)(x), \pi(t)(x)) \]
   there exists \( B \) such that
   \[ M_s(A) = \pi(t)(x) : B, \]
   \[ \hat{s}_s \vdash \{ x \} \vdash \text{list}(\pi(t)(x), B) \]
   Correspondingly, on the atomic side:
   \[ (r := \text{pop}(s), \Sigma) \longrightarrow (r := x, \Sigma'), (\Sigma, \Sigma') \in \mathcal{G}(t)^* \]
   where
   \[ \Pi'(t)(x) = \pi(t)(x), M'_s(A) = B \]
   From \( (D.4) \) and \( (D.5) \), we have
   \[ \hat{s}_s \vdash \{ x \} \vdash \text{list}(m'_s(S), M'_s(A)) \]
   Since \( (D.1) \) and \( \hat{s}_s \) is a sub-state of \( m_s \{ \{ \} \} \), we know
   \[ \hat{s}_s \vdash \{ x \} \leq m'_s \{ \{ \} \} \]
   From \( (D.6) \) and \( (D.7) \), we have
   \[ \pi(t)(x) \pi'(t)(x) \]
   From \( (D.2) \) and \( (D.3) \), we have
   \[ m_s(x) \neq \pi(t)(x) \]
   Thus from \( (D.1) \), \( (D.3) \), \( (D.9) \) and the premise 3 we know
   \[ (r := \text{pop}^1(s), \sigma', \mathcal{R}(t), \mathcal{G}(t)) \preceq_{\alpha, \alpha} \]
   \[ (r := x, \Sigma', \mathcal{R}(t), \mathcal{G}(t)) \]
   (b) if \( m_s(S) \neq \pi(t)(x) \), then
   \[ m'_s = m_s, \pi' = \pi \{ t \leadsto \pi(t) \{ d \leadsto 0 \} \} \]
   Correspondingly, the atomic code does not go any step. Then from the premise 2 we know
   \[ (r := \text{pop}^1(s), \sigma', \mathcal{R}(t), \mathcal{G}(t)) \preceq_{\alpha, \alpha} \]
   \[ (r := \text{pop}(s), \Sigma, \mathcal{R}(t), \mathcal{G}(t)) \]
Case: Both sides are \( r := x \), the proof is trivial. \( \Box \)

D.4 Lock-Coupling List

To prove \( \text{add}(e) \) refines \( \text{ADD}(e) \), we analyze the algorithm step by step and find out the commands whose executions correspond to the high-level single atomic step \( i.e., \) the linearization points. Since we require the elements in the concrete list are those in the abstract set, we pick line 15 as the linearization point of a successful call where the new node containing the value \( e \) is inserted into the list. For unsuccessful calls \( e \) is already in the set, we choose lines 3 and 9 where the value \( e \) is read from an existing list node. Similarly, for \( \text{remove}(e) \), we choose line 13 (for successful calls) and lines 3 and 9 (for unsuccessful calls) as linearization points.

From the definition of \( \mathcal{G}_{\text{list}}(t) \), we can find that when the thread \( t \) holds the lock of a node, it can only delete the node from the list, update its \( \text{next} \) field or release the lock; otherwise, it cannot update the node’s fields nor delete its next node. The data field of a list node will never be updated. The algorithm takes advantage of these knowledges and safely reads a node’s data field when holding only its predecessor’s lock. We successfully handle these subtle issues in our proofs. Moreover, our proofs illustrate that after the current thread releases the lock of a node, it does not care about the node any more, which coincides with the fact that the environment can then manipulate the node. We also deal with ownership transfers and dynamic allocation and deallocation in our proofs.
Lemma 33. For all $(\sigma, \Sigma) \in \alpha$ where $\sigma = (\pi, m_s)$ and $\Sigma = (\Pi, M_s)$,
1. $(t.\texttt{add}(e), \sigma, \mathcal{T}(t), \mathcal{G}(t)) \preceq_{\alpha, 0} (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
2. if there exists $x$ such that $\pi(t(x)) = m_s(\texttt{Head}) = x$, then $(t.\texttt{add}^1(e), \sigma, \mathcal{T}(t), \mathcal{G}(t)) \preceq_{\alpha, 0} (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
3. if there exist $x$ and $z$ such that $\pi(t(x)) = m_s(\texttt{Head}) = x$ and $m_s(z) = (t.\texttt{MIN-VAL}, z)$, then $(t.\texttt{add}^2(e), \sigma, \mathcal{T}(t), \mathcal{G}(t)) \preceq_{\alpha, 0} (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
4. if there exist $x$ and $z$ such that $\pi(t(x)) = m_s(\texttt{Head}) = x$ and $m_s(z) = (t.\texttt{MIN-VAL}, z)$ and $\pi(t(z)) = z$, then $(t.\texttt{add}^3(e), \sigma, \mathcal{T}(t), \mathcal{G}(t)) \preceq_{\alpha, 0} (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
5. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
6. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
7. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
8. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
9. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
10. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
11. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
12. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
13. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
14. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
15. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
16. if there exist $x, z, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{add}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$

Lemma 34. For all $(\sigma, \Sigma) \in \alpha$ where $\sigma = (\pi, m_s)$ and $\Sigma = (\Pi, M_s)$,
1. $(t.\texttt{mv}(e), \sigma, \mathcal{T}(t), \mathcal{G}(t)) \preceq_{\alpha, 0} (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
2. if there exists $x$ such that $\pi(t(x)) = m_s(\texttt{Head}) = x$, then $(t.\texttt{mv}^1(e), \sigma, \mathcal{T}(t), \mathcal{G}(t)) \preceq_{\alpha, 0} (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
3. if there exist $x$ and $y$ such that $\pi(t(x)) = m_s(\texttt{Head}) = x$ and $m_s(x) = (t.\texttt{MIN-VAL}, y)$, then $(t.\texttt{mv}^2(e), \sigma, \mathcal{T}(t), \mathcal{G}(t)) \preceq_{\alpha, 0} (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
4. if there exist $x$ and $y$ such that $\pi(t(x)) = m_s(\texttt{Head}) = x$ and $m_s(x) = (t.\texttt{MIN-VAL}, y)$ and $\pi(t(y)) = y$, then $(t.\texttt{mv}^3(e), \sigma, \mathcal{T}(t), \mathcal{G}(t)) \preceq_{\alpha, 0} (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
5. if there exist $x, y, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
6. if there exist $x, y, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
7. if there exist $x, y, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
8. if there exist $x, y, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
9. if there exist $x, y, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
10. if there exist $x, y, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
11. if there exist $x, y, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
12. if there exist $x, y, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$
13. if there exist $x, y, e, v$ and $u$ such that $\pi(t(e)) = e$ and $\pi(t(z)) = z$ and $\pi(t(u)) = u$ and $m_s(x) = (t.\texttt{mv}(e), \Sigma, \mathcal{T}(t), \mathcal{G}(t));$

Proof. By co-induction. □
tion which introduces a local variable k.

Correctness of the two transformations are formalized as follows:

is transformed to the target new boundary outside the loop.

while(k<r) {
  m = π
  if (m) {
    m = t
    rmv
    x
  }
}

Medium-Level

\begin{align*}
\text{D.5 Strength Reduction and Induction Variable Elimination}
\end{align*}

\begin{itemize}
  \item \( C_1 \)
  \item \( C_2 \)
  \item \( t \)
  \item \( t \)
  \item \( t \)
  \item \( t \)
  \item \( t \)
\end{itemize}

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 while-condition. We assume \( n \) and \( r \) will not be updated by the target environment, so we can compute the new boundary outside the loop.

\( R_2 \triangleq \{ \sigma_2(k) = \sigma_2(k) \wedge \sigma_2(r) = \sigma_2(r) \wedge \sigma_2(n) = \sigma_2(n) \} \)

\( R_1 \triangleq \{ \sigma_1(k) = \sigma_1(k) \land \sigma_1(n) = \sigma_1(n) \} \)

\( G \triangleq \text{True} \)

\( \text{D.5 Strength Reduction and Induction Variable Elimination} \)

\begin{itemize}
  \item \( \{\text{Source-Level } C \} \)
  \item \( \{\text{Medium-Level } C_1 \} \)
  \item \( \{\text{Target-Level } C_2 \} \)
\end{itemize}

Correctness of the two transformations are formalized as follows:

\( (C_2, R_2, G) \leq_{\alpha, \beta, \kappa, \beta} (C_1, R_1, G), (C_1, R_1, G) \leq_{\beta, \beta, \kappa, \beta} (C, R, G) \)

where

\( \alpha \triangleq \{ (\sigma_2(s_1) \mid \sigma_2(k) = \sigma_1(k) \land \sigma_2(n) = \sigma_1(n) \wedge \sigma_2(x) = \sigma_1(x) \} \)

\( \beta \triangleq \{ (\sigma_1(s_1) \mid \sigma_1(s_1) = \sigma_1(s_1) \land \sigma_1(n) = \sigma_1(n) \wedge \sigma_1(x) = \sigma_1(x) \} \)

The proofs are not difficult by the \( \text{RGSim} \) definition or by the optimization rules.
A GC thread is introduced on the low-level which can use privilege commands to control the mutator threads and manage the heap, e.g., \( x := \text{get_root}(y) \) allows the GC to read the values of all the pointer variables in the thread \( y \)'s store at once and \( \text{free}(x) \) allows to reclaim an object. The stop-the-world phase can be implemented by \textit{atomic} \( \{C\} \) in which the GC does some work \( C \) without being interrupted by mutator threads.

An object has \( m \) pointer fields and a data field from the high-level view, whereas a concrete object has two auxiliary fields \( \text{color} \) and \( \text{dirty} \) for the collection. We give each object a dirty card whose value can be 0 (not dirty) or 1 (dirty). The \( \text{color} \) field has three possible values and is used for two purposes: for marking, we use \( \text{BLACK} \) for a marked object and \( \text{WHITE} \) for an unmarked one; and for allocation, we use \( \text{BLUE} \) for an unallocated object which will neither be traced nor be reclaimed, but can be allocated later. New objects are created \( \text{BLACK} \), and when reclaiming an object, we just set its color to \( \text{BLUE} \).

The high-level language is typed in the sense that heap locations and integers are regarded as distinct kinds of values. But on the low-level machine, they are not distinguished to allow the GC to perform pointer arithmetics. On the other hand, every 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 roots. Note that 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 \( \text{nil} \) for null pointers and it will be translated to 0 on the low-level machine. We assume there is a bijective function from high-level locations to low-level integers:

\[
\text{Loc2Int} : \text{Loc} \mapsto [0..M]
\]

which satisfies \( \text{Loc2Int}(\text{nil}) = 0 \).

We present the high-level operational semantics rules and the detailed definition of \( \text{AbsGCStep} \) in Figure 24. Here we use \( \text{same_type}(V, V') \) to mean that the two values \( V \) and \( V' \) are of the same type (\text{Int} or \text{Loc}).

---

**Figure 24. A High-level Garbage-Collected Machine**
\[
\begin{align*}
\varphi_1(s, b) &= \begin{cases} n & \text{if } b = 0 \text{ or } b = 2 \\ 0 & \text{if } b = 1 \text{ and } n = 0 \\ ∅ & \text{otherwise} \end{cases} \\
\varphi_2(s, b) &= \begin{cases} n & \text{if } s(x) = (n, b') \text{ and } (b = b' \lor b = 2) \\ ∅ & \text{otherwise} \end{cases} \\
[E_1 + E_2](s, b) &= \begin{cases} n_1 + n_2 & \text{if } [E_1](s, b) = n_1 \text{ and } [E_2](s, b) = n_2 \text{ and } (b = 0 \lor b = 2) \\ ∅ & \text{otherwise} \end{cases} \\
[E_1 = E_2](s, b) &= \begin{cases} \text{true} & \text{if } [E_1](s, b) = n_1 \text{ and } [E_2](s, b) = n_2 \text{ and } n_1 = n_2 \text{ and } (b = 0 \lor b = 2) \\ \text{false} & \text{if } [E_1](s, b) = n_1 \text{ and } [E_2](s, b) = n_2 \text{ and } n_1 \neq n_2 \text{ and } (b = 0 \lor b = 2) \\ ∅ & \text{otherwise} \end{cases} \\
\text{is_empty}(x)(s, b) &= \begin{cases} \text{true} & \text{if } b = 0 \text{ and } s(x) = ϵ \\ \text{false} & \text{if } b = 0 \text{ and } s(x) = n::A \\ ∅ & \text{otherwise} \end{cases}
\end{align*}
\]

Figure 25. Expression Evaluations on the Low-Level Machine

For the low-level machine, we need to prohibit mutators from pointer arithmetics (although the GC is allowed to do so). Thus an expression is evaluated (shown in Figure 25) under the store with an extra bit \( b \) to indicate whether it is used as an object location in the heap. When \( b = 2 \), 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 26.

E.2 The GC Code

```c
int WHITE = 0;
int BLACK = 1;
int BLUE = 2;

Collection() {
  local mstk: Seq(Int); // initial: EMPTY
  while (true) {
    Initialize();
    Trace();
    CleanCard();
    atomic{ ScanRoot(); CleanCard(); }
    Sweep();
  }
}

Initialize() {
  local i: [1..M], c: {BLACK, WHITE, BLUE};
  i: =1 ;
  while (i <= M) {
    i.dirty := 0;
    c := i.color;
    if (c = BLACK) { i.color := WHITE; }
    i: = i+1;
  }
}

Trace() { // non-recursive
  local t: [1..N], rt: Set(Int), i: [0..M];
  t := 1;
  while (t <= N) { // for each thread
    rt := get_root(t);
    foreach i in rt do {
      MarkAndPush(i);
    }
    t := t + 1;
    TraceStack();
  }
}

TraceStack() {
  local i: [1..M], j: [0..M];
  while (is_empty(mstk)) {
    i := pop(mstk);
    j := i.pt1; MarkAndPush(j);
    ...
    j := i.ptm; MarkAndPush(j);
  }
}

Mark(i) {
  local c: {BLACK, WHITE, BLUE};
  if (i != 0) {
    c := i.color;
    if (c = WHITE) {
      i.color := BLACK;
      push(i, mstk);
    }
  }
}

CleanCard() {
  local i: [1..M], c: {BLACK, WHITE, BLUE}, d: {1, 0};
  i: =1 ;
  while (i <= M) {
    c := i.color;
    d := i.dirty;
    if (d = 1) {
      i.dirty := 0;
      if (c = BLACK) {
        push(i, mstk);
      }
    }
    i: = i+1;
  }
  TraceStack();
}

ScanRoot() {
  local t: [1..N], rt: Set(Int), i: [0..M];
  t := 1;
  while (t <= N) {
    rt := get_root(t);
    foreach i in rt do {
      MarkAndPush(i);
    }
    t := t + 1;
  }
}
```

33 2011/11/5
\[
\begin{align*}
\text{tid} \in [1..N] \quad & s(x) = (n', b) \quad & |E|_{s,h} = n \\
(\text{tid}(x := E), (s,h)) & \rightarrow (\text{tid} \cdot \text{skip}, (s(x \mapsto (n,b), h))) \\
\text{tid} = t_{gc} \quad & s(x) = (n', b) \quad & |E|_{(n,2)} = n \\
(\text{tid}(x := E), (s,h)) & \rightarrow (\text{tid} \cdot \text{skip}, (s(x \mapsto (n,b), h))) \\
s(y) = (n_y, 1) \quad & h(n_y)(id) = n \quad & s(x) = (n_x, b) \quad id \in \{p_1, \ldots, p_m\} \implies b = 1 \quad id \in \{\text{data}\} \implies b = 0 \\
(x := y.id, (s,h)) & \rightarrow (\text{skip}, (s(x \mapsto (n,b), h))) \\
& \quad \text{if} \quad y \notin \text{dom}(s) \quad \text{or} \quad \text{fst}(s(y)) \notin \text{dom}(h) \quad \text{or} \quad \text{snd}(s(y)) \neq 1 \\
& \quad \text{or} \quad id \notin \{p_1, \ldots, p_m\} \implies \text{snd}(s(x)) \neq 1 \quad \text{or} \quad id \in \{\text{data}\} \implies \text{snd}(s(x)) \neq 0 \\
(\text{tid}(x := E), (s,h)) & \rightarrow \text{abort} \\
\text{tid} = t_{gc} \quad & s(y) = (t_0) \quad & \pi(t) = s_t \quad s(x) = (n', 0) \quad S = \{n \mid \exists x. s(x) = (n,1)\} \quad s' = s(x \mapsto (S,0)) \\
(\text{tid}(x := \text{get_root}(y)), (\pi \cup \{\text{tid} \mapsto s\}, h)) & \rightarrow (\text{tid} \cdot \text{skip}, (\pi \cup \{\text{tid} \mapsto s'\}, h)) \\
\text{tid} \neq t_{gc} \quad \text{or} \quad x \notin \text{dom}(s) \quad \text{or} \quad \text{snd}(s(x)) \neq 0 \quad \text{or} \quad y \notin \text{dom}(s) \quad \text{or} \quad \text{snd}(s(y)) \neq 0 \quad \text{or} \quad \text{fst}(s(y)) \notin \text{dom}(\pi) \\
(\text{tid}(x := \text{get_root}(y)), (\pi \cup \{\text{tid} \mapsto s\}, h)) & \rightarrow \text{abort} \\
\text{tid} \in \{1..N\} \quad & s(x) = (n', 1) \quad & h(n)(\text{color}) = \text{BLUE} \\
(\text{tid}(x := \text{new}(x)), (s,h)) & \rightarrow (\text{skip}, (s(x \mapsto (n,1)), h) \\
\text{tid} \in \{1..N\} \quad & s(x) = (n', 1) \quad & h(n)(\text{color}) = \text{BLUE} \\
(\text{tid}(x := \text{new}(x)), (s,h)) & \rightarrow (\text{skip}, (s(x \mapsto (0, 1)), h)) \\
\text{tid} \notin \{1..N\} \quad \text{or} \quad x \notin \text{dom}(s) \quad \text{or} \quad \text{snd}(s(x)) \neq 1 \\
(\text{tid}(x := \text{new}(x)), (s,h)) & \rightarrow \text{abort} \\
\text{free}(x), (s,h)) & \rightarrow (\text{skip}, (s, h(n \mapsto o(\text{color} \mapsto \text{BLUE})))) \\
s(x) = (n, 1) \quad h(n) = o \\
(\text{push}(x,y), (s,h)) & \rightarrow (\text{skip}, (s\{y \mapsto \text{BLUE} \mapsto A, 0\}, h)) \\
\text{tid} \in \{1..N\} \quad & s(x) = (n', b) \quad & s(y) = (A, 0) \\
(\text{tid}(x := \text{pop}(y)), (s,h)) & \rightarrow (\text{skip}, (s(x \mapsto (n,b), y \mapsto (A,0)), h)) \\
\text{tid} \notin \{1..N\} \quad \text{or} \quad x \notin \text{dom}(s) \quad \text{or} \quad \text{fst}(s(y)) \notin \text{Seq}(\text{Val}) \quad \text{or} \quad \text{snd}(s(y)) \neq 0 \\
(\text{tid}(x := \text{pop}(y)), (s,h)) & \rightarrow \text{abort} \\
\text{Figure 26. Selected Operational Semantics Rules on the Low-Level Machine} \\
\end{align*}
\]
Sweep() { 
  local i: [1..M], c: {BLACK, WHITE, BLUE};
  i := 1; 
  while (i ≤ M) {
    c := i.color;
    if (c = WHITE) {
      free(i); // append a node to the free list 
    }
    i := i + 1;
  }
}

E.3 The Logic for the GC thread

E.3.1 Assertions
We define the semantics of the assertions in Figures 27 and 28. The logical variable mapping $i$ maps a logical variables to a value and a bit to indicate whether it is a pointer (like the store mapping). We lift the original rely and guarantee conditions over state pairs to $\text{LState} \times \text{LvMap}$ pairs with identity transitions on $\text{LvMap}$.

As shown in Figure 27(d), 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 higher-order partial functions, they can be transformed to an uncurried form by the uncurry operator. We then use $h_1 \uplus h_2$ to denote the union when their domains of uncurry($h_1$) and 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.

E.3.2 Inference Rules
We present the inference rules for reasoning about sequential programs in Figure 29 following separation logic.

The concurrency rules are presented in Figure 30 where stability is defined in a traditional way.

**Definition 35 (Stability).** Sta($p$, $a$) holds iff, for all $\sigma$, $i$, $\sigma'$ and $i'$, if $p(\sigma, i)$ and $a(\sigma, i, (\sigma', i'))$, then $p(\sigma', i')$.

E.3.3 Soundness
The semantics for the judgment $\{p\}C\{q\}$ is standard, except we also require that no external events are generated.

**Definition 36 (Seq-Semantics).** $\models \{p\}C\{q\}$ iff, for any $\sigma$ and $i$ such that $p(\sigma, i)$, the following are true:

1. $\neg((C, \sigma) \rightarrow \ast \text{abort})$;
2. $\exists C', \sigma', e. ((C, \sigma) \rightarrow \ast (C', \sigma'))$;
3. if $(C, \sigma) \rightarrow \ast (\text{skip}, \sigma')$, then $q(\sigma', i)$.

**Lemma 37 (Seq-Soundness).** If $\{p\}C\{q\}$, then $\models \{p\}C\{q\}$.

Lemma 37 is proved by induction over the derivation of the judgment $\{p\}C\{q\}$. The whole proof consists of the soundness proof for each individual rules. Here we only present the proofs for soundness of the GETRT, FREE and FOREACH rules. Others are following previous works on sequent separation logic and omitted here.

**Lemma 38 (GETRT-Sound).** Let $p \triangleq x, y; \bullet x \leftarrow X' \land 1 \leq y \leq N$ and $q \triangleq x, y; \bullet x \leftarrow X \land 1 \leq y \leq N \land \text{root}(y, X)$. If $\{p\}x := \text{get\_root}(y, q)$, then $\models \{p\}x := \text{get\_root}(y, q)$.

**Proof.** By Definition 36, we need to prove that, for all $\sigma$ and $i$ such that $p(\sigma, i)$, we have

(i) $\neg((x := \text{get\_root}(y, \sigma)) \rightarrow \ast \text{abort})$;
(ii) $\exists C', \sigma', e. ((x := \text{get\_root}(y, \sigma)) \rightarrow \ast (C', \sigma'))$;
(iii) if $(x := \text{get\_root}(y, \sigma)) \rightarrow \ast (\text{skip}, \sigma')$, then $q(\sigma', i)$.

Suppose $\sigma = (\tau \uplus \{r \leftarrow s\}, h)$. Since $\sigma \models p$, there exists $t$ and $n'$ such that $s(y) = (t, 0), t \in [1..N], s(x) = (n', 0)$ and $\pi = \pi \cup \{x \leftarrow \text{root}(y, X)\}$. Then $(x := \text{get\_root}(y, \sigma)) \rightarrow (\text{skip}, \sigma')$, where $\sigma' = (\pi \uplus \{r \leftarrow s', t \leftarrow s_1\}, h), S = \{n \mid \exists x, s_x(x) = (n, 1)\}$ and $s' = s(x \leftarrow (S, 0))$. Thus (i) and (ii) are proved. Since aux is an auxiliary variable added only for proof, it is not counted in $S$ actually when the program is executed. Thus $q(\sigma', i)$, i.e., (iii) is proved.

**Lemma 39 (FREE-Sound).** Let $p \triangleq \bullet x \leftarrow x.color \rightarrow \_ \land q \triangleq \bullet x \leftarrow x.color \rightarrow \_$. If $\{p\}\text{free}(x, q)$, then $\models \{p\}\text{free}(x, q)$.

**Proof.** By Definition 36, we need to prove that, for all $\sigma$ and $i$ such that $p(\sigma, i)$, we have

(i) $\neg((\text{free}(x, \sigma) \rightarrow \ast \text{abort})$;
(ii) $\exists C', \sigma', e. ((\text{free}(x, \sigma) \rightarrow \ast (C', \sigma'))$;
(iii) if $(\text{free}(x, \sigma) \rightarrow \ast (\text{skip}, \sigma')$, then $q(\sigma', i)$.

Suppose $\sigma = (\tau \uplus \{t_e \leftarrow s\}, h)$. Since $\sigma \models p$, there exists $t$ and $n$ such that $s(y) = (t, 0), t \in [1..N], s(x) = (n', 0)$ and $\pi = \pi \cup \{x \leftarrow \text{root}(y, X)\}$. Then $(\text{free}(x, \sigma) \rightarrow \ast (\text{skip}, \sigma')$, where $\sigma' = (\pi \uplus \{t_e \leftarrow s', t \leftarrow s_1\}, h), S = \{n \mid \exists x, s_x(x) = (n, 1)\}$ and $s' = s(x \leftarrow (S, 0))$. Thus (i) and (ii) are proved. Also $q(\sigma', i)$ holds, i.e., (iii) is proved.

Suppose the (FOREACH) rule is applied to derive $(p \ast \text{own}(x))\text{FOREACH} x \in y \rightarrow \_ C \land \text{root}(y, X) \land y = \phi$. We want to prove $\models \{p \ast \text{own}(x)\}\text{FOREACH} x \in y \rightarrow \_ C \land \text{own}(x) \land y = \phi$. By inversion of the (FOREACH) rule, we know $p \ast \text{own}(x) \land \text{root}(y, X) \land y = \_ \rightarrow \{p \ast \text{own}(x)\}$.

**Lemma 40 (FOREACH-Sound).** If $p \rightarrow \text{own}(y)$ and $\{p \ast \text{own}(x) \land \text{root}(y, X) \land y = \_ \rightarrow \{p \ast \text{own}(x)\}$, then $\models \{p \ast \text{own}(x)\}\text{FOREACH} x \in y \rightarrow \_ C \land \text{own}(x) \land y = \phi$.

**Proof.** By Definition 36, we need to prove that, for all $n \geq 0$, for all $\sigma$ and $i$ such that $(p \ast \text{own}(x))(\sigma, i)$, we have

(i) $\neg((\text{foreach} x \in y \rightarrow C, \sigma) \rightarrow \ast \text{abort})$;
(ii) $\exists C', \sigma', e. ((\text{foreach} x \in y \rightarrow C, \sigma) \rightarrow \ast (C', \sigma'))$;
(iii) if $(\text{foreach} x \in y \rightarrow C, \sigma) \rightarrow \ast (\text{skip}, \sigma')$, then $p(\sigma, i)$.

Suppose $\sigma = (\pi \uplus \{t_e \leftarrow s\}, h)$. Since $\sigma \models p \ast \text{own}(x)$ and $p \rightarrow \text{own}(y)$, we know $x \in \text{dom}(s)$ and $s(y) = \_ \rightarrow \_ \rightarrow \phi(\sigma', i)$.

Perform induction over $n$.

**Base Case:** When $n = 0$, it is trivial. When $n = 1$, assume there's a type checker ensuring the value of $y$ is a set (or we can extend the assertion language to know this), we can prove (i) and (ii) from the operational semantics of FOREACH. If $(\text{foreach} x \in y \rightarrow C, \sigma) \rightarrow (\text{skip}, \sigma')$, then $\sigma' = \sigma$ and $s(y) = \{\_\}$. Thus $(p \ast \text{own}(x) \land y = \phi(\sigma', i))$.

**Inductive Step:** Assume (i), (ii) and (iii) are true when $n \leq m$. From the operational semantics of FOREACH, we know

$(\text{foreach} x \in y \rightarrow C, \sigma) \rightarrow (\text{skip}, \sigma')$.

By the assumption, (i) and (ii) are true when $n = m + 1$.

If $(\text{foreach} x \in y \rightarrow C, \sigma) \rightarrow (\text{skip}, \sigma')$, then $s(y) = \{n_1, \ldots, n_k\} \rightarrow \_ \rightarrow \phi(\sigma', i))$. 


Then \((p \circ \text{own}(x) \land x \in y) (\sigma_1, i, \text{If})\). If

\[(C; y := y - \{x\}; \text{foreach } x \in y \text{ do } C, \sigma_1) \rightarrow^*\]

(\text{foreach } x \in y \text{ do } C, \sigma_2)

then \((p \circ \text{own}(x))(\sigma_2, i).\) Thus if

(\text{foreach } x \in y \text{ do } C, \sigma_2) \rightarrow^k \text{ (skip, } \sigma')\]

where \(k \leq m\), then \((p \circ \text{own}(x) \land y = \emptyset)(\sigma', i, \text{), (iii) is proved.}

We have defined the semantics of \(R; G \vdash \{p\}C\{q\}\) in Definition 24 and 25 here we only extend it with the logical variable mapping.

**Lemma 41.** If \((C, \sigma, R)\) guarantees \(n+1, G\), there does not exist \(j\) such that \(j < n\) and \((C, \sigma) \vdash^k \text{abort}\).

**Theorem 42 (Soundness).** If \(R; G \vdash \{p\}C\{q\}\), then \(R; G \vdash \{p\}C\{q\}\).

Theorem 42 is proved by induction over the derivation of the judgment \(R; G \vdash \{p\}C\{q\}\). The whole proof consists of the soundness proof for each individual rules. Here we only present the proofs for soundness of the ATOMIC rules. Others are similar to the traditional RG logic and omitted here.

Suppose the ATOMIC rule is applied to derive

\(R; G \vdash \{p\}\text{atomic}(\{q\}G)\).

We want to prove \(R; G \vdash \{p\}\text{atomic}(\{q\}G)\). By inversion of the ATOMIC rule, we know \(p \Rightarrow p', \{p\}C\{q'\}, p' \times q' \Rightarrow G, q' \Rightarrow q\) and \(\text{Sta}(\{p, q\}, R)\).

**Lemma 43 (ATOMIC-Sound).** If \(p \Rightarrow p', \vdash \{p\}C[q'], p' \times q' \Rightarrow G, q' \Rightarrow q\) and \(\text{Sta}(\{p, q\}, R)\), then \(R; G \vdash \{p\}\text{atomic}(\{q\}G)\).

**Proof:** By Definition 25 and 26 we need to prove that, for all \(\sigma\) and \(i\) such that \(p(\sigma, i)\), we have

(i) if \(\text{atomic}(\{C\}, \sigma) \vdash_{\text{R}}^\ast \text{ (skip, } \sigma')\), then \(q(\sigma', i)\);

(ii) \(\forall n. (\text{atomic}(\{C\}, \sigma, R)\text{ guarantees}_n G)\).

Since \(\text{atomic}(\{C\}, \sigma) \vdash_{\text{R}}^\ast \text{ (skip, } \sigma')\), there exist \(\sigma_1\) and \(\sigma_2\) such that \(\text{atomic}(\{C\}, \sigma_1) \rightarrow \text{ (skip, } \sigma_2)\). By the definition of the semantics of sequential rules, we know \(q'(\sigma_2, i)\). Then \(q(\sigma_2, i)\). Since \(\text{Sta}(\{q, R\})\), we know \(q(\sigma', i)\). Thus (i) is proved.

If \(p'(\sigma_1, i)\) and \(\text{atomic}(\{C\}, \sigma_1) \rightarrow \text{ (skip, } \sigma_2)\), then \(q'(\sigma_2, i)\), thus \(\sigma_1, \sigma_2 \in G\) because \(p' \times q' \Rightarrow G\). We can prove (ii) by induction over \(n\).

\[\square\]
\[ \text{obj}(x) \triangleq x \cdot pt_1 \rightarrow \ldots \rightarrow x \cdot pt_m \rightarrow * x \cdot data \rightarrow * x \cdot color \rightarrow * x \cdot dirty \rightarrow _{\text{}} \]

\[ \text{blueobj}(x) \triangleq x \cdot pt_1 \rightarrow \ldots \rightarrow x \cdot pt_m \rightarrow * x \cdot data \rightarrow * x \cdot color \rightarrow \text{BLUE} \rightarrow * x \cdot dirty \rightarrow _{\text{}} \]

\[ \text{newobj}(x) \triangleq x \cdot pt_1 \rightarrow 0 \ldots \rightarrow x \cdot pt_m \rightarrow 0 \ldots \rightarrow * x \cdot data \rightarrow 0 \ldots \rightarrow * x \cdot color \rightarrow \text{BLACK} \rightarrow * x \cdot dirty \rightarrow _{\text{}} \]

\[ \text{black}(x) \triangleq x \cdot color \rightarrow \text{BLACK} \]

\[ \text{white}(x) \triangleq x \cdot color \rightarrow \text{WHITE} \]

\[ \text{dirty}(x) \triangleq x \cdot dirty \rightarrow 1 \]

\[ \text{not_blue}(x) \triangleq \exists c. (x \cdot color \leftarrow c \land c \neq \text{BLUE}) \]

\[ \text{not_white}(x) \triangleq \exists c. (x \cdot color \leftarrow c \land c \neq \text{WHITE}) \]

\[ \text{not_dirty}(x) \triangleq x \cdot dirty \rightarrow 0 \]

\[ \text{instk}(n, A) \triangleq \exists n', A = n' :: A' \land (n = n' \lor \text{instk}(n, A')) \]

\[ \text{stk_black}(A) \triangleq \forall x. \text{instk}(x, A) \equiv \text{black}(x) \]

\[ \text{root}(t, S) \triangleq \lambda(x, i). x = (\pi \cup \{ t \rightarrow s_i \}, h) \land S = \{ n \mid \exists x. n_i(x) = (n, 1) \land x \neq \text{aux} \} \]

\[ \text{edge}(x, y) \triangleq \exists \text{id} \in \{ pt_1, \ldots, pt_m \} \cdot (x.\text{id} \leftarrow y) \]

\[ \text{path}_k(x, y) \triangleq \begin{cases} x = y & \text{if } k = 0 \\ \exists x. \text{edge}(x, z) \land \text{path}_{k-1}(z, y) & \text{if } k > 0 \end{cases} \]

\[ \text{path}(x, y) \triangleq \exists k. \text{path}_k(x, y) \]

\[ \text{reachable}(t, x) \triangleq \exists y, \text{root}(t, S) \land y \in S \land \text{path}(y, x) \land x \neq 0 \]

\[ \text{reachable}(x) \triangleq \exists x \in [1..N]. \text{reachable}(t, x) \]

\[ \text{wfsstate} \triangleq \forall x \in [1..m]. \text{obj}(x) \times \text{true} \land (\forall y. \text{reachable}(x) \rightarrow \neg \text{blue}(x)) \]

\[ \text{white_edge}(x, y) \triangleq \exists \text{id} \in \{ pt_1, \ldots, pt_m \} \cdot (x.\text{id} \leftarrow y \land \text{white}(y)) \]

\[ \text{white_path}_k(x, y) \triangleq \begin{cases} x = y & \text{if } k = 0 \\ \exists y. \text{white_edge}(x, z) \land \text{white_path}_{k-1}(z, y) & \text{if } k > 0 \end{cases} \]

\[ \text{white_path}(x, y) \triangleq \exists k. \text{white_path}_k(x, y) \]

\[ \text{wvp}(x, y) \triangleq \text{white}(x) \land \text{white_path}(x, y) \]

\[ \text{rt_wvp}(t, x) \triangleq \exists y, \text{root}(t, S) \land y \in S \land \text{wvp}(y, x) \]

\[ \text{rt_wvp}(x) \triangleq \exists x \in [1..N]. \text{rt_wvp}(t, x) \]

\[ \text{dt_bwp}(x, y) \triangleq \text{black}(x) \land \text{dirty}(x) \land \text{white_path}(x, y) \]

\[ \text{stk_bwp}(x, y, A) \triangleq \text{black}(x) \land \text{instk}(x, A) \land \text{white_path}(x, y) \]

\[ \text{reach_inv} \triangleq \forall x. \text{reachable}(x) \land \text{white}(x) \implies \text{rt_wvp}(x) \lor \exists x'. \text{dt_bwp}(x', x) \]

\[ \text{reach_stk}(A) \triangleq \forall x. \text{reachable}(x) \land \text{white}(x) \implies \text{rt_wvp}(x) \lor \exists x'. \text{dt_bwp}(x', x) \lor \exists x'. \text{stk_bwp}(x', x, A) \]

\[ \text{reach_tnswstk}(A) \triangleq \forall x. \text{reachable}(x) \land \text{white}(x) \implies \exists x'. \text{dt_bwp}(x', x) \lor \exists x'. \text{stk_bwp}(x', x, A) \lor \exists x'. \text{tnswstk}(x', x, A) \]

\[ \text{popped_bwp}(x, y, S_i) \triangleq \text{black}(x) \land \exists z. z.\text{id} \in S_i \land \{ pt_1, \ldots, pt_m \} \land x.\text{id} \rightarrow z \land \text{wvp}(x, y) \]

\[ \text{reach_tnsw}(A, x, S_i, x_w) \triangleq \exists x_w. \exists x. \text{reachable}(x) \implies \exists z. z.\text{id} \in S_i \land \{ pt_1, \ldots, pt_m \} \land x.\text{id} \rightarrow z \land \text{wvp}(x, y) \]

\[ \text{reach_black} \triangleq \forall x. \text{reachable}(x) \implies \text{black}(x) \]

\[ \text{ptfd_sta}(x, y, A) \triangleq \exists n, x.\text{id} \rightarrow n \land (y = n \lor \text{dirty}(x) \lor n = 0 \lor \exists x'. (t.\text{aux} = x \land t.x' = n \land \text{town}(x'))) \]

\[ \text{newobj_sta}(x) \triangleq \text{obj}(x) \times \text{true} \land \text{black}(x) \land \forall \text{id} \in \{ pt_1, \ldots, pt_m \}. \text{ptfd_sta}(x.\text{id}, 0) \]

\[ \text{rt_not_white} \triangleq \exists y, \text{root}(t, S) \land \forall n \in S. \text{not_white}(n) \]

\[ \text{mark_till}(n) \triangleq \forall t \in [1..N]. \text{rt_not_white}(t) \]

\[ \text{clear_color_till}(n) \triangleq \forall x \in [1..N]. (x.\text{color} \rightarrow \text{BLACK} \implies \text{newobj_sta}(x)) \]

\[ \text{clear_dirty_till}(n) \triangleq \forall x \in [1..N]. \text{not_dirty}(x) \]

\[ \text{clear_till}(n) \triangleq \forall x \in [1..N]. \text{not_white}(x) \]

\[ \text{NOTE: Here we use } \_ \text{ for an unspecified integer } n \text{ that } 0 \leq n \leq M. \text{ Some assertions are already shown in Figure}\left[19\right] \]

**Figure 28.** Useful Assertions for Verifying Boehm et al. GC

### E.4 Proofs of the GC Code

Since each instruction in the GC code is executed atomically, we need to stabilize the pre and post conditions when verifying it (required by the ATOMIC rule). For example, when reading a pointer field of an object to a local variable, the postcondition should be stabilized since mutators might update the field.

\[ \text{ptfd_sta}(i, pt_1, X) \text{ says the } pt_1 \text{ field of } i \text{ was once } X \text{ and if it is not } X \text{ now, it must have been updated by a write barrier. Similarly, when reading the color of an object, the postcondition should take into account the mutators’ possible update of the color field in allocation and the updates of pointer fields after allocation.} \]

\[
\begin{align*}
\mathcal{R}_{gc: G_\text{gc}} \vdash & \quad \exists X, Y. c = X \land \text{i.color} \leftarrow Y \\
 & \quad c := \text{i.color}; \\
 & \quad \exists X, Y. c = X \lor \text{i.color} \leftarrow Y \\
 & \quad (X = Y \lor X = \text{BLUE} \land \text{newobj_sta}(i)) \end{align*}
\]

where newobj_sta(i) says i points to a new object whose color field is BLACK and all the pointer fields were once. Both the predicates ptfd_sta and newobj_sta are defined in Figure[28].

We present the key proof of each module in Figures 31, 32, 33, 34, 35, 36 and 37. Figure[28] defines the assertions used in the proofs.

In Initialize() (shown in Figure[31], the GC scans each object in the heap and colors the black object to white. We use clear_color_till(n) to mean the GC has done color-clearing from 1 to n, but there might still be black objects since the mutators...
\{O_0; O_1 \vdash x = X' \land X = E \land \text{emp}_b\} \xrightarrow{\text{ASSN}} x := E\{O_0; O_1 \vdash x = X \land \text{emp}_b\}

\{O_0; O_1 \vdash x = X \land y.id \rightarrow Y\} x := y.id\{O_0; O_1 \vdash x = Y \land y.id \rightarrow Y\} \xrightarrow{\text{READ}}

\{O_0; O_1 \vdash x.id \rightarrow \_, X = E\} x.id := E\{O_0; O_1 \vdash x.id \rightarrow X\} \xrightarrow{\text{WRITE}}

{x, y; \bullet \vdash x = X' \land 1 \leq y \leq N} x := \text{get}_\text{root}(y)\{x, y; \bullet \vdash x = X \land 1 \leq y \leq N \land \text{root}(y, X)\} \xrightarrow{\text{GETRT}}

\{\bullet; x \vdash x.color \rightarrow \_, \text{free}(x); \bullet; x \vdash x.color \rightarrow \text{BLUE}\} \xrightarrow{\text{FREE}}

\{y, O_0; O_1 \vdash x = X \land y = Y\} \xrightarrow{\text{PUSH}} \text{push}(x, y)\{y, O_0; O_1 \vdash x = X \land y = Y\}

\{y, O_0; O_1 \vdash x = X \land y = X'.\bot\} x := \text{pop}(y)\{y, O_0; O_1 \vdash x = X' \land y = Y\} \xrightarrow{\text{POP}}

\begin{align*}
p \Rightarrow B &= B \quad \{p \land B\} C_1\{q\} \quad \{p \land \neg B\} C_2\{q\} \quad (IF) \quad p \Rightarrow B = B \quad \{p \land B\} C\{p\} \quad \{p\} \text{while} \quad (B)\{C\}{p \land \neg B} \quad (WHILE) \\
p \Rightarrow \text{own}_{\text{up}}(y) &= \{p \land \text{own}(x) \land x \in y\} C; y := y - \{x\}\{p \land \text{own}(x)\} \quad \{p \land \text{own}(x) \land y = \phi\} \quad (FOREACH)
\end{align*}

\begin{align*}
\{p\} \text{skip} \quad (SKIP) \quad \{p\} & C\{q\} \quad \{p\} C\{q\}' \quad \{p \land p\}' C\{q \land q\}' \quad (SEQ) \quad \{p\} C\{q\}' \quad \{q\}' \Rightarrow q \quad (CONSEQ)
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig29.png}
\caption{Inference Rules - Sequential Rules}
\end{figure}

\begin{align*}
p \Rightarrow p' &= p' \quad \{p'\} C\{q'\} \quad p' \times q' \Rightarrow G \quad q' \Rightarrow q \quad \text{Sta}(p, q, R) \quad \text{(ATOMIC)} \\
p \Rightarrow B &= B \quad R; G \vdash \{p \land B\} C_1\{q\} \quad R; G \vdash \{p \land \neg B\} C_2\{q\} \quad (P-IF) \\
p \Rightarrow B &= B \quad R; G \vdash \{p \land B\} C\{p\} \quad (P-WHILE) \\
p \Rightarrow \text{own}_{\text{up}}(y) &= R; G \vdash \{p \land \text{own}(x) \land x \in y\} C; y := y - \{x\}\{p \land \text{own}(x)\} \quad (P-FOREACH)
\end{align*}

\begin{align*}
R; G \vdash \{p\} C\{q\} \quad (P-SEQ)
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig30.png}
\caption{Inference Rules - Concurrency Rules}
\end{figure}

\textbf{Initialize()} {
    \begin{align*}
    &\text{local } i: [1..M], c: \{\text{BLACK, WHITE, BLUE}\}; \\
    &i := 1; \\
    &\text{Loop Invariant: } \{\text{wfstate} \land \text{clear_color}_{\text{tili}}(1 - 1) \land 1 \leq 1 \leq M + 1 \land *\text{own}_{\text{up}}(c)\} \\
    &\text{while } (1 \leq i \leq M) \{ \\
    &\quad \text{i.dirty} := 0; \\
    &\quad c := 1.\text{color}; \\
    &\quad \text{if } (c = \text{BLACK}) \{} \\
    &\quad \quad \text{i.color} := \text{WHITE}; \\
    &\quad \} \\
    &i := i + 1; \\
    \}
\end{align*}

\textbf{wfstate} \text{using Lemma } \text{Lem10}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig31.png}
\caption{Proof Outline of Initialize()}
\end{figure}
\{(wfstat ∧ reach_inv) * (own_{\text{np}}(mstk) ∧ mstk = c)\}

\textbf{Trace()}

\begin{verbatim}
local t: [1..N], rt: Set(Int), i: [0..M];
i := 1;
\end{verbatim}

\textit{Loop Invariant:} \{ (wfstat ∧ reach_inv) * (own_{\text{np}}(mstk) ∧ mstk = c) * (own_{\text{np}}(t) ∧ 1 ≤ t ≤ N + 1) * own_{\text{np}}(rt) * own_{\text{np}}(i) \}

\begin{verbatim}
while (t <= N) {
    rt := get_root(t);
    \{Flw ∧ i ∈ rt\}
    MarkAndPush(i);
    \{Flw ∧ i ∈ rt\}
    t := t + 1;
\}
\end{verbatim}

\textit{Foreach Invariant:} \{Flw\}

\begin{verbatim}
foreset i in rt do {
    \{Flw ∧ i ∈ rt\}
    MarkAndPush(i);
    \{Flw ∧ i ∈ rt\}
}
\end{verbatim}

\{ (wfstat ∧ reach_inv) * (own_{\text{np}}(mstk) ∧ mstk = c) \}

where \textit{Flw} \triangleq \exists X. (wfstat ∧ reach_stk(X) ∧ stk_black(X)) * (own_{\text{np}}(mstk) ∧ mstk = X) * (own_{\text{np}}(t) ∧ 1 ≤ t ≤ N + 1) * own_{\text{np}}(rt) * own_{\text{np}}(i)

---

\textbf{Figure 32.} Proof Outline of Trace()
\{(wfstate ∧ reach inv) ∧ (ownup(mstk) ∧ mstk = ε)\} ScanRoot() {
  local t: [1..N], rt: Set(Int), i: [0..M];
  t := 1;
  Loop Invariant:
  \{∃X. (wfstate ∧ reach stk(X) ∧ stk_black(X) ∧ mark rt_till(t − 1) ∧ 1 ≤ t ≤ N + 1) * (ownup(mstk) ∧ mstk = X) * ownp(i) * ownup(rt)\}
  while (t <= N) {
    rt := get_root(t);
    Foreach Invariant:
    \{∃X,Y. (wfstate ∧ reach stk(X) ∧ stk_black(X) ∧ mark rt_till(t − 1) ∧ 1 ≤ t ≤ N ∧ root(t,Y) ∧ ∀n ∈ (Y − rt). not_white(n) ∧ rt ⊆ Y) \}
    foreach i in rt do {
      MarkAndPush(i);
    }
    t := t + 1;
  }
  \{∃X. (wfstate ∧ reach rtnw stk(X) ∧ stk_black(X)) * (ownup(mstk) ∧ mstk = X)\}
}

\{∃X. (wfstate ∧ reach rtnw stk(X) ∧ stk_black(X)) * (ownup(mstk) ∧ mstk = X)\} CleanCard() {
  local i: [1..M], c: {BLACK, WHITE, BLUE}, d: {1, 0};
  i := 1;
  Loop Invariant:
  \{∃X. (wfstate ∧ reach rtnw stk(X) ∧ stk_black(X) ∧ clear dirty_till(i − 1) ∧ 1 ≤ i ≤ M + 1) * (ownup(mstk) ∧ mstk = X) * ownup(c) * ownup(d)\}
  while (i <= M) {
    c := i.color;
    d := i.dirty;
    if (d = 1) {
      i.dirty := 0;
      if (c = BLACK) {
        push(i, mstk);
      }
    }
    i := i + 1;
  }
  \{∃X. (wfstate ∧ reach rtnw stk(X) ∧ stk_black(X) ∧ clear dirty_till(M)) * (ownup(mstk) ∧ mstk = X) * ownup(c) * ownup(d)\}
  TraceStack();
  \{(wfstate ∧ reach black) * (ownup(mstk) ∧ mstk = φ)\}
}

\{wfstate ∧ reach black\} Sweep() {
  local i: [1..M], c: {BLACK, WHITE, BLUE};
  i := 1;
  Loop Invariant: \{(wfstate ∧ reach black ∧ reclaim_till(i − 1) ∧ 1 ≤ i ≤ M + 1) * ownup(c)\}
  while (i <= M) {
    c := i.color;
    if (c = WHITE) {
      free(i);
    }
    i := i + 1;
  }
  \{wfstate ∧ reach black ∧ reclaim_till(M)\}
}

Figure 35. Proof Outline of ScanRoot() in an Atomic Block

Figure 36. Verification of CleanCard() in an Atomic Block

Figure 37. Proof Outline of Sweep()}
could allocate an black object after the GC’s clearing. When all the objects’ color has been “cleared”, we know \( \text{reach}_{\text{inv}} \) holds.

**Lemma 44.** \( \text{wfs} \land \text{clear.color}_{\text{top}}(M) \implies \text{reach}_{\text{inv}} \).

When the object \( i \) is white, \( \text{MarkAndPush}(i) \) colors it black and pushes it onto the mark stack. Since this module will be called several times, we use unified pre and post conditions.

\[
\begin{align*}
R_{\text{gc}}; G_{\text{gc}} \vdash \left\{ \begin{array}{l}
\exists X. W_{\text{state}} \land \text{reach}_{\text{tomk}}(X, x_p, i_d, i) \\
\land \text{stk}_{\text{black}}(X) \land (i = 0 \lor \text{obj}(i))
\end{array} \right\} \\
\text{MarkAndPush}(i) \\
\exists X. W_{\text{state}} \land \text{reach}_{\text{tomk}}(X, x_p, i_d, 0) \\
\land \text{stk}_{\text{black}}(X) \land (i = 0 \lor \text{not.white}(i))
\end{align*}
\]

(\( E.1 \))

Here \( \text{reach}_{\text{tomk}}(A, x_{p_i}, i_d, x_w) \) means, any reachable white object \( x \) must satisfy one of the following conditions:

- \( \text{rt}_w(x) \): \( x \) is reachable from a white root by a white path (i.e., all the objects in the path are white);
- \( \exists x', \text{dt}_{\text{bwp}}(x', x) \): \( x \) is reachable from a dirty black object by a white path;
- \( \exists x', \text{stk}_{\text{bwp}}(x', x, A) \): \( x \) is reachable from a black object by a white path and that object is on the stack \( A \);
- \( \text{pushed}_{\text{bwp}}(x_p, x, i_d) \): \( x \) is reachable from the black object \( x_p \) by a white path, but the first edge in the path (i.e., the edge starts from \( x_p \)) must be a field in \( i_d \).
- \( \text{wwp}(x_{w_i}, x) \): \( x \) is reachable from \( x_{w_i} \) by a white path and \( x_{w_i} \) is white as well.

We can find that the first two cases are the same as in \( \text{reach}_{\text{inv}} \). The third case will be useful during tracing when some objects have been colored black and pushed onto the stack. We define \( \text{reach}_{\text{stk}} \) to express that only these three cases are satisfied for reachable white objects. We will discuss the uses of the last two cases later.

\( \text{Trace()} \) in the concurrent mark-phase (Figure 32) first gets every mutator thread’s root set, marks and pushes every root object, and then calls the module \( \text{TraceStack()} \) to perform the depth-first traversal. We need the following two lemmas to relate the unified pre/post conditions of \( \text{MarkAndPush}(i) \) and the actual pre/post conditions when calling the module.

**Lemma 45.** \( \text{reach}_{\text{stk}}(X) \implies \text{reach}_{\text{tomk}}(X, 0, \phi, i) \).

**Lemma 46.** \( \text{reach}_{\text{tomk}}(X, 0, \phi, 0) \implies \text{reach}_{\text{stk}}(X) \).

Then by the \text{CONSEQ} rule, we can reuse the proof of \( \text{MarkAndPush}(i) \).

In \( \text{TraceStack()} \) (Figure 33), the GC pops every object in the mark stack and marks its children if needed, until the stack becomes empty. It seems subtle why \( \text{reach}_{\text{stk}}(X) \) holds as a loop invariant, at each time before popping an object. Suppose the reachable white object \( x \) is only traced from \( i \) by a white path which is the top object on the mark stack. Then the GC does the following things in order:

1. Pop \( i \). Then \( x \) is reachable from the black object \( i \) which is not on the stack now.
2. Read the \( p_t \) field of \( i \) to a local variable \( j \). As we explained before, \( i.p_t \) might not equal \( j \) since mutators could update this field. We only know that \( \text{ptfd}_{\text{sta}}(i, p_t, j) \) holds. Then is \( x \) still reachable from \( i \)? Not necessarily. Actually \( x \) is probably only reachable from \( j \) while \( j \) might not be a child of \( i \). If \( x \) is reachable from the current \( i.p_t \) but not \( j \), then \( i \) has been updated by a write barrier indicating that \( x \) might be reachable from the dirty black object \( i \). One may argue that it’s even possible that \( x \) is not reachable from \( i \) nor \( j \), but reachable from some other object. If so, then mutators must have used the write barrier to update some object but that \( x \) is reachable by another path without going though \( i \) nor \( j \). In all the cases, we can get \( \text{reach}_{\text{tomk}}(\text{ark}_{\text{stk}}, i, \{p_t, \ldots, p_{m_i}\}, j) \) holds. Formally, the following lemma holds:

**Lemma 47.**

\[
\begin{align*}
(a) & \, \text{reach}_{\text{stk}}(i \colon X) \implies \text{reach}_{\text{tomk}}(X, i, \{p_t, \ldots, p_{m_i}\}, 0); \\
(b) & \, \text{reach}_{\text{tomk}}(X, i, i_d, 0) \implies \text{reach}_{\text{tomk}}(X, i, i_d, j); \\
(c) & \, \text{reach}_{\text{tomk}}(X, i, i_d, j) \land \text{ptfd}_{\text{sta}}(i, i_d, j) \land i_d \in i_d \implies \text{reach}_{\text{tomk}}(X, i, i_d, j). \\
\end{align*}
\]

3. \( \text{MarkAndPush}(j) \). We can reuse the proof of this module again.

4. Mark and push other children. The proof is similar to the above two steps, so we omit the discussions. Finally, \( \text{reach}_{\text{stk}}(X) \) holds because no reachable white object need to rely on the reachability from \( i \) (it could be reachable from a child of \( i \) which is on the stack now).

In the concurrent pre-cleaning phase \( \text{CleanCard()} \) (Figure 34), dirty objects are pushed onto the mark stack and then \( \text{TraceStack()} \) is called again. We reuse the proof of \( \text{TraceStack()} \) via the frame rule.

The stop-the-world phase is implemented by an atomic block. Mutators can be suspended without requiring safe points. The GC first marks and pushes the roots of each thread onto the mark stack in \( \text{ScanRoot()} \) (Figure 35). The atomic \( \text{MarkAndPush}(i) \) is proved similarly to the concurrent one (\( E.1 \)) with the same pre/post conditions. Then the GC performs the atomic \( \text{CleanCard()} \) (Figure 36). We do not present the proof for the atomic \( \text{TraceStack()} \) since it is similar to the proof of the concurrent one.

Finally, the concurrent \( \text{Sweep()} \) is verified in Figure 37.

**E.5 Correctness of the Write Barrier**

The relation \( \zeta(t) \) defined in Figure 17 can be preserved under the environment:

**Lemma 48.** For all \( \sigma, \Sigma, \sigma', \Sigma' \), if \( (\sigma, \Sigma) \in \zeta(t), (\sigma', \Sigma') \in R(t), (\Sigma, \Sigma') \in R(t) \) and \( (\sigma', \Sigma') \in \alpha \), then \( (\sigma, \Sigma') \in \zeta(t) \).

**Proof.** By co-induction.

Let \( S = \{(t, \text{skip}, \sigma, \Sigma), \Sigma, \Sigma' \} \mid (\sigma, \Sigma) \in \zeta(t) \}. \) We prove \( S \subseteq F(S) \) where \( F \) is defined by the simulation.

Then it’s easy to prove RGSim for \( \text{skip} \).

**Lemma 49.** For all \( \sigma, \Sigma \), if \( (\sigma, \Sigma) \in \zeta(t), (t, \text{skip}, \sigma, \Sigma, \text{Id}) \prec_{\alpha, \zeta(t)} (t, \text{skip}, \sigma, \Sigma, \text{Id}) \).

**Proof.** By co-induction.

Let \( S = \{(t, \text{skip}, \sigma, \Sigma), \Sigma, \Sigma' \} \mid (\sigma, \Sigma) \in \zeta(t) \}. \) We prove \( S \subseteq F(S) \) where \( F \) is defined by the simulation.

We use some denotations as follows:

\[
\begin{align*}
\text{set}_{\text{dirty}}(x) & \triangleq \text{atomic}(x.\text{dirty}=1; \text{aux}=0) \\
G_t & \triangleq G_{\text{sync,gt}} \cup G_{\text{set}_{\text{dirty}}} \\
G_t & \triangleq G_{\text{sync,gt}} \\
\end{align*}
\]

**Lemma 50.** For all \( \sigma, \Sigma, \) for all \( i \in \{p_t, \ldots, p_{m_i}\}, \) for all \( E \) and \( E' \) such that \( T(E) = E' \),

1. if \( (\sigma, \Sigma) \in \zeta(t) \), then

\[
\begin{align*}
(t.\text{update}(x.\text{id}, E), \sigma, \Sigma, \text{Id}, G_t) & \prec_{\alpha, \zeta(t)} (t.\text{update}(x.\text{id}, E), \Sigma, \Sigma, \text{Id}, G_t) \\
\end{align*}
\]
2. If \((\sigma, \Sigma) \in \alpha\) and \(\exists n. \sigma.s\sigma(t)(\text{aux}) = \sigma.s\sigma(t)(x) = (n, 1)\),
then
\[ (t.set\_dirty(x), \sigma, \mathcal{R}(t), G_\ell) \preceq_{s, \zeta(t)} (t.sk\_\pi, \Sigma, \mathcal{R}(t), G_t) \]

**Proof:** For each case, by co-induction.

**Case:** The environments are executed. Similar to the proof of Lemma 43

**Case:** The low-level code goes one step (let \(\sigma = (\pi, h), s = \pi(t)\)
\(\Sigma = (\Pi, H)\) and \(S = \Pi(t)\)):

1. If \((t.update(x, t, E), (\pi, h)) \rightarrow (t.set\_dirty(x), (\pi', h'))\),
then \(s(x) = (n, 1), h(n) = o, [E]_{k, 1} = n', \pi' = \pi(t \cdot s(\text{aux} \leadsto (n, 1)))\) and \(h' = h(n \cdot o(\text{id} \leadsto n'))\).
Since \([E]_{k, 1} = n'\), we know \(n' = 0\) or \(\exists x.s(x) = (n', 1)\).
Thus we have \(((\pi, h), (\pi', h')) \in G_\ell\).
Since \((\sigma, \Sigma) \in \alpha\), we know \(\text{wfs}(\sigma)\), thus \(h(n)(\text{color}) \neq \text{BLUE}\). Moreover, \(S(x) = l, \text{Loc2Int}(l) = n, \text{H}(l) = O, [E]_{k, 1} = l'\) (where \(\text{Loc2Int}(l') = n', \text{O}(\text{id}) = l'',\) and \(l'' = \text{nfl}\) or \(\exists x.s(x) = l'\).
Thus \((t.(x, \text{id} := E), (\Pi, H)) \Rightarrow (t.set\_dirty, (\Pi', H'))\) where
\(\Pi' = \Pi \text{ and } H' = H\{l \cdot o(\text{id} \leadsto l')\}\).
We have \(((\pi', h'), (\Pi', H')) \in \alpha, \pi(t)(\text{aux}) = \pi'(t)(x) = (n, 1)\), which are the premises of the second case.

2. If \((\sigma, \Sigma) \in \alpha\) and \(\exists n. \sigma.s\sigma(t)(\text{aux}) = \sigma.s\sigma(t)(x) = (n, 1),\)
then \(n \in \text{dom}(h)\) and \(h(n)(\text{color}) \neq \text{BLUE}\),
thus \((t.set\_dirty(x), (\pi, h)) \rightarrow (t.sk\_\pi, (\pi', h'))\)
where \(\pi' = \pi(t \cdot s(\text{aux} \leadsto 0))\) and \(h' = h(n \cdot o(\text{dirty} \leadsto 1))\).
We can use \(((\pi', h'), \Sigma) \in \zeta(t)\), which is the premise of Lemma 43.

**Case:** The low-level code aborts.
If \((t.update(x, t, E), (\pi, h)) \rightarrow \text{abort}\), then \(x \notin \text{dom}(s)\), or
\(\text{fst}(s(x)) \notin \text{dom}(h)\), or \(\text{snd}(s(x)) \neq 1\), or \([E]_{k, 1} = 1\).
Since \((\sigma, \Sigma) \in \alpha\), we have \(x \notin \text{dom}(S)\), or \(S(x) \notin \text{dom}(H)\), or
\(\exists n. \exists E_k = l\).
Thus \((t.(x, \text{id} := E), (\Pi, H)) \rightarrow \text{abort}\).
The premises of the second case ensure that \((t.set\_dirty(x), \sigma)\)
will not abort.

Finally, we can conclude the correctness of the write barrier:
\[
(t.update(x, t, E), \mathcal{R}(t), G_\ell^{\text{write}} \cup G_\ell^{\text{dirty}}) \preceq_{s, \zeta(t), \zeta'(t)} (t.(x, \text{id} := E), \mathcal{R}(t), G_\ell^{\text{write}})
\]
where \(id \in \{t_1, \ldots, t_m\}\) and \(T(E) = E\).