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 $\mathbf{T}$, 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 [15]. 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 **atomic**$\{\mathbb{C}\}$, so that programmers can assume the atomicity of the execution of $\mathbb{C}$. 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 $\mathbf{T}$, we follow Leroy's approach [19] and define a refinement relation $\sqsubseteq$ 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:

$$\mathsf{Correct}(\mathbf{T}) \triangleq \forall C, \mathbb{C}.\, C = \mathbf{T}(\mathbb{C}) \implies C \sqsubseteq \mathbb{C}\,. \qquad (1.1)$$

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

- Since the target $\mathbf{T}(\mathbb{C})$ 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 $\mathbf{T}$ is syntax-directed (and it is usually the case for parallel compositions, *i.e.*, $\mathbf{T}(\mathbb{C} \parallel \mathbb{C}') = \mathbf{T}(\mathbb{C}) \parallel \mathbf{T}(\mathbb{C}')$), a *compositional* refinement is of particular importance for modular verification of $\mathbf{T}$.

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, 16, 19, 21], but they are usually designed for sequential programs (except [21, 25], which we will discuss in Section 8). 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 compositional *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 [17] with relational proofs based on RGSim.
- We verify the Boehm *et al.* concurrent garbage collection algorithm [7] 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, verify atomicity of concurrent objects in Section 6, and prove the correctness of concurrent GCs in Section 7. Finally we discuss related work and conclude in Section 8.

2. Challenges and Our Approach

The major challenge we face is to have a compositional refinement relation $\sqsubseteq$ between concurrent programs, *i.e.*, we should be able to know $\mathbf{T}(\mathbb{C}_1) \parallel \mathbf{T}(\mathbb{C}_2) \sqsubseteq \mathbb{C}_1 \parallel \mathbb{C}_2$ if we have $\mathbf{T}(\mathbb{C}_1) \sqsubseteq \mathbb{C}_1$ and $\mathbf{T}(\mathbb{C}_2) \sqsubseteq \mathbb{C}_2$.

2.1 Sequential Refinement Loses Parallel Compositionality

Observable behaviors of sequential imperative programs usually refer to their control effects (*e.g.*, termination and exceptions) and final program states. However, refinement relations defined correspondingly cannot be preserved after parallel compositions. It has been a well-known fact in the compiler community that sound optimizations for sequential programs may change the behaviors of multi-threaded programs [5]. The Dekker's algorithm shown in Figure 1(a) has been widely used to demonstrate the problem. Reordering 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.

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

(a) Dekker's Mutual Exclusion Algorithm

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

(b) Different Granularities of Atomic Operations

Figure 1. Equivalence Lost after Parallel Composition

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++ [6]) 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**$\{\mathbb{C}\}$ 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.

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 reordering 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 $(C, \mathcal{R}, \mathcal{G}) \preceq (\mathbb{C}, \mathbb{R}, \mathbb{G})$ talks about not only the target C and the source $\mathbb{C}$, but also the interference $\mathcal{R}$ and $\mathcal{G}$ between C and its target-level environment, and $\mathbb{R}$ and $\mathbb{G}$ between $\mathbb{C}$ and its environment at the source level. Informally, $(C, \mathcal{R}, \mathcal{G}) \preceq (\mathbb{C}, \mathbb{R}, \mathbb{G})$ says the executions of C under the environment $\mathcal{R}$ do not exhibit more observable behaviors than the executions of $\mathbb{C}$ under the environment $\mathbb{R}$, and the state transitions of C and $\mathbb{C}$ satisfy $\mathcal{G}$ and $\mathbb{G}$

$$(\textit{Events})\quad e \ ::= \ \ldots \qquad\qquad (\textit{Labels})\quad o \ ::= \ e \mid \tau$$

(a) Events and Transition Labels

$$
\begin{array}{llcl}
(\textit{LState}) & \sigma & ::= & \ldots \\
(\textit{LExpr}) & E & \in & \textit{LState} \rightarrow \textit{Int}_{\perp} \\
(\textit{LBExp}) & B & \in & \textit{LState} \rightarrow \{\textbf{true}, \textbf{false}\}_{\perp} \\
(\textit{LInstr}) & c & \in & \textit{LState} \rightharpoonup \mathcal{P}((\textit{Labels} \times \textit{LState}) \cup \{\textbf{abort}\}) \\
(\textit{LStmt}) & C & ::= & \textbf{skip} \mid c \mid C_1; C_2 \mid \textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2 \\
 & & \mid & \textbf{while}\ (B)\ C \mid C_1 \,\|\, C_2 \\
(\textit{LStep}) & \longrightarrow_L & \in & \mathcal{P}((\textit{LStmt}/\{\textbf{skip}\} \times \textit{LState}) \times \textit{Labels} \\
 & & & \quad \times ((\textit{LStmt} \times \textit{LState}) \cup \{\textbf{abort}\}))
\end{array}
$$

(b) The Low-Level Language

$$
\begin{array}{llcl}
(\textit{HState}) & \Sigma & ::= & \ldots \\
(\textit{HExpr}) & \mathbb{E} & \in & \textit{HState} \rightarrow \textit{Int}_{\perp} \\
(\textit{HBExp}) & \mathbb{B} & \in & \textit{HState} \rightarrow \{\textbf{true}, \textbf{false}\}_{\perp} \\
(\textit{HInstr}) & \mathrm{c} & \in & \textit{HState} \rightharpoonup \mathcal{P}((\textit{Labels} \times \textit{HState}) \cup \{\textbf{abort}\}) \\
(\textit{HStmt}) & \mathbb{C} & ::= & \textit{\textbf{skip}} \mid \mathrm{c} \mid \mathbb{C}_1;;\mathbb{C}_2 \mid \textit{\textbf{if}}\ \mathbb{B}\ \textit{\textbf{then}}\ \mathbb{C}_1\ \textit{\textbf{else}}\ \mathbb{C}_2 \\
 & & \mid & \textit{\textbf{while}}\ \mathbb{B}\ \textit{\textbf{do}}\ \mathbb{C} \mid \mathbb{C}_1 \,|||\, \mathbb{C}_2 \\
(\textit{HStep}) & \longrightarrow_L & \in & \mathcal{P}((\textit{HStmt}/\{\textit{\textbf{skip}}\} \times \textit{HState}) \times \textit{Labels} \\
 & & & \quad \times ((\textit{HStmt} \times \textit{HState}) \cup \{\textbf{abort}\}))
\end{array}
$$

(c) The High-Level Language

Figure 2. Generic Languages at Target and Source Levels

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

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

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

Also different from fully abstract semantics for threads, which assumes arbitrary behaviors of environments, RGSim allows us to instantiate the interference $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$ and $\mathbb{G}$ differently for different assumptions about environments, therefore it can 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 $\mathbb{R}$ and $\mathbb{G}$. 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 τ, 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 label and state pairs, describing the state transitions and the events it generates. We use $\mathcal{P}(\_)$ to denote the power set. Unsafe executions lead to **abort**. Note that the semantics of an instruction could be non-deterministic. Moreover, it might be undefined on some states, making it possible to model blocking operations such as acquiring a lock.

Statements are either primitive instructions or compositions of them. **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 $\_ \overset{\_}{\longrightarrow}_L \_$, 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 **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. $\mathbb{C}_1;;\mathbb{C}_2$ and $\mathbb{C}_1|||\mathbb{C}_2$ are sequential and parallel compositions of $\mathbb{C}_1$ and $\mathbb{C}_2$ respectively. Note that we choose to use the same set of compound statements in the two languages for simplicity only. This is not required by our simulation relation, although the analogous program constructs of the two languages (*e.g.*, parallel compositions $C_1 \parallel C_2$ and $\mathbb{C}_1|||\mathbb{C}_2$) make it convenient for us to discuss the compositionality later.

Figure 3 shows part of the definition of $\_ \overset{\_}{\longrightarrow}_H \_$, which gives the high-level operational semantics of statements. We often omit the subscript H (or L) in $\_ \overset{\_}{\longrightarrow}_H \_$ (or $\_ \overset{\_}{\longrightarrow}_L \_$) and the label on top of the arrow when it is τ. The semantics is mostly standard. Note that when a primitive instruction c is blocked at state Σ (*i.e.*, $\Sigma \notin dom(\mathrm{c})$), we let the program configuration reduce to itself. For example, the instruction `lock(l)` would be blocked when `l` is not 0, making it be repeated until `l` becomes 0; whereas `unlock(l)` simply sets `l` to 0 at any time and would never be blocked. Primitive instructions in the high-level and low-level languages are *atomic* in the interleaving semantics. Below we use $\_ \longrightarrow^* \_$ for zero or multiple-step transitions with no events generated, and $\_ \overset{e}{\longrightarrow}{}^* \_$ for multiple-step transitions with *only one* event e generated.

3.2 The Event Trace Refinement

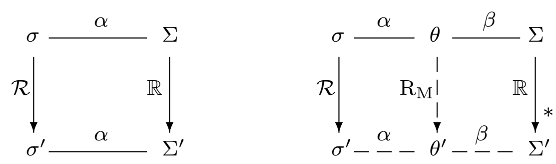
Now we can formally define the refinement relation $\sqsubseteq$ that relates the set of externally observable event traces generated by the target and the source programs. A trace is a sequence of events e, and may end with a termination marker **done** or a fault marker **abort**.

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

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

- $\mathit{ETrSet}_0(C,\sigma) \triangleq \{\epsilon\}$;
- $\mathit{ETrSet}_{n+1}(C,\sigma) \triangleq$
$$\begin{aligned}\{\mathcal{E} \mid\ & (C,\sigma) \longrightarrow (C',\sigma') \wedge \mathcal{E} \in \mathit{ETrSet}_n(C',\sigma') \\ \vee\ & (C,\sigma) \overset{e}{\longrightarrow} (C',\sigma') \wedge \mathcal{E}' \in \mathit{ETrSet}_n(C',\sigma') \wedge \mathcal{E} = e::\mathcal{E}' \\ \vee\ & (C,\sigma) \longrightarrow \mathbf{abort} \wedge \mathcal{E} = \mathbf{abort} \\ \vee\ & C = \mathbf{skip} \wedge \mathcal{E} = \mathbf{done}\}\,.\end{aligned}$$

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



(a) α-Related Transitions (b) The Side Condition of TRANS

Figure 4. Related Transitions

We overload the notation and use $\mathit{ETrSet}(\mathbb{C},\Sigma)$ for the high-level language. Then we define an event trace refinement as the subset relation between event trace sets, which is similar to Leroy's refinement property [19].

Definition 2 (Event Trace Refinement). We say (C,σ) is an *e-trace refinement* of $(\mathbb{C},\Sigma)$, *i.e.*, $(C,\sigma) \sqsubseteq (\mathbb{C},\Sigma)$, if and only if

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

The refinement is defined for program configurations instead of for code only because the initial states may affect the behaviors of programs. In this case, the transformation $\mathbf{T}$ should translate states as well as code. We overload the notation and use $\mathbf{T}(\Sigma)$ to represent the state transformation, and use $C \sqsubseteq_{\mathbf{T}} \mathbb{C}$ for

$$\forall \sigma, \Sigma.\ \sigma = \mathbf{T}(\Sigma) \Longrightarrow (C,\sigma) \sqsubseteq (\mathbb{C},\Sigma)\,,$$

then $\mathsf{Correct}(\mathbf{T})$ defined in formula (1.1) can be reformulated as

$$\mathsf{Correct}(\mathbf{T}) \triangleq \forall C, \mathbb{C}.\ C = \mathbf{T}(\mathbb{C}) \implies C \sqsubseteq_{\mathbf{T}} \mathbb{C}\,. \qquad (3.1)$$

From the above e-trace refinement definition, we can derive an *e-trace equivalence* relation as follows:

$$(C,\sigma) \approx (\mathbb{C},\Sigma) \triangleq (C,\sigma) \sqsubseteq (\mathbb{C},\Sigma) \wedge (\mathbb{C},\Sigma) \sqsubseteq (C,\sigma)\,,$$

and use $C \approx_{\mathbf{T}} \mathbb{C}$ for $\forall \sigma, \Sigma.\ \sigma = \mathbf{T}(\Sigma) \Longrightarrow (C,\sigma) \approx (\mathbb{C},\Sigma)$.

4. 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 $\mathbf{T}$.

4.1 The Definition

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

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

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

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

Before we formally define RGSim in Definition 4, we first introduce the α-related transitions as follows.

Definition 3 (α-Related Transitions).

$$\begin{aligned}\langle \mathcal{R},\mathbb{R}\rangle_\alpha \triangleq \{((\sigma,\sigma'),(\Sigma,\Sigma')) \mid & (\sigma,\sigma') \in \mathcal{R} \wedge (\Sigma,\Sigma') \in \mathbb{R} \\ & \wedge (\sigma,\Sigma) \in \alpha \wedge (\sigma',\Sigma') \in \alpha\}\,.\end{aligned}$$

$$\frac{(\tau, \Sigma') \in \mathrm{c}\,\Sigma}{(\mathrm{c}, \Sigma) \longrightarrow (\textit{\textbf{skip}}, \Sigma')} \qquad \frac{(e, \Sigma') \in \mathrm{c}\,\Sigma}{(\mathrm{c}, \Sigma) \stackrel{e}{\longrightarrow} (\textit{\textbf{skip}}, \Sigma')} \qquad \frac{\textbf{abort} \in \mathrm{c}\,\Sigma}{(\mathrm{c}, \Sigma) \longrightarrow \textbf{abort}} \qquad \frac{\Sigma \notin dom(\mathrm{c})}{(\mathrm{c}, \Sigma) \longrightarrow (\mathrm{c}, \Sigma)}$$

$$\frac{\mathbb{B}\,\Sigma = \textbf{true}}{(\textit{\textbf{if}}\ \mathbb{B}\ \textit{\textbf{then}}\ \mathbb{C}_1\ \textit{\textbf{else}}\ \mathbb{C}_2, \Sigma) \longrightarrow (\mathbb{C}_1, \Sigma)} \qquad \frac{\mathbb{B}\,\Sigma = \textbf{false}}{(\textit{\textbf{if}}\ \mathbb{B}\ \textit{\textbf{then}}\ \mathbb{C}_1\ \textit{\textbf{else}}\ \mathbb{C}_2, \Sigma) \longrightarrow (\mathbb{C}_1, \Sigma)} \qquad \frac{\mathbb{B}\,\Sigma = \perp}{(\textit{\textbf{if}}\ \mathbb{B}\ \textit{\textbf{then}}\ \mathbb{C}_1\ \textit{\textbf{else}}\ \mathbb{C}_2, \Sigma) \longrightarrow \textbf{abort}}$$

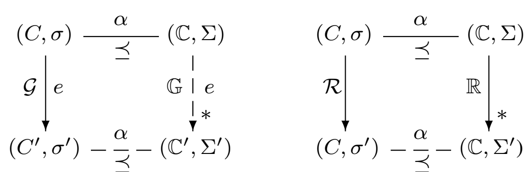
$$\frac{\mathbb{B}\,\Sigma = \textbf{true}}{(\textit{\textbf{while}}\ \mathbb{B}\ \textit{\textbf{do}}\ \mathbb{C}, \Sigma) \longrightarrow (\mathbb{C}; \textit{\textbf{while}}\ \mathbb{B}\ \textit{\textbf{do}}\ \mathbb{C}, \Sigma)} \qquad \frac{\mathbb{B}\,\Sigma = \textbf{false}}{(\textit{\textbf{while}}\ \mathbb{B}\ \textit{\textbf{do}}\ \mathbb{C}, \Sigma) \longrightarrow (\textbf{skip}, \Sigma)} \qquad \frac{\mathbb{B}\,\Sigma = \perp}{(\textit{\textbf{while}}\ \mathbb{B}\ \textit{\textbf{do}}\ \mathbb{C}, \Sigma) \longrightarrow \textbf{abort}}$$

$$\frac{(\mathbb{C}, \Sigma) \longrightarrow (\mathbb{C}', \Sigma')}{(\mathbb{C};;\mathbb{C}'', \Sigma) \longrightarrow (\mathbb{C}';;\mathbb{C}'', \Sigma')} \qquad \frac{(\mathbb{C}, \Sigma) \stackrel{e}{\longrightarrow} (\mathbb{C}', \Sigma')}{(\mathbb{C};;\mathbb{C}'', \Sigma) \stackrel{e}{\longrightarrow} (\mathbb{C}';;\mathbb{C}'', \Sigma')} \qquad \frac{}{(\textit{\textbf{skip}};;\mathbb{C}', \Sigma) \longrightarrow (\mathbb{C}', \Sigma)} \qquad \frac{(\mathbb{C}, \Sigma) \longrightarrow \textbf{abort}}{(\mathbb{C};;\mathbb{C}', \Sigma) \longrightarrow \textbf{abort}}$$

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

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

Figure 3. Operational Semantics of the High-Level Language



(a) Program Steps (b) Environment Steps

Figure 5. Simulation Diagrams of RGSim

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

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

1. if $(C, \sigma) \longrightarrow (C', \sigma')$, then there exist $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma) \longrightarrow^* (\mathbb{C}', \Sigma')$, $((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha$ and $(C', \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}', \Sigma', \mathbb{R}, \mathbb{G})$;
2. if $(C, \sigma) \stackrel{e}{\longrightarrow} (C', \sigma')$, then there exist $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma) \stackrel{e}{\longrightarrow}{}^* (\mathbb{C}', \Sigma')$, $((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha$ and $(C', \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}', \Sigma', \mathbb{R}, \mathbb{G})$;
3. if $C = \textbf{skip}$, then there exists Σ' such that $(\mathbb{C}, \Sigma) \longrightarrow^* (\textit{\textbf{skip}}, \Sigma')$, $((\sigma, \sigma), (\Sigma, \Sigma')) \in \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha$, $(\sigma, \Sigma') \in \gamma$ and $\gamma \subseteq \alpha$;
4. if $(C, \sigma) \longrightarrow \textbf{abort}$, then $(\mathbb{C}, \Sigma) \longrightarrow^* \textbf{abort}$;
5. if $((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha$, then $(C, \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma', \mathbb{R}, \mathbb{G})$.

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

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

- Starting from α-related states, each step of C corresponds to zero or multiple steps of $\mathbb{C}$, and the resulting states are α-related too. If an external event is produced in the step of C, the same event should be produced by $\mathbb{C}$. We show the simulation diagram with events generated by the program steps in Figure 5(a), where solid lines denote hypotheses and dashed lines denote conclusions, following Leroy's notations [19].
- The α relation reflects the abstractions from the low-level machine model to the high-level one, and is preserved by the related transitions at the two levels (so it is an *invariant*). For instance, when verifying a fine-grained implementation of sets, the α relation may relate a concrete representation in memory (*e.g.*, a linked-list) at the low level to the corresponding abstract mathematical set at the high level.
- The corresponding transitions of C and $\mathbb{C}$ need to be in $\langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha$. That is, for each step of C, its state transition should satisfy the guarantee $\mathcal{G}$, and the corresponding transition made by the multiple steps of $\mathbb{C}$ should be in the transitive closure of $\mathbb{G}$. The guarantees are abstractions of the programs' behaviors. As we will show later in the PAR rule in Figure 7, they will serve as the rely conditions of the sibling threads at the time of parallel compositions. Note that we do not need each step of $\mathbb{C}$ to be in $\mathbb{G}$, although we could do so. This is because we only care about the coarse-grained behaviors (with mumbling) of 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 $\mathbb{C}$ terminates as well, and the final states should be related by the postcondition γ. We require $\gamma \subseteq \alpha$, *i.e.*, the final state relation is not weaker than the step invariant.
- C is not safe only if $\mathbb{C}$ is not safe either. This means the 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 $\mathbb{R}$ do, as long as the state transitions are α-related, they should not affect the simulation between C and $\mathbb{C}$, as shown in Figure 5(b). Here a step in $\mathcal{R}$ may correspond to zero or multiple steps of $\mathbb{R}$. Note that different from the program steps, for the environment steps we do not require each step of $\mathcal{R}$ to correspond to zero or multiple steps of $\mathbb{R}$. On the other hand, only requiring that $\mathcal{R}$ be simulated by $\mathbb{R}$ is not sufficient for parallel compositionality, which we will explain later in Section 4.2.

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

$$B \Leftrightarrow \mathbb{B} \triangleq \{(\sigma, \Sigma) \mid B\,\sigma = \mathbb{B}\,\Sigma\} \qquad B \wedge\!\!\!\wedge \mathbb{B} \triangleq \{(\sigma, \Sigma) \mid B\,\sigma \wedge \mathbb{B}\,\Sigma\}$$

$$\mathsf{Intuit}(\alpha) \triangleq \forall \sigma, \Sigma, \sigma', \Sigma'.\ (\sigma, \Sigma) \in \alpha \wedge \sigma \subseteq \sigma' \wedge \Sigma \subseteq \Sigma' \implies (\sigma', \Sigma') \in \alpha$$

$$\eta \,\#\, \alpha \triangleq (\eta \cap \alpha) \subseteq (\eta \uplus \alpha)$$

$$\beta \circ \alpha \triangleq \{(\sigma, \Sigma) \mid \exists \theta.\ (\sigma, \theta) \in \alpha \wedge (\theta, \Sigma) \in \beta\}$$

$$\alpha \uplus \beta \triangleq \{(\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2) \mid (\sigma_1, \Sigma_1) \in \alpha \wedge (\sigma_2, \Sigma_2) \in \beta\}$$

$$\mathsf{Id} \triangleq \{(\sigma, \sigma) \mid \sigma \in \mathit{LState}\} \qquad \mathsf{True} \triangleq \{(\sigma, \sigma') \mid \sigma, \sigma' \in \mathit{LState}\}$$

$$\begin{aligned}
&\mathrm{R_M}\ \mathsf{isMidOf}\ (\alpha, \beta; \mathcal{R}, \mathbb{R}) \triangleq \forall \sigma, \sigma', \Sigma, \Sigma'.\\
&((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{R}, \mathbb{R} \rangle_{\beta \circ \alpha}\\
&\implies \forall \theta.\ (\sigma, \theta) \in \alpha \wedge (\theta, \Sigma) \in \beta\\
&\implies \exists \theta'.\ ((\sigma, \sigma'), (\theta, \theta')) \in \langle \mathcal{R}, \mathrm{R_M} \rangle_\alpha \wedge ((\theta, \theta'), (\Sigma, \Sigma')) \in \langle \mathrm{R_M}, \mathbb{R} \rangle_\beta
\end{aligned}$$

Figure 6. Auxiliary Definitions for RGSim

Then based on the simulation, we hide the states by the precondition ζ and define the RGSim relation between programs only. By the definition we know $\zeta \subseteq \alpha$ if $(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})$, *i.e.*, the precondition needs to be no weaker than the step invariant.

If two programs are simulated by each other, then they are called *R-G-based similar*, as defined below.

$$\begin{aligned}
&(C, \sigma, \mathcal{R}, \mathcal{G}) \simeq_{\alpha;\zeta} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G}) \triangleq\\
&(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G}) \wedge (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G}) \preceq_{\alpha^{-1};\gamma^{-1}} (C, \sigma, \mathcal{R}, \mathcal{G})
\end{aligned}$$

Here α^{-1} and γ^{-1} are the inverse of α and γ respectively. Then,

$$\begin{aligned}
&(C, \mathcal{R}, \mathcal{G}) \simeq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \triangleq\\
&\quad \forall \sigma, \Sigma.\ (\sigma, \Sigma) \in \zeta \implies (C, \sigma, \mathcal{R}, \mathcal{G}) \simeq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})
\end{aligned}$$

RGSim is sound *w.r.t.* the e-trace refinement (Definition 2). That is, $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$ ensures that (C, σ) does not have more observable behaviors than $(\mathbb{C}, \Sigma)$.

Theorem 5 (Soundness). *For all C, $\mathbb{C}$, σ and Σ,*

1. *if there exist $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$, $\mathbb{G}$, α and γ such that $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, then $(C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)$.*
2. *if there exist $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$, $\mathbb{G}$, α and γ such that $(C, \sigma, \mathcal{R}, \mathcal{G}) \simeq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, then $(C, \sigma) \approx (\mathbb{C}, \Sigma)$.*

The soundness theorem can be proved by first strengthening the relies to the identity transitions and weakening the guarantees to the universal relations. Then we prove that the resulting simulation under identity environments implies the e-trace refinement.

For program transformations, since the initial state for the target program is transformed from the initial state for the source, we use $\mathsf{InitRel}_{\mathbf{T}}(\zeta)$ (defined in Figure 6) to say the transformation $\mathbf{T}$ over states ensures the binary precondition ζ.

Corollary 6. *If there exist $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$, $\mathbb{G}$, α, ζ and γ such that $\mathsf{InitRel}_{\mathbf{T}}(\zeta)$ and $(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})$, then $C \sqsubseteq_{\mathbf{T}} \mathbb{C}$.*

4.2 Compositionality Rules

RGSim is compositional *w.r.t.* various program constructs, including parallel compositions. We present the compositionality rules in Figure 7, which gives us a relational proof method for concurrent program transformations.

As in the R-G logic [17], we require that the pre- and post-conditions be *stable* under the interference from the environments. Here we introduce the concept of stability of a relation ζ *w.r.t.* a set of transition pairs $\Lambda \in \mathcal{P}((\mathit{LState} \times \mathit{LState}) \times (\mathit{HState} \times \mathit{HState}))$.

Definition 7 (Stability). $\mathsf{Sta}(\zeta, \Lambda)$ holds iff for all σ, σ', Σ and Σ', if $(\sigma, \Sigma) \in \zeta$ and $((\sigma, \sigma'), (\Sigma, \Sigma')) \in \Lambda$, then $(\sigma', \Sigma') \in \zeta$.

Usually we need $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$, which says whenever ζ holds initially and $\mathcal{R}$ and $\mathbb{R}^*$ perform related actions, the resulting states still satisfy ζ. By unfolding $\langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha$, we could see that α itself is stable *w.r.t.* any α-related transitions, *i.e.*, $\mathsf{Sta}(\alpha, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$. Another simple example is given below, where both environments could increment `x` and the unary stable assertion $\{\mathtt{x} \geq 0\}$ is lifted to the relation ζ:

$$\begin{aligned}
&\zeta \triangleq \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x}) \geq 0\} \qquad \alpha \triangleq \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}\\
&\mathcal{R} \triangleq \{(\sigma, \sigma') \mid \sigma' = \sigma\{\mathtt{x} \rightsquigarrow \sigma(\mathtt{x}) + 1\}\}\\
&\mathbb{R} \triangleq \{(\Sigma, \Sigma') \mid \Sigma' = \Sigma\{\mathtt{x} \rightsquigarrow \Sigma(\mathtt{x}) + 1\}\}
\end{aligned}$$

We can prove $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$. 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 $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$ in the SKIP rule. We also require implicitly that the relies and guarantees are closed over identity transitions, since stuttering steps will not affect observable event traces.

In Figure 7, the rules SKIP, SEQ, IF and WHILE reveal a high degree of similarity to the corresponding inference rules in Hoare logic. In the SEQ rule, γ serves as the postcondition of C_1 and $\mathbb{C}_1$ and the precondition of C_2 and $\mathbb{C}_2$ at the same time. The IF rule requires the boolean conditions of both sides to be evaluated to the same value under the precondition ζ. We give the definitions of the sets $B \Leftrightarrow \mathbb{B}$ and $B \wedge\!\!\!\wedge \mathbb{B}$ in Figure 6. The rule also requires the precondition ζ to imply the step invariant α. In the WHILE rule, the γ relation is viewed as a loop invariant preserved at the loop entry point, which needs to ensure $B \Leftrightarrow \mathbb{B}$.

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

Note that, although RGSim does not require every step of the high-level program to be in its guarantee (see the first two conditions in Definition 4), this relaxation does not affect the parallel compositionality. This is because the target could have less behaviors than the source. To let $\mathbb{C}_1 \,|||\, \mathbb{C}_2$ simulate $C_1 \,\|\, C_2$, we only need a subset of the interleavings of $\mathbb{C}_1$ and $\mathbb{C}_2$ to simulate those of C_1 and C_2. Thus the high-level relies and guarantees need to ensure the existence of those interleavings only. Below we give a simple example to explain this subtle issue. We can prove

$$(\mathtt{x:=x+2}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathtt{x:=x+1;x:=x+1}, \mathbb{R}, \mathbb{G})\,, \qquad (4.1)$$

where the relies and the guarantees say `x` can be increased by 2 and α, ζ and γ relate `x` of the two sides:

$$\begin{aligned}
&\mathcal{R} = \mathcal{G} \triangleq \{(\sigma, \sigma') \mid \sigma' = \sigma \vee \sigma' = \sigma\{\mathtt{x} \rightsquigarrow \sigma(\mathtt{x}) + 2\}\}\,;\\
&\mathbb{R} = \mathbb{G} \triangleq \{(\Sigma, \Sigma') \mid \Sigma' = \Sigma \vee \Sigma' = \Sigma\{\mathtt{x} \rightsquigarrow \Sigma(\mathtt{x}) + 2\}\}\,;\\
&\alpha = \zeta = \gamma \triangleq \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}\,.
\end{aligned}$$

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 $(\mathtt{print(x)}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathtt{print(x)}, \mathbb{R}, \mathbb{G})$. Then by the PAR rule, we get

$$\begin{aligned}
&(\mathtt{x:=x+2} \,\|\, \mathtt{print(x)}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma}\\
&((\mathtt{x:=x+1;x:=x+1}) \,|||\, \mathtt{print(x)}, \mathbb{R}, \mathbb{G})\,,
\end{aligned}$$

which does not violate the natural meaning of refinements. That is, all the possible external events produced by the low-level side can

$$\frac{\zeta \subseteq \alpha}{(\mathbf{skip}, \mathcal{R}, \mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\zeta} (\boldsymbol{skip}, \mathbb{R}, \mathsf{Id})}\ (\text{SKIP}) \qquad \frac{(C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}_1, \mathbb{R}, \mathbb{G}) \quad (C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma\ltimes\eta} (\mathbb{C}_2, \mathbb{R}, \mathbb{G})}{(C_1; C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\eta} (\mathbb{C}_1;; \mathbb{C}_2, \mathbb{R}, \mathbb{G})}\ (\text{SEQ})$$

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

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

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

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

$$\frac{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \quad \zeta' \subseteq \zeta \quad \gamma \subseteq \gamma' \subseteq \alpha \quad \mathcal{R}' \subseteq \mathcal{R} \quad \mathbb{R}' \subseteq \mathbb{R} \quad \mathcal{G} \subseteq \mathcal{G}' \quad \mathbb{G} \subseteq \mathbb{G}'}{(C, \mathcal{R}', \mathcal{G}') \preceq_{\alpha;\zeta'\ltimes\gamma'} (\mathbb{C}, \mathbb{R}', \mathbb{G}')}\ (\text{CONSEQ})$$

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

Figure 7. Compositionality Rules for RGSim

also be produced by the high-level side, although the latter could have more external behaviors due to its finer granularity.

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

$$\begin{array}{l}\text{If } (\sigma, \sigma') \in \mathcal{R}, \text{then there exists } \Sigma' \text{ such that} \\ (\Sigma, \Sigma') \in \mathbb{R}^* \text{ and } (C, \sigma', \mathcal{R}, \mathcal{G}) \preceq'_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \Sigma', \mathbb{R}, \mathbb{G}).\end{array} \quad (4.2)$$

We refer to this modified simulation definition as $\preceq'$. Unfortunately, $\preceq'$ does not have parallel compositionality. As a counter-example, if the invariant α says the left-side x is not greater than the right-side x and the precondition ζ requires x of the two sides are equal, *i.e.*,

$$\alpha \triangleq \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) \leq \Sigma(\mathtt{x})\} \qquad \zeta \triangleq \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\},$$

we could prove the following:

$$\begin{array}{c}(\mathtt{x:=x+1}, \mathsf{Id}, \mathsf{True}) \preceq'_{\alpha;\zeta\ltimes\alpha} (\mathtt{x:=x+2}, \mathsf{Id}, \mathsf{True}); \\ (\mathtt{print(x)}, \mathsf{True}, \mathsf{Id}) \preceq'_{\alpha;\zeta\ltimes\alpha} (\mathtt{print(x)}, \mathsf{True}, \mathsf{Id}).\end{array}$$

Here we use Id and True (defined in Figure 6) for the sets of identity transitions and arbitrary transitions respectively, and overload the notations at the low level to the high level. However, the following refinement does *not* hold after parallel composition:

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

This is because the rely $\mathcal{R}$ (or $\mathbb{R}$) is an abstraction of all the permitted behaviors in the environment of a thread. But a concrete sibling thread that runs in parallel may produce less transitions than $\mathcal{R}$ (or $\mathbb{R}$). To obtain parallel compositionality, we need to ensure that the simulation holds for all concrete sibling threads. With *our* definition $\preceq$, the refinement $(\mathtt{print(x)}, \mathsf{True}, \mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\alpha} (\mathtt{print(x)}, \mathsf{True}, \mathsf{Id})$ is not provable because, after the environments' α-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 STREN-α rule allows us to replace the invariant α by a stronger invariant α'. We need to check that α' is indeed an invariant preserved by the related program steps, *i.e.*, $\mathsf{Sta}(\alpha', \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha)$ holds. Symmetrically, the WEAKEN-α rule requires α to be preserved by environment steps related by the weaker invariant α'. As usual, the pre/post conditions, the relies and the guarantees can be strengthened or weakened by the CONSEQ rule.

The FRAME rule allows us to use local specifications. When verifying the simulation between C and $\mathbb{C}$, we need to only talk about the locally-used resource in α, ζ and γ, and the local relies and guarantees $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$ and $\mathbb{G}$. Then the proof can be reused in contexts where some extra resource η is used, and the accesses of it respect the invariant β and $\mathcal{R}_1$, $\mathcal{G}_1$, $\mathbb{R}_1$ and $\mathbb{G}_1$. We give the auxiliary definitions in Figure 6. The disjoint union $\uplus$ between states is lifted to state pairs. An intuitionistic state relation is monotone *w.r.t.* the extension of states. The disjointness $\eta \# \alpha$ says that any state pair satisfying both η and α can be split into two disjoint state pairs satisfying η and α respectively. For example, let $\eta \triangleq \{(\sigma, \Sigma) \mid \sigma(\mathtt{y}) = \Sigma(\mathtt{y})\}$ and $\alpha \triangleq \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}$, then both η and α are intuitionistic and $\eta \# \alpha$ holds. We also require η to be stable under interference from the programs (*i.e.*, the programs do not change the extra resource) and the extra environments. We use $\eta \# \{\zeta, \gamma, \alpha\}$ as a shorthand for $(\eta \# \zeta) \wedge (\eta \# \gamma) \wedge (\eta \# \alpha)$. Similar representations are used in this rule.

Finally, the transitivity rule TRANS allows us to verify a transformation by using an intermediate level as a bridge. The intermediate environment $\mathrm{R_M}$ should be chosen with caution so that the $(\beta \circ \alpha)$-related transitions can be decomposed into β-related and

α-related transitions, as illustrated in Figure 4(b). Here $\circ$ defines the composition of two relations and $\mathsf{isMidOf}$ defines the side condition over the environments, as shown in Figure 6. We use θ for a middle-level state.

We give all the soundness proofs in Appendix A and B. The proofs [20] 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.

$$(C, \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathsf{Id}, \mathsf{True}) \tag{4.3}$$

The refinement (4.4) assumes arbitrary environments, which makes the interference constraints in the PAR rule trivially true. But this assumption is too strong: usually (4.4) cannot be satisfied in practice.

$$(C, \mathsf{True}, \mathsf{True}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathsf{True}, \mathsf{True}) \tag{4.4}$$

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 $\mathbb{C}_1 \,|||\, \mathbb{C}_2$ is transformed to $C_1 \parallel C_2$, using a lock `l` to synchronize the accesses of the shared variable `x`. We aim to prove $C_1 \parallel C_2 \sqsubseteq_{\mathbf{T}} \mathbb{C}_1 |||\mathbb{C}_2$. That is, although `x:=x+2` is implemented by two steps of incrementing `x` in C_2, the parallel observer C_1 will not print unexpected values. Here we view output events as externally observable behaviors.

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

To facilitate the proof, we introduce an auxiliary shared variable `X` at the low level to record the value of `x` at the time when releasing the lock. It specifies the value of `x` outside every critical section, thus should match the value of the high-level `x` after every corresponding action. Here $\langle C\rangle$ means C is executed atomically.

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 α, which only cares about the value of `x` when the lock is free.

$$\alpha \;\triangleq\; \{(\sigma,\Sigma) \mid \sigma(\mathtt{X}) = \Sigma(\mathtt{x}) \wedge (\sigma(\mathtt{l})=0 \Longrightarrow \sigma(\mathtt{x}) = \sigma(\mathtt{X}))\}\,.$$

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

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

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

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

Following the definition, we can prove $(C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha\ltimes\alpha} (\mathbb{C}_1, \mathbb{R}, \mathbb{G})$ and $(C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha\ltimes\alpha} (\mathbb{C}_2, \mathbb{R}, \mathbb{G})$. By applying the PAR rule and from the soundness of RGSim (Corollary 6), we know $C_1 \parallel C_2 \sqsubseteq_{\mathbf{T}} \mathbb{C}_1 |||\mathbb{C}_2$ holds for any $\mathbf{T}$ that respects α.

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

$$\frac{\mathcal{R}; \mathcal{G} \vdash \{p\}C\{q\}}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\mathsf{Id};[p]\ltimes[q]} (C, \mathcal{R}, \mathcal{G})}$$

For the code which is unchanged after optimizations, we can prove the simulation by the judgment in 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, $[p] \triangleq \{(\sigma,\sigma) \mid p\ \sigma\}$.

Sequential* skip *Law

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

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

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

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

That is, **skip**s could be arbitrarily introduced and eliminated.

Common Branch

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

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

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

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

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

Dead While

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

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

Loop Unrolling

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

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

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

Dead Code Elimination

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

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

Redundancy Introduction

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

As we lifted sequential dead code elimination, we can also lift sequential redundant code introduction to the concurrent setting, so long as the pre- and post-conditions are stable *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_1)

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

$\Leftarrow$

Source Code (C)

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

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 [3]. But in a concurrent setting, safely hoisting the invariant code `t:=x+1` also requires that the environment should not update `x` nor `t`.

$$\mathcal{R} \triangleq \{(\sigma,\sigma') \mid \sigma(\mathtt{x}) = \sigma'(\mathtt{x}) \land \sigma(\mathtt{t}) = \sigma'(\mathtt{t})\}\,.$$

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

$$\alpha \triangleq \{(\sigma_1,\sigma) \mid \sigma_1(\mathtt{i}) = \sigma(\mathtt{i}) \land \sigma_1(\mathtt{n}) = \sigma(\mathtt{n}) \land \sigma_1(\mathtt{x}) = \sigma(\mathtt{x})\}\,.$$

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

$$(C_1,\mathcal{R},\mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (C,\mathcal{R},\mathsf{True})\,. \tag{5.1}$$

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:

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

where γ and η specify the states at the specific program points:

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

After adding **skip**s to C_1 and C to make them the same "shape", we can prove the simulation by the compositionality rules SEQ and WHILE. Finally, we remove all the **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.

```
ADD(e) :
0  atom {
      S := S ∪ {e};
   }
```

```
RMV(e) :
0  atom {
      S := S − {e};
   }
```

(a) An Abstract Set

```
add(e) :
    local x,y,z,u;
 0  <x := Head;>
 1  lock(x);
 2  <z := x.next;>
 3  <u := z.data;>
 4  while (u < e) {
 5    lock(z);
 6    unlock(x);
 7    x := z;
 8    <z := x.next;>
 9    <u := z.data;>
    }
10  if (u != e) {
11    y := new();
12    y.lock := 0;
13    y.data := e;
14    y.next := z;
15    <x.next := y;>
    }
16  unlock(x);
```

```
rmv(e) :
    local x,y,z,v;
 0  <x := Head;>
 1  lock(x);
 2  <y := x.next;>
 3  <v := y.data;>
 4  while (v < e) {
 5    lock(y);
 6    unlock(x);
 7    x := y;
 8    <y := x.next;>
 9    <v := y.data;>
    }
10  if (v = e) {
11    lock(y);
12    <z := y.next;>
13    <x.next := z;>
14    unlock(x);
15    free(y);
    } else {
16    unlock(x);
    }
```

(b) The Lock-Coupling List-Based Set

Figure 8. The Set Object

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 wil 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.

$$
\begin{array}{llll}
(\mathit{HMem}) & M_s, M_l & \in & (\mathit{Loc} \cup \mathit{PVar}) \rightharpoonup \mathit{HVal} \\
(\mathit{HThrds}) & \Pi & \in & \mathit{ThrdID} \to \mathit{HMem} \\
(\mathit{HState}) & \Sigma & \in & \mathit{HThrds} \times \mathit{HMem}
\end{array}
$$

The state consists of shared memory M_s (where the object resides) and a thread pool Π, which is a mapping from thread identifiers ($\mathtt{t} \in \mathit{ThrdID}$) to their memory M_l. The low-level state σ is defined similarly. We use m_s, m_l and π 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 $\textbf{\textit{atom}}\{\mathbb{C}\}_{\mathbb{A}}$ (or $\langle C \rangle_{\mathcal{A}}$ at the low level) to convert the shared memory to local and then execute $\mathbb{C}$ (or C) atomically. Following RGSep [29], an abstract transition $\mathbb{A} \in \mathcal{P}(\mathit{HMem} \times \mathit{HMem})$ (or $\mathcal{A} \in \mathcal{P}(\mathit{LMem} \times \mathit{LMem})$) 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) in Figure 2. We omit the annotations $\mathbb{A}$ and $\mathcal{A}$ 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.

$$
\begin{array}{llll}
(\mathit{HStmts}) & \mathbb{C} & ::= & \textbf{\textit{skip}} \mid \mathrm{c} \mid \textbf{\textit{atom}}\{\mathbb{C}\}_{\mathbb{A}} \mid \mathbb{C}_1; \mathbb{C}_2 \\
 & & & \mid \textbf{if } (\mathbb{B})\, \mathbb{C}_1 \textbf{ else } \mathbb{C}_2 \mid \textbf{while } (\mathbb{B})\{\mathbb{C}\} \\
(\mathit{HProg}) & \mathbb{W} & ::= & \mathtt{t}_1.\mathbb{C}_1 \parallel \ldots \parallel \mathtt{t}_n.\mathbb{C}_n \qquad (\mathit{ThrdID})\ \mathtt{t} \in \mathit{Nat} \\
(\mathit{HMem}) & M_s, M_l & \in & (\mathit{Loc} \cup \mathit{PVar}) \rightharpoonup \mathit{HVal} \\
(\mathit{HThrds}) & \Pi & \in & \mathit{ThrdID} \to \mathit{HMem} \\
(\mathit{HState}) & \Sigma & \in & \mathit{HThrds} \times \mathit{HMem} \\
(\mathit{HAtomG}) & \mathbb{A} & \in & \mathcal{P}(\mathit{HMem} \times \mathit{HMem})
\end{array}
$$

(a) The High-Level Language for Abstract Operations

$$
\begin{array}{llll}
(\mathit{LStmts}) & C & ::= & \textbf{skip} \mid c \mid \langle C \rangle_{\mathcal{A}} \mid C_1; C_2 \\
 & & & \mid \textbf{if } (B)\, C_1 \textbf{ else } C_2 \mid \textbf{while } (B)\{C\} \\
(\mathit{LProg}) & W & ::= & \mathtt{t}_1.C_1 \parallel \ldots \parallel \mathtt{t}_n.C_n \\
(\mathit{LMem}) & m_s, m_l & \in & (\mathit{Loc} \cup \mathit{PVar}) \rightharpoonup \mathit{LVal} \\
(\mathit{LThrds}) & \pi & \in & \mathit{ThrdID} \to \mathit{LMem} \\
(\mathit{LState}) & \sigma & \in & \mathit{LThrds} \times \mathit{LMem} \\
(\mathit{LAtomG}) & \mathcal{A} & \in & \mathcal{P}(\mathit{LMem} \times \mathit{LMem})
\end{array}
$$

(b) The Low-Level Language for Concrete Implementations

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. Traversing 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 α relation, the guarantees and the relies in Figure 11. The predicate $m_s \models \mathsf{list}(x, A)$ 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 α 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 $\mathcal{G}_{\text{lock}}$, $\mathcal{G}_{\text{unlock}}$, $\mathcal{G}_{\text{add}}$, $\mathcal{G}_{\text{rmv}}$ and $\mathcal{G}_{\text{local}}$ respectively, which are all parameterized with a thread identifier t. For example, $\mathcal{G}_{\text{rmv}}(\mathtt{t})$ 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 $\mathcal{R}(\mathtt{t})$. Similarly, the high-level $\mathbb{G}(\mathtt{t})$ and $\mathbb{R}(\mathtt{t})$ 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:

$$
\begin{array}{l}
(\mathtt{t.add(e)}, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha\ltimes\alpha} (\mathtt{t.ADD(e)}, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))\,; \\
(\mathtt{t.rmv(e)}, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha\ltimes\alpha} (\mathtt{t.RMV(e)}, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))\,.
\end{array}
$$

Detailed proofs are given in Appendix D.

$$\frac{(\mathbb{C}, (M_l \uplus M_s, \phi)) \longrightarrow^* (\textbf{skip}, (M_l'', \phi)) \quad M_l'' = M_l' \uplus M_s' \quad (M_s, M_s') \in \mathbb{A}}{(\textbf{\textit{atom}}\{\mathbb{C}\}_{\mathbb{A}}, (M_l, M_s)) \longrightarrow (\textbf{skip}, (M_l', M_s'))}$$

$$\frac{\neg\exists M_l''.\exists M_l'.\exists M_s'.\ (\mathbb{C}, (M_l \uplus M_s, \phi)) \longrightarrow^* (\textbf{skip}, (M_l'', \phi)) \wedge M_l'' = M_l' \uplus M_s' \wedge (M_s, M_s') \in \mathbb{A})}{(\textbf{\textit{atom}}\{\mathbb{C}\}_{\mathbb{A}}, (M_l, M_s)) \longrightarrow \textbf{abort}}$$

$$\frac{(\mathbb{C}, (M_l, M_s)) \longrightarrow (\mathbb{C}', (M_l', M_s'))}{(\mathsf{t}.\mathbb{C}, (\Pi \uplus \{\mathsf{t} \leadsto M_l\}, M_s)) \longrightarrow (\mathsf{t}.\mathbb{C}', (\Pi \uplus \{\mathsf{t} \leadsto M_l'\}, M_s'))} \qquad \frac{(\mathbb{C}, (M_l, M_s)) \longrightarrow \textbf{abort}}{(\mathsf{t}.\mathbb{C}, (\Pi \uplus \{\mathsf{t} \leadsto M_l\}, M_s)) \longrightarrow \textbf{abort}}$$

$$\frac{(\mathsf{t}_i.\mathbb{C}_i, \Sigma) \longrightarrow (\mathsf{t}_i.\mathbb{C}_i', \Sigma')}{(\mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \mathsf{t}_i.\mathbb{C}_i \ldots |||\, \mathsf{t}_n.\mathbb{C}_n, \Sigma) \longrightarrow (\mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \mathsf{t}_i.\mathbb{C}_i' \ldots |||\, \mathsf{t}_n.\mathbb{C}_n, \Sigma')} \qquad \frac{(\mathsf{t}_i.\mathbb{C}_i, \Sigma) \longrightarrow \textbf{abort}}{(\mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \mathsf{t}_i.\mathbb{C}_i \ldots |||\, \mathsf{t}_n.\mathbb{C}_n, \Sigma) \longrightarrow \textbf{abort}}$$

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

$$m_s \models \mathsf{list}(x, A) \triangleq (m_s = \phi \wedge x = \textbf{null} \wedge A = \epsilon) \vee (\exists m_s'.\exists v.\exists y.\exists A'.\ m_s = m_s' \uplus \{x \leadsto (\_, v, y)\} \wedge A = v :: A' \wedge m_s' \models \mathsf{list}(y, A'))$$

$$\mathsf{sorted}(A) \triangleq \begin{cases} \textbf{true} & \text{if } A = \epsilon \vee A = a :: \epsilon \\ (a < b) \wedge \mathsf{sorted}(b :: A') & \text{if } A = a :: b :: A' \end{cases}$$

$$\mathsf{elems}(A) \triangleq \begin{cases} \phi & \text{if } A = \epsilon \\ \{a\} \cup \mathsf{elems}(A') & \text{if } A = a :: A' \end{cases}$$

$$\mathsf{shared\_map}(m_s, M_s) \triangleq \exists m_s'.\exists A.\exists x.\ m_s = m_s' \uplus \{\mathtt{Head} \leadsto x\} \wedge (m_s' \models \mathsf{list}(x, \mathtt{MIN\_VAL} :: A :: \mathtt{MAX\_VAL})) \wedge \mathsf{sorted}(A) \wedge (\mathsf{elems}(A) = M_s(\mathtt{S}))$$

$$\mathsf{local\_map}(m_l, M_l) \triangleq m_l(\mathsf{e}) = M_l(\mathsf{e}) \wedge \exists m_l'.\ m_l = m_l' \uplus \{\mathtt{x} \leadsto \_, \mathtt{y} \leadsto \_, \mathtt{z} \leadsto \_, \mathtt{u} \leadsto \_, \mathtt{v} \leadsto \_\}$$

$$\alpha \triangleq \{((\pi, m_s), (\Pi, M_s)) \mid \mathsf{shared\_map}(m_s, M_s) \wedge \forall \mathsf{t} \in dom(\Pi).\ \mathsf{local\_map}(\pi(\mathsf{t}), \Pi(\mathsf{t}))\}$$

$$\mathcal{G}_{\text{lock}}(\mathsf{t}) \triangleq \{((\pi, m_s), (\pi, m_s')) \mid \exists x, v, y.\ m_s(x) = (0, v, y) \wedge m_s' = m_s\{x \leadsto (\mathsf{t}, v, y)\}\}$$

$$\mathcal{G}_{\text{unlock}}(\mathsf{t}) \triangleq \{((\pi, m_s), (\pi, m_s')) \mid \exists x, v, y.\ m_s(x) = (\mathsf{t}, v, y) \wedge m_s' = m_s\{x \leadsto (0, v, y)\}\}$$

$$\begin{aligned}\mathcal{G}_{\text{add}}(\mathsf{t}) \triangleq\ & \{((\pi \uplus \{\mathsf{t} \leadsto m_l\}, m_s), (\pi \uplus \{\mathsf{t} \leadsto m_l'\}, m_s')) \\ & \mid\ \exists x, y, z, u, v, w.\ m_s(x) = (\mathsf{t}, u, z) \wedge m_s(z) = (\_, w, \_) \\ & \quad \wedge m_s' = m_s\{x \leadsto (\mathsf{t}, u, y)\} \uplus \{y \leadsto (0, v, z)\} \wedge (m_l' \uplus \{y \leadsto (0, v, z)\} = m_l) \wedge u < v < w\}\end{aligned}$$

$$\begin{aligned}\mathcal{G}_{\text{rmv}}(\mathsf{t}) \triangleq\ & \{((\pi \uplus \{\mathsf{t} \leadsto m_l\}, m_s), (\pi \uplus \{\mathsf{t} \leadsto m_l'\}, m_s')) \\ & \mid\ \exists x, y, z, u, v.\ m_s(x) = (\mathsf{t}, u, y) \wedge m_s(y) = (\mathsf{t}, v, z) \\ & \quad \wedge m_s' \uplus \{y \leadsto (\mathsf{t}, v, z)\} = m_s\{x \leadsto (\mathsf{t}, u, z)\} \wedge m_l' = m_l \uplus \{y \leadsto (\mathsf{t}, v, z)\} \wedge v < \mathtt{MAX\_VAL}\}\end{aligned}$$

$$\mathcal{G}_{\text{local}}(\mathsf{t}) \triangleq \{((\pi \uplus \{\mathsf{t} \leadsto m_l\}, m_s), (\pi \uplus \{\mathsf{t} \leadsto m_l'\}, m_s)) \mid \pi \in (\mathit{ThrdID} \to \mathit{LMem}) \wedge m_l, m_l', m_s \in \mathit{LMem}\}$$

$$\mathcal{G}(\mathsf{t}) \triangleq \mathcal{G}_{\text{lock}}(\mathsf{t}) \cup \mathcal{G}_{\text{unlock}}(\mathsf{t}) \cup \mathcal{G}_{\text{add}}(\mathsf{t}) \cup \mathcal{G}_{\text{rmv}}(\mathsf{t}) \cup \mathcal{G}_{\text{local}}(\mathsf{t}) \qquad \mathcal{R}(\mathsf{t}) \triangleq \bigcup_{\mathsf{t}' \neq \mathsf{t}} \mathcal{G}(\mathsf{t}')$$

$$\mathbb{G}_{\text{add}}(\mathsf{t}) \triangleq \{((\Pi \uplus \{\mathsf{t} \leadsto M_l\}, M_s), (\Pi \uplus \{\mathsf{t} \leadsto M_l'\}, M_s')) \mid \exists e.\ M_s' = M_s\{\mathtt{S} \leadsto M_s(\mathtt{S}) \cup \{e\}\}\}$$

$$\mathbb{G}_{\text{rmv}}(\mathsf{t}) \triangleq \{((\Pi \uplus \{\mathsf{t} \leadsto M_l\}, M_s), (\Pi \uplus \{\mathsf{t} \leadsto M_l'\}, M_s')) \mid \exists e.\ M_s' = M_s\{\mathtt{S} \leadsto M_s(\mathtt{S}) - \{e\}\}\}$$

$$\mathbb{G}_{\text{local}}(\mathsf{t}) \triangleq \{((\Pi \uplus \{\mathsf{t} \leadsto M_l\}, M_s), (\Pi \uplus \{\mathsf{t} \leadsto M_l'\}, M_s)) \mid \Pi \in (\mathit{ThrdID} \to \mathit{HMem}) \wedge M_l, M_l', M_s \in \mathit{HMem}\}$$

$$\mathbb{G}(\mathsf{t}) \triangleq \mathbb{G}_{\text{add}}(\mathsf{t}) \cup \mathbb{G}_{\text{rmv}}(\mathsf{t}) \cup \mathbb{G}_{\text{local}}(\mathsf{t}) \qquad \mathbb{R}(\mathsf{t}) \triangleq \bigcup_{\mathsf{t}' \neq \mathsf{t}} \mathbb{G}(\mathsf{t}')$$

Figure 11. Useful Definitions for the Lock-Coupling List

By the compositionality and the soundness of RGSim, we know that the fine-grained operations (under the parallel environment $\mathcal{R}$) are simulated by the corresponding atomic operations (under the high-level environment $\mathbb{R}$), while $\mathcal{R}$ 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 D, 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 [26], 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 $\mathbf{T}$ from a high-level garbage-collected language to a low-level language. A standard atomic memory operation at the source level is transformed into the corresponding barrier code at the target level. In the source level, we assume there is an *abstract GC thread* that magically turns unreachable objects into reusable memory. The abstract collector *AbsGC* is transformed into the concrete GC code C_{gc} running concurrently with the target mutators. That is,

$$\mathbf{T}(\mathsf{t}_{\mathrm{gc}}.AbsGC \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \,|||\, \mathsf{t}_n.\mathbb{C}_n) \triangleq \mathsf{t}_{\mathrm{gc}}.C_{\mathrm{gc}} \,\|\, \mathsf{t}_1.\mathbf{T}(\mathbb{C}_1) \,\|\, \ldots \,\|\, \mathsf{t}_n.\mathbf{T}(\mathbb{C}_n)\,,$$

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

Then we reduce the correctness of the concurrent garbage collecting system to $\mathsf{Correct}(\mathbf{T})$, 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 $\mathsf{Correct}(\mathbf{T})$, 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 $\mathsf{AbsGCStep}$:

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

That is, the abstract GC thread always makes $\mathsf{AbsGCStep}$ to change the high-level state. We can choose different $\mathsf{AbsGCStep}$ for different GCs, but usually $\mathsf{AbsGCStep}$ 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 $\mathcal{R}_{\mathrm{gc}}; \mathcal{G}_{\mathrm{gc}} \vdash \{p_{\mathrm{gc}}\}C_{\mathrm{gc}}\{q_{\mathrm{gc}}\}$ in a standard Rely-Guarantee logic with proper $\mathcal{R}_{\mathrm{gc}}$, $\mathcal{G}_{\mathrm{gc}}$, p_{gc} and q_{gc}, as long as $\mathcal{G}_{\mathrm{gc}}$ is a concrete representation of $\mathsf{AbsGCStep}$. The judgment says given an initial state satisfying the precondition p_{gc}, if the environment's behaviors satisfy $\mathcal{R}_{\mathrm{gc}}$, then each step of C_{gc} satisfies $\mathcal{G}_{\mathrm{gc}}$, and the postcondition q_{gc} holds at the end if C_{gc} terminates. In general, the collector never terminates, thus we can let q_{gc} be **false**. $\mathcal{G}_{\mathrm{gc}}$ 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. $\mathcal{R}_{\mathrm{gc}}$ encodes the possible behaviors of mutators, which can be derived, as we will show below.

Verifying mutators. For the mutator thread, since $\mathbf{T}$ is syntax-directed on $\mathbb{C}$, we can reduce the refinement 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 $\mathbb{G}(\mathsf{t.c})$ and $\mathcal{G}(\mathsf{t}.\mathbf{T}(\mathsf{c}))$ denote the guarantees of the source instruction c and the target code $\mathbf{T}(\mathsf{c})$ respectively. Then we can define the general guarantees for a mutator thread t:

$$\begin{array}{lcl} \mathbb{G}(\mathsf{t}) & \triangleq & \bigcup_{\mathsf{c}} \mathbb{G}(\mathsf{t.c})\,; \\ \mathcal{G}(\mathsf{t}) & \triangleq & \bigcup_{\mathsf{c}} \mathcal{G}(\mathsf{t}.\mathbf{T}(\mathsf{c}))\,. \end{array} \tag{7.1}$$

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

$$\begin{array}{lcl} \mathbb{R}(\mathsf{t}) & \triangleq & \mathsf{AbsGCStep} \cup \bigcup_{\mathsf{t}' \neq \mathsf{t}} \mathbb{G}(\mathsf{t}')\,; \\ \mathcal{R}(\mathsf{t}) & \triangleq & \mathcal{G}_{\mathrm{gc}} \cup \bigcup_{\mathsf{t}' \neq \mathsf{t}} \mathcal{G}(\mathsf{t}')\,. \end{array} \tag{7.2}$$

The $\mathcal{R}_{\mathrm{gc}}$ used to verify the GC code can now be defined below:

$$\mathcal{R}_{\mathrm{gc}} \triangleq \bigcup_{\mathsf{t}} \mathcal{G}(\mathsf{t})\,. \tag{7.3}$$

The refinement proof also needs definitions of binary α, ζ and γ relations. The invariant α relates the low-level and the high-level states and needs to be preserved by each low-level step. In general, a high-level state Σ can be mapped to a low-level state σ by giving a concrete local store for the GC thread, adding additional structures in the heap (to record information for collection), renaming heap cells (for copying GCs), *etc.*. For each mutator thread t, the relations $\zeta(\mathsf{t})$ and $\gamma(\mathsf{t})$ need to hold at the beginning and the end of each basic transformation unit (every high-level primitive instruction in this case) respectively. We let $\gamma(\mathsf{t})$ be the same as $\zeta(\mathsf{t})$ to support sequential compositions. We require $\mathsf{InitRel}_{\mathbf{T}}(\zeta(\mathsf{t}))$ (see Figure 6), *i.e.*, $\zeta(\mathsf{t})$ holds over the initial states. In addition, the target and the source boolean expressions should be evaluated to the same value under related states, as required in the IF and WHILE rules in Figure 7.

$$\mathsf{Good}_{\mathbf{T}}(\zeta(\mathsf{t})) \triangleq \mathsf{InitRel}_{\mathbf{T}}(\zeta(\mathsf{t})) \wedge \forall \mathbb{B}.\, \zeta(\mathsf{t}) \subseteq (\mathbf{T}(\mathbb{B}) \Leftrightarrow \mathbb{B}) \tag{7.4}$$

Theorem 8 (Verifying Concurrent Garbage Collecting Systems). *If there exist $\mathcal{R}_{\mathrm{gc}}$, $\mathcal{G}_{\mathrm{gc}}$, p_{gc}, $\mathcal{R}(\mathsf{t})$, $\mathbb{R}(\mathsf{t})$, $\zeta(\mathsf{t})$ and α such that (7.1), (7.2), (7.3), (7.4) and the following hold:*

1. *(Verification of the GC code)* $\mathcal{R}_{\mathrm{gc}}; \mathcal{G}_{\mathrm{gc}} \vdash \{p_{\mathrm{gc}}\}C_{\mathrm{gc}}\{\mathbf{false}\}$;
2. *(Correctness of $\mathbf{T}$ on mutator instructions)* $\forall \mathsf{c}.\ (\mathsf{t}.\mathbf{T}(\mathsf{c}), \mathcal{R}(\mathsf{t}), \mathcal{G}(\mathsf{t})) \preceq_{\alpha;\zeta(\mathsf{t})\ltimes\zeta(\mathsf{t})} (\mathsf{t.c}, \mathbb{R}(\mathsf{t}), \mathbb{G}(\mathsf{t}))$;
3. *(Side Conditions)* $\mathcal{G}_{\mathrm{gc}} \circ \alpha^{-1} \subseteq \alpha^{-1} \circ (\mathsf{AbsGCStep})^*$; $\forall \sigma, \Sigma.\ \sigma = \mathbf{T}(\Sigma) \implies p_{\mathrm{gc}}\ \sigma$;

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

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

- Define the α and $\zeta(\mathsf{t})$ relations, and prove the correctness of $\mathbf{T}$ on high-level primitive instructions. Since $\mathbf{T}$ preserves the syntax on most instructions, it's often immediate to prove the target instructions are simulated by their sources. But for instructions that are transformed to barriers, we need to verify the barriers that they implement both the source instructions (by RGSim) and the interaction mechanism (shown in their guarantees).
- Find some proper $\mathcal{G}_{\mathrm{gc}}$ and p_{gc}, and verify the GC code by R-G reasoning. We require the GC's guarantee $\mathcal{G}_{\mathrm{gc}}$ should not contain more behaviors than $\mathsf{AbsGCStep}$ (the first side condition), and C_{gc} can start its execution from any state σ transformed from a high-level one (the second side condition).

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 E.

```
  {wfstate}
0 Collection() {
1   local mstk: Seq(Int);
    Loop Invariant: {wfstate ∗ (own_np(mstk) ∧ mstk = ε)}
2   while (true) {
3     Initialize();
      {(wfstate ∧ reach_inv) ∗ (own_np(mstk) ∧ mstk = ε)}
4     Trace();
      {(wfstate ∧ reach_inv) ∗ (own_np(mstk) ∧ mstk = ε)}
5     CleanCard();
      {(wfstate ∧ reach_inv) ∗ (own_np(mstk) ∧ mstk = ε)}
      atomic{
6       ScanRoot();
        {∃X.(wfstate ∧ reach_rtnw_stk(X) ∧ stk_black(X))
          ∗(own_np(mstk) ∧ mstk = X)}
7       CleanCard();
      }
      {(wfstate ∧ reach_black) ∗ (own_np(mstk) ∧ mstk = ε)}
8     Sweep();
    }
  }
  {false}
```

Figure 12. Outline of the GC Code and Proof Sketch

```
update(x.id, E) { // id ∈ {pt1, ..., 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* (`mstk`) 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 C_{gc} is given in Appendix E.2. C_{gc} can use privilege commands to control the mutator threads and manage the heap, *e.g.*, use $x := \textbf{get\_root}(y)$ to read all the pointer variables in the thread y's store and use $\textbf{free}(x)$ 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

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

(a) The Language

(*Loc*)	l	$\in$	$\{L_1, \ldots, L_M, \textbf{nil}\}$
(*HVal*)	V	$\in$	*Int* $\cup$ *Loc*
(*HStore*)	S	$\in$	*PVar* $\rightharpoonup$ *HVal*
(*HObj*)	O	$\in$	*HField* $\rightarrow$ *HVal*
(*HHeap*)	H	$\in$	*Loc* $\rightharpoonup$ *HObj*
(*HThrds*)	Π	$\in$	*MutID* $\rightarrow$ *HStore*
(*HState*)	Σ	$\in$	*HThrds* $\times$ *HHeap*

(b) Program States

Figure 14. The High-level Language and State Model

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

(a) The Language

(*LVal*)	v	$\in$	*Int* $\cup$ *Set(LVal)* $\cup$ *Seq(LVal)*
(*LStore*)	s	$\in$	*PVar* $\rightharpoonup$ *LVal* $\times \{0, 1\}$
(*LObj*)	o	$\in$	*LField* $\rightharpoonup$ *LVal*
(*LHeap*)	h	$\in$	$[1..M] \rightharpoonup$ *LObj*
(*LThrds*)	π	$\in$	$(\textit{MutID} \cup \{\mathrm{t}_{\mathrm{gc}}\}) \rightarrow$ *LStore*
(*LState*)	σ	$\in$	*LThrds* $\times$ *LHeap*

(b) Program States

Figure 15. The Low-level Language and State Model

descriptions of the machine models. The behaviors of the high-level abstract GC thread are defined as follows:

$$\mathsf{AbsGCStep} \triangleq \{((\Pi, \mathrm{H}), (\Pi, \mathrm{H}')) \mid \forall l.\ \mathsf{reachable}(l)(\Pi, \mathrm{H}) \Longrightarrow \mathrm{H}(l) = \mathrm{H}'(l)\}\,,$$

saying that, the mutator stores and the reachable objects in the heap are remained unmodified. Here $\mathsf{reachable}(l)(\Pi, \mathrm{H})$ means the object at the location l is reachable in H from the roots in Π.

The transformation. The transformation $\mathbf{T}$ 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 $x.\mathrm{id} := \mathbb{E}$ 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*:

$$\mathsf{wfstate}(\Pi, \mathrm{H}) \quad \triangleq \quad \forall l.\ \mathsf{reachable}(l)(\Pi, \mathrm{H}) \Longrightarrow l \in dom(\mathrm{H})\,.$$

  That is, reachable locations cannot be dangling pointers.

- High-level locations are transformed to integers by a bijective function $\mathsf{Loc2Int} : Loc \leftrightarrow [0..M]$ satisfying $\mathsf{Loc2Int}(\mathbf{nil}) = 0$.
- 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^{np} and v^{p} short for $(v, 0)$ and $(v, 1)$ respectively.
- High-level objects are transformed to the low level by adding the `color` and `dirty` fields with initial values `WHITE` and 0 respectively. Other addresses in the low-level heap domain $[1..M]$ are filled out using unallocated objects whose `color`s are `BLUE` and all the other fields are initialized by 0. Here we use `BLACK` and `WHITE` for marked and unmarked objects respectively, and `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), ϵ (for the mark stack `mstk`) or ϕ (for the root set `rt`).

$$s_{\mathrm{gc\_init}} \quad \triangleq \quad \{\mathtt{mstk} \rightsquigarrow \epsilon^{\mathrm{np}}, \mathtt{rt} \rightsquigarrow \phi^{\mathrm{np}}, \mathtt{i} \rightsquigarrow 0^{\mathrm{p}}, \mathtt{j} \rightsquigarrow 0^{\mathrm{p}},\\ \mathtt{c} \rightsquigarrow 0^{\mathrm{np}}, \mathtt{d} \rightsquigarrow 0^{\mathrm{np}}, \mathtt{t} \rightsquigarrow 0^{\mathrm{np}}\}$$

To prove $\mathsf{Correct}(\mathbf{T})$ in our framework, we apply Theorem 8, 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 α and $\zeta(\mathsf{t})$ relations.

$$\alpha \triangleq \{((\pi \uplus \{\mathsf{t}_{\mathrm{gc}} \rightsquigarrow \_\}, h), (\Pi, \mathrm{H})) \mid \\ \forall \mathsf{t} \in dom(\Pi).\ \mathsf{store\_map}(\pi(\mathsf{t}), \Pi(\mathsf{t})) \\ \wedge\ \mathsf{heap\_map}(h, \mathrm{H}) \wedge \mathsf{wfstate}(\Pi, \mathrm{H})\}\,.$$

In α, the relation between low-level and high-level stores and heaps are enforced by $\mathsf{store\_map}$ and $\mathsf{heap\_map}$ respectively. Their definitions reflect the state transformations we describe above, where we consider well-formed states only and use $\mathsf{Loc2Int}$ to relate integers and locations. The difference between α and $\mathbf{T}$ only lies in that, in α 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 α in Figure 17.

For each mutator thread t, the $\zeta(\mathsf{t})$ relation enforced at the beginning and the end of each transformation unit (each high-level instruction) is stronger than α. It requires that the value of the auxiliary variable `aux` (see Figure 13) be a null pointer (0^{p}):

$$\zeta(\mathsf{t}) \triangleq \alpha \cap \{((\pi, h), (\Pi, \mathrm{H})) \mid \pi(\mathsf{t})(\mathtt{aux}) = 0^{\mathrm{p}}\}\,.$$

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:

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

where $\mathcal{G}^{\mathsf{t}}_{\mathrm{write\_barrier}} \triangleq \mathcal{G}^{\mathsf{t}}_{\mathrm{write\_pt}} \cup \mathcal{G}^{\mathsf{t}}_{\mathrm{set\_dirty}}$ and $\mathbb{G}^{\mathsf{t}}_{\mathrm{write\_pt}}$ are the guarantees of the two-step write barrier and the high-level atomic write operation respectively. The proof is given in Appendix E.5. Since the transformation of other high-level instructions is identity, the proofs of the refinement are simple. For example, it's not difficult to prove:

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

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.*, $x := \mathbf{get\_root}(y)$) and adapting some heap manipulation rules to our low-level machine model (*e.g.*, $\mathbf{free}(x)$ just sets the object's color to `BLUE`). We give the inference rules and the soundness proofs in Appendix E.3.

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

$$p, q ::= B \mid \mathsf{t.own_p}(x) \mid \mathsf{t.own_{np}}(x) \mid E_1.id \mapsto E_2 \mid p * q \mid \ldots$$

Following Parkinson *et al.* [23], we treat program variables as resource and use $\mathsf{t.own_p}(x)$ and $\mathsf{t.own_{np}}(x)$ for the thread t's ownerships of pointers and non-pointers respectively. Also in B we can use $\mathsf{t}.x$ to denote the thread t's local variable x. We omit the thread identifiers if these predicates hold for the current thread. We use $E_1.id \mapsto E_2$ to specify a single-object single-field heap with E_2 stored in the field id of the object E_1. The separating conjunction $p * q$ means p and q hold on disjoint states. We use $E_1.id \hookrightarrow E_2$ for $E_1.id \mapsto E_2 * \mathbf{true}$ and $\circledast_{x \in S}.p(x)$ 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.*, $p_{\mathrm{gc}} \triangleq \mathsf{wfstate}$. Just corresponding to the high-level $\mathsf{wfstate}$ definition, the low-level $\mathsf{wfstate}$ predicate says that the heap contains M objects and none of the reachable objects are `BLUE`. We define the low-level $\mathsf{wfstate}$ predicate in Figure 19, It's easy to see that any low-level initial state is well-formed. We define $\mathcal{G}_{\mathrm{gc}}$ as follows:

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

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

$$\lfloor o \rfloor \triangleq \{\mathrm{pt}_1 \rightsquigarrow o(\mathrm{pt}_1), \ldots, \mathrm{pt}_m \rightsquigarrow o(\mathrm{pt}_m), \mathrm{data} \rightsquigarrow o(\mathrm{data})\}$$

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 followings 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 $\mathsf{reach\_inv}$.
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 `mstk`. 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, $\mathsf{reach\_inv}$ 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

$$\mathbf{T}(\mathbb{E}) \triangleq \begin{cases} 0 & \text{if } \mathbb{E} = \textbf{nil} \\ E & \text{otherwise} \end{cases}$$

$$\mathbf{T}(\mathbb{B}) \triangleq \begin{cases} \mathbf{T}(\mathbb{E}_1) = \mathbf{T}(\mathbb{E}_2) & \text{if } \mathbb{B} = (\mathbb{E}_1 = \mathbb{E}_2) \\ B & \text{otherwise} \end{cases}$$

$$\mathbf{T}(\mathrm{c}) \triangleq \begin{cases} \mathtt{update}(x.id, \mathbf{T}(\mathbb{E})) & \text{if } \mathrm{c} = (x.\mathrm{id} := \mathbb{E}) \wedge \mathrm{id} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \\ c & \text{otherwise} \end{cases}$$

$$\mathbf{T}(\mathbb{C}) \triangleq \begin{cases} \textbf{skip} & \text{if } \mathbb{C} = \textbf{\textit{skip}} \\ \mathbf{T}(\mathrm{c}) & \text{if } \mathbb{C} = \mathrm{c} \\ \mathbf{T}(\mathbb{C}'); \mathbf{T}(\mathbb{C}'') & \text{if } \mathbb{C} = \mathbb{C}';;\mathbb{C}'' \\ \textbf{if } (\mathbf{T}(\mathbb{B}))\ \{\mathbf{T}(\mathbb{C}')\}\ \textbf{else}\ \{\mathbf{T}(\mathbb{C}'')\} & \text{if } \mathbb{C} = \textbf{\textit{if}}\ \mathbb{B}\ \textbf{\textit{then}}\ \mathbb{C}'\ \textbf{\textit{else}}\ \mathbb{C}'' \\ \textbf{while } (\mathbf{T}(\mathbb{B}))\ \{\mathbf{T}(\mathbb{C}')\} & \text{if } \mathbb{C} = \textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}' \end{cases}$$

$$\mathbf{T}(\mathbb{W}) \triangleq \mathsf{t}_{\mathrm{gc}}.C_{\mathrm{gc}} \parallel \mathsf{t}_1.\mathbf{T}(\mathbb{C}_1) \parallel \ldots \parallel \mathsf{t}_n.\mathbf{T}(\mathbb{C}_n) \qquad \text{if } \mathbb{W} = \mathsf{t}_{\mathrm{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\ldots|||\, \mathsf{t}_n.\mathbb{C}_n$$

(a) $\mathbf{T}$ on Programs

$$\mathbf{T}(\mathrm{S})(x) \triangleq \begin{cases} n^{\mathrm{np}} & \text{if } \mathrm{S}(x) = n \\ n^{\mathrm{p}} & \text{if } \mathrm{S}(x) = l \wedge \mathsf{Loc2Int}(l) = n \\ 0^{\mathrm{p}} & \text{if } x = \mathtt{aux} \\ \bot & \text{if } x \notin dom(\mathrm{S}) \wedge x \neq \mathtt{aux} \end{cases}$$

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

$$\mathbf{T}(\Sigma) \triangleq \begin{cases} (\{\mathsf{t} \rightsquigarrow \mathbf{T}(\mathrm{S}) \mid (\mathsf{t} \rightsquigarrow \mathrm{S}) \in \Pi\} \uplus \{\mathsf{t}_{\mathrm{gc}} \rightsquigarrow s_{\mathrm{gc\_init}}\}, \mathbf{T}(\mathrm{H})) & \text{if } \Sigma = (\Pi, \mathrm{H}) \wedge \mathsf{wfstate}(\Sigma) \\ \bot & \text{otherwise} \end{cases}$$

(b) $\mathbf{T}$ on Initial States

Figure 16. The Transformation $\mathbf{T}$ for Boehm *et al.* GC

$$\mathsf{store\_map}(s, \mathrm{S}) \triangleq \forall x \neq \mathtt{aux}.\ (\forall n.\ s(x) = n^{\mathrm{np}} \iff \mathrm{S}(x) = n) \wedge (\forall n.\ s(x) = n^{\mathrm{p}} \iff \exists l.\ \mathsf{Loc2Int}(l) = n \wedge \mathrm{S}(x) = l)$$

$$\begin{aligned} \mathsf{heap\_map}(h, \mathrm{H}) \triangleq\ & \forall i, l, n, n_1, \ldots, n_m.\ 1 \leq i \leq M \wedge \mathsf{Loc2Int}(l) = i \\ & \implies (h(i)(\mathtt{color}) \neq \mathtt{BLUE} \wedge h(i) = \{\mathrm{pt}_1 \rightsquigarrow n_1, \ldots, \mathrm{pt}_m \rightsquigarrow n_m, \mathrm{data} \rightsquigarrow n, \mathtt{color} \rightsquigarrow \_, \mathtt{dirty} \rightsquigarrow \_\} \\ & \qquad \iff \exists l_1, \ldots, l_m.\ \mathsf{Loc2Int}(l_1) = n_1 \wedge \ldots \wedge \mathsf{Loc2Int}(l_m) = n_m \wedge \mathrm{H}(l) = \{\mathrm{pt}_1 \rightsquigarrow l_1, \ldots, \mathrm{pt}_m \rightsquigarrow l_m, \mathrm{data} \rightsquigarrow n\}) \\ & \quad \wedge (h(i) = \{\mathrm{pt}_1 \rightsquigarrow n_1, \ldots, \mathrm{pt}_m \rightsquigarrow n_m, \mathrm{data} \rightsquigarrow \_, \mathtt{color} \rightsquigarrow \mathtt{BLUE}, \mathtt{dirty} \rightsquigarrow \_\} \iff l \notin dom(\mathrm{H})) \end{aligned}$$

$$\alpha \triangleq \{((\pi \uplus \{\mathsf{t}_{\mathrm{gc}} \rightsquigarrow \_\}, h), (\Pi, \mathrm{H})) \mid \forall \mathsf{t} \in dom(\Pi).\ \mathsf{store\_map}(\pi(\mathsf{t}), \Pi(\mathsf{t})) \wedge \mathsf{heap\_map}(h, \mathrm{H}) \wedge \mathsf{wfstate}(\Pi, \mathrm{H})\}$$

Figure 17. The α Relation for Boehm *et al.* GC

unmarked object and its descendants are traced using the mark stack. Since any mutator updates during this phase still dirty the corresponding object, we can conclude $\mathsf{reach\_inv}$ is maintained at the end of this phase.

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, $\mathsf{reach\_rtnw\_stk}(X)$ holds, saying that all white reachable objects can be traced either from an object on the stack X or a black dirty object. Moreover, all the objects on the stack are black ($\mathsf{stk\_black}(X)$).
   (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 $\mathsf{reach\_black}$).
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 $\mathcal{G}_{\mathrm{gc}}$. After sweep, the state is still well-formed.

The predicates $\mathsf{reach\_inv}$, $\mathsf{reach\_rtnw\_stk}(X)$ and $\mathsf{reach\_black}$ 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.

$$
\begin{array}{lll}
\mathbb{G}^{\mathsf{t}}_{\mathrm{assgn\_int}} & \triangleq & \{((\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathrm{S_t}\}, \mathrm{H}), (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathrm{S'_t}\}, \mathrm{H})) \\
& & \mid \exists x, n, n'. \, \mathrm{S_t}(x) = n \wedge \mathrm{S'_t} = \mathrm{S_t}\{x \rightsquigarrow n'\}\} \\
\mathbb{G}^{\mathsf{t}}_{\mathrm{assgn\_pt}} & \triangleq & \{((\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathrm{S_t}\}, \mathrm{H}), (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathrm{S'_t}\}, \mathrm{H})) \\
& & \mid \exists x, l, l'. \, \mathrm{S_t}(x) = l \wedge \mathrm{S'_t} = \mathrm{S_t}\{x \rightsquigarrow l'\} \wedge (l' = \mathbf{nil} \vee (\exists x'. \, \mathrm{S_t}(x') = l') \vee (\exists y, l'', \mathrm{id}. \, \mathrm{S_t}(y) = l'' \wedge \mathrm{H}(l'')(\mathrm{id}) = l'))\} \\
\mathbb{G}^{\mathsf{t}}_{\mathrm{write\_data}} & \triangleq & \{((\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathrm{S_t}\}, \mathrm{H}), (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathrm{S_t}\}, \mathrm{H'})) \\
& & \mid \exists x, l, \mathrm{O}. \, \mathrm{S_t}(x) = l \wedge \mathrm{H}(l) = \mathrm{O} \wedge \mathrm{H'} = \mathrm{H}\{l \rightsquigarrow \mathrm{O}\{\mathrm{data} \rightsquigarrow \_\}\}\} \\
\mathbb{G}^{\mathsf{t}}_{\mathrm{write\_pt}} & \triangleq & \{((\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathrm{S_t}\}, \mathrm{H}), (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathrm{S_t}\}, \mathrm{H'})) \\
& & \mid \exists x, l, \mathrm{id}, \mathrm{O}, l', l''. \, \mathrm{S_t}(x) = l \wedge \mathrm{H}(l) = \mathrm{O} \wedge \mathrm{O}(\mathrm{id}) = l' \wedge \mathrm{H'} = \mathrm{H}\{l \rightsquigarrow \mathrm{O}\{\mathrm{id} \rightsquigarrow l''\}\} \wedge (l'' = \mathbf{nil} \vee \exists x'. \, \mathrm{S_t}(x') = l'')\} \\
\mathbb{G}^{\mathsf{t}}_{\mathrm{new}} & \triangleq & \{((\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathrm{S_t}\}, \mathrm{H}), (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathrm{S'_t}\}, \mathrm{H'})) \\
& & \mid \exists x, l, l'. \, \mathrm{S_t}(x) = l \wedge \mathrm{S'_t} = \mathrm{S_t}\{x \rightsquigarrow l'\} \wedge l' \notin dom(\mathrm{H}) \wedge \mathrm{H'} = \mathrm{H} \uplus \{l' \rightsquigarrow \{\mathrm{pt}_1 \rightsquigarrow \mathbf{nil}, \ldots, \mathrm{pt}_m \rightsquigarrow \mathbf{nil}, \mathrm{data} \rightsquigarrow 0\}\}\} \\
\\
\mathbb{G}(\mathsf{t}) & \triangleq & \mathbb{G}^{\mathsf{t}}_{\mathrm{assgn\_int}} \cup \mathbb{G}^{\mathsf{t}}_{\mathrm{assgn\_pt}} \cup \mathbb{G}^{\mathsf{t}}_{\mathrm{write\_data}} \cup \mathbb{G}^{\mathsf{t}}_{\mathrm{write\_pt}} \cup \mathbb{G}^{\mathsf{t}}_{\mathrm{new}}
\end{array}
$$

(a) High-Level Guarantees

$$
\begin{array}{lll}
\mathcal{G}^{\mathsf{t}}_{\mathrm{assgn\_int}} & \triangleq & \{((\pi \uplus \{\mathsf{t} \rightsquigarrow s_\mathsf{t}\}, h), (\pi \uplus \{\mathsf{t} \rightsquigarrow s'_\mathsf{t}\}, h)) \\
& & \mid s_\mathsf{t}(\mathtt{aux}) = 0^{\mathrm{p}} \wedge \exists x, n, n'. \, s_\mathsf{t}(x) = n^{\mathrm{np}} \wedge s'_\mathsf{t} = s_\mathsf{t}\{x \rightsquigarrow n'^{\mathrm{np}}\}\} \\
\mathcal{G}^{\mathsf{t}}_{\mathrm{assgn\_pt}} & \triangleq & \{((\pi \uplus \{\mathsf{t} \rightsquigarrow s_\mathsf{t}\}, h), (\pi \uplus \{\mathsf{t} \rightsquigarrow s'_\mathsf{t}\}, h)) \\
& & \mid s_\mathsf{t}(\mathtt{aux}) = 0^{\mathrm{p}} \wedge \exists x, n, n'. \, s_\mathsf{t}(x) = n^{\mathrm{p}} \wedge s'_\mathsf{t} = s_\mathsf{t}\{x \rightsquigarrow n'^{\mathrm{p}}\} \\
& & \quad \wedge (n' = 0 \vee (\exists x'. \, s_\mathsf{t}(x') = n'^{\mathrm{p}}) \vee (\exists y, n'', id. \, id \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \wedge s_\mathsf{t}(y) = n''^{\mathrm{p}} \wedge h(n'')(id) = n'))\} \\
\mathcal{G}^{\mathsf{t}}_{\mathrm{write\_data}} & \triangleq & \{((\pi \uplus \{\mathsf{t} \rightsquigarrow s_\mathsf{t}\}, h), (\pi \uplus \{\mathsf{t} \rightsquigarrow s_\mathsf{t}\}, h')) \\
& & \mid s_\mathsf{t}(\mathtt{aux}) = 0^{\mathrm{p}} \wedge \exists x, n, o. \, s_\mathsf{t}(x) = n^{\mathrm{p}} \wedge h(n) = o \wedge h' = h\{l \rightsquigarrow o\{\mathrm{data} \rightsquigarrow \_\}\}\} \\
\mathcal{G}^{\mathsf{t}}_{\mathrm{write\_pt}} & \triangleq & \{((\pi \uplus \{\mathsf{t} \rightsquigarrow s_\mathsf{t}\}, h), (\pi \uplus \{\mathsf{t} \rightsquigarrow s'_\mathsf{t}\}, h')) \\
& & \mid s_\mathsf{t}(\mathtt{aux}) = 0^{\mathrm{p}} \wedge \exists x, n, id, o, n''. \, id \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \wedge s_\mathsf{t}(x) = n^{\mathrm{p}} \wedge h(n) = o \wedge h' = h\{n \rightsquigarrow o\{id \rightsquigarrow n''\}\} \\
& & \quad \wedge (n'' = 0 \vee \exists x'. \, s_\mathsf{t}(x') = n''^{\mathrm{p}}) \wedge s'_\mathsf{t} = s_\mathsf{t}\{\mathtt{aux} \rightsquigarrow n^{\mathrm{p}}\}\} \\
\mathcal{G}^{\mathsf{t}}_{\mathrm{set\_dirty}} & \triangleq & \{((\pi \uplus \{\mathsf{t} \rightsquigarrow s_\mathsf{t}\}, h), (\pi \uplus \{\mathsf{t} \rightsquigarrow s'_\mathsf{t}\}, h')) \\
& & \mid s_\mathsf{t}(\mathtt{aux}) = n^{\mathrm{p}} \wedge \exists n, o. \, h(n) = o \wedge h' = h\{n \rightsquigarrow o\{\mathtt{dirty} \rightsquigarrow 1\}\} \wedge s'_\mathsf{t} = s_\mathsf{t}\{\mathtt{aux} \rightsquigarrow 0^{\mathrm{p}}\}\} \\
\mathcal{G}^{\mathsf{t}}_{\mathrm{new}} & \triangleq & \{((\pi \uplus \{\mathsf{t} \rightsquigarrow s_\mathsf{t}\}, h), (\pi \uplus \{\mathsf{t} \rightsquigarrow s'_\mathsf{t}\}, h')) \\
& & \mid s_\mathsf{t}(\mathtt{aux}) = 0^{\mathrm{p}} \wedge \exists x, n, n'. \, s_\mathsf{t}(x) = n^{\mathrm{p}} \wedge s'_\mathsf{t} = s_\mathsf{t}\{x \rightsquigarrow n'^{\mathrm{p}}\} \wedge h(n')(\mathtt{color}) = \mathtt{BLUE} \\
& & \quad \wedge h' = h\{n' \rightsquigarrow \{\mathrm{pt}_1 \rightsquigarrow 0, \ldots, \mathrm{pt}_m \rightsquigarrow 0, \mathrm{data} \rightsquigarrow 0, \mathtt{color} \rightsquigarrow \mathtt{BLACK}, \mathtt{dirty} \rightsquigarrow 0\}\}\} \\
\\
\mathcal{G}(\mathsf{t}) & \triangleq & \mathcal{G}^{\mathsf{t}}_{\mathrm{assgn\_int}} \cup \mathcal{G}^{\mathsf{t}}_{\mathrm{assgn\_pt}} \cup \mathcal{G}^{\mathsf{t}}_{\mathrm{write\_data}} \cup \mathcal{G}^{\mathsf{t}}_{\mathrm{write\_pt}} \cup \mathcal{G}^{\mathsf{t}}_{\mathrm{set\_dirty}} \cup \mathcal{G}^{\mathsf{t}}_{\mathrm{new}}
\end{array}
$$

(b) Low-Level Guarantees

Figure 18. Guarantees of Mutator Instructions

$$
\begin{array}{lll}
\mathsf{obj}(x) & \triangleq & \exists n_1, \ldots, n_m. \, (x.\mathrm{pt}_1 \mapsto n_1 * \ldots * x.\mathrm{pt}_m \mapsto n_m * x.\mathrm{data} \mapsto \_ * x.\mathtt{color} \mapsto \_ * x.\mathtt{dirty} \mapsto \_) \\
& & \quad \wedge 0 \le n_1 \le M \wedge \ldots \wedge 0 \le n_m \le M \\
\mathsf{wfstate} & \triangleq & \circledast_{x \in [1..M]}.\mathsf{obj}(x) * \mathbf{true} \wedge (\forall x. \, \mathsf{reachable}(x) \Longrightarrow \mathsf{not\_blue}(x)) \\
\mathsf{rt\_wp}(x) & \triangleq & \exists t \in [1..N]. \, \exists S, y. \, \mathsf{root}(t, S) \wedge y \in S \wedge \mathsf{white}(y) \wedge \mathsf{white\_path}(y, x) \\
\mathsf{dt\_bwp}(x, y) & \triangleq & \mathsf{black}(x) \wedge \mathsf{dirty}(x) \wedge \mathsf{white\_path}(x, y) \\
\mathsf{stk\_bwp}(x, y, A) & \triangleq & \mathsf{black}(x) \wedge \mathsf{instk}(x, A) \wedge \mathsf{white\_path}(x, y) \\
\mathsf{reach\_inv} & \triangleq & \forall x. \, \mathsf{reachable}(x) \wedge \mathsf{white}(x) \implies \mathsf{rt\_wp}(x) \vee \exists x'.\mathsf{dt\_bwp}(x', x) \\
\mathsf{reach\_rtnw\_stk}(A) & \triangleq & \forall x. \, \mathsf{reachable}(x) \wedge \mathsf{white}(x) \implies \exists x'.\mathsf{dt\_bwp}(x', x) \vee \exists x'.\mathsf{stk\_bwp}(x', x, A) \\
\mathsf{reach\_black} & \triangleq & \forall x. \, \mathsf{reachable}(x) \Longrightarrow \mathsf{black}(x)
\end{array}
$$

Figure 19. Boehm *et al.* GC Predicates

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, Lochbihler [21] 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], Ševčík *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 optimizations. Also the proof rules of RGSim for sequential compositions, conditional statements and loops coincide with those in relational Hoare logic [3] and relational separation logic [32].

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

Turon and Wand [27] propose a refinement-based proof method to verify concurrent objects. They first propose a simple refinement based on Brookes' fully abstract trace semantics [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.* [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 [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. Vechev *et al.* [30] define transformations to generate concurrent GCs from an abstract collector. Afterwards, Pavlovic *et al.* [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.* [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.* [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 NSFC 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

[1] M. Abadi and G. Plotkin. A model of cooperative threads. In *Proc. 36th ACM Symp. on Principles of Prog. Lang. (POPL'09)*, pages 29–40. ACM Press, January 2009.

[2] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst.*, 27(6):1097–1146, 2005.

[3] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. 31th ACM Symp. on Principles of Prog. Lang. (POPL'04)*, pages 14–25. ACM Press, January 2004.

[4] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. 14th ACM Int'l Conf. on Functional Prog. (ICFP'09)*, pages 97–108. ACM Press, September 2009.

[5] H.-J. Boehm. Threads cannot be implemented as a library. In *Proc. 2005 ACM Conf. on Prog. Lang. Design and Impl. (PLDI'05)*, pages 261–268. ACM Press, June 2005.

[6] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proc. 2008 ACM Conf. on Prog. Lang. Design and Impl. (PLDI'08)*, pages 68–78. ACM Press, June 2008.

[7] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proc. 1991 ACM Conf. on Prog. Lang. Design and Impl. (PLDI'91)*, pages 157–164. ACM Press, June 1991.

[8] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

[9] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In *Proc. 19th Int'l Conf. on Compiler Construction (CC'10)*, volume 6011 of *Lecture Notes in Computer Science*, pages 104–123. Springer, March 2010.

[10] Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.3, October 2010.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. 20th Int'l Symp. on Distributed Computing (DISC'06)*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer, September 2006.

[12] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *Proc. 16th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 296–311. Springer, March 2010.

[13] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *Proc. 18th European Symp. on Prog. (ESOP'09)*, volume 5502 of *Lecture Notes in Computer Science*, pages 252–266. Springer, March 2009.

[14] D. S. Gladstein and M. Wand. Compiler correctness for concurrent languages. In *Proc. 1st Int'l Conf. on Coordination Languages*

and Models (COORDINATION'96), volume 1061 of *Lecture Notes in Computer Science*, pages 231–248. Springer, April 1996.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, April 2008.

[16] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *Proc. 38th ACM Symp. on Principles of Prog. Lang. (POPL'11)*, pages 133–146. ACM Press, January 2011.

[17] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[18] K. Kapoor, K. Lodaya, and U. Reddy. Fine-grained concurrency with separation logic. *J. Philosophical Logic*, 40(5):583–632, 2011.

[19] X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

[20] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. Technical report (with Coq implementation), University of Science and Technology of China, October 2011. `http://kyhcs.ustcsz.edu.cn/relconcur/rgsim`.

[21] A. Lochbihler. Verifying a compiler for java threads. In *Proc. 20th European Symp. on Prog. (ESOP'10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 427–447. Springer, March 2010.

[22] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Proc. 2007 ACM Conf. on Prog. Lang. Design and Impl. (PLDI'07)*, pages 468–479. ACM Press, June 2007.

[23] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *Proc. 21th IEEE Symp. on Logic In Computer Science (LICS'06)*, pages 137–146. IEEE Computer Society, August 2006.

[24] D. Pavlovic, P. Pepper, and D. R. Smith. Formal derivation of concurrent garbage collectors. In *Proc. 10th Int'l Conf. on Mathematics of Program Construction (MPC'10)*, volume 6120 of *Lecture Notes in Computer Science*, pages 353–376. Springer, June 2010.

[25] J. Ševčík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proc. 38th ACM Symp. on Principles of Prog. Lang. (POPL'11)*, pages 43–54. ACM Press, January 2011.

[26] R. K. Treiber. System programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[27] A. Turon and M. Wand. A separation logic for refining concurrent objects. In *Proc. 38th ACM Symp. on Principles of Prog. Lang. (POPL'11)*, pages 247–258. ACM Press, January 2011.

[28] V. Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[29] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. 18th Int'l Conf. on Concurrency Theory (CONCUR'07)*, volume 4703, pages 256–271. Springer, 2007.

[30] M. T. Vechev, E. Yahav, and D. F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *Proc. 2006 ACM Conf. on Prog. Lang. Design and Impl. (PLDI'06)*, pages 341–353. ACM Press, June 2006.

[31] M. Wand. Compiler correctness for parallel languages. In *Proc. Conf. on Functional Prog. Lang. and Computer Architecture (FPCA'95)*, pages 120–134. ACM Press, June 1995.

[32] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

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', σ and σ', if $(C,\sigma) \longrightarrow^k (C',\sigma')$, then $\mathit{ETrSet}_m(C',\sigma') \subseteq \mathit{ETrSet}_{k+m}(C,\sigma)$.*

Proof:. By induction over k.
Base Case: $k = 0$, then $C' = C$ and $\sigma' = \sigma$, trivial.
Inductive Step: $k = n + 1$.
By unfolding $(C,\sigma) \longrightarrow^{n+1} (C',\sigma')$, we know there exists C'' and σ'' such that

$$(C,\sigma) \longrightarrow (C'',\sigma'') \tag{A.1}$$

and

$$(C'',\sigma'') \longrightarrow^n (C',\sigma'). \tag{A.2}$$

From (A.1) and Definition 1, we know

$$\mathit{ETrSet}_{n+m}(C'',\sigma'') \subseteq \mathit{ETrSet}_{n+m+1}(C,\sigma). \tag{A.3}$$

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

$$\mathit{ETrSet}_m(C',\sigma') \subseteq \mathit{ETrSet}_{n+m}(C'',\sigma''). \tag{A.4}$$

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

Lemma 10. *For all $k \geq 0$ and $m \geq 0$, for all $\mathcal{E}$, for all C, C', σ and σ', if $(C,\sigma) \xrightarrow{e}{}^k (C',\sigma')$ and $\mathcal{E} \in \mathit{ETrSet}_m(C',\sigma')$, then $e::\mathcal{E} \in \mathit{ETrSet}_{k+m}(C,\sigma)$.*

Proof:. By induction over k.
Base Case: $k = 0$, trivial.
Inductive Step: $k = n + 1$.
By unfolding $(C,\sigma) \xrightarrow{e}{}^{n+1} (C',\sigma')$, one of the following two cases holds:

1. there exists C'' and σ'' such that

$$(C,\sigma) \longrightarrow (C'',\sigma'') \tag{A.1}$$

and

$$(C'',\sigma'') \xrightarrow{e}{}^n (C',\sigma'). \tag{A.2}$$

From (A.1) and Definition 1, we know

$$\mathit{ETrSet}_{n+m}(C'',\sigma'') \subseteq \mathit{ETrSet}_{n+m+1}(C,\sigma). \tag{A.3}$$

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

$$e::\mathcal{E} \in \mathit{ETrSet}_{n+m}(C'',\sigma''). \tag{A.4}$$

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

2. there exists C'' and σ'' such that

$$(C,\sigma) \xrightarrow{e} (C'',\sigma'') \tag{A.5}$$

and

$$(C'',\sigma'') \longrightarrow^n (C',\sigma'). \tag{A.6}$$

From (A.6) and Lemma 9, we know

$$\mathit{ETrSet}_m(C',\sigma') \subseteq \mathit{ETrSet}_{n+m}(C'',\sigma''). \tag{A.7}$$

Thus

$$\mathcal{E} \in \mathit{ETrSet}_{n+m}(C'',\sigma''). \tag{A.8}$$

From (A.5), (A.8) and Definition 1, we know $e :: \mathcal{E} \in \mathit{ETrSet}_{n+m+1}(C,\sigma)$.

In both cases, we can get the conclusion. □

Lemma 11. *For all $k \geq 0$, for all C, $\mathbb{C}$, σ, Σ, $\mathcal{R}$, $\mathbb{R}$, $\mathcal{G}$, $\mathbb{G}$, α and γ, if $(C,\sigma,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C},\Sigma,\mathbb{R},\mathbb{G})$, then $\mathit{ETrSet}_k(C,\sigma) \subseteq \mathit{ETrSet}(\mathbb{C},\Sigma)$.*

Proof:. By induction over k.
Base Case: $k = 0$. We know $\{\epsilon\} \subseteq \mathit{ETrSet}(\mathbb{C},\Sigma)$ always holds.
Inductive Step: $k = n + 1$.
For all $\mathcal{E} \in \mathit{ETrSet}_{n+1}(C,\sigma)$, by Definition 1, we have four cases:

1. If $C = \textbf{skip}$, then $\mathcal{E} = \textbf{done}$. By unfolding $(\textbf{skip},\sigma,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C},\Sigma,\mathbb{R},\mathbb{G})$, we know there exists Σ' such that

$$(\mathbb{C}, \Sigma) \longrightarrow^* (\textit{\textbf{skip}}, \Sigma').$$

From Lemma 9 and $\mathcal{E} \in \textit{ETrSet}_1(\textit{\textbf{skip}}, \Sigma')$, we know

$$\mathcal{E} \in \textit{ETrSet}(\mathbb{C}, \Sigma). \quad \text{(A.1)}$$

2. If $(C, \sigma) \longrightarrow (C', \sigma')$ and $\mathcal{E} \in \textit{ETrSet}_n(C', \sigma')$, then by unfolding $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, we know there exist $\mathbb{C}'$ and Σ' such that

$$(\mathbb{C}, \Sigma) \longrightarrow^* (\mathbb{C}', \Sigma') \quad \text{(A.2)}$$

and

$$(C', \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}', \Sigma', \mathbb{R}, \mathbb{G}). \quad \text{(A.3)}$$

From (A.3) and the induction hypothesis, we know $\textit{ETrSet}_n(C', \sigma') \subseteq \textit{ETrSet}(\mathbb{C}', \Sigma')$.
Thus

$$\mathcal{E} \in \textit{ETrSet}(\mathbb{C}', \Sigma'). \quad \text{(A.4)}$$

By (A.2), Lemma 9 and (A.4), we know

$$\mathcal{E} \in \textit{ETrSet}(\mathbb{C}, \Sigma). \quad \text{(A.5)}$$

3. If $(C, \sigma) \xrightarrow{e} (C', \sigma')$, $\mathcal{E} = e :: \mathcal{E}'$ and $\mathcal{E}' \in \textit{ETrSet}_n(C', \sigma')$, then by unfolding $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, we know there exist $\mathbb{C}'$ and Σ' such that

$$(\mathbb{C}, \Sigma) \xrightarrow{e}{}^* (\mathbb{C}', \Sigma') \quad \text{(A.6)}$$

and

$$(C', \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}', \Sigma', \mathbb{R}, \mathbb{G}). \quad \text{(A.7)}$$

From (A.7) and the induction hypothesis, we know $\textit{ETrSet}_n(C', \sigma') \subseteq \textit{ETrSet}(\mathbb{C}', \Sigma')$.
Thus

$$\mathcal{E}' \in \textit{ETrSet}(\mathbb{C}', \Sigma'). \quad \text{(A.8)}$$

By (A.6), Lemma 10 and (A.8), we know

$$\mathcal{E} \in \textit{ETrSet}(\mathbb{C}, \Sigma). \quad \text{(A.9)}$$

4. If $(C, \sigma) \longrightarrow$ **abort** and $\mathcal{E} =$ **abort**, then by unfolding $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, we know $(\mathbb{C}, \Sigma) \longrightarrow^*$ **abort**. Then we can prove

$$\mathcal{E} \in \textit{ETrSet}(\mathbb{C}, \Sigma). \quad \text{(A.10)}$$

From (A.1), (A.5), (A.9) and (A.10), we get the conclusion. □

We get Theorem 5 immediately from Lemma 11.

B. Soundness of Compositionality Rules

B.1 Soundness of the SEQ Rule

Lemma 12. *For all C_1, C_2, $\mathbb{C}_1$ and $\mathbb{C}_2$, for all σ and Σ, if*

1. $(C_1, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}_1, \Sigma, \mathbb{R}, \mathbb{G})$*; and*
2. *for all σ_2 and Σ_2, if $(\sigma_2, \Sigma_2) \in \gamma$, then $(C_2, \sigma_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\eta} (\mathbb{C}_2, \Sigma_2, \mathbb{R}, \mathbb{G})$;*

then

$$(C_1; C_2, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\eta} (\mathbb{C}_1;;\mathbb{C}_2, \Sigma, \mathbb{R}, \mathbb{G}).$$

Proof:. By co-induction.
Let

$$S = \{((C_1; C_2, \sigma), (\mathbb{C}_1;;\mathbb{C}_2, \Sigma)) \mid \text{the premises hold}\} \cup \{((C_2, \sigma_2), (\mathbb{C}_2, \Sigma_2)) \mid (\sigma_2, \Sigma_2) \in \gamma\}.$$

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 $((C_2, \sigma_2), (\mathbb{C}_2, \Sigma_2))$ satisfies the simulation.
For all $((C_1; C_2, \sigma), (\mathbb{C}_1;;\mathbb{C}_2, \Sigma)) \in S$, we know $(\sigma, \Sigma) \in \alpha$.

1. If $(C_1; C_2, \sigma) \longrightarrow (C', \sigma')$, then according to the operational semantics, we have two possible cases:
   - $C_1 \neq$ **skip**. Thus $C' = C_1'; C_2$ and
     $$(C_1, \sigma) \longrightarrow (C_1', \sigma').$$
     From the 1st premise, we know $(\sigma, \sigma') \in \mathcal{G}$ and there exist $\mathbb{C}_1'$ and Σ' such that the followings hold:
     $$(\mathbb{C}_1, \Sigma) \longrightarrow^* (\mathbb{C}_1', \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}^*$$
     $$(C_1', \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}_1', \Sigma', \mathbb{R}, \mathbb{G})$$
     Thus we know $((C_1'; C_2, \sigma'), (\mathbb{C}_1';;\mathbb{C}_2, \Sigma')) \in S$.
   - $C_1 =$ **skip**. Thus $C' = C_2$ and $\sigma' = \sigma$.
     Since $\mathcal{G}$ contains identity transitions, we know $(\sigma, \sigma') \in \mathcal{G}$.
     From the 1st premise, we know there exists Σ' such that
     $$(\mathbb{C}_1, \Sigma) \longrightarrow^* (\textit{\textbf{skip}}, \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}^*, (\sigma, \Sigma') \in \gamma$$
     Thus $(\mathbb{C}_1;;\mathbb{C}_2, \Sigma) \longrightarrow^* (\mathbb{C}_2, \Sigma')$ and
     $$((C_2, \sigma), (\mathbb{C}_2, \Sigma')) \in S.$$
2. If $(C_1; C_2, \sigma) \xrightarrow{e} (C', \sigma')$, the proof is similar to the previous case.
3. If $(\sigma, \sigma') \in \mathcal{R}$, $(\Sigma, \Sigma') \in \mathbb{R}^*$ and $(\sigma', \Sigma') \in \alpha$, then from the 1st premise, we have
   $$(C_1, \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}_1, \Sigma', \mathbb{R}, \mathbb{G}).$$
   Thus $((C_1; C_2, \sigma'), (\mathbb{C}_1;;\mathbb{C}_2, \Sigma')) \in S$.
4. $C_1; C_2$ cannot be **skip**, so this case is vacantly true.
5. If $(C_1; C_2, \sigma) \longrightarrow$ **abort**, then $(\mathbb{C}_1;;\mathbb{C}_2, \Sigma) \longrightarrow^*$ **abort** is immediate from the 1st premise.

Then we have $((C_1; C_2, \sigma), (\mathbb{C}_1;;\mathbb{C}_2, \Sigma)) \in F(S)$. Thus $(C_1; C_2, \sigma)$ and $(\mathbb{C}_1;;\mathbb{C}_2, \Sigma)$ satisfy the largest simulation RGSim. □

Then we can conclude soundness of the SEQ rule.

B.2 Soundness of the IF Rule

Lemma 13. *For all C_1, C_2, $\mathbb{C}_1$ and $\mathbb{C}_2$, for all σ and Σ, if*

1. *for all σ_1 and Σ_1, if $(\sigma_1, \Sigma_1) \in \zeta_1 = (\zeta \cap (B \mathbin{\char"2A53} \mathbb{B}))$, then $(C_1, \sigma_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}_1, \Sigma_1, \mathbb{R}, \mathbb{G})$;*
2. *for all σ_2 and Σ_2, if $(\sigma_2, \Sigma_2) \in \zeta_2 = (\zeta \cap (\neg B \mathbin{\char"2A53} \neg\mathbb{B}))$, then $(C_2, \sigma_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}_2, \Sigma_2, \mathbb{R}, \mathbb{G})$;*
3. $\zeta \subseteq (B \Leftrightarrow \mathbb{B})$*; and*
4. $(\sigma, \Sigma) \in \zeta \subseteq \alpha$,

then

$$(\textbf{if } (B)\ C_1 \textbf{ else } C_2, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\textit{\textbf{if}}\ \mathbb{B}\ \textit{\textbf{then}}\ \mathbb{C}_1\ \textit{\textbf{else}}\ \mathbb{C}_2, \Sigma, \mathbb{R}, \mathbb{G})$$

Proof:. By co-induction.
Let

$$S = \{((\textbf{if } (B)\ C_1 \textbf{ else } C_2, \sigma), (\textit{\textbf{if}}\ \mathbb{B}\ \textit{\textbf{then}}\ \mathbb{C}_1\ \textit{\textbf{else}}\ \mathbb{C}_2, \Sigma)) \mid \text{the premises hold}\} \cup \{((C_1, \sigma_1), (\mathbb{C}_1, \Sigma_1)) \mid (\sigma_1, \Sigma_1) \in \zeta_1\} \cup \{((C_2, \sigma_2), (\mathbb{C}_2, \Sigma_2)) \mid (\sigma_2, \Sigma_2) \in \zeta_2\}.$$

We prove $S \subseteq F(S)$ where F is defined by the simulation.
For all $((\textbf{if } (B)\ C_1 \textbf{ else } C_2, \sigma), (\textit{\textbf{if}}\ \mathbb{B}\ \textit{\textbf{then}}\ \mathbb{C}_1\ \textit{\textbf{else}}\ \mathbb{C}_2, \Sigma)) \in S$,

1. If $(\textbf{if } (B)\ C_1 \textbf{ else } C_2, \sigma) \longrightarrow (C', \sigma')$, then according to the operational semantics, we have two possible cases:

- $B\ \sigma = \textbf{true}$. Thus $C' = C_1$ and $\sigma' = \sigma$.
  Since $\mathcal{G}$ contains identity transitions, we know $(\sigma, \sigma') \in \mathcal{G}$.
  From $\zeta \subseteq (B \Leftrightarrow\!\!\Leftrightarrow \mathbb{B})$, we know $\mathbb{B}\ \Sigma = \textbf{true}$. Thus
  $$(\textbf{\textit{if}}\ \mathbb{B}\ \textbf{\textit{then}}\ \mathbb{C}_1\ \textbf{\textit{else}}\ \mathbb{C}_2, \Sigma) \longrightarrow (\mathbb{C}_1, \Sigma), (\Sigma, \Sigma) \in \mathbb{G}^*.$$
  Since $(\sigma, \Sigma) \in \zeta_1$, we know $((C_1, \sigma), (\mathbb{C}_1, \Sigma)) \in S$.
- $B\ \sigma = \textbf{false}$. The proof is similar to the previous case.

2. The case for $(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, \sigma) \stackrel{e}{\longrightarrow} (C', \sigma')$ is vacantly true.
3. If $(\sigma, \sigma') \in \mathcal{R}$, $(\Sigma, \Sigma') \in \mathbb{R}^*$ and $(\sigma', \Sigma') \in \alpha$, then from $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$, we know $(\sigma', \Sigma') \in \zeta$. Thus $((\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, \sigma'), (\textbf{\textit{if}}\ \mathbb{B}\ \textbf{\textit{then}}\ \mathbb{C}_1\ \textbf{\textit{else}}\ \mathbb{C}_2, \Sigma')) \in S$.
4. $(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2)$ is not **skip**, so this case is vacantly true.
5. If $(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, \sigma) \longrightarrow \textbf{abort}$, then $B\ \sigma = \perp$.
   Since $\zeta \subseteq (B \Leftrightarrow\!\!\Leftrightarrow \mathbb{B})$ and $(\sigma, \Sigma) \in \zeta$, we know $\mathbb{B}\ \Sigma = \perp$.
   Thus $(\textbf{\textit{if}}\ \mathbb{B}\ \textbf{\textit{then}}\ \mathbb{C}_1\ \textbf{\textit{else}}\ \mathbb{C}_2, \sigma) \longrightarrow \textbf{abort}$.

Then we have $(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, \sigma), (\textbf{\textit{if}}\ \mathbb{B}\ \textbf{\textit{then}}\ \mathbb{C}_1\ \textbf{\textit{else}}\ \mathbb{C}_2, \Sigma)) \in F(S)$. Thus $(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, \sigma)$ and $(\textbf{\textit{if}}\ \mathbb{B}\ \textbf{\textit{then}}\ \mathbb{C}_1\ \textbf{\textit{else}}\ \mathbb{C}_2, \Sigma)$ satisfy the largest simulation RGSim. □

Then we can conclude soundness of the IF rule.

B.3 Soundness of the WHILE Rule

Lemma 14. *For all C and $\mathbb{C}$, for all σ and Σ, if*

1. *for all σ_1 and Σ_1, if $(\sigma_1, \Sigma_1) \in \gamma_1 = (\gamma \cap (B \wedge\!\!\!\wedge \mathbb{B}))$, then $(C, \sigma_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma_1, \mathbb{R}, \mathbb{G})$;*
2. *$\gamma \subseteq (B \Leftrightarrow\!\!\Leftrightarrow \mathbb{B})$;*
3. *$\gamma_2 = (\gamma \cap (\neg B \wedge\!\!\!\wedge \neg \mathbb{B}))$; and*
4. *$(\sigma, \Sigma) \in \gamma$,*

then
$$(\textbf{while}\ (B)\ C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma_2} (\textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G}).$$

Proof:. By co-induction.
Let
$$\begin{aligned} S = \{&((\textbf{while}\ (B)\ C, \sigma), (\textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma)) \\ &\mid \text{the premises hold}\} \\ \cup\ \{&((C; \textbf{while}\ (B)\ C, \sigma_1), (\mathbb{C};; \textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma_1)) \\ &\mid (\sigma_1, \Sigma_1) \in \gamma_1\} \\ \cup\ \{&((\textbf{skip}, \sigma_2), (\textbf{\textit{skip}}, \Sigma_2)) \mid (\sigma_2, \Sigma_2) \in \gamma_2\}. \end{aligned}$$

We prove $S \subseteq F(S)$ where F is defined by the simulation.
From Lemma 12, we can get that if for all $(\sigma, \Sigma) \in \gamma$ we have $((\textbf{while}\ (B)\ \ C, \sigma), (\textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma))$ satisfies the simulation, then for all $(\sigma_1, \Sigma_1) \in \gamma_1$,
$$((C; \textbf{while}\ (B)\ C, \sigma_1), (\mathbb{C};; \textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma_1))$$
satisfies the simulation.
Since $\mathsf{Sta}(\gamma_2, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$, we can prove that if $(\sigma_2, \Sigma_2) \in \gamma_2$, then
$$(\textbf{skip}, \sigma_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma_2} (\textbf{\textit{skip}}, \Sigma_2, \mathbb{R}, \mathbb{G})$$
That is, $((\textbf{skip}, \sigma_2), (\textbf{\textit{skip}}, \Sigma_2))$ satisfies the simulation.
For all $((\textbf{while}\ (B)\ C, \sigma), (\textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma)) \in S$, since $\gamma \subseteq \alpha$, we know $(\sigma, \Sigma) \in \alpha$.

1. If $(\textbf{while}\ (B)\ C, \sigma) \longrightarrow (C', \sigma')$, then according to the operational semantics, we have two possible cases:
   - $B\ \sigma = \textbf{true}$. Thus $C' = C; \textbf{while}\ (B)\ C$ and $\sigma' = \sigma$.
     Since $\mathcal{G}$ contains identity transitions, we know $(\sigma, \sigma') \in \mathcal{G}$.
     From $\zeta \subseteq (B \Leftrightarrow\!\!\Leftrightarrow \mathbb{B})$, we know $\mathbb{B}\ \Sigma = \textbf{true}$. Thus
     $$(\textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma) \longrightarrow (\mathbb{C};; \textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma)$$
     and $(\Sigma, \Sigma) \in \mathbb{G}^*$. Since $(\sigma, \Sigma) \in \gamma_1$, we know
     $$((C; \textbf{while}\ (B)\ C, \sigma_1), (\mathbb{C};; \textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma_1)) \in S.$$
   - $B\ \sigma = \textbf{false}$. Thus $C' = \textbf{skip}$ and $\sigma' = \sigma$.
     Since $\mathcal{G}$ contains identity transitions, we know $(\sigma, \sigma') \in \mathcal{G}$.
     From $\zeta \subseteq (B \Leftrightarrow\!\!\Leftrightarrow \mathbb{B})$, we know $\mathbb{B}\ \Sigma = \textbf{false}$. Thus
     $$(\textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma) \longrightarrow (\textbf{\textit{skip}}, \Sigma), (\Sigma, \Sigma) \in \mathbb{G}^*.$$
     Since $(\sigma, \Sigma) \in \gamma_2$, we know $((\textbf{skip}, \sigma), (\textbf{\textit{skip}}, \Sigma)) \in S$.
2. The case for $(\textbf{while}\ (B)\ C, \sigma) \stackrel{e}{\longrightarrow} (C', \sigma')$ is vacantly true.
3. If $(\sigma, \sigma') \in \mathcal{R}$, $(\Sigma, \Sigma') \in \mathbb{R}^*$ and $(\sigma', \Sigma') \in \alpha$, then from $\mathsf{Sta}(\gamma, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$, we know $(\sigma', \Sigma') \in \gamma$.
   Thus $((\textbf{while}\ (B)\ C, \sigma'), (\textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma')) \in S$.
4. $(\textbf{while}\ (B)\ C)$ is not **skip**, so this case is vacantly true.
5. If $(\textbf{while}\ (B)\ C, \sigma) \longrightarrow \textbf{abort}$, then $B\ \sigma = \perp$.
   Since $\zeta \subseteq (B \Leftrightarrow\!\!\Leftrightarrow \mathbb{B})$ and $(\sigma, \Sigma) \in \zeta$, we know $\mathbb{B}\ \Sigma = \perp$.
   Thus $(\textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \sigma) \longrightarrow \textbf{abort}$.

Then we have $((\textbf{while}\ (B)\ C, \sigma), (\textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma)) \in F(S)$. Thus $(\textbf{while}\ (B)\ C, \sigma)$ and $(\textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \Sigma)$ 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 C_1, C_2, $\mathbb{C}_1$, $\mathbb{C}_2$, σ and Σ, if*

1. *$(C_1, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\gamma_1} (\mathbb{C}_1, \Sigma, \mathbb{R}_1, \mathbb{G}_1)$;*
2. *$(C_2, \sigma, \mathcal{R}_2, \mathcal{G}_2) \preceq_{\alpha;\gamma_2} (\mathbb{C}_2, \Sigma, \mathbb{R}_2, \mathbb{G}_2)$; and*
3. *$\mathcal{G}_1 \subseteq \mathcal{R}_2$; $\mathcal{G}_2 \subseteq \mathcal{R}_1$; $\mathbb{G}_1 \subseteq \mathbb{R}_2$; $\mathbb{G}_2 \subseteq \mathbb{R}_1$,*

then
$$\begin{aligned} &(C_1 \,\|\, C_2, \sigma, \mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2) \preceq_{\alpha;(\gamma_1 \cap \gamma_2)} \\ &(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma, \mathbb{R}_1 \cap \mathbb{R}_2, \mathbb{G}_1 \cup \mathbb{G}_2) \end{aligned}$$

Proof:. By co-induction.
Let
$$S = \{((C_1 \,\|\, C_2, \sigma), (\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma)) \mid \text{the premises hold}\}$$
We prove $S \subseteq F(S)$ where F is defined by the simulation.
For all $((C_1 \,\|\, C_2, \sigma), (\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma)) \in S$,

1. If $(C_1 \,\|\, C_2, \sigma) \longrightarrow (C', \sigma')$, then according to the operational semantics, we have three possible cases:
   - $(C_1, \sigma) \longrightarrow (C_1', \sigma')$ and $C' = C_1' \,\|\, C_2$.
     From the 1st premise, we know
     $$(\sigma, \sigma') \in \mathcal{G}_1 \tag{B.1}$$
     and there exist $\mathbb{C}_1'$ and Σ' such that the followings hold:
     $$(\mathbb{C}_1, \Sigma) \longrightarrow^* (\mathbb{C}_1', \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}_1^* \tag{B.2}$$
     $$(C_1', \sigma', \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\gamma_1} (\mathbb{C}_1', \Sigma', \mathbb{R}_1, \mathbb{G}_1) \tag{B.3}$$
     From (B.1), we know $(\sigma, \sigma') \in \mathcal{G}_1 \cup \mathcal{G}_2$.
     From (B.2), we know $(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma) \longrightarrow^* (\mathbb{C}_1' \,|||\, \mathbb{C}_2, \Sigma')$ and $(\Sigma, \Sigma') \in (\mathbb{G}_1 \cup \mathbb{G}_2)^*$.
     Since $(\sigma, \sigma') \in \mathcal{G}_1 \subseteq \mathcal{R}_2$, $(\Sigma, \Sigma') \in \mathbb{G}_1^* \subseteq \mathbb{R}_2^*$ and $(\sigma', \Sigma') \in \alpha$, from the 2nd premise, we know
     $$(C_2', \sigma', \mathcal{R}_2, \mathcal{G}_2) \preceq_{\alpha;\gamma_2} (\mathbb{C}_2', \Sigma', \mathbb{R}_2, \mathbb{G}_2) \tag{B.4}$$
     From (B.3) and (B.4), we know

$$((C_1' \,\|\, C_2, \sigma'), (\mathbb{C}_1' \,|||\, \mathbb{C}_2, \Sigma')) \in S.$$

- $(C_2, \sigma) \longrightarrow (C_2', \sigma')$ and $C' = C_1 \,\|\, C_2'$.
  Similar to the previous case.
- $C_1 = \textbf{skip}$, $C_2 = \textbf{skip}$ and $(\textbf{skip} \,\|\, \textbf{skip}, \sigma) \longrightarrow (\textbf{skip}, \sigma)$.
  From the 1st premise, we know

$$(\mathbb{C}_1, \Sigma) \longrightarrow^* (\textbf{\textit{skip}}, \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}_1^* \tag{B.5}$$

$$(\sigma, \Sigma') \in \gamma_1 \tag{B.6}$$

From (B.5), we know

$$(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma) \longrightarrow^* (\textbf{\textit{skip}} \,|||\, \mathbb{C}_2, \Sigma'), (\Sigma, \Sigma') \in (\mathbb{G}_1 \cup \mathbb{G}_2)^* \tag{B.7}$$

and $(\Sigma, \Sigma') \in \mathbb{R}_2^*$. From (B.6), we know $(\sigma, \Sigma') \in \alpha$. Then from the 2nd premise, we know

$$(\textbf{skip}, \sigma, \mathcal{R}_2, \mathcal{G}_2) \preceq_{\alpha;\gamma_2} (\mathbb{C}_2, \Sigma', \mathbb{R}_2, \mathbb{G}_2) \tag{B.8}$$

Thus $(\mathbb{C}_2, \Sigma') \longrightarrow^* (\textbf{\textit{skip}}, \Sigma'')$, $(\Sigma', \Sigma'') \in \mathbb{G}_2^*$ and

$$(\sigma, \Sigma'') \in \gamma_2 \tag{B.9}$$

Then from (B.7), we know $(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma) \longrightarrow^* (\textbf{\textit{skip}}, \Sigma'')$ and $(\Sigma, \Sigma'') \in (\mathbb{G}_1 \cup \mathbb{G}_2)^*$.
On the other hand, $(\Sigma', \Sigma'') \in \mathbb{G}_2^* \subseteq \mathbb{R}_1^*$. From (B.6) and $\mathsf{Sta}(\gamma_1, \langle \mathcal{R}_1, \mathbb{R}_1^* \rangle_\alpha)$, we know

$$(\sigma, \Sigma'') \in \gamma_1 \tag{B.10}$$

From (B.9) and (B.10), we know $(\sigma, \Sigma'') \in \gamma_1 \cap \gamma_2$.

2. If $(C_1 \,\|\, C_2, \sigma) \stackrel{e}{\longrightarrow} (C', \sigma')$, the proof is similar to the previous case.
3. If $(\sigma, \sigma') \in \mathcal{R}_1 \cap \mathcal{R}_2$, $(\Sigma, \Sigma') \in (\mathbb{R}_1 \cap \mathbb{R}_2)^* \subseteq \mathbb{R}_1^* \cap \mathbb{R}_2^*$ and $(\sigma', \Sigma') \in \alpha$, then

$$(C_1, \sigma', \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\gamma_1} (\mathbb{C}_1, \Sigma', \mathbb{R}_1, \mathbb{G}_1)$$

$$(C_2, \sigma', \mathcal{R}_2, \mathcal{G}_2) \preceq_{\alpha;\gamma_2} (\mathbb{C}_2, \Sigma', \mathbb{R}_2, \mathbb{G}_2)$$

   Thus $((C_1 \,\|\, C_2, \sigma'), (\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma')) \in S$.
4. $C_1 \,\|\, C_2 \neq \textbf{skip}$. This case is vacantly true.
5. If $(C_1 \,\|\, C_2, \sigma) \longrightarrow \textbf{abort}$, then $(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma) \longrightarrow^* \textbf{abort}$ is immediate from the premises.

Then we have $((C_1 \,\|\, C_2, \sigma), (\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma)) \in F(S)$. Thus $(C_1 \,\|\, C_2, \sigma)$ and $(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma)$ satisfy the largest simulation RGSim. □

Thus we can conclude soundness of the PAR rule.

B.5 Soundness of the STREN-α Rule

Lemma 16. *For all C, $\mathbb{C}$, σ and Σ, if*

1. $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$*;*
2. $(\sigma, \Sigma) \in \alpha'$*,*
3. $\alpha' \subseteq \alpha$*; and*
4. $\mathsf{Sta}(\alpha', \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha)$*,*

then $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha';\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$.

Proof:. By co-induction.
Let $S = \{((C, \sigma), (\mathbb{C}, \Sigma)) \mid \text{the premises hold}\}$. We prove $S \subseteq F(S)$ where F is defined by the simulation.
For all $((C, \sigma), (\mathbb{C}, \Sigma)) \in S$,

1. If $(C, \sigma) \longrightarrow (C', \sigma')$, then from the 1st premise, we know $(\sigma, \sigma') \in \mathcal{G}$ and there exist $\mathbb{C}'$ and Σ' such that the followings hold:

$$(\mathbb{C}, \Sigma) \longrightarrow^* (\mathbb{C}', \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}^*$$

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

   We know $(\sigma', \Sigma') \in \alpha$. Then by applying the 4th premise, we can get $(\sigma', \Sigma') \in \alpha'$. Thus we know $((C', \sigma'), (\mathbb{C}', \Sigma')) \in S$.
2. If $(C, \sigma) \stackrel{e}{\longrightarrow} (C', \sigma')$, the proof is similar to the previous case.
3. If $(\sigma, \sigma') \in \mathcal{R}$, $(\Sigma, \Sigma') \in \mathbb{R}^*$ and $(\sigma', \Sigma') \in \alpha'$, then we know $(\sigma', \Sigma') \in \alpha$. Using the 1st premise, we have

$$(C, \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma', \mathbb{R}, \mathbb{G}).$$

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

$$(\mathbb{C}, \Sigma) \longrightarrow^* (\textbf{\textit{skip}}, \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}^*, (\sigma, \Sigma') \in \gamma$$

5. If $(C, \sigma) \longrightarrow \textbf{abort}$, then $(\mathbb{C}, \Sigma) \longrightarrow^* \textbf{abort}$ is immediate from the 1st premise.

Then we have $((C, \sigma), (\mathbb{C}, \Sigma)) \in F(S)$. Thus (C, σ) and $(\mathbb{C}, \Sigma)$ satisfy the largest simulation RGSim. □

Since $\zeta \subseteq \alpha'$, we can conclude soundness of the STREN-α rule.

B.6 Soundness of the WEAKEN-α Rule

Lemma 17. *For all C, $\mathbb{C}$, σ and Σ, if*

1. $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$*;*
2. $\alpha \subseteq \alpha'$*; and*
3. $\mathsf{Sta}(\alpha, \langle \mathcal{R}, \mathbb{R}^* \rangle_{\alpha'})$*,*

then $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha';\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$.

Proof:. By co-induction.
Let $S = \{((C, \sigma), (\mathbb{C}, \Sigma)) \mid \text{the premises hold}\}$. We prove $S \subseteq F(S)$ where F is defined by the simulation.
For all $((C, \sigma), (\mathbb{C}, \Sigma)) \in S$, we know $(\sigma, \Sigma) \in \alpha \subseteq \alpha'$.

1. If $(C, \sigma) \longrightarrow (C', \sigma')$, then from the 1st premise, we know $(\sigma, \sigma') \in \mathcal{G}$ and there exist $\mathbb{C}'$ and Σ' such that the followings hold:

$$(\mathbb{C}, \Sigma) \longrightarrow^* (\mathbb{C}', \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}^*$$

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

   Thus we know $((C', \sigma'), (\mathbb{C}', \Sigma')) \in S$.
2. If $(C, \sigma) \stackrel{e}{\longrightarrow} (C', \sigma')$, the proof is similar to the previous case.
3. If $(\sigma, \sigma') \in \mathcal{R}$, $(\Sigma, \Sigma') \in \mathbb{R}^*$ and $(\sigma', \Sigma') \in \alpha'$, then from the 3rd premise, we know $(\sigma', \Sigma') \in \alpha$. Using the 1st premise, we have

$$(C, \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma', \mathbb{R}, \mathbb{G}).$$

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

$$(\mathbb{C}, \Sigma) \longrightarrow^* (\textbf{\textit{skip}}, \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}^*, (\sigma, \Sigma') \in \gamma$$

5. If $(C, \sigma) \longrightarrow \textbf{abort}$, then $(\mathbb{C}, \Sigma) \longrightarrow^* \textbf{abort}$ is immediate from the 1st premise.

Then we have $((C, \sigma), (\mathbb{C}, \Sigma)) \in F(S)$. Thus (C, σ) and $(\mathbb{C}, \Sigma)$ satisfy the largest simulation RGSim. □

Thus we can conclude soundness of the WEAKEN-α rule.

B.7 Soundness of the FRAME Rule

Lemma 18. *For all C, $\mathbb{C}$, σ and Σ, if*

1. $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$;
2. $(\sigma, \Sigma) \in \alpha \uplus \eta$,
3. $\eta \subseteq \beta$;
4. $\eta \# \{\gamma, \alpha\}$;
5. $\mathsf{Intuit}(\{\alpha, \gamma, \beta, \eta, \mathcal{R}, \mathbb{R}, \mathcal{R}', \mathbb{R}'\})$; *and*
6. $\mathsf{Sta}(\eta, \{\langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha, \langle \mathcal{R}', \mathbb{R}'^* \rangle_\beta\})$,

then $(C, \sigma, \mathcal{R} \uplus \mathcal{R}', \mathcal{G} \uplus \mathcal{G}') \preceq_{\alpha\uplus\beta;\gamma\uplus\eta} (\mathbb{C}, \Sigma, \mathbb{R} \uplus \mathbb{R}', \mathbb{G} \uplus \mathbb{G}')$.

Proof:. By co-induction.
Let $S = \{((C, \sigma), (\mathbb{C}, \Sigma)) \mid \text{the premises hold}\}$. We prove $S \subseteq F(S)$ where F is defined by the simulation.
For all $((C, \sigma), (\mathbb{C}, \Sigma)) \in S$, from the 2nd and 3rd premises, we know $(\sigma, \Sigma) \in \alpha \uplus \beta$.

1. If $(C, \sigma) \longrightarrow (C', \sigma')$, then from the 1st premise, we know $(\sigma, \sigma') \in \mathcal{G}$ and there exist $\mathbb{C}'$ and Σ' such that the followings hold:
$$(\mathbb{C}, \Sigma) \longrightarrow^* (\mathbb{C}', \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}^*$$
$$(C', \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}', \Sigma', \mathbb{R}, \mathbb{G})$$
Since $(\phi, \phi) \in \mathsf{Id} \subseteq \mathcal{G}'$, we know
$$\mathcal{G} \subseteq (\mathcal{G} \uplus \mathcal{G}').$$
Thus $(\sigma, \sigma') \in \mathcal{G} \uplus \mathcal{G}'$. Similarly, $(\Sigma, \Sigma') \in (\mathbb{G} \uplus \mathbb{G}')^*$.
Also we know $(\sigma', \Sigma') \in \alpha$. From $\mathsf{Intuit}(\eta)$, we know
$$(\sigma, \Sigma) \in \alpha \uplus \eta \subseteq \eta.$$
From $\mathsf{Sta}(\eta, \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha)$, we know $(\sigma', \Sigma') \in \eta$. Since $\eta \# \alpha$, we know
$$(\sigma', \Sigma') \in \alpha \uplus \eta.$$
Thus we know $((C', \sigma'), (\mathbb{C}', \Sigma')) \in S$.
2. If $(C, \sigma) \xrightarrow{e} (C', \sigma')$, the proof is similar to the previous case.
3. If $(\sigma, \sigma') \in \mathcal{R} \uplus \mathcal{R}'$, $(\Sigma, \Sigma') \in (\mathbb{R} \uplus \mathbb{R}')^*$ and $(\sigma', \Sigma') \in \alpha \uplus \beta$, then from $\mathsf{Intuit}(\{\mathcal{R}, \mathbb{R}, \alpha\})$, we have
$$(\sigma, \sigma') \in \mathcal{R}, (\Sigma, \Sigma') \in \mathbb{R}^*, (\sigma', \Sigma') \in \alpha$$
Then from the 1st premise, we have
$$(C, \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma', \mathbb{R}, \mathbb{G}).$$
From $\mathsf{Intuit}(\eta)$, we know $(\sigma, \Sigma) \in \alpha \uplus \eta \subseteq \eta$. On the other hand, from $\mathsf{Intuit}(\{\mathcal{R}', \mathbb{R}', \beta\})$, we have
$$(\sigma, \sigma') \in \mathcal{R}', (\Sigma, \Sigma') \in \mathbb{R}'^*, (\sigma', \Sigma') \in \beta$$
From $\mathsf{Sta}(\eta, \langle \mathcal{R}', \mathbb{R}'^* \rangle_\beta)$, we know $(\sigma', \Sigma') \in \eta$. Since $\eta \# \alpha$, we know
$$(\sigma', \Sigma') \in \alpha \uplus \eta.$$
Thus $((C, \sigma'), (\mathbb{C}, \Sigma')) \in S$.
4. If $C = \textbf{skip}$, then from the 1st premise, we know there exists Σ' such that the followings hold:
$$(\mathbb{C}, \Sigma) \longrightarrow^* (\textbf{\textit{skip}}, \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}^*, (\sigma, \Sigma') \in \gamma$$
We have $(\Sigma, \Sigma') \in (\mathbb{G} \uplus \mathbb{G}')^*$ and $(\sigma, \Sigma') \in \alpha$.
From $\mathsf{Intuit}(\eta)$, we know $(\sigma, \Sigma) \in \alpha \uplus \eta \subseteq \eta$.
From $\mathsf{Sta}(\eta, \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha)$, we know $(\sigma, \Sigma') \in \eta$.
Since $\eta \# \gamma$, we know $(\sigma, \Sigma') \in \gamma \uplus \eta$.
5. If $(C, \sigma) \longrightarrow \textbf{abort}$, then $(\mathbb{C}, \Sigma) \longrightarrow^* \textbf{abort}$ is immediate from the 1st premise.

Then we have $((C, \sigma), (\mathbb{C}, \Sigma)) \in F(S)$. Thus (C, σ) and $(\mathbb{C}, \Sigma)$ satisfy the largest simulation RGSim. ☐

Since $(\zeta \cup \gamma) \subseteq \alpha$ and $\eta \subseteq \beta$, we have
$$(\zeta \uplus \eta) \subseteq (\alpha \uplus \beta),\ (\gamma \uplus \eta) \subseteq (\alpha \uplus \beta),\ (\zeta \uplus \eta) \subseteq (\alpha \uplus \eta).$$
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 σ_1 and σ_2, if*

1. $\zeta = (\zeta \cap (\textbf{true} \Lbag \neg B))$;
2. $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathcal{R}'^* \rangle_\alpha)$;
3. $(\sigma_1, \sigma_2) \in \zeta \subseteq \alpha$,

then $(\textbf{skip}, \sigma_1, \mathcal{R}, \mathsf{Id}) \preceq_{\alpha;\zeta} (\textbf{while}\ (B)\{C\}, \sigma_2, \mathcal{R}', \mathsf{Id})$.

Proof:. By co-induction.
Since $B\, \sigma_2 = \textbf{false}$, we know $(\textbf{while}\ (B)\{C\}, \sigma_2) \longrightarrow (\textbf{skip}, \sigma_2)$ and $(\sigma_2, \sigma_2) \in \mathsf{Id}$.
The case for the environments' transitions is immediate from $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathcal{R}'^* \rangle_\alpha)$. ☐

Lemma 20 (Dead Code Elimination). *For all σ_1 and σ_2, if*

1. $(\textbf{skip}, \sigma_1, \mathsf{Id}, \mathsf{Id}) \preceq_{\alpha;\gamma} (C, \sigma_2, \mathsf{Id}, \mathcal{G})$;
2. $\mathsf{Sta}(\{\zeta, \gamma\}, \langle \mathcal{R}, \mathcal{R}'^* \rangle_\alpha)$;
3. $(\sigma_1, \sigma_2) \in \zeta$,

then $(\textbf{skip}, \sigma_1, \mathcal{R}, \mathsf{Id}) \preceq_{\alpha;\gamma} (C, \sigma_2, \mathcal{R}', \mathcal{G})$.

Proof:. By co-induction.
If $C = \textbf{skip}$, then $(\sigma_1, \sigma_2) \in \gamma$. From $\mathsf{Sta}(\gamma, \langle \mathcal{R}, \mathcal{R}'^* \rangle_\alpha)$, we can prove the conclusion.
Otherwise, there exists σ_2' such that $(C, \sigma_2) \longrightarrow^* (\textbf{skip}, \sigma_2')$, $(\sigma_2, \sigma_2') \in \mathcal{G}^*$ and $(\sigma_1, \sigma_2') \in \gamma$.
Finally, the case for the environments' transitions is immediate from $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathcal{R}'^* \rangle_\alpha)$. ☐

Lemma 21 (Redundancy Introduction). *For all σ_1 and σ_2, if*

1. $(c, \sigma_1, \mathsf{Id}, \mathcal{G}) \preceq_{\alpha;\gamma} (\textbf{skip}, \sigma_2, \mathsf{Id}, \mathsf{Id})$;
2. $\mathsf{Sta}(\{\zeta, \gamma\}, \langle \mathcal{R}, \mathcal{R}'^* \rangle_\alpha)$;
3. $(\sigma_1, \sigma_2) \in \zeta$,

then $(c, \sigma_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\textbf{skip}, \sigma_2, \mathcal{R}', \mathsf{Id})$.

Proof:. By co-induction.
Since c is an instruction, by its operational semantics, we only have four cases:

1. If $(c, \sigma_1) \longrightarrow (\textbf{skip}, \sigma_1')$, then $(\sigma_1, \sigma_1') \in \mathcal{G}$ and $(\textbf{skip}, \sigma_2) \longrightarrow^0 (\textbf{skip}, \sigma_2)$, $(\sigma_2, \sigma_2) \in \mathsf{Id}$. From
$$(\textbf{skip}, \sigma_1', \mathsf{Id}, \mathcal{G}) \preceq_{\alpha;\gamma} (\textbf{skip}, \sigma_2, \mathsf{Id}, \mathsf{Id}),$$
we know $(\sigma_1', \sigma_2) \in \gamma$. Since $\mathsf{Sta}(\gamma, \langle \mathcal{R}, \mathcal{R}'^* \rangle_\alpha)$, it's not difficult to prove
$$(\textbf{skip}, \sigma_1', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\textbf{skip}, \sigma_2, \mathcal{R}', \mathsf{Id}).$$
2. $(c, \sigma_1) \xrightarrow{e} (\textbf{skip}, \sigma_1')$ is impossible.

3. If $(c, \sigma_1) \longrightarrow (c, \sigma_1)$, trivial.

4. $(c, \sigma_1) \longrightarrow$ **abort** is impossible.

Finally, the case for the environments' transitions is immediate from $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathcal{R}'^* \rangle_\alpha)$. $\Box$

C. Proof of Theorem 8

We prove for any high-level mutator program $\mathbb{W}$, $\mathbf{T}(\mathbb{W}) \sqsubseteq_{\mathbf{T}} \mathbb{W}$. Suppose $\mathbb{W} = \mathsf{t}_{\text{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots |||\, \mathsf{t}_n.\mathbb{C}_n$.

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 α, ζ and γ such that:

$$(\mathsf{t}_{\text{gc}}.C_{\text{gc}} \parallel \mathsf{t}_1.\mathbf{T}(\mathbb{C}_1) \parallel \ldots \parallel \mathsf{t}_n.\mathbf{T}(\mathbb{C}_n), \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\zeta \ltimes \gamma}$$
$$(\mathsf{t}_{\text{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots |||\, \mathsf{t}_n.\mathbb{C}_n, \mathsf{Id}, \mathsf{True})$$

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. $(C_{\text{gc}}, \mathcal{R}_{\text{gc}}, \mathcal{G}_{\text{gc}}) \preceq_{\alpha;\zeta_{\text{gc}} \ltimes \zeta_{\text{gc}}} (\mathit{AbsGC}, \mathsf{True}, \mathsf{AbsGCStep})$*;*
2. $\forall \mathsf{c}.\ (\mathsf{t}.\mathbf{T}(\mathsf{c}), \mathcal{R}(\mathsf{t}), \mathcal{G}(\mathsf{t}.\mathbf{T}(\mathsf{c}))) \preceq_{\alpha;\zeta(\mathsf{t}) \ltimes \zeta(\mathsf{t})} (\mathsf{t.c}, \mathbb{R}(\mathsf{t}), \mathbb{G}(\mathsf{t.c}))$*;*
3. $\mathcal{G}(\mathsf{t}) = \bigcup_{\mathsf{c}} \mathcal{G}(\mathsf{t}.\mathbf{T}(\mathsf{c}))$*;* $\mathbb{G}(\mathsf{t}) = \bigcup_{\mathsf{c}} \mathbb{G}(\mathsf{t.c})$*;*
   $\mathcal{R}(\mathsf{t}) = \mathcal{G}_{\text{gc}} \cup \bigcup_{\mathsf{t}' \neq \mathsf{t}} \mathcal{G}(\mathsf{t}')$*;* $\mathbb{R}(\mathsf{t}) = \mathsf{AbsGCStep} \cup \bigcup_{\mathsf{t}' \neq \mathsf{t}} \mathbb{G}(\mathsf{t}')$*;*
   $\mathcal{R}_{\text{gc}} = \bigcup_{\mathsf{t}} \mathcal{G}(\mathsf{t})$*;* $\zeta = \zeta_{\text{gc}} \cap \bigcap_{\mathsf{t}} \zeta(\mathsf{t})$*;*
4. $\forall \mathsf{t}, \mathbb{B}.\ \zeta(\mathsf{t}) \subseteq (\mathbf{T}(\mathbb{B}) \Leftrightarrow \mathbb{B})$,

then

$$(\mathsf{t}_{\text{gc}}.C_{\text{gc}} \parallel \mathsf{t}_1.\mathbf{T}(\mathbb{C}_1) \parallel \ldots \parallel \mathsf{t}_n.\mathbf{T}(\mathbb{C}_n), \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\zeta \ltimes \zeta}$$
$$(\mathsf{t}_{\text{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots |||\, \mathsf{t}_n.\mathbb{C}_n, \mathsf{Id}, \mathsf{True}).$$

Proof:. By induction over the high-level program structure. $\Box$

If $\mathsf{InitRel}_{\mathbf{T}}(\zeta(\mathsf{t}))$ and $\mathsf{InitRel}_{\mathbf{T}}(\zeta_{\text{gc}})$, then $\mathsf{InitRel}_{\mathbf{T}}(\zeta)$. Thus from soundness of RGSim (Corollary 6), we can conclude $\mathsf{Correct}(\mathbf{T})$.

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

$$\zeta_{\text{gc}} \triangleq \{(\sigma, \Sigma) \mid \sigma = \mathbf{T}(\Sigma)\}.$$

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

Lemma 23. *If*

1. *(Verification of the GC code)*
   $\mathcal{R}_{\text{gc}}; \mathcal{G}_{\text{gc}} \vdash \{p_{\text{gc}}\} C_{\text{gc}} \{\textbf{false}\}$*;*
2. *(Side Conditions)*
   $\mathcal{G}_{\text{gc}} \circ \alpha^{-1} \subseteq \alpha^{-1} \circ (\mathsf{AbsGCStep})^*$*;*
   $\forall \sigma, \Sigma.\ (\sigma, \Sigma) \in \zeta_{\text{gc}} \Longrightarrow p_{\text{gc}}\ \sigma$*;* $\zeta_{\text{gc}} \subseteq \alpha$*,*

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

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

Definition 24 (Non-Interference).
$(C, \sigma, \mathcal{R})\ \mathbf{guarantees}_0\ \mathcal{G}$ always holds;
$(C, \sigma, \mathcal{R})\ \mathbf{guarantees}_{n+1}\ \mathcal{G}$ holds iff
$\neg((C, \sigma) \longrightarrow \textbf{abort})$, $\neg \exists C', \sigma', e.((C, \sigma) \stackrel{e}{\longrightarrow} (C', \sigma'))$, and

1. for all σ', if $(\sigma, \sigma') \in \mathcal{R}$, then $(C, \sigma', \mathcal{R})\ \mathbf{guarantees}_n\ \mathcal{G}$;
2. for all σ', if $(C, \sigma) \longrightarrow (C', \sigma')$, then $(\sigma, \sigma') \in \mathcal{G}$ and $(C', \sigma', \mathcal{R})\ \mathbf{guarantees}_n\ \mathcal{G}$.

Then, $(C, \sigma, \mathcal{R})\ \mathbf{guarantees}\ \mathcal{G} \triangleq \forall k.\ (C, \sigma, \mathcal{R})\ \mathbf{guarantees}_k\ \mathcal{G}$.

Definition 25 (Semantics). $\mathcal{R}; \mathcal{G} \models \{p\} C \{q\}$ iff, for any σ such that $p\ \sigma$, the following are true:

1. if $(C, \sigma) \stackrel{\mathcal{R}}{\longmapsto}{}^* (\textbf{skip}, \sigma')$, then $q\ \sigma'$;
2. $(C, \sigma, \mathcal{R})\ \mathbf{guarantees}\ \mathcal{G}$,

where $(C, \sigma) \stackrel{\mathcal{R}}{\longmapsto} (C', \sigma')$ is defined by:

$$\frac{(C, \sigma) \longrightarrow (C', \sigma')}{(C, \sigma) \stackrel{\mathcal{R}}{\longmapsto} (C', \sigma')} \qquad \frac{(\sigma, \sigma') \in \mathcal{R}}{(C, \sigma) \stackrel{\mathcal{R}}{\longmapsto} (C, \sigma')}$$

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

Lemma 26. *For all C, $\mathcal{R}$, $\mathcal{G}$, σ and Σ, if*

1. $(C, \sigma, \mathcal{R})\ \mathbf{guarantees}\ \mathcal{G}$*;*
2. $\mathcal{G} \circ \alpha^{-1} \subseteq \alpha^{-1} \circ (\mathsf{AbsGCStep})^*$*;*
3. $(\sigma, \Sigma) \in \alpha$*; and*
4. $\neg \exists \sigma'.((C, \sigma) \stackrel{\mathcal{R}}{\longmapsto}{}^* (\textbf{skip}, \sigma'))$,

then $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathit{AbsGC}, \Sigma, \mathsf{True}, \mathsf{AbsGCStep})$.

Proof:. By co-induction.
Let $S \triangleq \{(C, \sigma, \Sigma) \mid$ the premises hold$\}$. We prove $S \subseteq F(S)$ where F is defined by the simulation. For all $(C, \sigma, \Sigma) \in S$, we only need to consider two cases:

1. If $(C, \sigma) \longrightarrow (C', \sigma')$, then
   $(\sigma, \sigma') \in \mathcal{G}$ and $(C', \sigma', \mathcal{R})\ \mathbf{guarantees}\ \mathcal{G}$. By the premise 2 and the operational semantics of *AbsGC*, we know there must exist Σ' such that $(\mathit{AbsGC}, \Sigma) \longrightarrow^* (\mathit{AbsGC}, \Sigma')$, $(\Sigma, \Sigma') \in (\mathsf{AbsGCStep})^*$ and $(\sigma', \Sigma') \in \alpha$.
   Thus $(C', \sigma', \Sigma') \in S$.
2. If $(\sigma, \sigma') \in \mathcal{R}$, $(\Sigma, \Sigma') \in \mathsf{True}$ and $(\sigma', \Sigma') \in \alpha$, then we have $(C, \sigma', \mathcal{R})\ \mathbf{guarantees}\ \mathcal{G}$ and $(\sigma', \Sigma') \in \alpha$.
   Thus $(C, \sigma', \Sigma') \in S$.

Then we have $(C, \sigma, \Sigma) \in F(S)$. Thus (C, σ) and (AbsGC, Σ) satisfy the largest simulation RGSim. $\Box$

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_l(x)` synchronizes reading and writing `x` by a global lock `l`. We can view the CAS instruction $x := \textbf{cas}(\&y, E_1, E_2)$ as a syntax sugar of $\langle \textbf{if}\ (y = E_1)\ \{y := E_2; x := 1\}\ \textbf{else}\ x := 0 \rangle$.

To observe the value of `x`, we use the standard $\textbf{print}(E)$ operation which will produce an external event $\textbf{out}(n)$ if E evaluates to n. The source `PRT(x)` which directly prints out the value of `x` is transformed to two targets: `prt(x)` performs print in a fine-grained manner; while `prt_l(x)` uses the global lock `l` to protect accesses of `x`.

```
INC(x):   atom{ x := x+1; }
```

(a) Source Code

```
inc(x):
     local d, t;
0    d := 0;
1    while (d = 0) {
2      <t := x;>
3      d := cas(&x,t,t+1);
     }
```

```
inc_l(x):
0    lock(l);
1    x := x-1;
2    x := x+2;
3    unlock(l);
```

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

Figure 20. Incrementing a Shared Variable

```
PRT(x)    ≜   print(x);
prt(x)    ≜   local t; <t := x;> print(t);
prt_l(x)  ≜   lock(l); print(x); unlock(l);
```

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 α relation between low-level and high-level states, where only the values of x are concerned:

$$\alpha \quad \triangleq \quad \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}.$$

Both inc(x) and 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.

$$\begin{array}{lcl}
\mathcal{R} & \triangleq & \{(\sigma, \sigma') \mid \sigma'(\mathtt{t}) = \sigma(\mathtt{t}) \wedge \sigma'(\mathtt{d}) = \sigma(\mathtt{d})\} \\
\mathcal{G} & \triangleq & \{(\sigma, \sigma') \mid \sigma' = \sigma\{\mathtt{t} \leadsto _, \mathtt{d} \leadsto _\} \\
 & & \quad \vee\, \sigma' = \sigma\{\mathtt{x} \leadsto \sigma(\mathtt{x}) + 1, \mathtt{t} \leadsto _, \mathtt{d} \leadsto _\}\} \\
\mathbb{R} & \triangleq & \{(\Sigma, \Sigma') \mid \Sigma, \Sigma' \in \mathit{HState}\} \\
\mathbb{G} & \triangleq & \{(\Sigma, \Sigma') \mid \Sigma' = \Sigma \vee \Sigma' = \Sigma\{\mathtt{x} \leadsto \Sigma(\mathtt{x}) + 1\}\}
\end{array}$$

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

$$(\mathtt{inc(x)}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha \ltimes \alpha} (\mathtt{INC(x)}, \mathbb{R}, \mathbb{G})$$

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

$$(\mathtt{inc(x)}, \mathcal{R}, \mathcal{G}) \simeq_{\alpha;\alpha \ltimes \alpha} (\mathtt{INC(x)}, \mathbb{R}, \mathbb{G}). \qquad \text{(D.1)}$$

In other words, inc(x) and INC(x) behave just the same.

Similarly, we can prove that prt(x) and PRT(x) are equivalent:

$$(\mathtt{prt(x)}, \mathcal{R}, \mathcal{G}) \simeq_{\alpha;\alpha \ltimes \alpha} (\mathtt{PRT(x)}, \mathbb{R}, \mathbb{G}). \qquad \text{(D.2)}$$

As a simple illustration, we go on to show that the non-blocking 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 $(\mathtt{inc(x)} \parallel \mathtt{inc(x)}); \mathtt{prt(x)}$ and $(\mathtt{INC(x)} \parallel \mathtt{INC(x)}); \mathtt{PRT(x)}$ have the same observable events when the initial values of x are the same.

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

$$\begin{array}{l}
((\mathtt{inc(x)} \parallel \mathtt{inc(x)}); \mathtt{prt(x)}, \mathcal{R}, \mathcal{G}) \simeq_{\alpha;\alpha \ltimes \alpha} \\
((\mathtt{INC(x)} \parallel \mathtt{INC(x)}); \mathtt{PRT(x)}, \mathbb{R}, \mathbb{G}).
\end{array}$$

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

$$(\mathtt{inc(x)} \parallel \mathtt{inc(x)}); \mathtt{prt(x)} \approx_{\mathbf{T}} (\mathtt{INC(x)} \parallel \mathtt{INC(x)}); \mathtt{PRT(x)},$$

for any $\mathbf{T}$ that respects α. 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 inc(x) and INC(x) and prove their relations by co-induction. Here we use $\mathtt{inc}^l\mathtt{(x)}$ to denote the code from line l to the end of the program (loops might be unrolled if needed), *e.g.*, $\mathtt{inc}^3\mathtt{(x)}$ 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 **skip**; C (for some C) and even do not distinguish C and **skip**; C in the proofs.

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

1. $(\mathtt{inc(x)}, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\mathtt{INC(x)}, \Sigma, \mathbb{R}, \mathbb{G})$*;*
2. *if* $\sigma(\mathtt{d}) = 0$*, then* $(\mathtt{inc}^1\mathtt{(x)}, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\mathtt{INC(x)}, \Sigma, \mathbb{R}, \mathbb{G})$*;*
3. *if* $\sigma(\mathtt{d}) = 1$*, then* $(\mathtt{inc}^1\mathtt{(x)}, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\mathbf{skip}, \Sigma, \mathbb{R}, \mathcal{G})$*;*
4. $(\mathtt{inc}^2\mathtt{(x)}, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\mathtt{INC(x)}, \Sigma, \mathbb{R}, \mathbb{G})$*;*
5. $(\mathtt{inc}^3\mathtt{(x)}, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\mathtt{INC(x)}, \Sigma, \mathbb{R}, \mathbb{G})$*;*
6. $(\mathbf{skip}, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\mathbf{skip}, \Sigma, \mathbb{R}, \mathbb{G})$.

Proof:. For each case, by co-induction.
Case: The environments are executed. The proof is trivial since $\mathcal{R}$ does not update d.
Case: The non-blocking counter code goes one step.

1. If $(\mathtt{inc(x)}, \sigma) \longrightarrow (\mathtt{inc}^1\mathtt{(x)}, \sigma')$, then $\sigma'(\mathtt{d}) = 0$ and $\sigma'(\mathtt{x}) = \sigma(\mathtt{x})$. Correspondingly, INC(x) does not go any step:
$$(\mathtt{INC(x)}, \Sigma) \longrightarrow^0 (\mathtt{INC(x)}, \Sigma),\ (\Sigma, \Sigma) \in \mathbb{G}^*.$$
From the premise 2, we know
$$(\mathtt{inc}^1\mathtt{(x)}, \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\mathtt{INC(x)}, \Sigma, \mathbb{R}, \mathbb{G}).$$
2. If $\sigma(\mathtt{d}) = 0$ and $(\mathtt{inc}^1\mathtt{(x)}, \sigma) \longrightarrow (\mathtt{inc}^2\mathtt{(x)}, \sigma')$, then $\sigma' = \sigma$. Correspondingly:
$$(\mathtt{INC(x)}, \Sigma) \longrightarrow^0 (\mathtt{INC(x)}, \Sigma),\ (\Sigma, \Sigma) \in \mathbb{G}^*.$$
From the premise 4, we know
$$(\mathtt{inc}^2\mathtt{(x)}, \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\mathtt{INC(x)}, \Sigma, \mathbb{R}, \mathbb{G}).$$
3. If $\sigma(\mathtt{d}) = 1$ and $(\mathtt{inc}^1\mathtt{(x)}, \sigma) \longrightarrow (\mathbf{skip}, \sigma')$, then $\sigma' = \sigma$. Correspondingly:
$$(\mathbf{skip}, \Sigma) \longrightarrow^0 (\mathbf{skip}, \Sigma),\ (\Sigma, \Sigma) \in \mathbb{G}^*.$$
From the premise 6, we know
$$(\mathbf{skip}, \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\mathbf{skip}, \Sigma, \mathbb{R}, \mathbb{G}).$$
4. If $(\mathtt{inc}^2\mathtt{(x)}, \sigma) \longrightarrow (\mathtt{inc}^3\mathtt{(x)}, \sigma')$, then $\sigma(\mathtt{x}) = \sigma'(\mathtt{x})$. Correspondingly:
$$(\mathtt{INC(x)}, \Sigma) \longrightarrow^0 (\mathtt{INC(x)}, \Sigma),\ (\Sigma, \Sigma) \in \mathbb{G}^*.$$
From the premise 5, we know
$$(\mathtt{inc}^3\mathtt{(x)}, \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\mathtt{INC(x)}, \Sigma, \mathbb{R}, \mathbb{G}).$$
5. If $(\mathtt{inc}^3\mathtt{(x)}, \sigma) \longrightarrow (\mathtt{inc}^1\mathtt{(x)}, \sigma')$, then
   (a) if $\sigma(\mathtt{x}) = \sigma(\mathtt{t})$, we have
   $$\sigma' = \sigma\{\mathtt{x} \leadsto \sigma(\mathtt{x}) + 1, \mathtt{d} \leadsto 1\}.$$
   Thus $(\sigma, \sigma') \in \mathcal{G}$. Correspondingly:
   $$(\mathtt{INC(x)}, \Sigma) \longrightarrow (\mathbf{skip}, \Sigma'),\ \Sigma' = \Sigma\{\mathtt{x} \leadsto \Sigma(\mathtt{x}) + 1\}.$$

Thus we have $(\Sigma, \Sigma') \in \mathbb{G}^*$ and $(\sigma', \Sigma') \in \alpha$. From the premise 3, we know

$$(\texttt{inc}^1(\texttt{x}), \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma', \mathbb{R}, \mathbb{G}).$$

(b) if $\sigma(\texttt{x}) \neq \sigma(\texttt{t})$, then $\sigma'(\texttt{d}) = 0$ and $\sigma'(\texttt{x}) = \sigma(\texttt{x})$. Correspondingly:

$$(\texttt{INC}(\texttt{x}), \Sigma) \longrightarrow^0 (\texttt{INC}(\texttt{x}), \Sigma), (\Sigma, \Sigma) \in \mathbb{G}^*.$$

From the premise 2, we know

$$(\texttt{inc}^1(\texttt{x}), \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha} (\texttt{INC}(\texttt{x}), \Sigma, \mathbb{R}, \mathbb{G}).$$

Case: Both the non-blocking and the atomic sides are **skip**, then they are corresponding trivially. □

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

1. $(\texttt{INC}(\texttt{x}), \Sigma, \mathbb{R}, \mathbb{G}) \preceq_{\alpha^{-1};\alpha^{-1}} (\texttt{inc}(\texttt{x}), \sigma, \mathcal{R}, \mathcal{G})$;
2. $(\textbf{skip}, \Sigma, \mathbb{R}, \mathbb{G}) \preceq_{\alpha^{-1};\alpha^{-1}} (\textbf{skip}, \sigma, \mathcal{R}, \mathcal{G})$.

Proof:. By co-induction.
Case: The environments are executed. Trivial.
Case: The atomic counter code goes one step.
If $(\texttt{INC}(\texttt{x}), \Sigma) \longrightarrow (\textbf{skip}, \Sigma')$, then $\Sigma'(\texttt{x}) = \Sigma(\texttt{x}) + 1$. For $\texttt{inc}(\texttt{x})$, without the environment's interference (*i.e.*, `x` will not be modified), the statement `d:=cas(&x,t,t+1)` just after $\langle$`t:=x`$\rangle$ will find the values of `x` and `t` are the same and succeed in updating `x`. Thus `inc(x)` can be executed to **skip**:

$$(\texttt{inc}(\texttt{x}), \sigma) \longrightarrow^* (\textbf{skip}, \sigma'), \sigma'(\texttt{x}) = \sigma(\texttt{x}) + 1$$

Thus $(\sigma, \sigma') \in \mathcal{G}^*$ and $(\sigma', \Sigma') \in \alpha$. Then from the premise 2, we know

$$(\textbf{skip}, \Sigma, \mathbb{R}, \mathbb{G}) \preceq_{\alpha^{-1};\alpha^{-1}} (\textbf{skip}, \sigma, \mathcal{R}, \mathcal{G}).$$

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

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 proof [20] in the Coq proof assistant [10].

D.1.3 Incrementing Several Shared Variables

We verified the transformations for `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:

$$(\texttt{inc}(\texttt{y}), \mathcal{R}_1, \mathcal{G}_1) \simeq_{\alpha_1;\alpha_1 \ltimes \alpha_1} (\texttt{INC}(\texttt{y}), \mathbb{R}_1, \mathbb{G}_1)$$

where $\alpha_1 \triangleq \{(\sigma, \Sigma) \mid \sigma(\texttt{y}) = \Sigma(\texttt{y})\}$ and $\mathcal{R}_1, \mathcal{G}_1, \mathbb{R}_1$ and $\mathbb{G}_1$ are defined similarly as $\mathcal{R}, \mathcal{G}, \mathbb{R}$ and $\mathbb{G}$ except all the occurrences of `x` are replaced by `y`.

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

$$(\texttt{inc}(\texttt{x}); \texttt{inc}(\texttt{y}); \texttt{prt}(\texttt{x}), \mathcal{R} \uplus \mathcal{R}_1, \mathcal{G} \uplus \mathcal{G}_1) \simeq_{\beta;\beta \ltimes \beta} (\texttt{INC}(\texttt{x}); \texttt{INC}(\texttt{y}); \texttt{PRT}(\texttt{x}), \mathbb{R} \uplus \mathbb{R}_1, \mathbb{G} \uplus \mathbb{G}_1)$$

where $\beta \triangleq \alpha \uplus \alpha_1 = \{(\sigma, \Sigma) \mid \sigma(\texttt{x}) = \Sigma(\texttt{x}) \wedge \sigma(\texttt{y}) = \Sigma(\texttt{y})\}$, the relies ensure that the environments cannot update any local variable used in incrementing `x` nor `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:

$$\texttt{inc}(\texttt{x}); \texttt{inc}(\texttt{y}); \texttt{prt}(\texttt{x}) \approx_{\mathbf{T}} \texttt{INC}(\texttt{x}); \texttt{INC}(\texttt{y}); \texttt{PRT}(\texttt{x}),$$

for any $\mathbf{T}$ that respects β.

A_1 :
```
    local d1;
    d1 := 0;
    while (d1 = 0) {
0     atom{
        if (a = b)
          d1 := 1;
        if (a > b)
          a := a-b;
      }
    }
```
||

A_2 :
```
    local d2;
    d2 := 0;
    while (d2 = 0) {
0     atom{
        if (b = a)
          d2 := 1;
        if (b > a)
          b := b-a;
      }
    }
```

(a) Source Code

C_1 :
```
    local d1, t11, t12;
    d1 := 0;
    while (d1 = 0) {
0     ⟨t11 := a;⟩
1     ⟨t12 := b;⟩
2     if (t11 = t12)
3       d1 := 1;
4     if (t11 > t12)
5       ⟨a := t11-t12;⟩
    }
```
||

C_2 :
```
    local d2, t21, t22;
    d2 := 0;
    while (d2 = 0) {
0     ⟨t21 := b;⟩
1     ⟨t22 := a;⟩
2     if (t21 = t22)
3       d2 := 1;
4     if (t21 > t22)
5       ⟨b := t21-t22;⟩
    }
```

(b) Target Code

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 $\texttt{a} > \texttt{b}$. Another thread does the reverse. When $\texttt{a} = \texttt{b}$, 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, `prt(a)` and `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.*, $(C_1 \,\|\, C_2); \texttt{prt}(\texttt{a})$ and $(A_1 \,\|\, A_2); \texttt{PRT}(\texttt{a})$ 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 α relation as:

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

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

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

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_1^0 and A_1^0:

$$(C_1^0, \mathcal{R}_1, \mathcal{G}_1) \simeq_{\alpha;\alpha\ltimes\alpha} (A_1^0, \mathbb{R}_1, \mathbb{G}_1).$$

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

$$(C_1, \mathcal{R}_1, \mathcal{G}_1) \simeq_{\alpha;\alpha\ltimes\alpha} (A_1, \mathbb{R}_1, \mathbb{G}_1).$$

Similarly, C_2 and A_2 are equivalent:

$$(C_2, \mathcal{R}_2, \mathcal{G}_2) \simeq_{\alpha;\alpha\ltimes\alpha} (A_2, \mathbb{R}_2, \mathbb{G}_2).$$

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 $C_1 \| C_2$ is computing their GCD. Its guarantee is just specified as a set of all the possible state transitions.

$$\begin{array}{lcl} \mathcal{R} & \triangleq & \{(\sigma, \sigma') \mid \sigma' = \sigma\} \\ \mathcal{G} & \triangleq & \{(\sigma, \sigma') \mid \sigma, \sigma' \in \mathit{LState}\} \\ \mathbb{R} & \triangleq & \{(\Sigma, \Sigma') \mid \Sigma' = \Sigma\} \\ \mathbb{G} & \triangleq & \{(\Sigma, \Sigma') \mid \Sigma, \Sigma' \in \mathit{HState}\} \end{array}$$

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

$$(\mathtt{prt(a)}, \mathcal{R}, \mathcal{G}) \simeq_{\alpha;\alpha\ltimes\alpha} (\mathtt{PRT(a)}, \mathbb{R}, \mathbb{G})$$

Then by using the rules PAR and SEQ, we can get:

$$((C_1 \| C_2); \mathtt{prt(a)}, \mathcal{R}, \mathcal{G}) \simeq_{\alpha;\alpha\ltimes\alpha} ((A_1 \| A_2); \mathtt{PRT(a)}, \mathbb{R}, \mathbb{G}).$$

By soundness of RGSim (Theorem 5) we obtain the final result:

$$(C_1 \| C_2); \mathtt{prt(a)} \approx_{\mathbf{T}} (A_1 \| A_2); \mathtt{PRT(a)},$$

for any $\mathbf{T}$ that respects α.

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's 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_1^0 and A_1^0 in Lemmas 29 and 30.

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

1. $(C_1^0, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (A_1^0, \Sigma, \mathbb{R}_1, \mathbb{G}_1)$;
2. *if* $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$, *then* $(C_1^1, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (A_1^0, \Sigma, \mathbb{R}_1, \mathbb{G}_1)$;
3. *if* $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$ *and* $\sigma(\mathtt{t11}) < \sigma(\mathtt{t12})$, *then* $(C_1^2, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma, \mathbb{R}_1, \mathbb{G}_1)$;
4. *if* $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$, $\sigma(\mathtt{t12}) = \sigma(\mathtt{b})$ *and* $\sigma(\mathtt{t11}) >= \sigma(\mathtt{t12})$, *then* $(C_1^2, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (A_1^0, \Sigma, \mathbb{R}_1, \mathbb{G}_1)$;
5. *if* $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$, $\sigma(\mathtt{t12}) = \sigma(\mathtt{b})$ *and* $\sigma(\mathtt{t11}) = \sigma(\mathtt{t12})$, *then* $(C_1^3, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (A_1^0, \Sigma, \mathbb{R}_1, \mathbb{G}_1)$;
6. *if* $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$, $\sigma(\mathtt{t11}) <= \sigma(\mathtt{t12})$ *and* $(\sigma(\mathtt{t11}) = \sigma(\mathtt{t12})) \Longrightarrow (\sigma(\mathtt{d1}) = 1)$, *then* $(C_1^4, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma, \mathbb{R}_1, \mathbb{G}_1)$;
7. *if* $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$, $\sigma(\mathtt{t12}) = \sigma(\mathtt{b})$ *and* $\sigma(\mathtt{t11}) > \sigma(\mathtt{t12})$, *then* $(C_1^4, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (A_1^0, \Sigma, \mathbb{R}_1, \mathbb{G}_1)$;
8. *if* $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$, $\sigma(\mathtt{t12}) = \sigma(\mathtt{b})$ *and* $\sigma(\mathtt{t11}) > \sigma(\mathtt{t12})$, *then* $(C_1^5, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (A_1^0, \Sigma, \mathbb{R}_1, \mathbb{G}_1)$;
9. $(\textbf{skip}, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma, \mathbb{R}_1, \mathbb{G}_1)$.

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 $\mathcal{R}_1$ and $\mathbb{R}_1$.
Case: The concrete GCD goes one step.

1. If $(C_1^0, \sigma) \longrightarrow (C_1^1, \sigma')$, then $\sigma' = \sigma\{\mathtt{t11} \rightsquigarrow \sigma(\mathtt{a})\}$, thus $(\sigma, \sigma') \in \mathcal{G}_1$. Correspondingly, the abstract code does not go any step:
$$(A_1^0, \Sigma) \longrightarrow^0 (A_1^0, \Sigma), (\Sigma, \Sigma) \in \mathbb{G}_1^*.$$
From the premise 2, we know
$$(C_1^1, \sigma', \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (A_1^0, \Sigma, \mathbb{R}_1, \mathbb{G}_1).$$
2. If $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$ and $(C_1^1, \sigma) \longrightarrow (C_1^2, \sigma')$, then $\sigma' = \sigma\{\mathtt{t12} \rightsquigarrow \sigma(\mathtt{b})\}$, thus $(\sigma, \sigma') \in \mathcal{G}_1$.
   (a) If $\sigma(\mathtt{t11}) < \sigma(\mathtt{t12})$, then $\sigma(\mathtt{a}) < \sigma(\mathtt{b})$. From the α relation, we know $\Sigma(\mathtt{a}) < \Sigma(\mathtt{b})$. Thus on the atomic side:
   $$(A_1^0, \Sigma) \longrightarrow^1 (\textbf{skip}, \Sigma), (\Sigma, \Sigma) \in \mathbb{G}_1^*.$$
   From the premise 3, we know
   $$(C_1^2, \sigma', \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma, \mathbb{R}_1, \mathbb{G}_1).$$
   (b) If $\sigma(\mathtt{t11}) >= \sigma(\mathtt{t12})$, then on the atomic side:
   $$(A_1^0, \Sigma) \longrightarrow^0 (A_1^0, \Sigma), (\Sigma, \Sigma) \in \mathbb{G}_1^*.$$
   From the premise 4, we know
   $$(C_1^2, \sigma', \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (A_1^0, \Sigma, \mathbb{R}_1, \mathbb{G}_1).$$
3. If $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$ and $\sigma(\mathtt{t11}) < \sigma(\mathtt{t12})$, then $(C_1^2, \sigma) \longrightarrow (C_1^4, \sigma)$. On the atomic side,
$$(\textbf{skip}, \Sigma) \longrightarrow^0 (\textbf{skip}, \Sigma), (\Sigma, \Sigma) \in \mathbb{G}_1.$$
From the premise 6, we know
$$(C_1^4, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma, \mathbb{R}_1, \mathbb{G}_1).$$
4. If $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$, $\sigma(\mathtt{t12}) = \sigma(\mathtt{b})$ and
   (a) $\sigma(\mathtt{t11}) = \sigma(\mathtt{t12})$, then $(C_1^2, \sigma) \longrightarrow (C_1^3, \sigma)$. Correspondingly, the atomic code does not go any step. From the premise 5, we know
   $$(C_1^3, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (A_1^0, \Sigma, \mathbb{R}_1, \mathbb{G}_1).$$
   (b) $\sigma(\mathtt{t11}) > \sigma(\mathtt{t12})$, then $(C_1^2, \sigma) \longrightarrow (C_1^4, \sigma)$. Correspondingly, the atomic code does not go any step. From the premise 7, we know
   $$(C_1^4, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (A_1^0, \Sigma, \mathbb{R}_1, \mathbb{G}_1).$$
5. If $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$, $\sigma(\mathtt{t12}) = \sigma(\mathtt{b})$ and $\sigma(\mathtt{t11}) = \sigma(\mathtt{t12})$, then $(C_1^3, \sigma) \longrightarrow (C_1^4, \sigma')$, and $\sigma' = \sigma\{\mathtt{d1} \rightsquigarrow 1\}$. From the α relation, we know $\Sigma(\mathtt{a}) = \Sigma(\mathtt{b})$.
Thus on the atomic side:
$$(A_1^0, \Sigma) \longrightarrow^1 (\textbf{skip}, \Sigma'), \Sigma' = \Sigma\{\mathtt{d1} \rightsquigarrow 1\}.$$
Thus $(\Sigma, \Sigma') \in \mathbb{G}_1^*$ and $(\sigma', \Sigma') \in \alpha$. From the premise 6, we know
$$(C_1^4, \sigma', \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma, \mathbb{R}_1, \mathbb{G}_1).$$
6. If $\sigma(\mathtt{t11}) = \sigma(\mathtt{a})$, $\sigma(\mathtt{t11}) <= \sigma(\mathtt{t12})$ and $(\sigma(\mathtt{t11}) = \sigma(\mathtt{t12})) \Longrightarrow (\sigma(\mathtt{d1}) = 1)$, then $(C_1^4, \sigma) \longrightarrow (\textbf{skip}, \sigma)$. From the premise 9, we know
$$(\textbf{skip}, \sigma, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma, \mathbb{R}_1, \mathbb{G}_1).$$

```
PUSH(v) :

  atom {
    A := v::A;
  }
```

```
POP() :
  local r;
  atom {
    if (A = ϵ) {
      r := EMPTY;
    }else {
      r := hd(A);
      A := tl(A);
    }
  }
  return r;
```

(a) An Abstract Stack

```
push(v) :

  local d, x, t;
  x := new Cell();
  x.data := v;
  d := 0;
  while (d = 0) {
    <t := S;>
    x.next := t;
    d := cas(&S,t,x);
  }
```

```
pop() :
  local r, d, x, t;
  d := 0;
  while (d = 0) {
    <t := S;>
    if (t = null) {
      r := EMPTY;
      d := 1;
    }else {
      r := t.data;
      x := t.next;
      d := cas(&S,t,x);
    }
  }
  return r;
```

(b) Treiber's Non-Blocking Stack

Figure 22. The Stack Object

7. If $\sigma(\texttt{t11}) = \sigma(\texttt{a})$, $\sigma(\texttt{t12}) = \sigma(\texttt{b})$ and $\sigma(\texttt{t11}) > \sigma(\texttt{t12})$, then from the premise 8, we can prove this case.

8. If $\sigma(\texttt{t11}) = \sigma(\texttt{a})$, $\sigma(\texttt{t12}) = \sigma(\texttt{b})$ and $\sigma(\texttt{t11}) > \sigma(\texttt{t12})$, then $(C_1^5, \sigma) \longrightarrow (\textbf{skip}, \sigma')$, and $\sigma' = \sigma\{\texttt{a} \rightsquigarrow (\sigma(\texttt{a}) - \sigma(\texttt{b}))\}$. From the α relation, we know $\Sigma(\texttt{a}) > \Sigma(\texttt{b})$. Thus on the atomic side:
$$(A_1^0, \Sigma) \longrightarrow^1 (\textbf{skip}, \Sigma'),\ \Sigma' = \Sigma\{\texttt{a} \rightsquigarrow (\Sigma(\texttt{a}) - \Sigma(\texttt{b}))\}$$
Thus $(\Sigma, \Sigma') \in \mathbb{G}_1^*$ and $(\sigma', \Sigma') \in \alpha$. From the premise 9, we know
$$(\textbf{skip}, \sigma', \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma', \mathbb{R}_1, \mathbb{G}_1).$$

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

Lemma 30. *For all* $(\sigma, \Sigma) \in \alpha$,

1. $(A_1^0, \Sigma, \mathbb{R}_1, \mathbb{G}_1) \preceq_{\alpha^{-1};\alpha^{-1}} (C_1^0, \sigma, \mathcal{R}_1, \mathcal{G}_1)$;
2. $(\textbf{skip}, \Sigma, \mathbb{R}_1, \mathbb{G}_1) \preceq_{\alpha^{-1};\alpha^{-1}} (\textbf{skip}, \sigma, \mathcal{R}_1, \mathcal{G}_1)$.

Proof:. By co-induction.
The proof is similar to that of Lemma 28, where the key is that without the environment's interference, C_1^0 does exactly the same things as A_1^0. □

D.3 Treiber's Non-Blocking Stack

We prove atomicity of Treiber's non-blocking stack. As shown in Figure 22, the stack interface consists of two operations: `PUSH(v)` and `POP()`. The abstract stack `A` is a value sequence and the operations are executed atomically. `PUSH(v)` and `POP()` are transformed to the non-blocking programs `push(v)` and `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.

We first define a predicate $m_s \models \mathsf{list}(x, A)$ 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.
$$\begin{array}{rcl} m_s \models \mathsf{list}(x, A) & \triangleq & (dom(m_s) = \phi \wedge x = \textbf{null} \wedge A = \epsilon) \\ & \vee & (\exists v. \exists y. \exists B. m_s(x) = (v, y) \wedge A = v :: B \\ & & \wedge\ m_s \backslash \{x\} \models \mathsf{list}(y, B)) \end{array}$$

For the high-level and low-level library states, we only consider the value sequence on the stack:
$$\begin{array}{rcl} \mathsf{shared\_map}(m_s, M_s) & \triangleq & \exists \hat{\sigma}_s.\ \hat{\sigma}_s \models \mathsf{list}(m_s(\texttt{S}), M_s(\texttt{A})) \\ & & \wedge\ \hat{\sigma}_s \subseteq m_s \backslash \{\texttt{S}\} \end{array}$$

It requires that the concrete library state m_s has a sub-state $\hat{\sigma}_s$ 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 $\hat{\sigma}_s$. 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`).
$$\begin{array}{rcl} \mathsf{local\_map}(m_l, M_l) & \triangleq & m_l(\texttt{v}) = M_l(\texttt{v}) \\ & & \wedge\ \exists m_l'.\ m_l = m_l' \uplus \{\texttt{d} \rightsquigarrow \_, \texttt{x} \rightsquigarrow \_, \texttt{t} \rightsquigarrow \_, \texttt{r} \rightsquigarrow \_\} \end{array}$$

Then α is defined as follows:
$$\begin{array}{l} \alpha \triangleq \{((\pi, m_s), (\Pi, M_s)) \mid \mathsf{shared\_map}(m_s, M_s) \\ \qquad\qquad \wedge\ \forall \texttt{t} \in dom(\Pi).\ \mathsf{local\_map}(\pi(\texttt{t}), \Pi(\texttt{t}))\} \end{array}$$

The program guarantees and relies can be specified as follows:
$$\begin{array}{rcl} \mathcal{G}_{\text{push}}(\texttt{t}) & \triangleq & \{((\pi \uplus \{\texttt{t} \rightsquigarrow m_l\}, m_s), (\pi \uplus \{\texttt{t} \rightsquigarrow m_l'\}, m_s')) \\ & & \mid\ \exists v. \exists x \in dom(m_l).\ x \notin dom(m_l') \\ & & \quad \wedge\ m_s' = m_s\{\texttt{S} \rightsquigarrow x\} \uplus \{x \rightsquigarrow (v, m_s(\texttt{S}))\}\} \\ \mathcal{G}_{\text{pop}}(\texttt{t}) & \triangleq & \{((\pi \uplus \{\texttt{t} \rightsquigarrow m_l\}, m_s), (\pi \uplus \{\texttt{t} \rightsquigarrow m_l'\}, m_s')) \\ & & \mid\ \exists x. \exists v. \exists y.\ m_s(\texttt{S}) = x \wedge m_s(x) = (v, y) \\ & & \quad \wedge\ m_s' = m_s\{\texttt{S} \rightsquigarrow y\}\} \\ \mathcal{G}_{\text{local}}(\texttt{t}) & \triangleq & \{((\pi \uplus \{\texttt{t} \rightsquigarrow m_l\}, m_s), (\pi \uplus \{\texttt{t} \rightsquigarrow m_l'\}, m_s))\} \\ \mathcal{G}(\texttt{t}) & \triangleq & \mathcal{G}_{\text{push}}(\texttt{t}) \cup \mathcal{G}_{\text{pop}}(\texttt{t}) \cup \mathcal{G}_{\text{local}}(\texttt{t}) \\ \mathcal{R}(\texttt{t}) & \triangleq & \bigcup_{\texttt{t}' \neq \texttt{t}} \mathcal{G}(\texttt{t}') \\ \\ \mathbb{G}_{\text{push}}(\texttt{t}) & \triangleq & \{((\Pi \uplus \{\texttt{t} \rightsquigarrow M_l\}, M_s), (\Pi \uplus \{\texttt{t} \rightsquigarrow M_l'\}, M_s')) \\ & & \mid\ \exists v.\ M_s' = M_s\{\texttt{A} \rightsquigarrow v :: M_s(\texttt{A})\}\} \\ \mathbb{G}_{\text{pop}}(\texttt{t}) & \triangleq & \{((\Pi \uplus \{\texttt{t} \rightsquigarrow M_l\}, M_s), (\Pi \uplus \{\texttt{t} \rightsquigarrow M_l'\}, M_s')) \\ & & \mid\ \exists v. \exists B.\ M_s(\texttt{A}) = v :: B \wedge M_s' = M_s\{\texttt{A} \rightsquigarrow B\}\} \\ \mathbb{G}_{\text{local}}(\texttt{t}) & \triangleq & \{((\Pi \uplus \{\texttt{t} \rightsquigarrow M_l\}, M_s), (\Pi \uplus \{\texttt{t} \rightsquigarrow M_l'\}, M_s))\} \\ \mathbb{G}(\texttt{t}) & \triangleq & \mathbb{G}_{\text{push}}(\texttt{t}) \cup \mathbb{G}_{\text{pop}}(\texttt{t}) \cup \mathbb{G}_{\text{local}}(\texttt{t}) \\ \mathbb{R}(\texttt{t}) & \triangleq & \bigcup_{\texttt{t}' \neq \texttt{t}} \mathbb{G}(\texttt{t}') \end{array}$$

The ownership transfer in `push(v)` is reflected in the guarantee $\mathcal{G}_{\text{push}}(\texttt{t})$, 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.
$$\begin{array}{l} (\texttt{t.push(v)}, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha\ltimes\alpha} (\texttt{t.PUSH(v)}, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t})); \\ (\texttt{t}.(r := \texttt{pop}()), \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha\ltimes\alpha} (\texttt{t}.(r := \texttt{POP}()), \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t})). \end{array}$$

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. $(\texttt{push(v)}, \sigma, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha} (\texttt{PUSH(v)}, \Sigma, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$;
2. *if there exists* x *such that* $\pi(\texttt{t})(\texttt{x}) = x$ *and* $\pi(\texttt{t})(x) = (\_, \_)$, *then* $(\texttt{push}^1(\texttt{v}), \sigma, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha} (\texttt{PUSH(v)}, \Sigma, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$;

3. *if there exists x such that $\pi(\mathtt{t})(\mathtt{x}) = x$ and $\pi(\mathtt{t})(x) = (\pi(\mathtt{t})(\mathtt{v}), \_)$,*
   then $(\mathtt{push}^2(\mathtt{v}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{PUSH}(\mathtt{v}), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
4. *if $\pi(\mathtt{t})(\mathtt{d}) = 1$,*
   then $(\mathtt{push}^3(\mathtt{v}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
5. *if $\pi(\mathtt{t})(\mathtt{d}) = 0$ and there exists x such that $\pi(\mathtt{t})(\mathtt{x}) = x$ and $\pi(\mathtt{t})(x) = (\pi(\mathtt{t})(\mathtt{v}), \_)$,*
   then $(\mathtt{push}^3(\mathtt{v}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{PUSH}(\mathtt{v}), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
6. *if there exists x such that $\pi(\mathtt{t})(\mathtt{x}) = x$ and $\pi(\mathtt{t})(x) = (\pi(\mathtt{t})(\mathtt{v}), \_)$,*
   then $(\mathtt{push}^4(\mathtt{v}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{PUSH}(\mathtt{v}), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
7. *if there exists x such that $\pi(\mathtt{t})(\mathtt{x}) = x$ and $\pi(\mathtt{t})(x) = (\pi(\mathtt{t})(\mathtt{v}), \_)$,*
   then $(\mathtt{push}^5(\mathtt{v}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{PUSH}(\mathtt{v}), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
8. *if there exists x such that $\pi(\mathtt{t})(\mathtt{x}) = x$ and $\pi(\mathtt{t})(x) = (\pi(\mathtt{t})(\mathtt{v}), \pi(\mathtt{t})(\mathtt{t}))$,*
   then $(\mathtt{push}^6(\mathtt{v}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{PUSH}(\mathtt{v}), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
9. $(\textbf{skip}, \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$.

Proof:. By co-induction.
Case: The environments are executed. Trivial.
Case: The concrete code goes one step.

1. If $(\mathtt{push}, \sigma) \longrightarrow (\mathtt{push}^1, \sigma')$, then there exists x such that
   $$\pi' = \pi\{\mathtt{t} \rightsquigarrow \pi(\mathtt{t})\{\mathtt{x} \rightsquigarrow x\} \uplus \{x \rightsquigarrow (\_, \_)\}\},\ m_s' = m_s.$$
   Correspondingly, the atomic code does not go any step:
   $$(\mathtt{PUSH}, \Sigma) \longrightarrow^0 (\mathtt{PUSH}, \Sigma),\ (\Sigma, \Sigma) \in \mathbb{G}(\mathtt{t})^*.$$
   From the premise 2, we know
   $$(\mathtt{push}^1, \sigma', \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{PUSH}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t})).$$
2. Similar to the first case (but using the premise 3) and omitted.
3. Similar to the first case (but using the premise 5) and omitted.
4. If $\pi(\mathtt{t})(\mathtt{d}) = 1$, then
   $$(\mathtt{push}^3, \sigma) \longrightarrow (\textbf{skip}, \sigma).$$
   Correspondingly, on the atomic side:
   $$(\textbf{skip}, \Sigma) \longrightarrow^0 (\textbf{skip}, \Sigma),\ (\Sigma, \Sigma) \in \mathbb{G}(\mathtt{t})^*.$$
   From the premise 9, we know
   $$(\textbf{skip}, \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t})).$$
5. Similar to the first case (but using the premise 6) and omitted.
6. Similar to the first case (but using the premise 7) and omitted.
7. Similar to the first case (but using the premise 8) and omitted.
8. If $(\mathtt{push}^6, \sigma) \longrightarrow (\mathtt{push}^3, \sigma')$, then
   (a) if $m_s(\mathtt{S}) = \pi(\mathtt{t})(\mathtt{t})$, then $(\sigma, \sigma') \in \mathcal{G}(\mathtt{t})$ and
   $$\pi' = \pi\{\mathtt{t} \rightsquigarrow \pi(\mathtt{t})\{\mathtt{d} \rightsquigarrow 1\}\}$$
   $$m_s' = m_s\{\mathtt{S} \rightsquigarrow x\} \uplus \{x \rightsquigarrow (\pi(\mathtt{t})(\mathtt{v}), m_s(\mathtt{S}))\}$$
   where $x = \pi(\mathtt{t})(\mathtt{x})$.
   Correspondingly, on the atomic side:
   $$(\mathtt{PUSH}, \Sigma) \longrightarrow (\textbf{skip}, \Sigma'),\ (\Sigma, \Sigma') \in \mathbb{G}(\mathtt{t})^*$$
   where
   $$\Pi' = \Pi$$
   $$M_s' = M_s\{\mathtt{A} \rightsquigarrow \Pi(\mathtt{t})(\mathtt{v}) :: M_s(\mathtt{A})\}$$
   We know $\pi(\mathtt{t})(\mathtt{v}) = \Pi(\mathtt{t})(\mathtt{v})$. And there exists a sub-state $\hat{\sigma}_s$ such that
   $$\hat{\sigma}_s \models \mathsf{list}(m_s(\mathtt{S}), M_s(\mathtt{A})). \tag{D.1}$$
   Let
   $$\hat{\sigma}_s' = \hat{\sigma}_s \uplus \{m_s'(\mathtt{S}) \rightsquigarrow (\pi(\mathtt{t})(\mathtt{v}), m_s(\mathtt{S}))\}. \tag{D.2}$$
   Then from (D.1) and (D.2), we can prove that
   $$\hat{\sigma}_s' \models \mathsf{list}(m_s'(\mathtt{S}), \Pi(\mathtt{t})(\mathtt{v}) :: M_s(\mathtt{A})). \tag{D.3}$$
   Moreover, $\hat{\sigma}_s'$ is a sub-state of $m_s' \backslash \{\mathtt{S}\}$.
   Thus $\mathsf{shared\_map}(m_s', M_s')$. As a result, we have $(\sigma', \Sigma') \in \alpha$. From the premise 4, we know
   $$(\mathtt{push}^3, \sigma', \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\textbf{skip}, \Sigma', \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t})).$$
   (b) if $m_s(\mathtt{S}) \neq \pi(\mathtt{t})(\mathtt{t})$, then
   $$\pi' = \pi\{\mathtt{t} \rightsquigarrow \pi(\mathtt{t})\{\mathtt{d} \rightsquigarrow 0\}\} \text{ and } m_s' = m_s.$$
   Correspondingly, the atomic code does not go any step. Then from the premise 5, we know
   $$(\mathtt{push}^3, \sigma', \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{PUSH}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t})).$$

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

For the POP operation, we assume the returned value $\mathtt{r}$ will be assigned to a variable r.

Lemma 32. *For all $(\sigma, \Sigma) \in \alpha$ where $\sigma = (\pi, m_s)$ and $\Sigma = (\Pi, M_s)$,*

1. $(r := \mathtt{pop}(), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{POP}(), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
2. *if $\pi(\mathtt{t})(\mathtt{d}) = 0$,*
   then
   $(r := \mathtt{pop}^1()(), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{POP}(), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
3. *if $\pi(\mathtt{t})(\mathtt{d}) = 1$ and $\pi(\mathtt{t})(\mathtt{r}) = \Pi(\mathtt{t})(\mathtt{r})$,*
   then $(r := \mathtt{pop}^1(), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{r}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
4. $(r := \mathtt{pop}^2(), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{POP}(), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
5. *if there exists x such that $\pi(\mathtt{t})(\mathtt{t}) = x$ and $m_s(x) = (\_, \_)$,*
   then
   $(r := \mathtt{pop}^3(), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{POP}(), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
6. *if $\pi(\mathtt{t})(\mathtt{t}) = \textbf{null}$ and $\Pi(\mathtt{t})(\mathtt{r}) = \mathtt{EMPTY}$,*
   then $(r := \mathtt{pop}^3(), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{r}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
7. *if $\Pi(\mathtt{t})(\mathtt{r}) = \mathtt{EMPTY}$,*
   then $(r := \mathtt{pop}^4(), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{r}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
8. *if $\pi(\mathtt{t})(\mathtt{r}) = \Pi(\mathtt{t})(\mathtt{r})$,*
   then $(r := \mathtt{pop}^5(), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{r}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
9. *if there exists x such that $\pi(\mathtt{t})(\mathtt{t}) = x$ and $m_s(x) = (\_, \_)$,*
   then
   $(r := \mathtt{pop}^6(), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{POP}(), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
10. *if there exists x such that $\pi(\mathtt{t})(\mathtt{t}) = x$, $m_s(x) = (\_, \_)$ and $m_s(\mathtt{S}) = x \implies m_s(x) = (\pi(\mathtt{t})(\mathtt{r}), \_)$,*
    then
    $(r := \mathtt{pop}^7(), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{POP}(), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
11. *if there exists x such that $\pi(\mathtt{t})(\mathtt{t}) = x$, $m_s(x) = (\_, \_)$ and $m_s(\mathtt{S}) = x \implies m_s(x) = (\pi(\mathtt{t})(\mathtt{r}), \pi(\mathtt{t})(\mathtt{x}))$,*
    then
    $(r := \mathtt{pop}^8(), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{POP}(), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
12. *if $\pi(\mathtt{t})(\mathtt{r}) = \Pi(\mathtt{t})(\mathtt{r})$,*
    then $(r := \mathtt{r}, \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{r}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$.

Proof:. By co-induction.
Case: The environments are executed. Trivial.
Case: The concrete pop operation goes one step.

1. If $(r := \mathtt{pop}(), \sigma) \longrightarrow (r := \mathtt{pop}^1(), \sigma')$, then
$$m'_s = m_s, \pi' = \pi\{\mathtt{t} \rightsquigarrow \pi(\mathtt{t})\{\mathtt{d} \rightsquigarrow 0\}\}.$$
Thus $(\sigma, \sigma') \in \mathcal{G}(\mathtt{t})$. Correspondingly, the atomic code does not go any step. From the premise 2, we know
$$(r := \mathtt{pop}^1(), \sigma', \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{POP}(), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$$
2. Similar and omitted (using the premise 4).
3. If $\pi(\mathtt{t})(\mathtt{d}) = 1$, then
$$(r := \mathtt{pop}^1(), \sigma) \longrightarrow (r := \mathtt{r}, \sigma).$$
Correspondingly, the atomic code does not go any step. From the premise 12, we know
$$(r := \mathtt{r}, \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{r}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t})).$$
4. If $(r := \mathtt{pop}^2(), \sigma) \longrightarrow (r := \mathtt{pop}^3(), \sigma')$, then
$$m'_s = m_s, \pi' = \pi\{\mathtt{t} \rightsquigarrow \pi(\mathtt{t})\{\mathtt{t} \rightsquigarrow m_s(\mathtt{S})\}\}.$$
   (a) If $m_s(\mathtt{S}) = \textbf{null}$, then $\pi'(\mathtt{t})(\mathtt{t}) = \textbf{null}$.
   Thus there exists $\hat{\sigma}_s$ such that
$$\hat{\sigma}_s \models \mathsf{list}(\textbf{null}, M_s(\mathtt{A})).$$
   Thus $M_s(\mathtt{A}) = \epsilon$.
   Correspondingly, on the atomic side:
$$(r := \mathtt{POP}(), \Sigma) \longrightarrow (r := \mathtt{r}, \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}(\mathtt{t})^*$$
   where
$$M'_s = M_s, \Pi' = \Pi\{\mathtt{t} \rightsquigarrow \Pi(\mathtt{t})\{\mathtt{r} \rightsquigarrow \mathtt{EMPTY}\}\}.$$
   From the premise 6, we know
$$(r := \mathtt{pop}^3(), \sigma', \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{r}, \Sigma', \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$$
   (b) If $m_s(\mathtt{S}) \neq \textbf{null}$, then from
$$\hat{\sigma}_s \models \mathsf{list}(m_s(\mathtt{S}), M_s(\mathtt{A})),$$
   we know there exists x such that $\pi'(\mathtt{t})(\mathtt{t}) = x$ and $m'_s(x) = (\_, \_)$.
   Correspondingly, the atomic code does not go any step. From the premise 5, we know
$$(r := \mathtt{pop}^3(), \sigma', \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{POP}(), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$$
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 $(r := \mathtt{pop}^8(), \sigma) \longrightarrow (r := \mathtt{pop}^1(), \sigma')$, then
   (a) if $m_s(\mathtt{S}) = \pi(\mathtt{t})(\mathtt{t}) = x$, then $(\sigma, \sigma') \in \mathcal{G}(\mathtt{t})$ and
$$\begin{array}{l}\pi' = \pi\{\mathtt{t} \rightsquigarrow \pi(\mathtt{t})\{\mathtt{d} \rightsquigarrow 1\}\} \\ m'_s = m_s\{\mathtt{S} \rightsquigarrow \pi(\mathtt{t})(\mathtt{x})\}.\end{array} \tag{D.1}$$
   From $(\sigma, \Sigma) \in \alpha$, there exists $\hat{\sigma}_s$ such that
$$\hat{\sigma}_s \models \mathsf{list}(x, M_s(\mathtt{A})). \tag{D.2}$$
   Since (D.2) and
$$m_s(x) = (\pi(\mathtt{t})(\mathtt{r}), \pi(\mathtt{t})(\mathtt{x})), \tag{D.3}$$
   there exists B such that
$$\begin{array}{l}M_s(\mathtt{A}) = \pi(\mathtt{t})(\mathtt{r}) :: B, \\ \hat{\sigma}_s \backslash \{x\} \models \mathsf{list}(\pi(\mathtt{t})(\mathtt{x}), B).\end{array} \tag{D.4}$$
   Correspondingly, on the atomic side:
$$(r := \mathtt{POP}(), \Sigma) \longrightarrow (r := \mathtt{r}, \Sigma'), (\Sigma, \Sigma') \in \mathbb{G}(\mathtt{t})^*$$
   where
$$\Pi'(\mathtt{t})(\mathtt{r}) = \pi(\mathtt{t})(\mathtt{r}), M'_s(\mathtt{A}) = B. \tag{D.5}$$
   From (D.4) and (D.5), we have
$$\hat{\sigma}_s \backslash \{x\} \models \mathsf{list}(m'_s(\mathtt{S}), M'_s(\mathtt{A})). \tag{D.6}$$
   Since (D.1) and $\hat{\sigma}_s$ is a sub-state of $m_s \backslash \{\mathtt{S}\}$, we know
$$\hat{\sigma}_s \backslash \{x\} \subseteq m'_s \backslash \{\mathtt{S}\}. \tag{D.7}$$
   From (D.6) and (D.7), we have
$$(\sigma', \Sigma') \in \alpha. \tag{D.8}$$
   From (D.1) and (D.5), we have
$$\pi'(\mathtt{t})(\mathtt{r}) = \Pi'(\mathtt{t})(\mathtt{r}). \tag{D.9}$$
   Thus from (D.1), (D.8), (D.9) and the premise 3, we know
$$(r := \mathtt{pop}^1(), \sigma', \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{r}, \Sigma', \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$$
   (b) if $m_s(\mathtt{S}) \neq \pi(\mathtt{t})(\mathtt{t})$, then
$$m'_s = m_s, \pi' = \pi\{\mathtt{t} \rightsquigarrow \pi(\mathtt{t})\{\mathtt{d} \rightsquigarrow 0\}\}.$$
   Correspondingly, the atomic code does not go any step. Then from the premise 2, we know
$$(r := \mathtt{pop}^1(), \sigma', \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (r := \mathtt{POP}(), \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$$

Case: Both sides are $r := \mathtt{r}$, the proof is trivial. □

D.4 Lock-Coupling List

To prove `add(e)` refines `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 `rmv(e)`, we choose line `13` (for successful calls) and lines `3` and `9` (for unsuccessful calls) as linearization points.

From the definition of $\mathcal{G}_{\text{lib}}(\mathtt{t})$, 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 `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. $(\mathtt{t.add(e)}, \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
2. *if there exists x such that $\pi(\mathtt{t})(\mathtt{x}) = m_s(\mathtt{Head}) = x$, then* $(\mathtt{t.add}^1(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
3. *if there exist x and z such that $\pi(\mathtt{t})(\mathtt{x}) = m_s(\mathtt{Head}) = x$ and $m_s(x) = (\mathtt{t}, \mathtt{MIN\_VAL}, z)$, then* $(\mathtt{t.add}^2(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
4. *if there exist x and z such that $\pi(\mathtt{t})(\mathtt{x}) = m_s(\mathtt{Head}) = x$, $m_s(x) = (\mathtt{t}, \mathtt{MIN\_VAL}, z)$ and $\pi(\mathtt{t})(\mathtt{z}) = z$, then* $(\mathtt{t.add}^3(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
5. *if there exist x, z, e, v and u such that $\pi(\mathtt{t})(\mathtt{e}) = e$, $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(\mathtt{u}) = u$, $m_s(x) = (\mathtt{t}, v, z)$, $m_s(z) = (\_, u, \_)$, $v < e$ and $u \neq e$, then* $(\mathtt{t.add}^4(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
6. *if there exist x, z, e and v such that $\pi(\mathtt{t})(\mathtt{e}) = \pi(\mathtt{t})(\mathtt{u}) = e$, $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $m_s(x) = (\mathtt{t}, v, z)$, $m_s(z) = (\_, e, \_)$ and $v < e$, then* $(\mathtt{t.add}^4(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.}\textbf{skip}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
7. *if there exist x, z and u such that $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(\mathtt{u}) = u$, $m_s(x) = (\mathtt{t}, \_, z)$, $m_s(z) = (\_, u, \_)$ and $u < \pi(\mathtt{t})(\mathtt{e})$, then* $(\mathtt{t.add}^5(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
8. *if there exist x, z and u such that $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(\mathtt{u}) = u$, $m_s(x) = (\mathtt{t}, \_, z)$, $m_s(z) = (\mathtt{t}, u, \_)$ and $u < \pi(\mathtt{t})(\mathtt{e})$, then* $(\mathtt{t.add}^6(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
9. *if there exist z and u such that $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(\mathtt{u}) = u$, $m_s(z) = (\mathtt{t}, u, \_)$ and $u < \pi(\mathtt{t})(\mathtt{e})$, then* $(\mathtt{t.add}^7(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
10. *if there exist z and u such that $\pi(\mathtt{t})(\mathtt{x}) = z$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(\mathtt{u}) = u$, $m_s(z) = (\mathtt{t}, u, \_)$ and $u < \pi(\mathtt{t})(\mathtt{e})$, then* $(\mathtt{t.add}^8(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
11. *if there exist x, z and u such that $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(\mathtt{u}) = u$, $m_s(x) = (\mathtt{t}, u, z)$ and $u < \pi(\mathtt{t})(\mathtt{e})$, then* $(\mathtt{t.add}^9(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
12. *if there exist x, z, e, v and u such that $\pi(\mathtt{t})(\mathtt{e}) = e$, $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(\mathtt{u}) = u$, $m_s(x) = (\mathtt{t}, v, z)$, $m_s(z) = (\_, u, \_)$, $v < e < u$, then* $(\mathtt{t.add}^{10}(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
13. *if there exist x, z, v and u such that $\pi(\mathtt{t})(\mathtt{e}) = \pi(\mathtt{t})(\mathtt{u}) = e$, $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $m_s(x) = (\mathtt{t}, v, z)$, $m_s(z) = (\_, e, \_)$ and $v < e$, then* $(\mathtt{t.add}^{10}(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.}\textbf{skip}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
14. *if there exist x, z, v and u such that $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(\mathtt{u}) = u$, $m_s(x) = (\mathtt{t}, v, z)$, $m_s(z) = (\_, u, \_)$ and $v < \pi(\mathtt{t})(\mathtt{e}) < u$, then* $(\mathtt{t.add}^{11}(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
15. *if there exist x, y, z, v and u such that $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(y) = (\_, \_, \_)$, $m_s(x) = (\mathtt{t}, v, z)$, $m_s(z) = (\_, u, \_)$ and $v < \pi(\mathtt{t})(\mathtt{e}) < u$, then* $(\mathtt{t.add}^{12}(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
16. *if there exist x, y, z, v and u such that $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(y) = (0, \_, \_)$, $m_s(x) = (\mathtt{t}, v, z)$, $m_s(z) = (\_, u, \_)$ and $v < \pi(\mathtt{t})(\mathtt{e}) < u$, then* $(\mathtt{t.add}^{13}(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
17. *if there exist x, y, z, e, v and u such that $\pi(\mathtt{t})(\mathtt{e}) = e$, $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(y) = (0, e, \_)$, $m_s(x) = (\mathtt{t}, v, z)$, $m_s(z) = (\_, u, \_)$ and $v < e < u$, then* $(\mathtt{t.add}^{14}(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
18. *if there exist x, y, z, e, v and u such that $\pi(\mathtt{t})(\mathtt{e}) = e$, $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{z}) = z$, $\pi(\mathtt{t})(y) = (0, e, z)$, $m_s(x) = (\mathtt{t}, v, z)$, $m_s(z) = (\_, u, \_)$ and $v < e < u$, then* $(\mathtt{t.add}^{15}(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.ADD(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
19. *if there exists x such that $\pi(\mathtt{t})(\mathtt{x}) = x$ and $m_s(x) = (\mathtt{t}, \_, \_)$, then* $(\mathtt{t.add}^{16}(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.}\textbf{skip}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
20. $(\mathtt{t.}\textbf{skip}, \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.}\textbf{skip}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;

Proof:. By co-induction. □

Lemma 34. *For all $(\sigma, \Sigma) \in \alpha$ where $\sigma = (\pi, m_s)$ and $\Sigma = (\Pi, M_s)$,*

1. $(\mathtt{t.rmv(e)}, \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.RMV(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
2. *if there exists x such that $\pi(\mathtt{t})(\mathtt{x}) = m_s(\mathtt{Head}) = x$, then* $(\mathtt{t.rmv}^1(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.RMV(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
3. *if there exist x and y such that $\pi(\mathtt{t})(\mathtt{x}) = m_s(\mathtt{Head}) = x$ and $m_s(x) = (\mathtt{t}, \mathtt{MIN\_VAL}, y)$, then* $(\mathtt{t.rmv}^2(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.RMV(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
4. *if there exist x and y such that $\pi(\mathtt{t})(\mathtt{x}) = m_s(\mathtt{Head}) = x$, $m_s(x) = (\mathtt{t}, \mathtt{MIN\_VAL}, y)$ and $\pi(\mathtt{t})(\mathtt{y}) = y$, then* $(\mathtt{t.rmv}^3(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.RMV(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
5. *if there exist x, y, e, v and u such that $\pi(\mathtt{t})(\mathtt{e}) = e$, $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{v}) = v$, $m_s(x) = (\mathtt{t}, u, y)$, $m_s(y) = (\_, v, \_)$, $u < e$ and $v \leq e$, then* $(\mathtt{t.rmv}^4(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.RMV(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
6. *if there exist x, y, e, v and u such that $\pi(\mathtt{t})(\mathtt{e}) = e$, $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{v}) = v$, $m_s(x) = (\mathtt{t}, u, y)$, $m_s(y) = (\_, v, \_)$, $u < e < v$, then* $(\mathtt{t.rmv}^4(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.}\textbf{skip}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
7. *if there exist x, y and v such that $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{v}) = v$, $m_s(x) = (\mathtt{t}, \_, y)$, $m_s(y) = (\_, v, \_)$ and $v < \pi(\mathtt{t})(\mathtt{e})$, then* $(\mathtt{t.rmv}^5(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.RMV(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
8. *if there exist x, y and v such that $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{v}) = v$, $m_s(x) = (\mathtt{t}, \_, y)$, $m_s(y) = (\mathtt{t}, v, \_)$ and $v < \pi(\mathtt{t})(\mathtt{e})$, then* $(\mathtt{t.rmv}^6(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.RMV(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
9. *if there exist y and v such that $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{v}) = v$, $m_s(z) = (\mathtt{t}, v, \_)$ and $v < \pi(\mathtt{t})(\mathtt{e})$, then* $(\mathtt{t.rmv}^7(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.RMV(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
10. *if there exist y and v such that $\pi(\mathtt{t})(\mathtt{x}) = y$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{v}) = v$, $m_s(y) = (\mathtt{t}, v, \_)$ and $v < \pi(\mathtt{t})(\mathtt{e})$, then* $(\mathtt{t.rmv}^8(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.RMV(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
11. *if there exist x, y and v such that $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{v}) = v$, $m_s(x) = (\mathtt{t}, v, y)$ and $v < \pi(\mathtt{t})(\mathtt{e})$, then* $(\mathtt{t.rmv}^9(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.RMV(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
12. *if there exist x, y, e and u such that $\pi(\mathtt{t})(\mathtt{e}) = \pi(\mathtt{t})(\mathtt{v}) = e$, $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $m_s(x) = (\mathtt{t}, u, y)$, $m_s(y) = (\_, e, \_)$ and $u < e$, then* $(\mathtt{t.rmv}^{10}(\mathtt{e}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq_{\alpha;\alpha} (\mathtt{t.RMV(e)}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}))$;
13. *if there exist x, y, e, v and u such that $\pi(\mathtt{t})(\mathtt{e}) = e$, $\pi(\mathtt{t})(\mathtt{x}) = x$, $\pi(\mathtt{t})(\mathtt{y}) = y$, $\pi(\mathtt{t})(\mathtt{v}) = v$, $m_s(x) = (\mathtt{t}, u, y)$, $m_s(y) =$*

$(\_, v, \_)$ *and* $u < e < v$*, then*
$(\texttt{t.rmv}^{10}(\texttt{e}), \sigma, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha} (\texttt{t}.\textbf{skip}, \Sigma, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$*;*

14. *if there exist* x*,* y*,* e *and* u *such that* $\pi(\texttt{t})(\texttt{e}) = \pi(\texttt{t})(\texttt{v}) = e$*,* $\pi(\texttt{t})(\texttt{x}) = x$*,* $\pi(\texttt{t})(\texttt{y}) = y$*,* $m_s(x) = (\texttt{t}, u, y)$*,* $m_s(y) = (\_, e, \_)$ *and* $u < e$*, then*
$(\texttt{t.rmv}^{11}(\texttt{e}), \sigma, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha} (\texttt{t.RMV}(\texttt{e}), \Sigma, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$*;*
15. *if there exist* x*,* y*,* e *and* u *such that* $\pi(\texttt{t})(\texttt{e}) = \pi(\texttt{t})(\texttt{v}) = e$*,* $\pi(\texttt{t})(\texttt{x}) = x$*,* $\pi(\texttt{t})(\texttt{y}) = y$*,* $m_s(x) = (\texttt{t}, u, y)$*,* $m_s(y) = (\texttt{t}, e, \_)$ *and* $u < e$*, then*
$(\texttt{t.rmv}^{12}(\texttt{e}), \sigma, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha} (\texttt{t.RMV}(\texttt{e}), \Sigma, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$*;*
16. *if there exist* x*,* y*,* z*,* e *and* u *such that* $\pi(\texttt{t})(\texttt{e}) = \pi(\texttt{t})(\texttt{v}) = e$*,* $\pi(\texttt{t})(\texttt{x}) = x$*,* $\pi(\texttt{t})(\texttt{y}) = y$*,* $\pi(\texttt{t})(\texttt{z}) = z$*,* $m_s(x) = (\texttt{t}, u, y)$*,* $m_s(y) = (\texttt{t}, e, z)$ *and* $u < e$*, then*
$(\texttt{t.rmv}^{13}(\texttt{e}), \sigma, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha} (\texttt{t.RMV}(\texttt{e}), \Sigma, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$*;*
17. *if there exist* x*,* y*,* z*,* e *and* u *such that* $\pi(\texttt{t})(\texttt{e}) = \pi(\texttt{t})(\texttt{v}) = e$*,* $\pi(\texttt{t})(\texttt{x}) = x$*,* $\pi(\texttt{t})(\texttt{y}) = y$*,* $\pi(\texttt{t})(\texttt{z}) = z$*,* $\pi(\texttt{t})(y) = (\texttt{t}, e, z)$*,* $m_s(x) = (\texttt{t}, u, z)$ *and* $u < e$*, then*
$(\texttt{t.rmv}^{14}(\texttt{e}), \sigma, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha} (\texttt{t}.\textbf{skip}, \Sigma, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$*;*
18. *if there exist* y *such that* $\pi(\texttt{t})(\texttt{y}) = y$ *and* $\pi(\texttt{t})(y) = (\_, \_, \_)$*, then*
$(\texttt{t.rmv}^{15}(\texttt{e}), \sigma, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha} (\texttt{t.RMV}(\texttt{e}), \Sigma, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$*;*
19. *if there exists* x *such that* $\pi(\texttt{t})(\texttt{x}) = x$ *and* $m_s(x) = (\texttt{t}, \_, \_)$*, then*
$(\texttt{t.rmv}^{16}(\texttt{e}), \sigma, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha} (\texttt{t}.\textbf{skip}, \Sigma, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$*;*
20. $(\texttt{t}.\textbf{skip}, \sigma, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha} (\texttt{t}.\textbf{skip}, \Sigma, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$*;*

Proof:. By co-induction. □

D.5 Strength Reduction and Induction Variable Elimination

Target-Level C_2 ⇐ Medium-Level C_1 ⇐ Source-Level C

```
Target-Level C2        Medium-Level C1        Source-Level C
local k, r;            local i, k;            local i;
k := 0;                i := 0;                i := 0;
r := 6*n;              k := 0;                while(i<n) {
while(k<r) {      <=   while(i<n) {      <=     x := x+6*i;
  x := x+k;              x := x+k;              i := i+1;
  k := k+6;              i := i+1;            }
}                        k := k+6;
                       }
```

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.

$$\begin{aligned}
\mathcal{R}_2 &\triangleq \{(\sigma_2, \sigma_2') \mid \sigma_2(\texttt{k}) = \sigma_2'(\texttt{k}) \wedge \sigma_2(\texttt{r}) = \sigma_2'(\texttt{r}) \wedge \sigma_2(\texttt{n}) = \sigma_2'(\texttt{n})\}\\
\mathcal{R}_1 &\triangleq \{(\sigma_1, \sigma_1') \mid \sigma_1(\texttt{i}) = \sigma_1'(\texttt{i}) \wedge \sigma_1(\texttt{k}) = \sigma_1'(\texttt{k})\}\\
\mathcal{R} &\triangleq \{(\sigma, \sigma') \mid \sigma(\texttt{i}) = \sigma'(\texttt{i})\}\\
\mathcal{G} &\triangleq \mathsf{True}
\end{aligned}$$

Correctness of the two transformations are formalized as follows:

$$(C_2, \mathcal{R}_2, \mathcal{G}) \preceq_{\alpha;\alpha\ltimes\alpha} (C_1, \mathcal{R}_1, \mathcal{G}),\ (C_1, \mathcal{R}_1, \mathcal{G}) \preceq_{\beta;\beta\ltimes\beta} (C, \mathcal{R}, \mathcal{G})$$

where

$$\begin{aligned}
\alpha &\triangleq \{(\sigma_2, \sigma_1) \mid \sigma_2(\texttt{k}) = \sigma_1(\texttt{k}) \wedge \sigma_2(\texttt{n}) = \sigma_1(\texttt{n}) \wedge \sigma_2(\texttt{x}) = \sigma_1(\texttt{x})\}\\
\beta &\triangleq \{(\sigma_1, \sigma) \mid \sigma_1(\texttt{i}) = \sigma(\texttt{i}) \wedge \sigma_1(\texttt{n}) = \sigma(\texttt{n}) \wedge \sigma_1(\texttt{x}) = \sigma(\texttt{x})\}
\end{aligned}$$

The proofs are not difficult by the RGSim definition or by the optimization rules.

```
C1                        C2
t1 := y;                  t2 := x;
if (t1 = 1)               if (t2 = 1)
  x := 1;          ||       y := 1;
t1 := x;                  t2 := y;
print(t1);                print(t2);
```

(a) Source Code

```
C'1                       C'2
t1 := x;           ||     t2 := y;
print(t1);                print(t2);
```

(b) Target Code

Figure 23. Dead Code Elimination Example

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

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

where

$$\beta \circ \alpha = \{(\sigma_2, \sigma) \mid \sigma_2(\texttt{n}) = \sigma(\texttt{n}) \wedge \sigma_2(\texttt{x}) = \sigma(\texttt{x})\}.$$

That is, the optimization phases are correct when the source program is executed in an environment that does not change `i` nor `n`.

D.6 Dead Code Elimination for Data-Race-Free Programs

As shown in Figure 23, C_1 and C_2 are transformed to C_1' and C_2' respectively, assuming the values of `x` and `y` are both 0 in the initial state.

$$\begin{aligned}
\alpha &\triangleq \{(\sigma_T, \sigma_S) \mid \sigma_T(\texttt{x}) = \sigma_S(\texttt{x}) \wedge \sigma_T(\texttt{y}) = \sigma_S(\texttt{y})\}\\
\zeta &\triangleq \{(\sigma_T, \sigma_S) \mid \sigma_T(\texttt{x}) = \sigma_S(\texttt{x}) = 0 \wedge \sigma_T(\texttt{y}) = \sigma_S(\texttt{y}) = 0\}
\end{aligned}$$

We assume $C_1 \parallel C_2$ is a closed program. Thus correctness of the transformation is formalized as follows:

$$(C_1' \parallel C_2', \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\zeta\ltimes\zeta} (C_1 \parallel C_2, \mathsf{Id}, \mathsf{True}) \qquad \text{(D.1)}$$

The transformation is syntax-directed, so we decompose (D.1) into single threads. Due to the symmetry between C_1 and C_2, we only prove:

$$(C_1', \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\zeta} (C_1, \mathcal{R}_1, \mathcal{G}_1)$$

where

$$\begin{aligned}
\mathcal{R}_1 &\triangleq \{(\sigma, \sigma') \mid \sigma(\texttt{t1}) = \sigma'(\texttt{t1}) \wedge \sigma(\texttt{x}) = \sigma'(\texttt{x}) \wedge \sigma(\texttt{y}) = \sigma'(\texttt{y})\}\\
\mathcal{G}_1 &\triangleq \{(\sigma, \sigma') \mid \sigma(\texttt{t2}) = \sigma'(\texttt{t2}) \wedge \sigma(\texttt{x}) = \sigma'(\texttt{x}) \wedge \sigma(\texttt{y}) = \sigma'(\texttt{y})\}
\end{aligned}$$

The proof is immediate by the RGSim definition. Since the other thread just guarantees $\mathcal{R}_1$ when executed in an environment satisfying $\mathcal{G}_1$, by the PAR rule of RGSim, we can get (D.1).

E. Verification of Boehm *et al.* Concurrent GC

E.1 The High-Level and Low-Level Machines

The high-level and low-level languages and state models are presented in Figures 14 and 15 respectively.

- High-level mutators can use $x := y.\mathrm{id}$ to read a field of an object, $x.\mathrm{id} := \mathbb{E}$ to write the value of $\mathbb{E}$ to a field of an object and $x :=$ ***new***$()$ to allocate a new object. If the instruction $x.\mathrm{id} := \mathbb{E}$ updates a pointer field (*i.e.*, $\mathrm{id} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}$), then it will be transformed to a write barrier (shown in Figure 13). The write barrier first modifies the pointer field and then sets the `dirty` field. Here we use an 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, which is write-only and can be safely deleted after the proof is completed.

$$\frac{\mathrm{S}(x)=\mathrm{V}' \quad [\![\mathbb{E}]\!]_\mathrm{S}=\mathrm{V} \quad \mathsf{same\_type}(\mathrm{V},\mathrm{V}')}{(x:=\mathbb{E},(\mathrm{S},\mathrm{H}))\longrightarrow(\textbf{\textit{skip}},(\mathrm{S}\{x\leadsto\mathrm{V}\},\mathrm{H}))} \qquad \frac{x\notin dom(\mathrm{S}) \quad \text{or} \quad [\![\mathbb{E}]\!]_\mathrm{S}=\bot \quad \text{or} \quad \neg\mathsf{same\_type}(\mathrm{S}(x),[\![\mathbb{E}]\!]_\mathrm{S})}{(x:=\mathbb{E},(\mathrm{S},\mathrm{H}))\longrightarrow\textbf{abort}}$$

$$\frac{\mathrm{S}(y)=l \quad \mathrm{H}(l)(\mathrm{id})=\mathrm{V} \quad \mathrm{S}(x)=\mathrm{V}' \quad \mathsf{same\_type}(\mathrm{V},\mathrm{V}')}{(x:=y.\mathrm{id},(\mathrm{S},\mathrm{H}))\longrightarrow(\textbf{\textit{skip}},(\mathrm{S}\{x\leadsto\mathrm{V}\},\mathrm{H}))}$$

$$\frac{y\notin dom(\mathrm{S}) \quad \text{or} \quad \mathrm{S}(y)\notin dom(\mathrm{H}) \quad \text{or} \quad x\notin dom(\mathrm{S}) \quad \text{or} \quad \neg\mathsf{same\_type}(\mathrm{S}(x),\mathrm{H}(\mathrm{S}(y))(\mathrm{id}))}{(x:=y.\mathrm{id},(\mathrm{S},\mathrm{H}))\longrightarrow\textbf{abort}}$$

$$\frac{\mathrm{S}(x)=l \quad \mathrm{H}(l)=\mathrm{O} \quad [\![\mathbb{E}]\!]_\mathrm{S}=\mathrm{V} \quad \mathrm{O}(\mathrm{id})=\mathrm{V}' \quad \mathsf{same\_type}(\mathrm{V},\mathrm{V}')}{(x.\mathrm{id}:=\mathbb{E},(\mathrm{S},\mathrm{H}))\longrightarrow(\textbf{\textit{skip}},(\mathrm{S},\mathrm{H}\{l\leadsto\mathrm{O}\{\mathrm{id}\leadsto\mathrm{V}\}\}))}$$

$$\frac{x\notin dom(\mathrm{S}) \quad \text{or} \quad \mathrm{S}(x)\notin dom(\mathrm{H}) \quad \text{or} \quad [\![\mathbb{E}]\!]_\mathrm{S}=\bot \quad \text{or} \quad \neg\mathsf{same\_type}(\mathrm{H}(\mathrm{S}(x))(\mathrm{id}),[\![\mathbb{E}]\!]_\mathrm{S})}{(x.\mathrm{id}:=\mathbb{E},(\mathrm{S},\mathrm{H}))\longrightarrow\textbf{abort}}$$

$$\frac{l\notin dom(\mathrm{H}) \quad l\neq\textbf{nil} \quad \mathrm{S}(x)=l'}{(x:=\textbf{\textit{new}}(),(\mathrm{S},\mathrm{H}))\longrightarrow(\textbf{\textit{skip}},(\mathrm{S}\{x\leadsto l\},\mathrm{H}\uplus\{l\leadsto\{\mathrm{pt}_1\leadsto\textbf{nil},\ldots,\mathrm{pt}_m\leadsto\textbf{nil},\mathrm{data}\leadsto 0\}\}))}$$

$$\frac{\neg(\exists l.l\notin dom(\mathrm{H})\wedge l\neq\textbf{nil}) \quad \mathrm{S}(x)=l'}{(x:=\textbf{\textit{new}}(),(\mathrm{S},\mathrm{H}))\longrightarrow(\textbf{\textit{skip}},(\mathrm{S}\{x\leadsto\textbf{nil}\},\mathrm{H}))} \qquad \frac{x\notin dom(\mathrm{S}) \quad \text{or} \quad \neg\exists l.\mathrm{S}(x)=l}{(x:=\textbf{\textit{new}}(),(\mathrm{S},\mathrm{H}))\longrightarrow\textbf{abort}}$$

$$\frac{(\mathbb{C},(\mathrm{S},\mathrm{H}))\longrightarrow(\mathbb{C}',(\mathrm{S}',\mathrm{H}'))}{(\mathsf{t}.\mathbb{C},(\Pi\uplus\{\mathsf{t}\leadsto\mathrm{S}\},\mathrm{H}))\longrightarrow(\mathsf{t}.\mathbb{C}',(\Pi\uplus\{\mathsf{t}\leadsto\mathrm{S}'\},\mathrm{H}'))} \qquad \frac{(\mathbb{C},(\mathrm{S},\mathrm{H}))\longrightarrow\textbf{abort}}{(\mathsf{t}.\mathbb{C},(\Pi\uplus\{\mathsf{t}\leadsto\mathrm{S}\},\mathrm{H}))\longrightarrow\textbf{abort}}$$

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

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

(a) Selected Operational Semantics Rules

$$\begin{array}{lcl}
\mathsf{root}(\mathsf{t},S) & \triangleq & \lambda\Sigma.\ \Sigma=(\Pi\uplus\{\mathsf{t}\leadsto\mathrm{S}_\mathsf{t}\},\mathrm{H})\wedge S=\{l\mid\exists x.\mathrm{S}_\mathsf{t}(x)=l\}\\
\mathsf{edge}(l_1,l_2) & \triangleq & \lambda\Sigma.\ \Sigma=(\Pi,\mathrm{H})\wedge\exists\mathrm{id}\in\{\mathrm{pt}_1,\ldots,\mathrm{pt}_m\}.\ \mathrm{H}(l_1)(\mathrm{id})=l_2\\
\mathsf{path}_0(l_1,l_2) & \triangleq & \lambda\Sigma.\ l_1=l_2\\
\mathsf{path}_{k+1}(l_1,l_2) & \triangleq & \lambda\Sigma.\ \exists l_3.\ \mathsf{edge}(l_1,l_3)(\Sigma)\wedge\mathsf{path}_k(l_3,l_2)(\Sigma)\\
\mathsf{path}(l_1,l_2) & \triangleq & \lambda\Sigma.\ \exists k.\ \mathsf{path}_k(l_1,l_2)(\Sigma)\\
\mathsf{reachable}(\mathsf{t},l) & \triangleq & \lambda\Sigma.\ \exists S.\ \mathsf{root}(\mathsf{t},S)(\Sigma)\wedge\exists l'\in S.\ \mathsf{path}(l',l)(\Sigma)\wedge l\neq\textbf{nil}\\
\mathsf{reachable}(l) & \triangleq & \exists\mathsf{t}\in[1..N].\ \mathsf{reachable}(\mathsf{t},l)\\
\mathsf{AbsGCStep} & \triangleq & \{((\Pi,\mathrm{H}),(\Pi,\mathrm{H}'))\mid\forall l.\ \mathsf{reachable}(l)(\Pi,\mathrm{H})\Longrightarrow\mathrm{H}(l)=\mathrm{H}'(l)\}
\end{array}$$

(b) Definition of AbsGCStep

Figure 24. A High-level Garbage-Collected Machine

- 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 := **get_root**(y) allows the GC to read the values of all the pointer variables in the thread y's store at once and **free**(x) allows to reclaim an object. The stop-the-world phase can be implemented by **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 `color` and `dirty` for the collection. We give each object a dirty card whose value can be 0 (not dirty) or 1 (dirty). The `color` field has three possible values and is used for two purposes: for marking, we use `BLACK` for a marked object and `WHITE` for an unmarked one; and for allocation, we use `BLUE` for an unallocated object which will neither be traced nor be reclaimed, but can be allocated later. New objects are created `BLACK`, and when reclaiming an object, we just set its color to `BLUE`.
- The 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 **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:

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

  which satisfies $\mathsf{Loc2Int}(\textbf{nil}) = 0$.

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

$$[\![n]\!]_{(s,b)} = \begin{cases} n & \text{if } b = 0 \text{ or } b = 2 \\ 0 & \text{if } b = 1 \text{ and } n = 0 \\ \bot & \text{otherwise} \end{cases}$$

$$[\![x]\!]_{(s,b)} = \begin{cases} n & \text{if } s(x) = (n, b') \text{ and } (b = b' \vee b = 2) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![E_1 + E_2]\!]_{(s,b)} = \begin{cases} n_1 + n_2 & \text{if } [\![E_1]\!]_{(s,b)} = n_1 \text{ and } [\![E_2]\!]_{(s,b)} = n_2 \text{ and } (b = 0 \vee b = 2) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![E_1 = E_2]\!]_{(s,b)} = \begin{cases} \textbf{true} & \text{if } [\![E_1]\!]_{(s,b)} = n_1 \text{ and } [\![E_2]\!]_{(s,b)} = n_2 \text{ and } n_1 = n_2 \text{ and } (b = 0 \vee b = 2) \\ \textbf{false} & \text{if } [\![E_1]\!]_{(s,b)} = n_1 \text{ and } [\![E_2]\!]_{(s,b)} = n_2 \text{ and } n_1 \neq n_2 \text{ and } (b = 0 \vee b = 2) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\texttt{is\_empty}(x)]\!]_{(s,b)} = \begin{cases} \textbf{true} & \text{if } b = 0 \text{ and } s(x) = \epsilon \\ \textbf{false} & \text{if } b = 0 \text{ and } s(x) = n :: A \\ \bot & \text{otherwise} \end{cases}$$

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

```
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;
  }
}
```

$$\frac{\texttt{tid} \in [1..N] \quad s(x) = (n', b) \quad [\![E]\!]_{(s,b)} = n}{(\texttt{tid}.(x := E), (s, h)) \longrightarrow (\texttt{tid}.\textbf{skip}, (s\{x \rightsquigarrow (n, b)\}, h))} \qquad \frac{\texttt{tid} \in [1..N] \quad (x \notin \mathit{dom}(s) \quad \text{or} \quad [\![E]\!]_{(s,\textbf{snd}(s(x)))} = \perp)}{(\texttt{tid}.(x := E), (s, h)) \longrightarrow \textbf{abort}}$$

$$\frac{\texttt{tid} = \texttt{t}_{gc} \quad s(x) = (n', b) \quad [\![E]\!]_{(s,2)} = n}{(\texttt{tid}.(x := E), (s, h)) \longrightarrow (\texttt{tid}.\textbf{skip}, (s\{x \rightsquigarrow (n, b)\}, h))} \qquad \frac{\texttt{tid} = \texttt{t}_{gc} \quad (x \notin \mathit{dom}(s) \quad \text{or} \quad [\![E]\!]_{(s,2)} = \perp)}{(\texttt{tid}.(x := E), (s, h)) \longrightarrow \textbf{abort}}$$

$$\frac{s(y) = (n_y, 1) \quad h(n_y)(id) = n \quad s(x) = (n_x, b) \quad id \in \{\text{pt}_1, \ldots, \text{pt}_m\} \Longrightarrow b = 1 \quad id \in \{\text{data}\} \Longrightarrow b = 0}{(x := y.id, (s, h)) \longrightarrow (\textbf{skip}, (s\{x \rightsquigarrow (n, b)\}, h))}$$

$$\frac{\begin{array}{c} y \notin \mathit{dom}(s) \quad \text{or} \quad \textbf{fst}(s(y)) \notin \mathit{dom}(h) \quad \text{or} \quad \textbf{snd}(s(y)) \neq 1 \\ \text{or} \quad x \notin \mathit{dom}(s) \quad \text{or} \quad id \in \{\text{pt}_1, \ldots, \text{pt}_m\} \Longrightarrow \textbf{snd}(s(x)) \neq 1 \quad \text{or} \quad id \in \{\text{data}\} \Longrightarrow \textbf{snd}(s(x)) \neq 0 \end{array}}{(x := y.id, (s, h)) \longrightarrow \textbf{abort}}$$

$$\frac{\begin{array}{c} s(x) = (n, 1) \quad h(n) = o \quad id \in \{\text{pt}_1, \ldots, \text{pt}_m\} \Longrightarrow [\![E]\!]_{(s,1)} = n' \\ id \in \{\text{data}\} \Longrightarrow [\![E]\!]_{(s,0)} = n' \quad id \in \{\texttt{color}, \texttt{dirty}\} \Longrightarrow [\![E]\!]_{(s,2)} = n' \end{array}}{(x.id := E, (s, h)) \longrightarrow (\textbf{skip}, (s, h\{n \rightsquigarrow o\{id \rightsquigarrow n'\}\}))}$$

$$\frac{\begin{array}{c} x \notin \mathit{dom}(s) \quad \text{or} \quad \textbf{fst}(s(x)) \notin \mathit{dom}(h) \quad \text{or} \quad \textbf{snd}(s(x)) \neq 1 \quad \text{or} \quad id \in \{\text{pt}_1, \ldots, \text{pt}_m\} \Longrightarrow [\![E]\!]_{(s,1)} = \perp \\ \text{or} \quad id \in \{\text{data}\} \Longrightarrow [\![E]\!]_{(s,0)} = \perp \quad \text{or} \quad id \in \{\texttt{color}, \texttt{dirty}\} \Longrightarrow [\![E]\!]_{(s,2)} = \perp \end{array}}{(x.id := E, (s, h)) \longrightarrow \textbf{abort}}$$

$$\frac{\texttt{tid} = \texttt{t}_{gc} \quad s(y) = (\texttt{t}, 0) \quad \pi(\texttt{t}) = s_{\texttt{t}} \quad s(x) = (n', 0) \quad S = \{n \mid \exists x. s_{\texttt{t}}(x) = (n, 1)\} \quad s' = s\{x \rightsquigarrow (S, 0)\}}{(\texttt{tid}.(x := \textbf{get\_root}(y)), (\pi \uplus \{\texttt{tid} \rightsquigarrow s\}, h)) \longrightarrow (\texttt{tid}.\textbf{skip}, (\pi \uplus \{\texttt{tid} \rightsquigarrow s'\}, h))}$$

$$\frac{\texttt{tid} \neq \texttt{t}_{gc} \quad \text{or} \quad x \notin \mathit{dom}(s) \quad \text{or} \quad \textbf{snd}(s(x)) \neq 0 \quad \text{or} \quad y \notin \mathit{dom}(s) \quad \text{or} \quad \textbf{snd}(s(y)) \neq 0 \quad \text{or} \quad \textbf{fst}(s(y)) \notin \mathit{dom}(\pi)}{(\texttt{tid}.(x := \textbf{get\_root}(y)), (\pi \uplus \{\texttt{tid} \rightsquigarrow s\}, h)) \longrightarrow \textbf{abort}}$$

$$\frac{x \in \mathit{dom}(s) \quad s(y) = (\{\}, 0)}{(\textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C, (s, h)) \longrightarrow (\textbf{skip}, (s, h))} \qquad \frac{x \notin \mathit{dom}(s) \quad \text{or} \quad \textbf{fst}(s(y)) \notin \mathit{Set}(\mathit{Val}) \quad \text{or} \quad \textbf{snd}(s(y)) \neq 0}{(\textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C, (s, h)) \longrightarrow \textbf{abort}}$$

$$\frac{s(x) = (n, b) \quad s(y) = (\{n_1, \ldots, n_k\}, 0)}{(\textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C, (s, h)) \longrightarrow (C; y := y - \{x\}; \textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C, (s\{x \rightsquigarrow (n_1, b)\}, h))}$$

$$\frac{(C, (s, h)) \longrightarrow^* (\textbf{skip}, (s', h'))}{(\textbf{atomic}\{C\}, (s, h)) \longrightarrow (\textbf{skip}, (s', h'))} \qquad \frac{(C, (s, h)) \longrightarrow^* \textbf{abort}}{(\textbf{atomic}\{C\}, (s, h)) \longrightarrow \textbf{abort}}$$

$$\frac{\texttt{tid} \in [1..N] \quad s(x) = (n', 1) \quad h(n)(\texttt{color}) = \texttt{BLUE}}{(\texttt{tid}.(x := \textbf{new}()), (s, h)) \longrightarrow (\textbf{skip}, (s\{x \rightsquigarrow (n, 1)\}, h\{n \rightsquigarrow \{\text{pt}_1 \rightsquigarrow 0, \ldots, \text{pt}_m \rightsquigarrow 0, \text{data} \rightsquigarrow 0, \texttt{color} \rightsquigarrow \texttt{BLACK}, \texttt{dirty} \rightsquigarrow 0\}\}))}$$

$$\frac{\texttt{tid} \in [1..N] \quad s(x) = (n', 1) \quad \neg(\exists n.\, h(n)(\texttt{color}) = \texttt{BLUE})}{(\texttt{tid}.(x := \textbf{new}()), (s, h)) \longrightarrow (\textbf{skip}, (s\{x \rightsquigarrow (0, 1)\}, h))} \qquad \frac{\texttt{tid} \notin [1..N] \quad \text{or} \quad x \notin \mathit{dom}(s) \quad \text{or} \quad \textbf{snd}(s(x)) \neq 1}{(\texttt{tid}.(x := \textbf{new}()), (s, h)) \longrightarrow \textbf{abort}}$$

$$\frac{s(x) = (n, 1) \quad h(n) = o}{(\textbf{free}(x), (s, h)) \longrightarrow (\textbf{skip}, (s, h\{n \rightsquigarrow o\{\texttt{color} \rightsquigarrow \texttt{BLUE}\}\}))} \qquad \frac{x \notin \mathit{dom}(s) \quad \text{or} \quad \textbf{fst}(s(x)) \notin \mathit{dom}(h) \quad \text{or} \quad \textbf{snd}(s(x)) \neq 1}{(\textbf{free}(x), (s, h)) \longrightarrow \textbf{abort}}$$

$$\frac{s(x) = (n', b) \quad s(y) = (A, 0)}{(\textbf{push}(x, y), (s, h)) \longrightarrow (\textbf{skip}, (s\{y \rightsquigarrow (n' :: A, 0)\}, h))}$$

$$\frac{x \notin \mathit{dom}(s) \quad \text{or} \quad y \notin \mathit{dom}(s) \quad \text{or} \quad \textbf{fst}(s(y)) \notin \mathit{Seq}(\mathit{Val}) \quad \text{or} \quad \textbf{snd}(s(y)) \neq 0}{(\textbf{push}(x, y), (s, h)) \longrightarrow \textbf{abort}}$$

$$\frac{s(x) = (n', b) \quad s(y) = (n :: A, 0)}{(x := \textbf{pop}(y), (s, h)) \longrightarrow (\textbf{skip}, (s\{x \rightsquigarrow (n, b), y \rightsquigarrow (A, 0)\}, h))} \qquad \frac{x \notin \mathit{dom}(s) \quad \text{or} \quad \neg\exists n, A. s(y) = (n :: A, 0)}{(x := \textbf{pop}(y), (s, h)) \longrightarrow \textbf{abort}}$$

Figure 26. Selected Operational Semantics Rules on the Low-Level Machine

```
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 *LState* $\times$ *LvMap* pairs with identity transitions on *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 $h_1 \oplus h_2$ to denote the union when their domains of $\mathsf{uncurry}(h_1)$ and $\mathsf{uncurry}(h_2)$ are disjoint. The disjoint union of states is defined based on the disjoint unions of the shared heaps and the stores for each thread.

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). $\mathtt{Sta}(p, a)$ holds iff, for all σ, i, σ' 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 σ and i such that $p(\sigma, i)$, the followings are true:

1. $\neg((C, \sigma) \longrightarrow^* \mathbf{abort})$;
2. $\neg\exists C', \sigma', e.\ ((C, \sigma) \xrightarrow{e}{}^* (C', \sigma'))$;
3. if $(C, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$, 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 sequential separation logic and omitted here.

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

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

(i) $\neg((x := \mathbf{get\_root}(y), \sigma) \longrightarrow^* \mathbf{abort})$;

(ii) $\neg\exists C', \sigma', e.\ ((x := \mathbf{get\_root}(y), \sigma) \xrightarrow{e}{}^* (C', \sigma'))$;

(iii) if $(x := \mathbf{get\_root}(y), \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$, then $q(\sigma', i)$.

Suppose $\sigma = (\pi' \uplus \{\mathtt{t}_{\mathrm{gc}} \leadsto s\}, h)$. Since $\sigma \models p$, there exists $\mathtt{t}$ and n' such that $s(y) = (\mathtt{t}, 0)$, $\mathtt{t} \in [1..N]$, $s(x) = (n', 0)$ and $\pi' = \pi \uplus \{\mathtt{t} \leadsto s_{\mathtt{t}}\}$. Then $(x := \mathbf{get\_root}(y), \sigma) \longrightarrow (\mathbf{skip}, \sigma')$, where $\sigma' = (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \leadsto s', \mathtt{t} \leadsto s_{\mathtt{t}}\}, h)$, $S = \{n \mid \exists x. s_{\mathtt{t}}(x) = (n, 1)\}$ and $s' = s\{x \leadsto (S, 0)\}$. 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* $p \triangleq \bullet; x \Vdash x.\mathtt{color} \mapsto \_$ *and* $q \triangleq \bullet; x \Vdash x.\mathtt{color} \mapsto \mathtt{BLUE}$*. If* $\{p\}\mathbf{free}(x)\{q\}$*, then* $\models \{p\}\mathbf{free}(x)\{q\}$.

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

(i) $\neg((\mathbf{free}(x), \sigma) \longrightarrow^* \mathbf{abort})$;

(ii) $\neg\exists C', \sigma', e.\ ((\mathbf{free}(x), \sigma) \xrightarrow{e}{}^* (C', \sigma'))$;

(iii) if $(\mathbf{free}(x), \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$, then $q(\sigma', i)$.

Suppose $\sigma = (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \leadsto s\}, h)$. Since $\sigma \models p$, there exists n and o such that $s(x) = (n, 1)$ and $h(n) = o$. Then $(\mathbf{free}(x), \sigma) \longrightarrow (\mathbf{skip}, \sigma')$, where $\sigma' = (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \leadsto s\}, h\{n \leadsto o\{\mathtt{color} \leadsto \mathtt{BLUE}\}\})$. Thus (i) and (ii) are proved. Also $q(\sigma', i)$ holds, *i.e.*, (iii) is proved. □

Suppose the (FOREACH) rule is applied to derive $\{p * \mathsf{own}(x)\}\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C\{p * \mathsf{own}(x) \wedge y = \phi\}$. We want to prove $\models \{p * \mathsf{own}(x)\}\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C\{p * \mathsf{own}(x) \wedge y = \phi\}$. By inversion of the (FOREACH) rule, we know $p \Longrightarrow \mathsf{own}_{\mathrm{np}}(y)$ and $\{p * \mathsf{own}(x) \wedge x \in y\}C; y := y - \{x\}\{p * \mathsf{own}(x)\}$.

Lemma 40 (FOREACH-Sound). *If* $p \Longrightarrow \mathsf{own}_{\mathrm{np}}(y)$ *and* $\models \{p * \mathsf{own}(x) \wedge x \in y\}C; y := y - \{x\}\{p * \mathsf{own}(x)\}$*, then* $\models \{p * \mathsf{own}(x)\}\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C\{p * \mathsf{own}(x) \wedge y = \phi\}$.

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

(i) $\neg((\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, \sigma) \longrightarrow^n \mathbf{abort})$;

(ii) $\neg\exists C', \sigma', e.\ ((\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, \sigma) \xrightarrow{e}{}^n (C', \sigma'))$;

(iii) if $(\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, \sigma) \longrightarrow^n (\mathbf{skip}, \sigma')$, then $(p * \mathsf{own}(x) \wedge y = \phi)\ (\sigma', i)$.

Suppose $\sigma = (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \leadsto s\}, h)$. Since $\sigma \models p * \mathsf{own}(x)$ and $p \Longrightarrow \mathsf{own}_{\mathrm{np}}(y)$, we know $x \in dom(s)$ and $s(y) = (\_, 0)$. 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 $(\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, \sigma) \longrightarrow (\mathbf{skip}, \sigma')$, then $\sigma' = \sigma$ and $s(y) = (\{\}, 0)$. Thus $(p * \mathsf{own}(x) \wedge y = \phi)(\sigma', i)$, *i.e.*, (iii) is proved.

Inductive Step: Assume (i), (ii) and (iii) are true when $n \leq m$, $m \geq 1$. From the operational semantics of FOREACH, we know $\neg((\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, \sigma) \longrightarrow \mathbf{abort})$ and $\neg\exists C', \sigma', e.\ ((\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, \sigma) \xrightarrow{e} (C', \sigma'))$.
By the assumption, (i) and (ii) are true when $n = m + 1$.
If $(\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, \sigma) \longrightarrow^{m+1} (\mathbf{skip}, \sigma')$, then $s(y) = (\{n_1, \ldots, n_k\}, 0)$ and

$$
\begin{array}{l}
(\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, \sigma) \longrightarrow \\
(C; y := y - \{x\}; \mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, \sigma_1) \\
\ \text{where } \sigma_1 = (\pi \uplus \{\mathtt{t}_{\mathrm{gc}} \leadsto s\{x \leadsto (n_1, b)\}\}, h)
\end{array}
$$

$$
\begin{array}{rrcl}
(\mathit{PVarList}) & O & ::= & \bullet \mid x, O \\
(\mathit{LvMap}) & i & \in & \mathit{LVar} \to \mathit{LVal} \times \{0,1\} \\
(\mathit{StateAssert}) & p, q & \in & \mathit{LState} \times \mathit{LvMap} \to \mathit{Prop} \\
(\mathit{Action}) & a, \mathcal{R}, \mathcal{G} & \in & \mathcal{P}((\mathit{LState} \times \mathit{LvMap}) \times (\mathit{LState} \times \mathit{LvMap})) \to \mathit{Prop}
\end{array}
$$

(a) Syntax of Assertions

$$
\begin{array}{lcl}
B & \triangleq & \lambda(\sigma,i).\ \sigma = (\pi,h) \land [\![B]\!]_{(\pi,i,2)} = \mathbf{true} \\
\mathsf{emp}_h & \triangleq & \lambda(\sigma,i).\ \sigma = (\pi,h) \land \mathit{dom}(h) = \phi \\
\mathsf{t.own}_{\mathrm{np}}(x) & \triangleq & \lambda(\sigma,i).\ \sigma = (\pi,h) \land \pi(\mathsf{t}) = s \land \mathit{dom}(s) = \{x\} \land s(x) = (\_,0) \\
\mathsf{t.own}_{\mathrm{p}}(x) & \triangleq & \lambda(\sigma,i).\ \sigma = (\pi,h) \land \pi(\mathsf{t}) = s \land \mathit{dom}(s) = \{x\} \land s(x) = (\_,1) \\
\mathsf{t.own}(x) & \triangleq & \lambda(\sigma,i).\ \sigma = (\pi,h) \land \pi(\mathsf{t}) = s \land \mathit{dom}(s) = \{x\} \\
p * q & \triangleq & \lambda(\sigma,i).\ \exists\sigma_1,\sigma_2.\ \sigma_1 \oplus \sigma_2 = \sigma \land p(\sigma_1,i) \land q(\sigma_2,i) \\
E_1.\mathit{id} \mapsto E_2 & \triangleq & \lambda(\sigma,i).\ \sigma = (\pi,h) \land \exists n,n'.\ [\![E_1]\!]_{(\pi,i,1)} = n' \land \mathit{dom}(h) = \{n'\} \land h(n')(\mathit{id}) = n \\
 & & \quad \land ([\![E_2]\!]_{(\pi,i,1)} = n \land \mathit{id} = \{\mathrm{pt}_1,\ldots,\mathrm{pt}_m\} \lor [\![E_2]\!]_{(\pi,i,0)} = n \land \mathit{id} = \mathrm{data} \lor [\![E_2]\!]_{(\pi,i,2)} = n \land (\mathit{id} = \mathtt{color} \lor \mathit{id} = \mathtt{dirty})) \\
E_1.\mathit{id} \hookrightarrow E_2 & \triangleq & (E_1.\mathit{id} \mapsto E_2) * \mathbf{true} \\
\exists X.p & \triangleq & \lambda(\sigma,i).\ \exists v.\ p(\sigma, i\{X \rightsquigarrow v\}) \\
O_0; O_1 \Vdash p & \triangleq & (\mathsf{own}_{\mathrm{np}}(x_1) * \ldots * \mathsf{own}_{\mathrm{np}}(x_i) * \mathsf{own}_{\mathrm{p}}(y_1) * \ldots * \mathsf{own}_{\mathrm{p}}(y_j)) \land p \qquad \text{where } O_0 = x_1,\ldots,x_i,\bullet \text{ and } O_1 = y_1,\ldots,y_j,\bullet \\
x \in S & \triangleq & \exists X.S = X \uplus \{x\} \\
\circledast_{x\in S}.p(x) & \triangleq & S = \phi \land \mathsf{emp} \lor \exists z.(S = \{z\} \uplus S') \land (\circledast_{x\in S'}.p(x)) * p(z)
\end{array}
$$

(b) State Assertions

$$
\begin{array}{lcl}
p \ltimes q & \triangleq & \lambda((\sigma,i),(\sigma',i')).\ p(\sigma,i) \land q(\sigma',i) \land i = i' \\
[p] & \triangleq & \lambda((\sigma,i),(\sigma',i')).\ \sigma = \sigma' \land i = i' \land p(\sigma,i)
\end{array}
$$

(c) Actions

$$
\begin{array}{lcl}
f_1 \bot f_2 & \triangleq & \mathit{dom}(f_1) \cap \mathit{dom}(f_2) = \phi \\
f_1 \uplus f_2 & \triangleq & \begin{cases} f_1 \cup f_2 & \text{if } f_1 \bot f_2 \\ \bot & \text{otherwise} \end{cases} \\
h_1 \oplus h_2 & \triangleq & \begin{cases} \mathsf{curry}(\mathsf{uncurry}(h_1) \cup \mathsf{uncurry}(h_2)) & \text{if } \mathsf{uncurry}(h_1) \bot \mathsf{uncurry}(h_2) \\ \bot & \text{otherwise} \end{cases} \\
\sigma_1 \oplus \sigma_2 & \triangleq & \begin{cases} (\{\mathsf{t} \rightsquigarrow (\pi_1(\mathsf{t}) \uplus \pi_2(\mathsf{t})) \mid \mathsf{t} \in \mathit{dom}(\pi_1)\}, h_1 \oplus h_2) \\ \qquad \text{if } \sigma_1 = (\pi_1,h_1) \land \sigma_2 = (\pi_2,h_2) \land \mathit{dom}(\pi_1) = \mathit{dom}(\pi_2) \\ \qquad \land \forall \mathsf{t} \in \mathit{dom}(\pi_1).\ \pi_1(\mathsf{t}) \bot \pi_2(\mathsf{t}) \land \mathsf{uncurry}(h_1) \bot \mathsf{uncurry}(h_2) \\ \bot \qquad \text{otherwise} \end{cases}
\end{array}
$$

(d) Disjoint Unions

Figure 27. Semantics of Basic Assertions

Then $(p * \mathsf{own}(x) \land x \in y)\ (\sigma_1, i)$. If

$$(C; y := y - \{x\}; \textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C, \sigma_1) \longrightarrow^* (\textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C, \sigma_2)$$

then $(p * \mathsf{own}(x))\ (\sigma_2, i)$. Thus if

$$(\textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C, \sigma_2) \longrightarrow^k (\textbf{skip}, \sigma')$$

where $k \leq m$, then $(p * \mathsf{own}(x) \land y = \phi)(\sigma', i)$. (iii) is proved. □

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

Lemma 41. *If $(C, \sigma, \mathcal{R})$ $\mathbf{guarantees}_{n+1}$ $\mathcal{G}$, there does not exist j such that $j < n$ and $(C, \sigma) \overset{\mathcal{R}}{\longmapsto}{}^j$* **abort**.

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

Theorem 42 is proved by induction over the derivation of the judgment $\mathcal{R}; \mathcal{G} \vdash \{p\}C\{q\}$. 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

$$\mathcal{R}; \mathcal{G} \vdash \{p\}\textbf{atomic}\{C\}\{q\}.$$

We want to prove $\mathcal{R}; \mathcal{G} \models \{p\}\textbf{atomic}\{C\}\{q\}$. By inversion of the ATOMIC rule, we know $p \Longrightarrow p'$, $\{p'\}C\{q'\}$, $p' \ltimes q' \Longrightarrow \mathcal{G}$, $q' \Longrightarrow q$ and $\mathsf{Sta}(\{p, q\}, \mathcal{R})$.

Lemma 43 (ATOMIC-Sound). *If $p \Longrightarrow p'$, $\models \{p'\}C\{q'\}$, $p' \ltimes q' \Longrightarrow \mathcal{G}$, $q' \Longrightarrow q$ and $\mathsf{Sta}(\{p, q\}, \mathcal{R})$, then $\mathcal{R}; \mathcal{G} \models \{p\}$***atomic***$\{C\}\{q\}$.*

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

(i) if $(\textbf{atomic}\{C\}, \sigma) \overset{\mathcal{R}}{\longmapsto}{}^*(\textbf{skip}, \sigma')$, then $q\ (\sigma', i)$;

(ii) $\forall n.\ (\textbf{atomic}\{C\}, \sigma, \mathcal{R})\ \mathbf{guarantees}_n\ \mathcal{G}$.

Since $(\textbf{atomic}\{C\}, \sigma) \overset{\mathcal{R}}{\longmapsto}{}^*(\textbf{skip}, \sigma')$, there exist σ_1 and σ_2 such that $(\textbf{atomic}\{C\}, \sigma_1) \longrightarrow (\textbf{skip}, \sigma_2)$, $(\sigma, \sigma_1) \in \mathcal{R}^*$ and $(\sigma_2, \sigma') \in \mathcal{R}^*$. Since $\mathsf{Sta}(p, \mathcal{R})$, we know $p\ (\sigma_1, i)$. Since $p \Longrightarrow p'$, we know $p'\ (\sigma_1, i)$. By the definition for the semantics of sequential rules, we know $q'\ (\sigma_2, i)$. Then $q\ (\sigma_2, i)$. Since $\mathsf{Sta}(q, \mathcal{R})$, we know $q\ (\sigma', i)$. Thus (i) is proved.
If $p'\ (\sigma_1, i)$ and $(\textbf{atomic}\{C\}, \sigma_1) \longrightarrow (\textbf{skip}, \sigma_2)$, then $q'\ (\sigma_2, i)$, thus $(\sigma_1, \sigma_2) \in \mathcal{G}$ because $p' \ltimes q' \Longrightarrow \mathcal{G}$. We can prove (ii) by induction over n. □

$\mathsf{obj}(x)$	$\triangleq$	$x.\mathrm{pt}_1 \mapsto \_ * \ldots * x.\mathrm{pt}_m \mapsto \_ * x.\mathrm{data} \mapsto \_ * x.\mathtt{color} \mapsto \_ * x.\mathtt{dirty} \mapsto \_$
$\mathsf{blueobj}(x)$	$\triangleq$	$x.\mathrm{pt}_1 \mapsto \_ * \ldots * x.\mathrm{pt}_m \mapsto \_ * x.\mathrm{data} \mapsto \_ * x.\mathtt{color} \mapsto \mathtt{BLUE} * x.\mathtt{dirty} \mapsto \_$
$\mathsf{newobj}(x)$	$\triangleq$	$x.\mathrm{pt}_1 \mapsto 0 * \ldots * x.\mathrm{pt}_m \mapsto 0 * x.\mathrm{data} \mapsto 0 * x.\mathtt{color} \mapsto \mathtt{BLACK} * x.\mathtt{dirty} \mapsto 0$
$\mathsf{black}(x)$	$\triangleq$	$x.\mathtt{color} \hookrightarrow \mathtt{BLACK}$
$\mathsf{white}(x)$	$\triangleq$	$x.\mathtt{color} \hookrightarrow \mathtt{WHITE}$
$\mathsf{dirty}(x)$	$\triangleq$	$x.\mathtt{dirty} \hookrightarrow 1$
$\mathsf{not\_blue}(x)$	$\triangleq$	$\exists c.\ (x.\mathtt{color} \hookrightarrow c \wedge c \neq \mathtt{BLUE})$
$\mathsf{not\_white}(x)$	$\triangleq$	$\exists c.\ (x.\mathtt{color} \hookrightarrow c \wedge c \neq \mathtt{WHITE})$
$\mathsf{not\_dirty}(x)$	$\triangleq$	$x.\mathtt{dirty} \hookrightarrow 0$
$\mathsf{instk}(n, A)$	$\triangleq$	$\exists n', A'.\ A = n' :: A' \wedge (n = n' \vee \mathsf{instk}(n, A'))$
$\mathsf{stk\_black}(A)$	$\triangleq$	$\forall x.\ \mathsf{instk}(x, A) \Longrightarrow \mathsf{black}(x)$
$\mathsf{root}(\mathsf{t}, S)$	$\triangleq$	$\lambda(\sigma, i).\ \sigma = (\pi \uplus \{\mathsf{t} \rightsquigarrow s_{\mathsf{t}}\}, h) \wedge S = \{n \mid \exists x. s_{\mathsf{t}}(x) = (n, 1) \wedge x \neq \mathsf{aux}\}$
$\mathsf{edge}(x, y)$	$\triangleq$	$\exists id \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}.\ (x.id \hookrightarrow y)$
$\mathsf{path}_k(x, y)$	$\triangleq$	$\begin{cases} x = y & \text{if } k = 0 \\ \exists z.\ \mathsf{edge}(x, z) \wedge \mathsf{path}_{k-1}(z, y) & \text{if } k > 0 \end{cases}$
$\mathsf{path}(x, y)$	$\triangleq$	$\exists k.\ \mathsf{path}_k(x, y)$
$\mathsf{reachable}(\mathsf{t}, x)$	$\triangleq$	$\exists S, y.\ \mathsf{root}(\mathsf{t}, S) \wedge y \in S \wedge \mathsf{path}(y, x) \wedge x \neq 0$
$\mathsf{reachable}(x)$	$\triangleq$	$\exists t \in [1..N].\ \mathsf{reachable}(t, x)$
$\mathsf{wfstate}$	$\triangleq$	$\circledast_{x \in [1..M]}.\mathsf{obj}(x) * \mathbf{true} \wedge (\forall x.\ \mathsf{reachable}(x) \Longrightarrow \mathsf{not\_blue}(x))$
$\mathsf{white\_edge}(x, y)$	$\triangleq$	$\exists id \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}.\ (x.id \hookrightarrow y \wedge \mathsf{white}(y))$
$\mathsf{white\_path}_k(x, y)$	$\triangleq$	$\begin{cases} x = y & \text{if } k = 0 \\ \exists z.\ \mathsf{white\_edge}(x, z) \wedge \mathsf{white\_path}_{k-1}(z, y) & \text{if } k > 0 \end{cases}$
$\mathsf{white\_path}(x, y)$	$\triangleq$	$\exists k.\ \mathsf{white\_path}_k(x, y)$
$\mathsf{wwp}(x, y)$	$\triangleq$	$\mathsf{white}(x) \wedge \mathsf{white\_path}(x, y)$
$\mathsf{rt\_wp}(\mathsf{t}, x)$	$\triangleq$	$\exists S, y.\ \mathsf{root}(\mathsf{t}, S) \wedge y \in S \wedge \mathsf{wwp}(y, x)$
$\mathsf{rt\_wp}(x)$	$\triangleq$	$\exists t \in [1..N].\ \mathsf{rt\_wp}(t, x)$
$\mathsf{dt\_bwp}(x, y)$	$\triangleq$	$\mathsf{black}(x) \wedge \mathsf{dirty}(x) \wedge \mathsf{white\_path}(x, y)$
$\mathsf{stk\_bwp}(x, y, A)$	$\triangleq$	$\mathsf{black}(x) \wedge \mathsf{instk}(x, A) \wedge \mathsf{white\_path}(x, y)$
$\mathsf{reach\_inv}$	$\triangleq$	$\forall x.\ \mathsf{reachable}(x) \wedge \mathsf{white}(x) \implies \mathsf{rt\_wp}(x) \vee \exists x'.\mathsf{dt\_bwp}(x', x)$
$\mathsf{reach\_stk}(A)$	$\triangleq$	$\forall x.\ \mathsf{reachable}(x) \wedge \mathsf{white}(x) \implies \mathsf{rt\_wp}(x) \vee \exists x'.\mathsf{dt\_bwp}(x', x) \vee \exists x'.\mathsf{stk\_bwp}(x', x, A)$
$\mathsf{reach\_rtnw\_stk}(A)$	$\triangleq$	$\forall x.\ \mathsf{reachable}(x) \wedge \mathsf{white}(x) \implies \exists x'.\mathsf{dt\_bwp}(x', x) \vee \exists x'.\mathsf{stk\_bwp}(x', x, A)$
$\mathsf{popped\_bwp}(x, y, S_{id})$	$\triangleq$	$\mathsf{black}(x) \wedge \exists id, z.\ id \in S_{id} \subseteq \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \wedge x.id \hookrightarrow z \wedge \mathsf{wwp}(z, y)$
$\mathsf{reach\_tomk}(A, x_p, S_{id}, x_w)$	$\triangleq$	$\forall x.\ \mathsf{reachable}(x) \wedge \mathsf{white}(x)$ $\implies \mathsf{rt\_wp}(x) \vee \exists x'.\ \mathsf{dt\_bwp}(x', x) \vee \exists x'.\ \mathsf{stk\_bwp}(x', x, A) \vee \mathsf{popped\_bwp}(x_p, x, S_{id}) \vee \mathsf{wwp}(x_w, x)$
$\mathsf{reach\_black}$	$\triangleq$	$\forall x.\ \mathsf{reachable}(x) \Longrightarrow \mathsf{black}(x)$
$\mathsf{ptfd\_sta}(x.id, y)$	$\triangleq$	$\exists n.\ x.id \hookrightarrow n \wedge (y = n \vee \mathsf{dirty}(x) \vee n = 0 \vee \exists \mathsf{t}, x'.\ (\mathsf{t.aux} = x \wedge \mathsf{t}.x' = n \wedge \mathsf{t.own_p}(x')))$
$\mathsf{newobj\_sta}(x)$	$\triangleq$	$\mathsf{obj}(x) * \mathbf{true} \wedge \mathsf{black}(x) \wedge \forall id \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}.\ \mathsf{ptfd\_sta}(x.id, 0)$
$\mathsf{rt\_not\_white}(t)$	$\triangleq$	$\exists S.\ \mathsf{root}(t, S) \wedge \forall n \in S.\ \mathsf{not\_white}(n)$
$\mathsf{rt\_not\_white}$	$\triangleq$	$\forall t \in [1..N].\ \mathsf{rt\_not\_white}(t)$
$\mathsf{mark\_rt\_till}(n)$	$\triangleq$	$\forall t \in [1..n].\ \mathsf{rt\_not\_white}(t)$
$\mathsf{clear\_color\_till}(n)$	$\triangleq$	$\forall x \in [1..n].\ (x.\mathtt{color} \hookrightarrow \mathtt{BLACK} \Longrightarrow \mathsf{newobj\_sta}(x))$
$\mathsf{clear\_dirty\_till}(n)$	$\triangleq$	$\forall x \in [1..n].\ \mathsf{not\_dirty}(x)$
$\mathsf{reclaim\_till}(n)$	$\triangleq$	$\forall x \in [1..n].\ \mathsf{not\_white}(x)$

NOTE: Here we use $\_$ for an unspecified integer n that $0 \leq n \leq M$. Some assertions are already shown in Figure 19.

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.

$$\mathcal{R}_{\mathrm{gc}}; \mathcal{G}_{\mathrm{gc}} \vdash \begin{array}{l} \{\ \exists X, Y.\ \mathtt{j} = Y \wedge \mathtt{i}.\mathrm{pt}_1 \hookrightarrow X\ \} \\ \quad \mathtt{j\ :=\ i.pt1;} \\ \{\ \exists X.\ \mathtt{j} = X \wedge \mathsf{ptfd\_sta}(\mathtt{i}.\mathrm{pt}_1, X)\ \} \end{array}$$

where $\mathsf{ptfd\_sta}(\mathtt{i}.\mathrm{pt}_1, X)$ says the pt_1 field of $\mathtt{i}$ was once X and if it is not X 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.

$$\mathcal{R}_{\mathrm{gc}}; \mathcal{G}_{\mathrm{gc}} \vdash \begin{array}{l} \{\ \exists X, Y.\ \mathtt{c} = X \wedge \mathtt{i.color} \hookrightarrow Y\ \} \\ \quad \mathtt{c\ :=\ i.color;} \\ \left\{ \begin{array}{l} \exists X, Y.\ \mathtt{c} = X \wedge \mathtt{i.color} \hookrightarrow Y \\ \wedge\ (X = Y \vee X = \mathtt{BLUE} \wedge \mathsf{newobj\_sta}(\mathtt{i})) \end{array} \right\} \end{array}$$

where $\mathsf{newobj\_sta}(\mathtt{i})$ says $\mathtt{i}$ points to an new object whose color field is $\mathtt{BLACK}$ and all the pointer fields were once 0. Both the predicates $\mathsf{ptfd\_sta}$ and $\mathsf{newobj\_sta}$ 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 $\mathsf{clear\_color\_till}(n)$ to mean the GC has done color-clearing from 1 to n, but there might still be black objects since the mutators

$$\frac{}{\{O_0; O_1 \Vdash x = X' \wedge X = E \wedge \mathsf{emp}_h\} x := E \{O_0; O_1 \Vdash x = X \wedge \mathsf{emp}_h\}} \text{ (ASSN)}$$

$$\frac{}{\{O_0; O_1 \Vdash x = X \wedge y.id \mapsto Y\} x := y.id \{O_0; O_1 \Vdash x = Y \wedge y.id \mapsto Y\}} \text{ (READ)}$$

$$\frac{}{\{O_0; O_1 \Vdash x.id \mapsto \_ \wedge X = E\} x.id := E \{O_0; O_1 \Vdash x.id \mapsto X\}} \text{ (WRITE)}$$

$$\frac{}{\{x, y; \bullet \Vdash x = X' \wedge 1 \le y \le N\} x := \textbf{get\_root}(y) \{x, y; \bullet \Vdash x = X \wedge 1 \le y \le N \wedge \mathsf{root}(y, X)\}} \text{ (GETRT)}$$

$$\frac{}{\{\bullet; x \Vdash x.\mathtt{color} \mapsto \_\} \textbf{free}(x) \{\bullet; x \Vdash x.\mathtt{color} \mapsto \mathtt{BLUE}\}} \text{ (FREE)}$$

$$\frac{}{\{y, O_0; O_1 \Vdash x = X \wedge y = Y\} \textbf{push}(x, y) \{y, O_0; O_1 \Vdash x = X \wedge y = X :: Y\}} \text{ (PUSH)}$$

$$\frac{}{\{y, O_0; O_1 \Vdash x = X \wedge y = X' :: Y\} x := \textbf{pop}(y) \{y, O_0; O_1 \Vdash x = X' \wedge y = Y\}} \text{ (POP)}$$

$$\frac{p \Longrightarrow B = B \quad \{p \wedge B\} C_1 \{q\} \quad \{p \wedge \neg B\} C_2 \{q\}}{\{p\} \textbf{if } (B)\ C_1 \textbf{ else } C_2 \{q\}} \text{ (IF)} \qquad \frac{p \Longrightarrow B = B \quad \{p \wedge B\} C \{p\}}{\{p\} \textbf{while } (B) \{C\} \{p \wedge \neg B\}} \text{ (WHILE)}$$

$$\frac{p \Longrightarrow \mathsf{own_{np}}(y) \quad \{p * \mathsf{own}(x) \wedge x \in y\} C; y := y - \{x\} \{p * \mathsf{own}(x)\}}{\{p * \mathsf{own}(x)\} \textbf{foreach } x \textbf{ in } y \textbf{ do } C \{p * \mathsf{own}(x) \wedge y = \phi\}} \text{ (FOREACH)}$$

$$\frac{}{\{p\} \textbf{skip} \{p\}} \text{ (SKIP)} \qquad \frac{\{p\} C_1 \{R\} \quad \{R\} C_2 \{q\}}{\{p\} C_1; C_2 \{q\}} \text{ (SEQ)} \qquad \frac{\{p\} C \{q\}}{\{\exists X.p\} C \{\exists X.q\}} \text{ (EXISTS)} \qquad \frac{\{p\} C \{q\}}{\{p * R\} C \{q * R\}} \text{ (FRM)}$$

$$\frac{\{p\} C \{q\} \quad \{p'\} C \{q'\}}{\{p \wedge p'\} C \{q \wedge q'\}} \text{ (CONJ)} \qquad \frac{\{p\} C \{q\} \quad \{p'\} C \{q'\}}{\{p \vee p'\} C \{q \vee q'\}} \text{ (DISJ)} \qquad \frac{p \Longrightarrow p' \quad \{p'\} C \{q'\} \quad q' \Longrightarrow q}{\{p\} C \{q\}} \text{ (CONSEQ)}$$

Figure 29. Inference Rules - Sequential Rules

$$\frac{p \Longrightarrow p' \qquad \{p'\} C \{q'\} \qquad p' \ltimes q' \Longrightarrow \mathcal{G} \qquad q' \Longrightarrow q \qquad \mathsf{Sta}(\{p, q\}, \mathcal{R})}{\mathcal{R}; \mathcal{G} \vdash \{p\} \textbf{atomic} \{C\} \{q\}} \text{ (ATOMIC)}$$

$$\frac{p \Longrightarrow B = B \qquad \mathcal{R}; \mathcal{G} \vdash \{p \wedge B\} C_1 \{q\} \qquad \mathcal{R}; \mathcal{G} \vdash \{p \wedge \neg B\} C_2 \{q\}}{\mathcal{R}; \mathcal{G} \vdash \{p\} \textbf{if } (B)\ C_1 \textbf{ else } C_2 \{q\}} \text{ (P-IF)}$$

$$\frac{p \Longrightarrow B = B \qquad \mathcal{R}; \mathcal{G} \vdash \{p \wedge B\} C \{p\}}{\mathcal{R}; \mathcal{G} \vdash \{p\} \textbf{while } (B) \{C\} \{p \wedge \neg B\}} \text{ (P-WHILE)}$$

$$\frac{p \Longrightarrow \mathsf{own_{np}}(y) \qquad \mathcal{R}; \mathcal{G} \vdash \{p * \mathsf{own}(x) \wedge x \in y\} C; y := y - \{x\} \{p * \mathsf{own}(x)\}}{\mathcal{R}; \mathcal{G} \vdash \{p * \mathsf{own}(x)\} \textbf{foreach } x \textbf{ in } y \textbf{ do } C \{p * \mathsf{own}(x) \wedge y = \phi\}} \text{ (P-FOREACH)}$$

$$\frac{\mathcal{R}; \mathcal{G} \vdash \{p\} C_1 \{r\} \qquad \mathcal{R}; \mathcal{G} \vdash \{r\} C_2 \{q\}}{\mathcal{R}; \mathcal{G} \vdash \{p\} C_1; C_2 \{q\}} \text{ (P-SEQ)}$$

Figure 30. Inference Rules - Concurrency Rules

```
{wfstate}
Initialize() {
  local i: [1..M], c: {BLACK, WHITE, BLUE};
  i := 1;
 Loop Invariant:  {(wfstate ∧ clear_color_till(i − 1) ∧ 1 ≤ i ≤ M + 1) ∗ own_np(c)}
  while (i <= M) {
    i.dirty := 0;
    c := i.color;
    if (c = BLACK) {
      i.color := WHITE;
    }
    i := i + 1;
  }
}
{wfstate ∧ reach_inv}    using Lemma 46
```

Figure 31. Proof Outline of `Initialize()`

```
{(wfstate ∧ reach_inv) ∗ (own_np(mstk) ∧ mstk = ε)}
Trace() {
  local t: [1..N], rt: Set(Int), i: [0..M];
  t := 1;
 Loop Invariant: {(wfstate ∧ reach_inv) ∗ (own_np(mstk) ∧ mstk = ε) ∗ (own_np(t) ∧ 1 ≤ t ≤ N + 1) ∗ own_np(rt) ∗ own_p(i)}
  while (t <= N) {
    rt := get_root(t);
   Foreach Invariant: {FInv}
    foreach i in rt do {
     {FInv ∧ i ∈ rt}
      MarkAndPush(i);
     {FInv ∧ i ∈ rt}
    }
    t := t + 1;
   {∃X. (wfstate ∧ reach_stk(X) ∧ stk_black(X)) ∗ (own_np(mstk) ∧ mstk = X) ∗ (own_np(t) ∧ 1 ≤ t ≤ N + 1) ∗ own_np(rt) ∗ own_p(i)}
    TraceStack();
   {(wfstate ∧ reach_inv) ∗ (own_np(mstk) ∧ mstk = ε) ∗ (own_np(t) ∧ 1 ≤ t ≤ N + 1) ∗ own_np(rt) ∗ own_p(i)}
  }
}
{(wfstate ∧ reach_inv) ∗ (own_np(mstk) ∧ mstk = ε)}
where FInv ≜ ∃X. (wfstate ∧ reach_stk(X) ∧ stk_black(X)) ∗ (own_np(mstk) ∧ mstk = X)
             ∗ (own_np(t) ∧ 1 ≤ t ≤ N) ∗ (own_np(rt) ∧ ∀n ∈ rt. 0 ≤ n ≤ M) ∗ own_p(i)
```

Figure 32. Proof Outline of `Trace()`

```
{∃X. (wfstate ∧ reach_stk(X) ∧ stk_black(X)) ∗ (own_np(mstk) ∧ mstk = X)}
TraceStack() {
  local i: [1..M], j: [0..M];
 Loop Invariant: {∃X. (wfstate ∧ reach_stk(X) ∧ stk_black(X)) ∗ (own_np(mstk) ∧ mstk = X) ∗ own_p(i) ∗ own_p(j)}
  while (!is_empty(mstk)) {
    i := pop(mstk);
   {∃X'. (wfstate ∧ reach_stk(i :: X') ∧ stk_black(X') ∧ obj(i)) ∗ (own_np(mstk) ∧ mstk = X') ∗ own_p(j)}
    j := i.pt1;
   {∃X'. (wfstate ∧ reach_stk(i :: X') ∧ stk_black(X') ∧ obj(i) ∧ ptfd_sta(i.pt_1, j) ∧ (j = 0 ∨ obj(j))) ∗ (own_np(mstk) ∧ mstk = X')}
   { ∃X', i. (wfstate ∧ reach_tomk(X', i, {pt_2, ..., pt_m}, j) ∧ stk_black(X') ∧ (j = 0 ∨ obj(j)))
      ∗ (own_np(mstk) ∧ mstk = X') ∗ (own_p(i) ∧ 1 ≤ i = i ≤ M) }        using Lemma 47
    MarkAndPush(j);
   { ∃X', i. (wfstate ∧ reach_tomk(X', i, {pt_2, ..., pt_m}, 0) ∧ stk_black(X') ∧ (j = 0 ∨ not_white(j)))
      ∗ (own_np(mstk) ∧ mstk = X') ∗ (own_p(i) ∧ 1 ≤ i = i ≤ M) }
    ...
    j := i.ptm; MarkAndPush(j);
   {∃X'. (wfstate ∧ reach_tomk(X', i, φ, 0) ∧ stk_black(X') ∧ (j = 0 ∨ not_white(j))) ∗ (own_np(mstk) ∧ mstk = X')}
  }
}
{(wfstate ∧ reach_inv) ∗ (own_np(mstk) ∧ mstk = ε)}
```

Figure 33. Proof Outline of `TraceStack()`

```
{(wfstate ∧ reach_inv) ∗ (own_np(mstk) ∧ mstk = ε)}
CleanCard() {
  local i: [1..M], c: {BLACK, WHITE, BLUE}, d: {1, 0};
  i := 1;
 Loop Invariant: {∃X. (wfstate ∧ reach_stk(X) ∧ stk_black(X)) ∗ (own_np(mstk) ∧ mstk = X) ∗ (own_p(i) ∧ 1 ≤ i ≤ M + 1) ∗ own_np(c) ∗ own_np(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_stk(X) ∧ stk_black(X)) ∗ (own_np(mstk) ∧ mstk = X) ∗ own_p(i) ∗ own_np(c) ∗ own_np(d)}
  TraceStack();
 {(wfstate ∧ reach_inv) ∗ (own_np(mstk) ∧ mstk = φ) ∗ own_p(i) ∗ own_np(c) ∗ own_np(d)}
}
{(wfstate ∧ reach_inv) ∗ (own_np(mstk) ∧ mstk = ε)}
```

Figure 34. Proof Outline of `CleanCard()`

$\{(\mathsf{wfstate} \wedge \mathsf{reach\_inv}) * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

```
ScanRoot() {
  local t: [1..N], rt: Set(Int), i: [0..M];
  t := 1;
```

Loop Invariant:

$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reach\_stk}(X) \wedge \mathsf{stk\_black}(X) \wedge \mathsf{mark\_rt\_till}(\mathtt{t}-1) \wedge 1 \leq \mathtt{t} \leq N+1) * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own_p}(\mathtt{i}) * \mathsf{own_{np}}(\mathtt{rt})\}$

```
  while (t <= N) {
    rt := get_root(t);
```

Foreach Invariant:

$$\left\{ \begin{array}{l} \exists X, Y.\ (\mathsf{wfstate} \wedge \mathsf{reach\_stk}(X) \wedge \mathsf{stk\_black}(X) \wedge \mathsf{mark\_rt\_till}(\mathtt{t}-1) \wedge 1 \leq \mathtt{t} \leq N \wedge \mathsf{root}(\mathtt{t}, Y) \wedge \forall n \in (Y - \mathtt{rt}).\ \mathsf{not\_white}(n) \wedge \mathtt{rt} \subseteq Y) \\ \quad * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own_p}(\mathtt{i}) \end{array} \right\}$$

```
    foreach i in rt do {
      MarkAndPush(i);
    }
    t := t + 1;
  }
}
```

$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reach\_rtnw\_stk}(X) \wedge \mathsf{stk\_black}(X)) * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)\}$

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

$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reach\_rtnw\_stk}(X) \wedge \mathsf{stk\_black}(X)) * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)\}$

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

Loop Invariant:

$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reach\_rtnw\_stk}(X) \wedge \mathsf{stk\_black}(X) \wedge \mathsf{clear\_dirty\_till}(\mathtt{i}-1) \wedge 1 \leq \mathtt{i} \leq M+1) * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own_{np}}(\mathtt{c}) * \mathsf{own_{np}}(\mathtt{d})\}$

```
  while (i <= M) {
    c := i.color;
    d := i.dirty;
    if (d = 1) {
      i.dirty := 0;
      if (c = BLACK) {
        push(i, mstk);
      }
    }
    i := i + 1;
  }
```

$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reach\_rtnw\_stk}(X) \wedge \mathsf{stk\_black}(X) \wedge \mathsf{clear\_dirty\_till}(M)) * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own_{np}}(\mathtt{c}) * \mathsf{own_{np}}(\mathtt{d})\}$

```
  TraceStack();
}
```

$\{(\mathsf{wfstate} \wedge \mathsf{reach\_black}) * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \phi)\}$

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

$\{\mathsf{wfstate} \wedge \mathsf{reach\_black}\}$

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

Loop Invariant: $\{(\mathsf{wfstate} \wedge \mathsf{reach\_black} \wedge \mathsf{reclaim\_till}(\mathtt{i}-1) \wedge 1 \leq \mathtt{i} \leq M+1) * \mathsf{own_{np}}(\mathtt{c})\}$

```
  while (i <= M) {
    c := i.color;
    if (c = WHITE) {
      free(i);
    }
    i := i + 1;
  }
}
```

$\{\mathsf{wfstate} \wedge \mathsf{reach\_black} \wedge \mathsf{reclaim\_till}(M)\}$

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 $\mathsf{reach\_inv}$ holds.

Lemma 44. $\mathsf{wfstate} \wedge \mathsf{clear\_color\_till}(M) \Longrightarrow \mathsf{reach\_inv}$.

When the object `i` is white, `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.

$$\mathcal{R}_{\text{gc}}; \mathcal{G}_{\text{gc}} \vdash \begin{array}{l} \left\{ \begin{array}{l} \exists X.\, \mathsf{wfstate} \wedge \mathsf{reach\_tomk}(X, x_p, S_{id}, \mathtt{i}) \\ \wedge\, \mathsf{stk\_black}(X) \wedge (\mathtt{i} = 0 \vee \mathsf{obj}(\mathtt{i})) \end{array} \right\} \\ \quad \mathtt{MarkAndPush(i);} \\ \left\{ \begin{array}{l} \exists X.\, \mathsf{wfstate} \wedge \mathsf{reach\_tomk}(X, x_p, S_{id}, 0) \\ \wedge\, \mathsf{stk\_black}(X) \wedge (\mathtt{i} = 0 \vee \mathsf{not\_white}(\mathtt{i})) \end{array} \right\} \end{array} \tag{E.1}$$

Here $\mathsf{reach\_tomk}(A, x_p, S_{id}, x_w)$ means, any reachable white object x must satisfy one of the following conditions:

- $\mathsf{rt\_wp}(x)$: x is reachable from a white root by a white path (*i.e.*, all the objects in the path are white);
- $\exists x'.\ \mathsf{dt\_bwp}(x', x)$: x is reachable from a dirty black object by a white path;
- $\exists x'.\ \mathsf{stk\_bwp}(x', x, A)$: x is reachable from a black object by a white path and that object is on the stack A;
- $\mathsf{popped\_bwp}(x_p, x, S_{id})$: 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 S_{id}.
- $\mathsf{wwp}(x_w, x)$: x is reachable from x_w by a white path and x_w is white as well.

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

`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 `TraceStack()` to perform the depth-first traversal. We need the following two lemmas to relate the unified pre/post conditions of `MarkAndPush(i)` and the actual pre/post conditions when calling the module.

Lemma 45. $\mathsf{reach\_stk}(X) \Longrightarrow \mathsf{reach\_tomk}(X, 0, \phi, i)$.

Lemma 46. $\mathsf{reach\_tomk}(X, 0, \phi, 0) \Longrightarrow \mathsf{reach\_stk}(X)$.

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

In `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 $\mathsf{reach\_stk}(X)$ 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 pt_1 field of `i` to a local variable `j`. As we explained before, $\mathtt{i}.\mathrm{pt}_1$ might not equal `j` since mutators could update this field. We only know that $\mathsf{ptfd\_sta}(\mathtt{i}.\mathrm{pt}_1, \mathtt{j})$ 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 $\mathtt{i}.\mathrm{pt}_1$ 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 so that x is reachable by another path without going though `i` nor `j`. In all the cases, we can get $\mathsf{reach\_tomk}(\mathtt{mstk}, \mathtt{i}, \{\mathrm{pt}_2, \ldots, \mathrm{pt}_m\}, \mathtt{j})$ holds. Formally, the following lemma holds:

   Lemma 47.

   (a) $\mathsf{reach\_stk}(i :: X) \Longleftrightarrow \mathsf{reach\_tomk}(X, i, \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}, 0)$;

   (b) $\mathsf{reach\_tomk}(X, i, S_{id}, 0) \Longrightarrow \mathsf{reach\_tomk}(X, i, S_{id}, j)$;

   (c) $\mathsf{reach\_tomk}(X, i, S_{id}, j) \wedge \mathsf{ptfd\_sta}(i.id, j) \wedge id \in S_{id} \Longrightarrow \mathsf{reach\_tomk}(X, i, S_{id} - \{id\}, j)$.
3. `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, $\mathsf{reach\_stk}(X)$ 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 `CleanCard()` (Figure 34), dirty objects are pushed onto the mark stack and then `TraceStack()` is called again. We reuse the proof of `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 `ScanRoot()` (Figure 35). The atomic `MarkAndPush(i)` is proved similarly to the concurrent one (E.1) with the same pre/post conditions. Then the GC performs the atomic `CleanCard()` (Figure 36). We do not present the proof for the atomic `TraceStack()` since it is similar to the proof of the concurrent one.

Finally, the concurrent `Sweep()` is verified in Figure 37.

E.5 Correctness of the Write Barrier

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

Lemma 48. *For all σ, Σ, σ' and Σ', if $(\sigma, \Sigma) \in \zeta(\mathsf{t})$, $(\sigma, \sigma') \in \mathcal{R}(\mathsf{t})$, $(\Sigma, \Sigma') \in \mathbb{R}(\mathsf{t})$ and $(\sigma', \Sigma') \in \alpha$, then $(\sigma', \Sigma') \in \zeta(\mathsf{t})$.*

Proof:. $\zeta(\mathsf{t}) = \alpha \cap \{((\pi, h), (\Pi, \mathrm{H})) \mid \pi(\mathsf{t})(\mathtt{aux}) = 0^{\mathsf{p}}\}$ and $\mathcal{R}(\mathsf{t})$ ensures not updating the store of the thread `t`. □

Then it's easy to prove RGSim for **skip**:

Lemma 49. *For all σ and Σ, if $(\sigma, \Sigma) \in \zeta(\mathsf{t})$, then*

$$(\mathsf{t}.\mathbf{skip}, \sigma, \mathcal{R}(\mathsf{t}), \mathsf{Id}) \preceq_{\alpha;\zeta(\mathsf{t})} (\mathsf{t}.\boldsymbol{skip}, \Sigma, \mathbb{R}(\mathsf{t}), \mathsf{Id})$$

Proof:. By co-induction.
Let $S = \{((\mathsf{t}.\mathbf{skip}, \sigma), (\mathsf{t}.\boldsymbol{skip}, \Sigma)) \mid (\sigma, \Sigma) \in \zeta(\mathsf{t})\}$. We prove $S \subseteq F(S)$ where F is defined by the simulation. □

We use some denotations as follows:

$$\begin{array}{rcl} \mathtt{set\_dirty(x)} & \triangleq & \mathbf{atomic}\{\mathtt{x.dirty:=1;aux:=0}\} \\ \mathcal{G}_{\mathsf{t}} & \triangleq & \mathcal{G}^{\mathsf{t}}_{\text{write\_pt}} \cup \mathcal{G}^{\mathsf{t}}_{\text{set\_dirty}} \\ \mathbb{G}_{\mathsf{t}} & \triangleq & \mathbb{G}^{\mathsf{t}}_{\text{write\_pt}} \end{array}$$

Lemma 50. *For all σ and Σ, for all $\mathrm{id} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}$, for all $\mathbb{E}$ and E such that $\mathbf{T}(\mathbb{E}) = E$,*

1. if $(\sigma, \Sigma) \in \zeta(\mathsf{t})$, then

$$(\mathsf{t}.\mathtt{update}(\mathtt{x}.id, E), \sigma, \mathcal{R}(\mathsf{t}), \mathcal{G}_{\mathsf{t}}) \preceq_{\alpha;\zeta(\mathsf{t})} (\mathsf{t}.(\mathtt{x}.\mathrm{id} := \mathbb{E}), \Sigma, \mathbb{R}(\mathsf{t}), \mathbb{G}_{\mathsf{t}})$$

2. *if* $(\sigma, \Sigma) \in \alpha$ *and* $\exists n.\ \sigma.ss(\mathtt{t})(\mathtt{aux}) = \sigma.ss(\mathtt{t})(\mathtt{x}) = (n, 1)$, *then*

$$(\mathtt{t.set\_dirty}(\mathtt{x}), \sigma, \mathcal{R}(\mathtt{t}), \mathcal{G}_{\mathtt{t}}) \preceq_{\alpha;\zeta(\mathtt{t})} (\mathtt{t}.\textbf{\textit{skip}}, \Sigma, \mathbb{R}(\mathtt{t}), \mathbb{G}_{\mathtt{t}})$$

Proof:. For each case, by co-induction.
Case: The environments are executed. Similar to the proof of Lemma 48.
Case: The low-level code goes one step (let $\sigma = (\pi, h)$, $s = \pi(\mathtt{t})$ $\Sigma = (\Pi, \mathrm{H})$ and $\mathrm{S} = \Pi(\mathtt{t})$):

1. If $(\mathtt{t.update}(\mathtt{x}.id, E), (\pi, h)) \longrightarrow (\mathtt{t.set\_dirty}(\mathtt{x}), (\pi', h'))$, then $s(\mathtt{x}) = (n, 1)$, $h(n) = o$, $[\![E]\!]_{(s,1)} = n'$, $\pi' = \pi\{\mathtt{t} \leadsto s\{\mathtt{aux} \leadsto (n, 1)\}\}$ and $h' = h\{n \leadsto o\{id \leadsto n'\}\}$.
Since $[\![E]\!]_{(s,1)} = n'$, we know $n' = 0$ or $\exists x.s(x) = (n', 1)$.
Thus we have $((\pi, h), (\pi', h')) \in \mathcal{G}_{\mathtt{t}}$.
Since $(\sigma, \Sigma) \in \alpha$, we know $\mathsf{wfstate}(\sigma)$, thus $h(n)(\mathtt{color}) \neq \mathtt{BLUE}$. Moreover, $\mathrm{S}(\mathtt{x}) = l$, $\mathsf{Loc2Int}(l) = n$, $\mathrm{H}(l) = \mathrm{O}$, $[\![\mathbb{E}]\!]_{\mathrm{S}} = l'$ (where $\mathsf{Loc2Int}(l') = n'$), $\mathrm{O}(\mathrm{id}) = l''$, and $l' = \textbf{nil}$ or $\exists x.\mathrm{S}(x) = l'$.
Thus $(\mathtt{t}.(\mathtt{x}.\mathrm{id} := \mathbb{E}), (\Pi, \mathrm{H})) \overset{}{\underset{\mathbb{G}_{\mathtt{t}}}{\Longrightarrow}} (\mathtt{t}.\textbf{\textit{skip}}, (\Pi', \mathrm{H}'))$ where $\Pi' = \Pi$ and $\mathrm{H}' = \mathrm{H}\{l \leadsto \mathrm{O}\{\mathrm{id} \leadsto l'\}\}$.
We have $((\pi', h'), (\Pi', \mathrm{H}')) \in \alpha$, $\pi'(\mathtt{t})(\mathtt{aux}) = \pi'(\mathtt{t})(\mathtt{x}) = (n, 1)$, which are the premises of the second case.

2. If $(\sigma, \Sigma) \in \alpha$ and $\exists n.\ \sigma.ss(\mathtt{t})(\mathtt{aux}) = \sigma.ss(\mathtt{t})(\mathtt{x}) = (n, 1)$, then $n \in dom(h)$ and $h(n)(\mathtt{color}) \neq \mathtt{BLUE}$,
thus $(\mathtt{t.set\_dirty}(\mathtt{x}), (\pi, h)) \longrightarrow (\mathtt{t}.\textbf{skip}, (\pi', h'))$
where $\pi' = \pi\{\mathtt{t} \leadsto s\{\mathtt{aux} \leadsto 0\}\}$ and $h' = h\{n \leadsto o\{\mathtt{dirty} \leadsto 1\}\}$. Thus we have $((\pi, h), (\pi', h')) \in \mathcal{G}_{\mathtt{t}}$.
We can see $((\pi', h'), \Sigma) \in \zeta(\mathtt{t})$, which is the premise of Lemma 49.

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

Finally, we can conclude the correctness of the write barrier:

$$(\mathtt{t.update}(x.id, E), \mathcal{R}(\mathtt{t}), \mathcal{G}^{\mathtt{t}}_{\text{write\_pt}} \cup \mathcal{G}^{\mathtt{t}}_{\text{set\_dirty}}) \preceq_{\alpha;\zeta(\mathtt{t})\ltimes\zeta(\mathtt{t})} (\mathtt{t}.(x.\mathrm{id} := \mathbb{E}), \mathbb{R}(\mathtt{t}), \mathbb{G}^{\mathtt{t}}_{\text{write\_pt}})$$

where $\mathrm{id} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}$ and $\mathbf{T}(\mathbb{E}) = E$.