Rely-Guarantee-Based Simulation for Compositional Verification of Concurrent Program Transformations

HONGJIN LIANG, XINYU FENG, and MING FU, University of Science and Technology of China

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 article, we propose a *R*ely-*G*uarantee-based *Sim*ulation (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 with respect to 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.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs, Formal methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrency, program transformation, rely-guarantee reasoning, simulation

ACM Reference Format:
Liang, H., Feng, X., and Fu, M. 2014. Rely-guarantee-based simulation for compositional verification of concurrent program transformations. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 3 (March 2014), 55 pages.
DOI:http://dx.doi.org/10.1145/2576235

1. INTRODUCTION

Many verification problems can be reduced to verifying program transformations, that is, proving the target program of the transformation has no more observable

This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under grant nos. 61379039, 61229201, 61103023, and 91318301, Program for New Century Excellent Talents in Universities (grant no. NCET-2010-0984), and the Fundamental Research Funds for the Central Universities (grant no. WK0110000031).
Authors' address: H. Liang, X. Feng (corresponding author), and M. Fu, School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230026, China and Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou, China 215123; email: xyfeng@ustc.edu.cn.

DOI:http://dx.doi.org/10.1145/2576235

behaviors than the source. Next 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 [Herlihy and Shavit 2008]. 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 [Dice et al. 2006]) 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 [Leroy 2009] 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.

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

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 [Benton and Hur 2009; Hur and Dreyer 2011; Leroy 2009; Lochbihler 2010], but they are usually designed for sequential programs (except Lochbihler [2010] and Ševčík et al. [2011], 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 article, we propose a *R*ely-*G*uarantee-based *Sim*ulation (RGSim) for compositional verification of concurrent transformations. By addressing the preceding problems, we make the following contributions.

— RGSim parametrizes the simulation between concurrent programs with rely/guarantee conditions [Jones 1983] which specify the interactions between the programs and their environments. This makes the corresponding refinement relation compositional with respect to parallel compositions, allowing us to decompose refinement proofs for multithreaded 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 nonblocking 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 [Jones 1983] with relational proofs based on RGSim.
— We verify the Boehm et al. concurrent garbage collection algorithm [Boehm et al. 1991] using our framework. As far as we know, it is the first time to formally prove the correctness of this algorithm.
— We give a mechanized formulation of RGSim, and prove its soundness and compositionality in the Coq proof assistant [2010]. Both the manual and mechanized proofs are available online[1].

This article extends the conference paper in POPL 2012 [Liang et al. 2012]. First, we add more examples, including strength reduction and induction variable elimination, the nonblocking concurrent counter, Treiber's stack algorithm, and the concurrent GCD algorithm. Second, we significantly expand the details for the concurrent GC verification, demonstrating that RGSim is a powerful proof technique for verifying program transformations which involve concurrent runtime systems.

In the rest of this article, 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

[1] `http://kyhcs.ustcsz.edu.cn/relconcur/rgsim`

```
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

Fig. 1. Equivalence lost after parallel composition.

optimizations in Section 5, verify fine-grained algorithms and 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, that is, we should be able to know $\mathbf{T}(\mathbb{C}_1) \,\|\, \mathbf{T}(\mathbb{C}_2) \sqsubseteq \mathbb{C}_1 \,\|\, \mathbb{C}_2$ 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 multithreaded programs [Boehm 2005]. The Dekker's algorithm shown in Figure 1(a) has been widely used to demonstrate the problem. Reordering the first two assignment statements of the thread on the left preserves its sequential behaviors, but the whole program can no longer ensure exclusive access to the critical region.

In addition to instruction reordering, the different granularities of atomic operations between the 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., [Abadi and Plotkin 2009; Brookes 1996]). 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++ [Boehm and Adve 2008]) 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 object, we can assume that all accesses to the object are made through the object's methods only, for example, a stack object can only be accessed through push and pop methods, and its internal data cannot be arbitrarily updated.
— Usually the implementation of STM (e.g., TL2 [Dice et al. 2006]) 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 arithmetic. 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 article we propose a *R*ely-*G*uarantee-based *Sim*ulation (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 [Jones 1983], 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, that is, the guarantee of each thread is permitted in the rely of every other. Then we can verify their parallel composition by separately verifying each thread, showing its behaviors under the rely condition indeed satisfy its guarantee. After parallel composition, the threads should be executed under their common environment (i.e., the intersection of their relies) and guarantee all the possible transitions made by them (i.e., the union of their guarantees).

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 $\mathbb{G}$ 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, that is, $\mathcal{G}_2 \subseteq \mathcal{R}_1$, $\mathcal{G}_1 \subseteq \mathcal{R}_2$, $\mathbb{G}_2 \subseteq \mathbb{R}_1$ and $\mathbb{G}_1 \subseteq \mathbb{R}_2$, we could get

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

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.

Example. Next we give an example of loop invariant hoisting to illustrate how RGSim works. The formal proofs are shown in Section 5.2.1.

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

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

(a) events and transition labels

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

(b) the low-level language

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

(c) the high-level language

Fig. 2. Generic languages at target and source levels.

Benton [2004] has proved that the optimized code C_1 preserves the *sequential* behaviors of the source C. In a concurrent setting, this optimization is incorrect within arbitrary environments. For instance, if other threads may update `x`, the final values of `i` might be different at the two levels. In fact, this optimization works only when the environments $\mathcal{R}$ at both levels do not update `x` nor `t`. The guarantees $\mathcal{G}$ of both C_1 and C can be specified as arbitrary transitions. Then we can prove the RGSim relation $(C_1, \mathcal{R}, \mathcal{G}) \preceq (C, \mathcal{R}, \mathcal{G})$ and conclude the correctness of the transformation.

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. Our RGSim relation, which will be formally defined in Section 4, is proposed as a proof technique for $\sqsubseteq$.

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

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

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

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

Fig. 3. Selected operational semantics rules of the high-level language.

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 $\perp$. Boolean expressions Bs are modeled similarly. An instruction c is a partial function from states to sets of label and state pairs, describing the state transitions and the events it generates. We use $\mathcal{P}(\_)$ to denote the power set. Unsafe executions lead to **abort**. Note that the semantics of an instruction could be nondeterministic. 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. When it is sequentially composed with other statements, it has no computational effects. A single-step execution of statements is modeled as a labeled transition $\_ \xrightarrow{\_}_L \_$, 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 $\_ \xrightarrow{\_}_H \_$, which gives the high-level operational semantics of statements. We often omit the subscript H (or L) in $\_ \xrightarrow{\_}_H \_$ (or $\_ \xrightarrow{\_}_L \_$) and the label on top of the arrow when it is τ. The semantics is mostly standard. We only show the rules for primitive instructions and parallel compositions here. Note that when a primitive instruction 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. Shortly

we use $\_ \longrightarrow^* \_$ for zero or multiple-step transitions with no events generated, and $\_ \xrightarrow{e}{}^* \_$ 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**.

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

Definition 3.1 (*Event Trace Set*). $\mathit{ETrSet}_n(C, \sigma)$ represents a set of external event traces produced by C in n steps from the state σ.

(1) $\mathit{ETrSet}_0(C, \sigma) \triangleq \{\epsilon\}$;

(2) $\mathit{ETrSet}_{n+1}(C, \sigma) \triangleq$
$\{\mathcal{E} \mid (C, \sigma) \longrightarrow (C', \sigma') \wedge \mathcal{E} \in \mathit{ETrSet}_n(C', \sigma')$
$\vee\ (C, \sigma) \xrightarrow{e} (C', \sigma') \wedge \mathcal{E}' \in \mathit{ETrSet}_n(C', \sigma') \wedge \mathcal{E} = e :: \mathcal{E}'$
$\vee\ (C, \sigma) \longrightarrow \mathbf{abort} \wedge \mathcal{E} = \mathbf{abort}$
$\vee\ C = \mathbf{skip} \wedge \mathcal{E} = \mathbf{done}\}$.

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

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

Then we define an event trace refinement as the subset relation between event trace sets, which is similar to Leroy's refinement property [Leroy 2009].

Definition 3.2 (*Event Trace Refinement*). We say (C, σ) is an *e-trace refinement* of $(\mathbb{C}, \Sigma)$, that is, $(C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)$, 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 Eq. (1.1) can be reformulated as

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

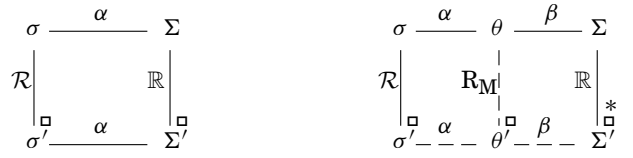
From the aforesaid e-trace refinement definition, we can derive an *e-trace equivalence* relation by requiring both directions hold

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

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.



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

Fig. 4. Related transitions.

However, as we explained before, it is *not* compositional with respect to parallel compositions. In this section we propose RGSim, which can be viewed as a compositional proof technique that allows us to derive the simple e-trace refinement and then verify the corresponding transformation **T**.

4.1. The Definition

Our co-inductively defined RGSim relation is in the form of $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, 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.

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

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.

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

Before we formally define RGSim in Definition 4.2, we first introduce the α-related transitions as follows.

Definition 4.1 (α-*Related Transitions*).
$\langle \mathcal{R}, \mathbb{R} \rangle_\alpha \triangleq \{((\sigma, \sigma'), (\Sigma, \Sigma')) \mid (\sigma, \sigma') \in \mathcal{R} \wedge (\Sigma, \Sigma') \in \mathbb{R} \wedge (\sigma, \Sigma) \in \alpha \wedge (\sigma', \Sigma') \in \alpha\}$.

$\langle \mathcal{R}, \mathbb{R} \rangle_\alpha$ 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 4.2 (*RGSim*). Whenever $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, then $(\sigma, \Sigma) \in \alpha$ and the following are true.

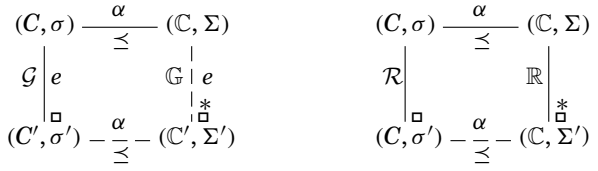
(1) If $(C, \sigma) \longrightarrow (C', \sigma')$, then there exist $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma) \longrightarrow^* (\mathbb{C}', \Sigma')$, $((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha$ and $(C', \sigma', \mathcal{R}, \mathcal{G})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

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

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

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

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



(a) program steps (b) environment steps

Fig. 5. Simulation diagrams of RGSim.

Then, $(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\,\square\,\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})$ 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* $\zeta \in \mathcal{P}(\mathit{LState} \times \mathit{HState})$ is used to relate the initial states σ and Σ.

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

- Starting from α-related states, each step of C corresponds to zero or multiple steps of $\mathbb{C}$, and the resulting states are α-related too. If an external event is produced in the step of C, the same event should be produced by $\mathbb{C}$. We show the simulation diagram with events generated by the program steps in Figure 5(a), where solid lines denote hypotheses and dashed lines denote conclusions, following Leroy's notations [Leroy 2009].
- 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 $\gamma \subseteq \alpha$, that is, 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, some steps of $\mathcal{R}$ may not correspond to steps of $\mathbb{R}$. On the other hand, only requiring that $\mathcal{R}$ be simulated by $\mathbb{R}$ (see (4.2) in Section 4.2) is not sufficient for parallel compositionality, which we will explain later in Section 4.2.

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

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

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

$$\alpha \uplus \beta \triangleq \{(\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2) \mid (\sigma_1, \Sigma_1) \in \alpha \wedge (\sigma_2, \Sigma_2) \in \beta\} \qquad \eta \,\#\, \alpha \triangleq (\eta \cap \alpha) \subseteq (\eta \uplus \alpha)$$

$$\beta \circ \alpha \triangleq \{(\sigma, \Sigma) \mid \exists \theta.\ (\sigma, \theta) \in \alpha \wedge (\theta, \Sigma) \in \beta\} \qquad \alpha^{-1} \triangleq \{(\Sigma, \sigma) \mid (\sigma, \Sigma) \in \alpha\}$$

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

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

Fig. 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 $(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\,\square\,\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})$, that is, the precondition needs to be no weaker than the step invariant. Usually in practice α is very weak and naturally implied by the pre- and postconditions ζ and γ, for example, ζ and γ are the same as α in examples in Section 6.

RGSim is sound with respect to the e-trace refinement (Definition 3.2). 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 4.3 (SOUNDNESS/ADEQUACY). *If there exist $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$, $\mathbb{G}$, α and γ such that $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, then $(C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)$.*

The soundness theorem shows that RGSim is a proof technique for the simple and natural refinement $\sqsubseteq$, which is what we ultimately care about. The theorem can be proved by first strengthening the relies to the identity transitions and weakening the guarantees to the universal relations. Then we prove that the resulting simulation under identity environments implies the e-trace refinement. The mechanized proof in the Coq proof assistant [2010] is available online.

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

4.2. Compositionality Rules

RGSim is compositional with respect to 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 [Jones 1983], we require that the pre- and postconditions be *stable* under the interference from the environments. Here we introduce the concept of stability of a relation ζ with respect to a set of transition pairs $\Lambda \in \mathcal{P}((\mathit{LState} \times \mathit{LState}) \times (\mathit{HState} \times \mathit{HState}))$.

Definition 4.5 (*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 $(\sigma', \Sigma') \in \zeta$.

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

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

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

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

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

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

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

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

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

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

Fig. 7. Compositionality rules for RGSim. At each proof rule, we implicitly assume that the pre- and postconditions are stable under the environments' interference (Definition 4.5), and the relies and guarantees are closed over identity transitions.

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 with respect to any α-related transitions, that is,

$\mathsf{Sta}(\alpha, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$. Another simple example is given next, where both environments could increment x and the unary stable assertion $\mathtt{x} \geq 0$ is lifted to the relation ζ.

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

We can prove $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$. Stability of the pre- and postconditions under the environments' interference is assumed as an implicit side condition at every proof rule in Figure 7, for example, 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. The definitions of the sets $B \Leftrightarrow \mathbb{B}$ and $B \nLeftrightarrow \mathbb{B}$ are given 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, and needs to ensure $B \Leftrightarrow \mathbb{B}$.

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

Note that, although RGSim does not require every step of the high-level program to be in its guarantee (see the first two conditions in Definition 4.2), 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 $C_1 \parallel C_2$, we only need a subset of the interleavings of $\mathbb{C}_1$ and $\mathbb{C}_2$ to simulate those of C_1 and C_2. Thus the high-level relies and guarantees need to ensure the existence of those interleavings only. Next we give a simple example to explain this subtle issue. We can prove

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

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

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

Note that the high-level program is actually finer grained than its guarantee, but to prove Eq. (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 $(\mathtt{print(x)}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \,\square\, \gamma} (\mathtt{print(x)}, \mathbb{R}, \mathbb{G})$. Here we use the instruction $\mathbf{print}(E)$ to observe the value of x, which will produce an external event $\mathbf{out}(n)$ if E evaluates to n. Then by the PAR rule, we get

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

which does not violate the natural meaning of refinements. That is, all the possible external events produced by the low-level side can also be produced by the high-level side, although the latter could have more external behaviors due to its finer granularity.

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

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

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, that is,

$$\alpha \triangleq \{(\sigma,\Sigma) \mid \sigma(\mathtt{x}) \le \Sigma(\mathtt{x})\},$$

we could prove the following.

$$(\mathtt{x:=x+1},\ \mathsf{Id},\ \mathsf{True}) \preceq'_{\alpha;\alpha\square\alpha} (\mathtt{x:=x+2},\ \mathsf{Id},\ \mathsf{True}); \tag{4.3}$$

$$(\mathtt{x:=0;print(x)},\ \mathsf{True},\ \mathsf{Id}) \preceq'_{\alpha;\alpha\square\alpha} (\mathtt{x:=0;print(x)},\ \mathsf{True},\mathsf{Id}). \tag{4.4}$$

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.

$$(\mathtt{x:=x+1} \parallel (\mathtt{x:=0;print(x)}),\ \mathsf{Id},\ \mathsf{True}) \preceq'_{\alpha;\alpha\square\alpha} (\mathtt{x:=x+2} \parallel\!\parallel (\mathtt{x:=0;print(x)}),\ \mathsf{Id},\ \mathsf{True}).$$

This is because the rely $\mathcal{R}$ (or $\mathbb{R}$) is an abstraction of all the permitted behaviors in the environment of a thread t. Any thread t′ whose behaviors are allowed in $\mathcal{R}$ (or $\mathbb{R}$) can run in parallel with t. Thus to obtain parallel compositionality, we have to ensure that the simulation is preserved with *any* possible sibling thread t′. With *our* definition $\preceq$, the refinement of Eq. (4.4) is not provable, because after some α-related transitions of environments, 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, that is, $\mathsf{Sta}(\alpha',\langle\mathcal{G},\mathbb{G}^*\rangle_\alpha)$ holds. Symmetrically, the WEAKEN-α rule requires α to be preserved by environment steps related by the weaker invariant α'. As usual, the pre- and postconditions, the relies and the guarantees can be strengthened or weakened by the CONSEQ rule.

The FRAME rule allows us to use local specifications [Reynolds 2002]. 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. A state relation α is intuitionistic, denoted by $\mathsf{Intuit}(\alpha)$, if it is monotone with respect to the extension of states. The disjointness $\eta \# \alpha$ says that any state pair satisfying both η and α can be split into two disjoint state pairs satisfying η and α respectively. For example, let $\eta \triangleq \{(\sigma,\Sigma) \mid \sigma(\mathtt{y}) = \Sigma(\mathtt{y})\}$ and $\alpha \triangleq \{(\sigma,\Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}$ where x and y are two distinct variables, 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 isMidOf defines the side condition over the environments, as shown in Figure 6. We use θ for a middle-level state.

Soundness. All the rules in Figure 7 are sound, that is, for each rule the premises imply the conclusion. We prove their soundness by co-induction, directly following the definition of RGSim. The proofs are checked in the Coq proof assistant [2010].

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 of Eq. (4.5) 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.

$$(C, \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\zeta \,\square\, \gamma} (\mathbb{C}, \mathsf{Id}, \mathsf{True}) \tag{4.5}$$

The refinement of Eq. (4.6) assumes arbitrary environments, which makes the interference constraints in the PAR rule trivially true. But this assumption is too strong: usually (4.6) cannot be satisfied in practice.

$$(C, \mathsf{True}, \mathsf{True}) \preceq_{\alpha;\zeta \,\square\, \gamma} (\mathbb{C}, \mathsf{True}, \mathsf{True}) \tag{4.6}$$

4.3. A Simple Example

Shortly 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. Its semantics follows RGSep [Vafeiadis 2008] (or see Section 6.2). The auxiliary variable is write-only and would not affect the external behaviors of the program [Abadi and Lamport 1991]. Thus in what follows we can focus on the instrumented target program with the auxiliary code.

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 postconditions 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.

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

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

Following the definition, we can prove $(C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha\square\alpha} (\mathbb{C}_1, \mathbb{R}, \mathbb{G})$ and $(C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha\square\alpha} (\mathbb{C}_2, \mathbb{R}, \mathbb{G})$. By applying the PAR rule and from the soundness of RGSim (Corollary 4.4), 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 $(\mathtt{print(x)}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha\square\alpha} (\mathtt{print(x)}, \mathbb{R}, \mathbb{G})$ with the current definitions of α, $\mathcal{R}$ and $\mathcal{G}$, even though the code of the two sides is syntactically identical.

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

$$(C_1, \mathcal{R}', \mathcal{G}') \preceq_{\alpha';\alpha'\square\alpha'} (\mathbb{C}_1, \mathbb{R}, \mathbb{G}), \tag{4.7}$$

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

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

But Eq. (4.7) does not hold because $\langle \mathcal{R}', \mathbb{R}^* \rangle_{\alpha'}$ (which is used in Definition 4.2(5)) permits unexpected transitions. For instance, we allow $((\sigma,\sigma'),(\Sigma,\Sigma')) \in \langle \mathcal{R}', \mathbb{R}^* \rangle_{\alpha'}$ for the following σ, σ', Σ and Σ'.

$$\sigma = \sigma' \triangleq \{\mathtt{x} \mapsto 0, \mathtt{l} \mapsto \mathtt{cid}\}; \qquad \Sigma \triangleq \{\mathtt{x} \mapsto 0\}; \qquad \Sigma' \triangleq \{\mathtt{x} \mapsto 1\}$$

The high-level environment is allowed to change x even if the thread holds the lock at the low level. Then the left thread may print out different values at the two levels, breaking the simulation (4.7).

It is possible to define the RGSim relation in another way that allows us to get rid of the auxiliary variable for this example. Instead of defining separate rely/guarantee relations at the two levels and using α to relate them, we can directly define "relational rely/guarantee" relations $r, g \in \mathcal{P}((\mathit{LState} \times \mathit{LState}) \times (\mathit{HState} \times \mathit{HState}))$ The new simulation is in the form of $C \preceq_{\alpha;\zeta\square\gamma;r;g} \mathbb{C}$ and defined by substituting r and g for $\langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha$

and $\langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha$ in Definition 4.2. It has all the nice properties of our current RGSim relation (including parallel compositionality) and we no longer need auxiliary variables to prove the simple example. We can prove the new simulations $C_1 \preceq_{\alpha';\alpha' \square \alpha';r;g} \mathbb{C}_1$ and $C'_2 \preceq_{\alpha';\alpha' \square \alpha';r;g} \mathbb{C}_2$. Here C'_2 results from removing `X` from C_2, α' is defined as given before, and r and g are as follows.

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

We can see that if the thread holds the lock at the low level, neither the high-level or the low-level environment can change `x`. This relational r does not permit the unexpected transitions discussed before. It is more expressive than $\langle \mathcal{R}', \mathbb{R}^* \rangle_{\alpha'}$, but is also much heavier. We choose to present the current RGSim relation because in practice it is usually easier to define separate rely/guarantee conditions at the two levels.

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 previous 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. In the following we adapt Benton's work [2004] on sequential optimizations to the concurrent setting.

5.1. Optimization Rules

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

Sequential Unit Laws

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

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

Common Branch

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

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

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

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

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

Dead While

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

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

Loop Peeling

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

Loop Unrolling

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

Dead Code Elimination

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

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

Redundancy Introduction

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

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 postconditions are stable with respect to 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. Examples

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

5.2.1. Invariant Hoisting. We first formally prove the example in Section 2.5. As we discussed, safely hoisting the invariant code `t:=x+1` requires that the environment should not update `x` nor `t`.

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

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

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

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

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

We could prove Eq. (5.1) directly by the RGSim definition and the operational semantics of the code. But shortly 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.

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

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

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

Then by the compositionality rules SEQ and WHILE, we can get $(C'_1, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\,\square\,\alpha} (C', \mathcal{R}, \mathsf{True})$ where C'_1 and C' result from adding **skip**s to C_1 and C.

C'_1 :

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

C' :

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

Besides, from sequential-unit laws and compositionality rules SEQ and WHILE, we can prove $(C_1, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\,\square\,\alpha} (C'_1, \mathcal{R}, \mathsf{True})$ and $(C', \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\,\square\,\alpha} (C, \mathcal{R}, \mathsf{True})$. Finally, by the TRANS rule, we can conclude Eq. (5.1), that is, the correctness of the optimization in appropriate contexts. Since the rely conditions 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.

5.2.2. Strength Reduction and Induction Variable Elimination

Target-Level C_2		Middle-Level C_1		Source-Level C
`local k, r;`		`local i, k;`		`local i;`
`k := 0;`		`i := 0;`		`i := 0;`
`r := 6*n;`		`k := 0;`		
`while(k<r) {`	$\Leftarrow$	`while(i<n) {`	$\Leftarrow$	`while(i<n) {`
`  x := x+k;`		`  x := x+k;`		`  x := x+6*i;`
`  k := k+6;`		`  i := i+1;`		`  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. Next, we give the environments $\mathcal{R}$, $\mathcal{R}_1$ and $\mathcal{R}_2$ at the source, intermediate, and target levels respectively.

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

For both transformations, we require that the common variables in the source and target have the same values. This is shown in the invariant relations α (for the transformation from C to C_1) and β (for the transformation from C_1 to C_2) next.

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

Thus we formalize the correctness of the two transformations as follows.

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

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

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

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

where $\alpha\circ\beta = \{(\sigma_2,\sigma) \mid \sigma_2(\mathtt{n}) = \sigma(\mathtt{n}) \wedge \sigma_2(\mathtt{x}) = \sigma(\mathtt{x})\}$. That is, the optimization phases are correct when the source program is executed in an environment that does not change i nor n (as shown in $\mathcal{R}$ and $\mathcal{R}_2$).

6. REFINEMENT-BASED VERIFICATION FOR CONCURRENT ALGORITHMS

The implementation of an abstract algorithm can be viewed as a transformation from an abstract operation to a concrete and executable program [Hoare 1972]. Verifying that the executable program refines the abstract operation gives us the correctness

```
A1 :                          A2 :
  local d1;                     local d2;
  d1 := 0;                      d2 := 0;
  while (d1 = 0) {              while (d2 = 0) {
0   atom{                     0   atom{
      if (a = b)                    if (b = a)
        d1 := 1;        |||           d2 := 1;
      if (a > b)                    if (b > a)
        a := a - b;                   b := b - a;
    }                             }
  }                             }
```

(a) source code

```
C1 :                          C2 :
  local d1, t11, t12;           local d2, t21, t22;
  d1 := 0;                      d2 := 0;
  while (d1 = 0) {              while (d2 = 0) {
0   t11 := a;                 0   t21 := b;
1   t12 := b;                 1   t22 := a;
2   if (t11 = t12)      ||    2   if (t21 = t22)
3     d1 := 1;                3     d2 := 1;
4   if (t11 > t12)            4   if (t21 > t22)
5     a := t11 - t12;         5     b := t21 - t22;
  }                             }
```

(b) target code

Fig. 8. Concurrent GCD.

of the implementation. In a concurrent setting, we can use RGSim to verify the fine-grained implementation of an algorithm.

Similarly, RGSim also gives us a refinement-based proof method to verify the *atomicity* of concurrent object implementations. A concurrent object provides a set of methods which can be called in parallel by clients as the only way to access 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.

In this section, we discuss four examples to illustrate how we use RGSim to verify the concurrent objects and fine-grained implementation of abstract algorithms: a concurrent GCD algorithm (calculating greatest common divisors) [Feng 2009], the lock-coupling list [Herlihy and Shavit 2008], the nonblocking concurrent counter [Turon and Wand 2011], and Treiber's stack algorithm [Treiber 1986].

6.1. Concurrent GCD

We first prove the correctness of a concurrent GCD program in Figure 8(b). The program uses two threads to compute the Greatest Common Divisor (GCD) of the shared variables a and b. One thread executes C_1 which reads the values of a and b, but only updates a if a $>$ b. The other thread executes C_2 which does the reverse. When a $=$ b, the two threads terminate. This fine-grained GCD program is transformed from the

program in Figure 8(a), where two threads atomically update a and b respectively. Here we use ***atom***{$\mathbb{C}$} to execute $\mathbb{C}$ atomically. Its semantics follows RGSep [Vafeiadis 2008] (or see Section 6.2).

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

By 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), that is, C_1^0 is equivalent to A_1^0 (and C_2^0 is equivalent to A_2^0), where C_1^0 (or C_2^0) denotes the code from line 0 to line 5 in C_1 (or C_2), and A_1^0 (or A_2^0) denotes the atomic block in A_1 (or A_2).

It is natural to define the α relation as

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

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

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

Then we can operationally prove the RGSim relations between C_1^0 and A_1^0 (here α^{-1} is the inverse relation of α, as defined in Figure 6).

$$(C_1^0, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha\Box\alpha} (A_1^0, \mathbb{R}_1, \mathbb{G}_1), \qquad (A_1^0, \mathbb{R}_1, \mathbb{G}_1) \preceq_{\alpha^{-1};\alpha^{-1}\Box\alpha^{-1}} (C_1^0, \mathcal{R}_1, \mathcal{G}_1).$$

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

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

Similarly, the relations hold between C_2 and A_2.

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

When C_1 and C_2 (or A_1 and A_2) are parallel composed to compute the GCD together, the environment of the whole GCD program should be the identity transition set Id because the shared variables a and b cannot be modified when $C_1 \parallel C_2$ is computing their GCD. Its guarantee is just specified as True, a set of all the possible state transitions. We can prove that both (print(a), Id, True) $\preceq_{\alpha;\alpha\Box\alpha}$ (print(a), Id, True) and the reverse direction hold. Then by the rules PAR and SEQ, we can get

$$((C_1 \parallel C_2); \mathtt{print(a)}, \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\alpha\Box\alpha} ((A_1 \parallel\!\!\parallel A_2); \mathtt{print(a)}, \mathsf{Id}, \mathsf{True}),$$

and also the reverse direction. By the soundness of RGSim (Theorem 4.3) we obtain the final result.

$$(C_1 \parallel C_2); \mathtt{print(a)} \approx_{\mathbf{T}} (A_1 \parallel\!\!\parallel A_2); \mathtt{print(a)},$$

This is 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 is not difficult to find that the abstract program really computes the GCD of a and b. So we can conclude

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

(a) an abstract set

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

(b) the lock-coupling list-based set

Fig. 9. The set object.

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.

6.2. Lock-Coupling List

In this section, we prove the atomicity of the lock-coupling list-based implementation for the set object. In Figure 9(a) we define two atomic set operations, `ADD(e)` and `RMV(e)`. Figure 9(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 [Vafeiadis 2008; Vafeiadis and Parkinson 2007]. Here we show that its atomicity can also be verified using our RGSim by proving the low-level methods refine the corresponding abstract operations. We will discuss the key difference between the previous proofs and ours in Section 8.

To support dynamically allocated memory and ownership transfers, we split the states into shared and thread-local parts. We first take the generic languages in

Figure 2, and instantiate the high-level program states as follows. The state Σ consists of shared memory M_s (where the object resides) and a thread pool Π, which is a mapping from thread identifiers ($\mathsf{t} \in \mathit{ThrdID}$) 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.

We show the high-level and low-level languages and the operational semantics in Figure 10. To allow ownership transfers between the shared memory and thread-local memory, we use $\boldsymbol{atom}\{\mathbb{C}\}_{\mathsf{A}}$ (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 [Vafeiadis and Parkinson 2007], 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 block.[2] 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 9, which are the same as the corresponding guarantees in Figure 11, as we will explain next.

In Figure 9, 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. Note that `lock(x)` and `unlock(x)` are instantiations of c. Their semantics has been explained in Section 3.1.

We define the α relation, the guarantees and the relies in Figure 11. The predicate $\mathsf{list}(x, A)(m_s)$ 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.

Before defining the rely and guarantee relations, we first introduce some syntactic sugar in Figure 11(b). We use $x \mapsto (n, v, y)$ and $x \overset{1}{\mapsto} (n, v, y)$ for nodes in the low-level shared memory m_s and the local memory m_l of the current thread respectively. *ltrue* means the thread-local memory m_l is arbitrary. The separating conjunction $p * q$ means p and q hold on disjoint memory. The action $p \ltimes_{\mathsf{t}} q$ represents the update of some memory (m_l, m_s) satisfying p to some memory satisfying q, and the memory of the threads other than the current thread t is unchanged. We overload the notations to the high-level machine, and use $x \Mapsto v$ to mean the value of x is v in the high-level shared memory M_s.

The atomic actions of the algorithm are specified by $\mathcal{G}_{\mathrm{lock}}$, $\mathcal{G}_{\mathrm{unlock}}$, $\mathcal{G}_{\mathrm{add}}$, $\mathcal{G}_{\mathrm{rmv}}$ and $\mathcal{G}_{\mathrm{local}}$ respectively, which are all parametrized with a thread identifier t. For example, $\mathcal{G}_{\mathrm{rmv}}(\mathsf{t})$ 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

[2]It is easy to prove that if the program $\mathbb{W}$ (or W) does not abort, then the event trace set of $\mathbb{W}$ (or W) under the instrumented semantics is the same as the event trace set of $\mathbb{W}$ (or W) under a standard (flat) operational semantics where we erase the annotations $\mathbb{A}$ (or A) and merge the local and shared memory. Thus by proving the event-trace refinement under the instrumented semantics, we can get the event-trace refinement under the flat semantics.

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

(a) the high-level language for abstract operations

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

(b) the low-level language for concrete implementations

$$
\frac{(\mathbb{C}, (\Pi \uplus \{\mathtt{t} \ \square\ (M_l \uplus M_s)\}, \emptyset)) \longrightarrow_{\mathtt{t}}^{*} (\mathbf{skip}, (\Pi \uplus \{\mathtt{t} \ \square\ M_l''\}, \emptyset)) \quad M_l'' = M_l' \uplus M_s' \quad (M_s, M_s') \in \mathbb{A}}{(\boldsymbol{atom}\{\mathbb{C}\}_{\mathsf{A}}, (\Pi \uplus \{\mathtt{t} \ \square\ M_l\}, M_s)) \longrightarrow_{\mathtt{t}} (\mathbf{skip}, (\Pi \uplus \{\mathtt{t} \ \square\ M_l'\}, M_s'))}
$$

$$
\frac{\begin{array}{c}(\mathbb{C}, (\Pi \uplus \{\mathtt{t} \ \square\ (M_l \uplus M_s)\}, \emptyset)) \longrightarrow_{\mathtt{t}}^{*} (\mathbf{skip}, (\Pi \uplus \{\mathtt{t} \ \square\ M_l''\}, \emptyset)) \\ \neg \exists M_l', M_s'.\ (M_l'' = M_l' \uplus M_s' \wedge (M_s, M_s') \in \mathbb{A})\end{array}}{(\boldsymbol{atom}\{\mathbb{C}\}_{\mathsf{A}}, (\Pi \uplus \{\mathtt{t} \ \square\ M_l\}, M_s)) \longrightarrow_{\mathtt{t}} \mathbf{abort}}
$$

$$
\frac{(\mathbb{C}, (\Pi \uplus \{\mathtt{t} \ \square\ (M_l \uplus M_s)\}, \emptyset)) \longrightarrow_{\mathtt{t}}^{*} \mathbf{abort}}{(\boldsymbol{atom}\{\mathbb{C}\}_{\mathsf{A}}, (\Pi \uplus \{\mathtt{t} \ \square\ M_l\}, M_s)) \longrightarrow_{\mathtt{t}} \mathbf{abort}} \qquad \frac{(\mathbb{C}_i, \Sigma) \longrightarrow_{\mathtt{t}_i} \mathbf{abort}}{(\mathtt{t}_1.\mathbb{C}_1 \,|||\, \ldots \mathtt{t}_i.\mathbb{C}_i \ldots \,|||\, \mathtt{t}_n.\mathbb{C}_n, \Sigma) \longrightarrow \mathbf{abort}}
$$

$$
\frac{(\mathbb{C}_i, \Sigma) \xrightarrow{o}_{\mathtt{t}_i} (\mathbb{C}_i', \Sigma')}{(\mathtt{t}_1.\mathbb{C}_1 \,|||\, \ldots \mathtt{t}_i.\mathbb{C}_i \ldots \,|||\, \mathtt{t}_n.\mathbb{C}_n, \Sigma) \xrightarrow{o} (\mathtt{t}_1.\mathbb{C}_1 \,|||\, \ldots \mathtt{t}_i.\mathbb{C}_i' \ldots \,|||\, \mathtt{t}_n.\mathbb{C}_n, \Sigma')}
$$

(c) selected operational semantics rules of the high-level language

Fig. 10. The languages for concurrent objects.

to the action performed by the code of line 13 in `rmv(e)` in Figure 9. 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 [Vafeiadis 2008].

$$\mathsf{list}(x,A) \triangleq \lambda m_s.\ (m_s = \emptyset \wedge x = \mathbf{null} \wedge A = \epsilon) \vee (\exists m_s', v, y, A'.\ m_s = m_s' \uplus \{x \mapsto (\_, v, y)\} \wedge A = v::A' \wedge \mathsf{list}(y, A')(m_s'))$$

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

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

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

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

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

(a) the α relation

$$x \mapsto (n,v,y) \triangleq \lambda(m_l, m_s).\ (dom(m_l) = \emptyset) \wedge (m_s = \{x \mapsto (n,v,y)\})$$
$$x \overset{\mathtt{l}}{\mapsto} (n,v,y) \triangleq \lambda(m_l, m_s).\ (m_l = \{x \mapsto (n,v,y)\}) \wedge (dom(m_s) = \emptyset)$$
$$ltrue \triangleq \lambda(m_l, m_s).\ (dom(m_s) = \emptyset)$$
$$p * q \triangleq \lambda(m_l, m_s).\ \exists m_l', m_s', m_l'', m_s''.\ p(m_l', m_s') \wedge q(m_l'', m_s'') \wedge (m_l = m_l' \uplus m_l'') \wedge (m_s = m_s' \uplus m_s'')$$
$$p \ltimes_{\mathtt{t}} q \triangleq \{((\pi \uplus \{\mathtt{t} \mapsto m_l\}, m_s), (\pi \uplus \{\mathtt{t} \mapsto m_l'\}, m_s')) \mid \exists m_{l1}, m_{s1}, m_{l2}, m_{s2}, m_{l1}', m_{s1}'.\ p(m_{l1}, m_{s1}) \wedge q(m_{l1}', m_{s1}') \wedge (m_l = m_{l1} \uplus m_{l2}) \wedge (m_s = m_{s1} \uplus m_{s2}) \wedge (m_l' = m_{l1}' \uplus m_{l2}) \wedge (m_s' = m_{s1}' \uplus m_{s2})\}$$

$$x \Mapsto v \triangleq \lambda(M_l, M_s).\ (dom(M_l) = \emptyset) \wedge (M_s = \{x \mapsto v\})$$
$$\boldsymbol{ltrue} \triangleq \lambda(M_l, M_s).\ (dom(M_s) = \emptyset)$$

(b) syntactic sugar (where $p, q \in LMem \times LMem \to Prop$)

$$\mathcal{G}_{\mathrm{lock}}(\mathtt{t}) \triangleq \exists x,v,y.\ (x \mapsto (0,v,y)) \ltimes_{\mathtt{t}} (x \mapsto (\mathtt{t},v,y))$$
$$\mathcal{G}_{\mathrm{unlock}}(\mathtt{t}) \triangleq \exists x,v,y.\ (x \mapsto (\mathtt{t},v,y)) \ltimes_{\mathtt{t}} (x \mapsto (0,v,y))$$
$$\mathcal{G}_{\mathrm{add}}(\mathtt{t}) \triangleq \exists x,y,z,u,v,w,n,z'.\ (x \mapsto (\mathtt{t},u,z) * y \overset{\mathtt{l}}{\mapsto} (0,v,z) * z \mapsto (n,w,z') \wedge u < v < w) \ltimes_{\mathtt{t}} (x \mapsto (\mathtt{t},u,y) * y \mapsto (0,v,z) * z \mapsto (n,w,z'))$$
$$\mathcal{G}_{\mathrm{rmv}}(\mathtt{t}) \triangleq \exists x,y,z,u,v.\ (x \mapsto (\mathtt{t},u,y) * y \mapsto (\mathtt{t},v,z) \wedge v < \mathtt{MAX\_VAL}) \ltimes_{\mathtt{t}} (x \mapsto (\mathtt{t},u,z) * y \overset{\mathtt{l}}{\mapsto} (\mathtt{t},v,z))$$
$$\mathcal{G}_{\mathrm{local}}(\mathtt{t}) \triangleq ltrue \ltimes_{\mathtt{t}} ltrue$$
$$\mathcal{G}(\mathtt{t}) \triangleq \mathcal{G}_{\mathrm{lock}}(\mathtt{t}) \cup \mathcal{G}_{\mathrm{unlock}}(\mathtt{t}) \cup \mathcal{G}_{\mathrm{add}}(\mathtt{t}) \cup \mathcal{G}_{\mathrm{rmv}}(\mathtt{t}) \cup \mathcal{G}_{\mathrm{local}}(\mathtt{t}) \qquad \mathcal{R}(\mathtt{t}) \triangleq \bigcup_{\mathtt{t}' \neq \mathtt{t}} \mathcal{G}(\mathtt{t}')$$

$$\mathbb{G}_{\mathrm{add}}(\mathtt{t}) \triangleq \exists S, e.\ (\mathtt{S} \Mapsto S) \ltimes_{\mathtt{t}} (\mathtt{S} \Mapsto S \cup \{e\}) \qquad \mathbb{G}_{\mathrm{rmv}}(\mathtt{t}) \triangleq \exists S, e.\ (\mathtt{S} \Mapsto S \cup \{e\}) \ltimes_{\mathtt{t}} (\mathtt{S} \Mapsto S)$$
$$\mathbb{G}_{\mathrm{local}}(\mathtt{t}) \triangleq \boldsymbol{ltrue} \ltimes_{\mathtt{t}} \boldsymbol{ltrue} \qquad \mathbb{G}(\mathtt{t}) \triangleq \mathbb{G}_{\mathrm{add}}(\mathtt{t}) \cup \mathbb{G}_{\mathrm{rmv}}(\mathtt{t}) \cup \mathbb{G}_{\mathrm{local}}(\mathtt{t}) \qquad \mathbb{R}(\mathtt{t}) \triangleq \bigcup_{\mathtt{t}' \neq \mathtt{t}} \mathbb{G}(\mathtt{t}')$$

(c) rely and guarantee relations

Fig. 11. Auxiliary definitions and specifications for the lock-coupling list.

We can prove that for any thread t, the following hold.

$$(\mathtt{add}(\mathtt{e}), \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq^{\mathtt{t}}_{\alpha;\alpha \square \alpha} (\mathtt{ADD}(\mathtt{e}), \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}));$$
$$(\mathtt{rmv}(\mathtt{e}), \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq^{\mathtt{t}}_{\alpha;\alpha \square \alpha} (\mathtt{RMV}(\mathtt{e}), \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t})).$$

Here $\preceq^{\mathtt{t}}_{\alpha;\alpha \square \alpha}$ is the RGSim relation in Definition 4.2 with the transitions $\longrightarrow$ being replaced by $\longrightarrow_{\mathtt{t}}$ (defined in Figure 10(c)). The proofs are done operationally based

```
atom{ x := x+1; }
```

(a) source code INC(x)

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

(b) target code inc(x)

Fig. 12. The atomic and nonblocking counters.

on the definition of RGSim. We analyze the implementation step by step and find out the instructions which correspond to the high-level single atomic steps (i.e., the *linearization points*). For the add(e) operation, since we require the elements in the concrete list are those in the abstract set, we can 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. We omit the detailed proofs here.

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.

6.3. Nonblocking Counter

The next example (in Figure 12) is counters which increase the value of a shared variable x atomically. The basic requirement is that the counter should not miss any increment when several threads update x concurrently. A simple abstract counter INC(x) increases the value of x in a coarse-grained atomic block. The concrete implementation inc(x) uses the Compare-And-Swap (CAS) instruction $\mathbf{cas}(\&x, t_1, t_2)$, which reads the value from the location of x, compares it with an expected value t_1, writes out a new value t_2 if the two match, and returns whether the update succeeds. Next we use RGSim to prove the atomicity of inc(x).

We first define the α relation between low-level and high-level states, where only the values of x are concerned.

$$\alpha \triangleq \{((\pi, m_s), (\Pi, M_s)) \mid m_s(\mathtt{x}) = M_s(\mathtt{x})\}$$

We let the pre- and postconditions be the same as the invariant α. Both inc(x) and INC(x) guarantee that a thread t only updates its local variables and/or increases the values of x. The rely conditions of thread t allow any other thread t′ to update x and thread-local variables of t′. Here we use the syntactic sugar in Figure 11 to define the rely and guarantee relations.

$$\mathcal{G}(\mathtt{t}) \triangleq (\exists n.\ (\mathtt{x} \mapsto n * \mathit{ltrue}) \ltimes_{\mathtt{t}} (\mathtt{x} \mapsto n+1 * \mathit{ltrue})) \vee \mathcal{G}_{\mathrm{local}}(\mathtt{t}) \qquad \mathcal{R}(\mathtt{t}) \triangleq \bigcup_{\mathtt{t}' \neq \mathtt{t}} \mathcal{G}(\mathtt{t}')$$

$$\mathbb{G}(\mathtt{t}) \triangleq (\exists n.\ (\mathtt{x} \mapsto n) \ltimes_{\mathtt{t}} (\mathtt{x} \mapsto n+1)) \vee \mathbb{G}_{\mathrm{local}}(\mathtt{t}) \qquad \mathbb{R}(\mathtt{t}) \triangleq \bigcup_{\mathtt{t}' \neq \mathtt{t}} \mathbb{G}(\mathtt{t}')$$

Then we can prove the RGSim relation holds.

$$(\texttt{inc(x)}, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq^{\texttt{t}}_{\alpha;\alpha\square\alpha} (\texttt{INC(x)}, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$$

It says that the fine-grained `inc(x)` does not have more behaviors than the atomic `INC(x)` in any environment, that is, `inc(x)` has atomicity. The proof is done operationally based on the RGSim definition. We find out the corresponding program points in `inc(x)` and `INC(x)`, and prove they are related no matter what the environments do.

Also we can prove $(\texttt{INC(x)}, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t})) \preceq^{\texttt{t}}_{\alpha^{-1};\alpha^{-1}\square\alpha^{-1}} (\texttt{inc(x)}, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t}))$, which says the implementation `inc(x)` has all the behaviors of `INC(x)`. Thus `inc(x)` and `INC(x)` behave just the same.

As a simple illustration of the atomicity, we go on to show that the nonblocking `inc(x)` can be used by two threads concurrently without missing any increment, as if `x` was updated by the threads one after the other. Formally, we prove that $(\texttt{inc(x)};\texttt{print(x)}) \parallel (\texttt{inc(x)};\texttt{print(x)})$ and $(\texttt{INC(x)};\texttt{print(x)}) \parallel\!\parallel\!\parallel (\texttt{INC(x)};\texttt{print(x)})$ have the same observable event traces when the initial values of `x` are the same.

We can prove that $(\texttt{print(x)}, \mathcal{R}(\texttt{t}), \mathcal{G}(\texttt{t})) \preceq_{\alpha;\alpha\square\alpha} (\texttt{print(x)}, \mathbb{R}(\texttt{t}), \mathbb{G}(\texttt{t}))$ and the reverse direction holds. Then by the rules SEQ, PAR and CONSEQ, we can get both

$$\begin{aligned}&((\texttt{inc(x)};\texttt{print(x)}) \parallel (\texttt{inc(x)};\texttt{print(x)}), \mathsf{Id}, \mathsf{True})\\ &\preceq_{\alpha;\alpha\square\alpha} ((\texttt{INC(x)};\texttt{print(x)}) \parallel\!\parallel\!\parallel (\texttt{INC(x)};\texttt{print(x)}), \mathsf{Id}, \mathsf{True})\end{aligned}$$

and the reverse direction. By the soundness of RGSim (Theorem 4.3), we know they are e-trace equivalent, and hence the transformation is correct.

$$(\texttt{inc(x)};\texttt{print(x)}) \parallel (\texttt{inc(x)};\texttt{print(x)}) \approx_{\mathbf{T}} (\texttt{INC(x)};\texttt{print(x)}) \parallel\!\parallel\!\parallel (\texttt{INC(x)};\texttt{print(x)})$$

This is for any $\mathbf{T}$ that respects α. That is, no matter how the two nonblocking threads interleave, they complete their operations as if both of them were executing the abstract atomic counter.

Incrementing several shared variables. We have verified the transformation from `INC(x)` to `inc(x)` without caring about other shared resource. 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 is easy to see: $(\texttt{inc(y)}, \mathcal{R}_1(\texttt{t}), \mathcal{G}_1(\texttt{t})) \preceq^{\texttt{t}}_{\alpha_1;\alpha_1\square\alpha_1} (\texttt{INC(y)}, \mathbb{R}_1(\texttt{t}), \mathbb{G}_1(\texttt{t}))$, where $\alpha_1 \triangleq \{((\pi, m_s), (\Pi, M_s)) \mid m_s(\texttt{y}) = M_s(\texttt{y})\}$ and $\mathcal{R}_1(\texttt{t})$, $\mathcal{G}_1(\texttt{t})$, $\mathbb{R}_1(\texttt{t})$ and $\mathbb{G}_1(\texttt{t})$ are defined similarly as $\mathcal{R}(\texttt{t})$, $\mathcal{G}(\texttt{t})$, $\mathbb{R}(\texttt{t})$ and $\mathbb{G}(\texttt{t})$ except all the occurrences of `x` are replaced by `y`.

By the rules FRAME and SEQ, we can get

$$\begin{aligned}&(\texttt{inc(x)};\texttt{inc(y)};\texttt{print(x)}, \mathcal{R}(\texttt{t}) \uplus \mathcal{R}_1(\texttt{t}), \mathcal{G}(\texttt{t}) \uplus \mathcal{G}_1(\texttt{t}))\\ &\preceq^{\texttt{t}}_{\beta;\beta\square\beta} (\texttt{INC(x)};\texttt{INC(y)};\texttt{print(x)}, \mathbb{R}(\texttt{t}) \uplus \mathbb{R}_1(\texttt{t}), \mathbb{G}(\texttt{t}) \uplus \mathbb{G}_1(\texttt{t}))\end{aligned} \tag{6.1}$$

where $\beta \triangleq \alpha \uplus \alpha_1 = \{((\pi, m_s), (\Pi, M_s)) \mid m_s(\texttt{x}) = M_s(\texttt{x}) \wedge m_s(\texttt{y}) = M_s(\texttt{y})\}$, the rely conditions 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.

Similarly, we can get the reverse direction of Eq. (6.1). Then by the soundness of RGSim, we can conclude the combined transformation is correct

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

for any $\mathbf{T}$ that respects β.

```
PUSH(v):

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

```
POP():
    local r;
 0  atom {
      if (A = ϵ) {
        r := EMPTY;
      }else {
        r := head(A);
        A := tail(A);
      }
    }
    return r;
```

(a) an abstract stack

```
push(v):

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

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

(b) Treiber's nonblocking implementation

Fig. 13. The stack object.

6.4. Treiber's Nonblocking Stack

The last example is to verify the atomicity of Treiber's nonblocking stack. The stack object provides two operations in its interface. The abstract PUSH(v) and POP(), defined in Figure 13(a), atomically operate on a value sequence. We implement the abstract stack by a singly-linked list pointed to by a shared variable S, and PUSH(v) and POP() by the nonblocking code push(v) and pop() respectively. As shown in Figure 13(b), the nonblocking implementation uses CAS instructions to obtain fine-grained atomicity.

We use RGSim to prove the atomicity of the nonblocking stack, that is, push(v) refines PUSH(v) and pop() refines POP() when they are executed in appropriate environments.

We define the α relation, the guarantees and the relies in Figure 14. The mapping shared_map between the low-level and the high-level shared memory is defined by only considering the value sequence on the stack. It requires that the concrete shared memory m_s contains a submemory $\widehat{m}_s$ of a linked list as the stack, and the concrete stack has the same value sequence as the abstract one. As in the lock-coupling list example

$$\mathsf{shared\_map}(m_s, M_s) \triangleq \exists \widehat{m}_s.\ \mathsf{list}(m_s(\mathtt{S}), M_s(\mathtt{A}))(\widehat{m}_s) \wedge \widehat{m}_s \subseteq m_s \backslash \{\mathtt{S}\}$$
$$\mathsf{local\_map}(m_l, M_l) \triangleq m_l(\mathtt{v}) = M_l(\mathtt{v}) \wedge \exists m_l'.\ m_l = m_l' \uplus \{\mathtt{d} \mapsto \_, \mathtt{x} \mapsto \_, \mathtt{t} \mapsto \_, \mathtt{r} \mapsto \_\}$$
$$\alpha \triangleq \{((\pi, m_s), (\Pi, M_s)) \mid \mathsf{shared\_map}(m_s, M_s) \wedge \forall \mathtt{t} \in dom(\Pi).\ \mathsf{local\_map}(\pi(\mathtt{t}), \Pi(\mathtt{t}))\}$$

$$\mathcal{G}_{\mathrm{push}}(\mathtt{t}) \triangleq \exists v, x, y.\ (\mathtt{S} \mapsto y * x \overset{1}{\mapsto} (v, y) * \mathit{ltrue}) \ltimes_{\mathtt{t}} (\mathtt{S} \mapsto x * x \mapsto (v, y) * \mathit{ltrue})$$
$$\mathcal{G}_{\mathrm{pop}}(\mathtt{t}) \triangleq \exists x, v, y.\ (\mathtt{S} \mapsto x * x \mapsto (v, y) * \mathit{ltrue}) \ltimes_{\mathtt{t}} (\mathtt{S} \mapsto y * x \mapsto (v, y) * \mathit{ltrue})$$
$$\mathcal{G}(\mathtt{t}) \triangleq \mathcal{G}_{\mathrm{push}}(\mathtt{t}) \cup \mathcal{G}_{\mathrm{pop}}(\mathtt{t}) \cup \mathcal{G}_{\mathrm{local}}(\mathtt{t}) \qquad \mathcal{R}(\mathtt{t}) \triangleq \bigcup_{\mathtt{t}' \neq \mathtt{t}} \mathcal{G}(\mathtt{t}')$$

$$\mathbb{G}_{\mathrm{push}}(\mathtt{t}) \triangleq \exists A, v.\ ((\mathtt{A} \mapsto A) * \boldsymbol{\mathit{ltrue}}) \ltimes_{\mathtt{t}} ((\mathtt{A} \mapsto v :: A) * \boldsymbol{\mathit{ltrue}})$$
$$\mathbb{G}_{\mathrm{pop}}(\mathtt{t}) \triangleq \exists A, v.\ ((\mathtt{A} \mapsto v :: A) * \boldsymbol{\mathit{ltrue}}) \ltimes_{\mathtt{t}} ((\mathtt{A} \mapsto A) * \boldsymbol{\mathit{ltrue}})$$
$$\mathbb{G}(\mathtt{t}) \triangleq \mathbb{G}_{\mathrm{push}}(\mathtt{t}) \cup \mathbb{G}_{\mathrm{pop}}(\mathtt{t}) \cup \mathbb{G}_{\mathrm{local}}(\mathtt{t}) \qquad \mathbb{R}(\mathtt{t}) \triangleq \bigcup_{\mathtt{t}' \neq \mathtt{t}} \mathbb{G}(\mathtt{t}')$$

Fig. 14. Auxiliary definitions and specifications for the nonblocking stack.

in Section 6.2, we use the predicate $\mathsf{list}(x, A)(\widehat{m}_s)$ to represent a singly-linked list in the shared memory $\widehat{m}_s$ whose head node's address is x and values form a sequence A. Since S is a shared variable containing the address of the top node, it itself is not in the domain of $\widehat{m}_s$. For the local memory, local_map defines the mapping of each thread. The value of v in the low-level local memory should be the same as in the high-level local memory, and the low-level local memory should provide enough additional space needed by the object operations (i.e., the local variables d, x, t and r). Then α relates the shared memory by shared_map and the local memory of each thread t by local_map.

Each thread guarantees that it performs push, pop, and local operations only, and its environment includes the operations made by all the other threads. The guarantees reflect the ownership transfers in push and pop operations. For example, $\mathcal{G}_{\mathrm{push}}(\mathtt{t})$ says that the node x is transferred from the thread-local memory to the shared memory. The definitions use the syntactic sugar in Figure 11.

We could prove the nonblocking stack operations are simulated by the corresponding atomic operations.

$$(\mathtt{push}(\mathtt{v}), \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq^{\mathtt{t}}_{\alpha;\alpha \square \alpha} (\mathtt{PUSH}(\mathtt{v}), \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t}));$$
$$(r := \mathtt{pop}(), \mathcal{R}(\mathtt{t}), \mathcal{G}(\mathtt{t})) \preceq^{\mathtt{t}}_{\alpha;\alpha \square \alpha} (r := \mathtt{POP}(), \mathbb{R}(\mathtt{t}), \mathbb{G}(\mathtt{t})).$$

This gives us the atomicity of the nonblocking implementation of the stack object.

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 [Boehm et al. 1991].

7.1. Correctness of Concurrent GCs

A concurrent GC is executed by a dedicated thread and performs the collection work in parallel with user threads (mutators), which access the shared heap via read, write, and allocation operations. To ensure that the GC and the mutators share a coherent view of the heap, the heap operations from mutators may be instrumented with extra operations, which provide an interaction mechanism to allow arbitrary mutators to cooperate with the GC. These instrumented heap operations are called barriers (e.g., read barriers, 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 preserve the mutators' intended behaviors and generate the abstract view for high-level programmers. Usually this part is examined by experts and then trusted.

Here we propose a more direct approach. We view a concurrent 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,

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

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

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 Verification Framework

The compositionality of RGSim allows us to develop a general framework to prove $\mathsf{Correct}(\mathbf{T})$, which is much more difficult using monolithic proof methods. By the parallel compositionality of RGSim (the PAR rule in Figure 7), we can decompose the refinement proofs into proofs for the GC thread and each mutator thread. For a mutator thread, we can further decompose the refinement proof into proof for each primitive instruction, using the compositionality of RGSim (the rules SEQ, IF and WHILE in Figure 7).

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

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

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 $\mathcal{R}_{\text{gc}}; \mathcal{G}_{\text{gc}} \vdash \{p_{\text{gc}}\} C_{\text{gc}} \{q_{\text{gc}}\}$ in a standard rely-guarantee logic with proper $\mathcal{R}_{\text{gc}}$, $\mathcal{G}_{\text{gc}}$, p_{gc} and q_{gc}, as long as $\mathcal{G}_{\text{gc}}$ is a concrete representation of $\mathsf{AbsGCStep}$. The judgment says given an initial state satisfying the precondition p_{gc}, if the environment's behaviors satisfy $\mathcal{R}_{\text{gc}}$, then each step of C_{gc} satisfies $\mathcal{G}_{\text{gc}}$, and the postcondition q_{gc} holds at the end if C_{gc} terminates. In general, the collector never terminates, thus

we can let q_{gc} be **false**. $\mathcal{G}_{\text{gc}}$ and p_{gc} should be provided by the verifier, where p_{gc} needs to be general enough so that it can be satisfied by any possible low-level initial state. $\mathcal{R}_{\text{gc}}$ encodes the possible behaviors of mutators, which can be derived, as we will show shortly.

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, following the compositionality of RGSim. The proof needs proper rely/guarantee conditions. Let $\mathbb{G}_{\text{c}}^{\text{t}}$ and $\mathcal{G}_{\mathbf{T}(\text{c})}^{\text{t}}$ denote the guarantees of the source instruction c and the target code $\mathbf{T}(\text{c})$ for the mutator thread t respectively. Then we can define the general guarantees for the thread.

$$\mathcal{G}(\text{t}) \triangleq \bigcup_{\text{c}} \mathcal{G}_{\mathbf{T}(\text{c})}^{\text{t}}\,; \qquad \mathbb{G}(\text{t}) \triangleq \bigcup_{\text{c}} \mathbb{G}_{\text{c}}^{\text{t}}\,. \tag{7.1}$$

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

$$\mathcal{R}(\text{t}) \triangleq \mathcal{G}_{\text{gc}} \cup \bigcup_{\text{t}' \neq \text{t}} \mathcal{G}(\text{t}')\,; \qquad \mathbb{R}(\text{t}) \triangleq \mathsf{AbsGCStep} \cup \bigcup_{\text{t}' \neq \text{t}} \mathbb{G}(\text{t}')\,. \tag{7.2}$$

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

$$\mathcal{R}_{\text{gc}} \triangleq \bigcup_{\text{t}} \mathcal{G}(\text{t}) \tag{7.3}$$

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

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

THEOREM 7.1 (VERIFYING CONCURRENT GARBAGE COLLECTING SYSTEMS). *If there exist* $\mathbb{G}_{\text{c}}^{\text{t}}$, $\mathcal{G}_{\mathbf{T}(\text{c})}^{\text{t}}$, $\zeta(\text{t})$, α, $\mathcal{G}_{\text{gc}}$ *and* p_{gc} *(for any* c *and* t*) such that the following hold (where* $\mathcal{G}(\text{t})$, $\mathbb{G}(\text{t})$, $\mathcal{R}(\text{t})$, $\mathbb{R}(\text{t})$ *and* $\mathcal{R}_{\text{gc}}$ *are defined in (7.1), (7.2) and (7.3), and* $\mathsf{Good}_{\mathbf{T}}(\zeta(\text{t}))$ *defined in (7.4) holds):*

(*1*) *(Correctness of* $\mathbf{T}$ *on mutator instructions)*
$\forall \text{t}, \text{c}.\ (\mathbf{T}(\text{c}), \mathcal{R}(\text{t}), \mathcal{G}(\text{t})) \preceq^{\text{t}}_{\alpha;\zeta(\text{t})\square\zeta(\text{t})} (\text{c}, \mathbb{R}(\text{t}), \mathbb{G}(\text{t}))$*;*

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

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

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

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

— Define the α and $\zeta(\text{t})$ relations, and prove the correctness of $\mathbf{T}$ on high-level primitive instructions. Since $\mathbf{T}$ preserves the syntax on most instructions, it's often immediate to prove the target instructions are simulated by their sources. But for

instructions that are transformed to barriers, we need to verify that the barriers implement both the source instructions (by RGSim) and the interaction mechanism (shown in their guarantees).
— Find some proper $\mathcal{G}_{\text{gc}}$ and p_{gc}, and verify the GC code by R-G reasoning. We require the GC's guarantee $\mathcal{G}_{\text{gc}}$ should not contain more behaviors than 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).

To prove Theorem 7.1, we first prove the following from (2) and (3).

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

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

$$\begin{array}{c}\forall \mathbb{C}_1, \ldots, \mathbb{C}_n.\ (\mathsf{t}_{\text{gc}}.C_{\text{gc}} \parallel \mathsf{t}_1.\mathbf{T}(\mathbb{C}_1) \parallel \ldots \parallel \mathsf{t}_n.\mathbf{T}(\mathbb{C}_n), \mathsf{Id}, \mathsf{True}) \\ \preceq_{\alpha;\zeta\,\square\,\zeta}\ (\mathsf{t}_{\text{gc}}.AbsGC \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \,|||\, \mathsf{t}_n.\mathbb{C}_n, \mathsf{Id}, \mathsf{True})\end{array}$$

Here $\zeta \triangleq \zeta_{\text{gc}} \cap \bigcap_{\mathsf{t}} \zeta(\mathsf{t})$. Finally, from the soundness of RGSim (Corollary 4.4), we can conclude Correct($\mathbf{T}$).

7.3. Application: Boehm et al. Concurrent GC Algorithm

We illustrate the applications of the framework (Theorem 7.1) by proving the correctness of a mostly concurrent mark-sweep garbage collector proposed by Boehm et al. [1991]. Variants of the algorithm have been used in practice (e.g., by IBM [Barabash et al. 2005]).

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

We show the code of the GC thread in Figure 15. We assume each object contains m pointer fields $\text{pt}_1, \ldots, \text{pt}_m$, a data field, and two auxiliary `color` and `dirty` fields. The `color` field has three possible values and is used for two purposes: for marking, we use `BLACK` for a marked object and `WHITE` for an unmarked one; and for allocation, we use `BLUE` for an unallocated object which will neither be traced nor be reclaimed, but can be allocated later. New objects are created `BLACK`, and when reclaiming an object, we just set its color to `BLUE`. The `dirty` field is the card bit whose value can be 0 (not dirty) or 1 (dirty). We also assume the total number of threads is `N` and the heap domain is `[1..M]`.

To make the GC code more readable, we divide it into several methods in Figure 15, which should be viewed as macros. The GC thread executes `Collection()` and repeats the collection cycle (the loop body in the method) forever. In each collection cycle, it first clears the `dirty` cards and resets the `colors` of all the objects (the method

```
 1  constant int WHITE, BLACK, BLUE; // colors
 2  constant int N; // total number of threads
 3  constant int M; // size of heap
 4
 5  Collection() {
 6    local mstk;
 7    while (true) {
 8      Initialize();
 9      Trace();
10      CleanCard();
11      atomic{ ScanRoot(); CleanCard(); }
12      Sweep();
13    }
14  }
15
16  Initialize() {
17    local i, c;
18    i := 1;
19    while (i <= M) {
20      i.dirty := 0;
21      c := i.color;
22      if (c = BLACK) { i.color := WHITE; }
23      i := i + 1;
24    }
25  }
26
27  Trace() {
28    local t, rt, i;
29    t := 1;
30    while (t <= N) {
31      rt := get_root(t);
32      foreach i in rt do {
33        MarkAndPush(i);
34      }
35      t := t + 1;
36      TraceStack();
37    }
38  }
39
40  TraceStack() {
41    local i, j;
42    while (!is_empty(mstk)) {
43      i := pop(mstk);
44      j := i.pt_1; MarkAndPush(j);
45      ...
46      j := i.pt_m; MarkAndPush(j);
47    }
48  }
49
50  MarkAndPush(i) {
51    local c;
52    if (i != 0) {
53      c := i.color;
54      if (c = WHITE) {
55        i.color := BLACK;
56        push(i, mstk);
57      }
58    }
59  }
60
61  CleanCard() {
62    local i, c, d;
63    i := 1;
64    while (i <= M) {
65      c := i.color;
66      d := i.dirty;
67      if (d = 1) {
68        i.dirty := 0;
69        if (c = BLACK) {
70          push(i, mstk);
71        }
72      }
73      i := i + 1;
74    }
75    TraceStack();
76  }
77
78  ScanRoot() {
79    local t, rt, i;
80    t := 1;
81    while (t <= N) {
82      rt := get_root(t);
83      foreach i in rt do {
84        MarkAndPush(i);
85      }
86      t := t + 1;
87    }
88  }
89
90  Sweep() {
91    local i, c;
92    i := 1;
93    while (i <= M) {
94      c := i.color;
95      if (c = WHITE) { free(i); }
96      i := i + 1;
97    }
98  }
```

Fig. 15. The code of Boehm et al. GC.

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

Fig. 16. The write barrier for Boehm et al. GC.

call of `Initialize()`). After the initialization, the GC enters the mark phase by calling `Trace()`. The command `rt` := **get_root**(`t`) (line 31) allows the GC to read the values of all the pointer variables in the thread `t`'s store at once to a set `rt`, and **foreach** i **in** `rt` **do** C allows to execute C for every value i in `rt`. Our atomic **get_root** tries to reflect the real-world GC implementation [Barabash et al. 2005], where the GC stops a mutator thread to scan its roots. A *mark stack* `mstk` is used to do the depth-first tracing in the method `TraceStack()`. For simplicity, we assume there are primitive commands **push**(x, `mstk`) and x := **pop**(`mstk`) to manipulate `mstk`. The stop-the-world phase (line 11) is implemented by **atomic**{C}. Here the roots are re-scanned in `ScanRoot()`, because the write barrier is not applied to the roots and we should assume conservatively that they have been modified. In the sweep phase (the call of `Sweep()` at line 12), the GC can use **free**(x) to reclaim the object x. Usually in practice, there is also a concurrent card-cleaning phase (the call of `CleanCard()` at line 10) before the stop-the-world card-cleaning (at line 11) to reduce the pause time of the latter.

The write barrier is shown in Figure 16, 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` for the purpose of verification only, which can be safely deleted after the proof is completed. We use `aux` to help specify some fine-grained and temporal property of the write barrier in the guarantees. For instance, a mutator should ensure that after it sets a pointer field of an object x to another object y, it must first set x's `dirty` field before updating other pointers (in particular, those pointing to y). Otherwise, the GC may not know that y is newly reachable from x and may finally reclaim y. In Figure 16 we set `aux` to the object x when its pointer field is updated, and specify in the mutator's guarantee ($\mathcal{G}^{\mathsf{t}}_{\text{set\_dirty}}$ in Figure 25(b)) that when `aux` $= x$, it must set x's `dirty` field. The GC does not use read barriers nor allocation barriers. Allocation can be implemented using a standard concurrent list algorithm. To be more focused on verifying the GC algorithm itself, we model allocation as an abstract instruction x := **new**() which can magically find an unallocated (`BLUE`) object in the heap.

7.3.2. The Transformation. We first present the detailed high-level and low-level languages and state models in Figures 17 and 18 respectively, which are instantiations of the generic languages in Figure 2.

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

— The behaviors of the high-level abstract GC thread are defined in Figure 19(a), saying that the mutator stores and the reachable objects in the heap remain unmodified. Here Reachable(l)(Π, H) means the object at the location l is reachable in H from the roots in Π.

— The low-level concrete GC thread could use privileged commands, such as x := **get_root**(y) and **free**(x), to control the mutator threads and manage the heap.

— High-level mutators can use $x := y$.fd to read a field of an object, x.fd := $\mathbb{E}$ to write the value of $\mathbb{E}$ to a field of an object and x := ***new***() to allocate a new object. If

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

(a) the language

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

(b) program states

Fig. 17. The high-level language and state model.

the instruction $x.\mathrm{fd} := \mathbb{E}$ updates a pointer field (i.e., $\mathrm{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}$), then it will be transformed to the write barrier in Figure 16. Note here $\mathbb{E}$ is restricted to be either **nil** (null pointers) or pointer variables.

— The high-level language is typed in the sense that heap locations and integers are regarded as distinct kinds (or types) of values. We present the high-level operational semantics in Figure 19(b). Here we use $\mathsf{SameType}(\mathrm{V}, \mathrm{V}')$ to mean that the two values V and V′ are of the same type.

— On the low-level machine, we allow the GC to perform pointer arithmetic, so we do not distinguish locations and integers. A low-level value v can be an integer, a set, or a sequence of integers. We use $\mathcal{P}(\_)$ for the power set and $\mathit{Seq}(\_)$ for the set of sequences. Every low-level variable is given an extra bit to preserve its high-level type information (0 for nonpointers and 1 for pointers), so that the GC can easily get the roots. The low-level mutators are still prohibited from pointer arithmetic. An expression E is evaluated (shown in Figure 20) under the store with an extra tag *tag* to indicate whether it is used as an object location in the heap ($\mathit{tag} = 1$ if E is used as a heap location; and $\mathit{tag} = 0$ otherwise). When $\mathit{tag} = 2$, we do not care about the usage of the expression, and such an expression will be used in the GC code since the GC has the privilege to use an integer as an address and vice versa. We present part of the low-level operational semantics rules in Figure 21. To formulate the semantics of **foreach** x **in** y **do** C, we assume x and y are temporary variables and not updated by C. At the beginning of each iteration, we set x to an

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

(a) the language

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

(b) program states

Fig. 18. The low-level language and state model.

arbitrary item in the set y, and after executing C we remove that item from y. The **foreach** loop terminates when y becomes empty.

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

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

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

The transformation $\mathbf{T}$ is defined as follows. For *code*, the high-level abstract GC thread is transformed to the GC thread shown in Figure 15. Each instruction $x.\mathrm{fd} := \mathbb{E}$ in mutators is transformed to the write barrier $\texttt{update}(x, \mathrm{fd}, \mathbf{T}(\mathbb{E}))$, where fd is a pointer field of x. $\mathbf{T}$ over expressions $\mathbb{E}$ returns 0 if $\mathbb{E}$ is **nil**, and keeps the syntax otherwise. Other instructions and the program structures of mutators are unchanged.

We also need to transform the initial high-level state to the low level. The transformation $\mathbf{T}(\Sigma)$ is defined in Figure 22.

— First we require the high-level initial state to be *well-formed* ($\mathsf{WfState}(\Sigma)$), that is, reachable locations cannot be dangling pointers.

— High-level locations are transformed to integers by the bijective function $\mathsf{Loc2Int}$.

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

(a) definition of AbsGCStep

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

$$
\frac{x \notin dom(\mathrm{S}) \quad \text{or} \quad \mathrm{S}(x) \notin dom(\mathrm{H}) \quad \text{or} \quad [\![\mathbb{E}]\!]_{\mathrm{S}} = \bot \quad \text{or} \quad \neg\mathsf{SameType}(\mathrm{H}(\mathrm{S}(x))(\mathrm{fd}), [\![\mathbb{E}]\!]_{\mathrm{S}})}{(x.\mathrm{fd} := \mathbb{E}, (\Pi \uplus \{\mathrm{t} \ \square \ \mathrm{S}\}, \mathrm{H})) \longrightarrow_{\mathrm{t}} \textbf{abort}}
$$

$$
\frac{l \notin dom(\mathrm{H}) \quad l \neq \mathbf{nil} \quad \mathrm{S}(x) = l' \quad \mathrm{S}' = \mathrm{S}\{x \ \square \ l\} \quad \mathrm{H}' = \mathrm{H} \uplus \{l \ \square \ \{\mathrm{pt}_1 \ \square \ \mathbf{nil}, \ldots, \mathrm{pt}_m \ \square \ \mathbf{nil}, \mathrm{data} \ \square \ 0\}\}}{(x := \textbf{\textit{new}}(), (\Pi \uplus \{\mathrm{t} \ \square \ \mathrm{S}\}, \mathrm{H})) \longrightarrow_{\mathrm{t}} (\textbf{\textit{skip}}, (\Pi \uplus \{\mathrm{t} \ \square \ \mathrm{S}'\}, \mathrm{H}'))}
$$

$$
\frac{\neg(\exists l.\, l \notin dom(\mathrm{H}) \wedge l \neq \mathbf{nil}) \quad \mathrm{S}(x) = l' \quad \mathrm{S}' = \mathrm{S}\{x \ \square \ \mathbf{nil}\}}{(x := \textbf{\textit{new}}(), (\Pi \uplus \{\mathrm{t} \ \square \ \mathrm{S}\}, \mathrm{H})) \longrightarrow_{\mathrm{t}} (\textbf{\textit{skip}}, (\Pi \uplus \{\mathrm{t} \ \square \ \mathrm{S}'\}, \mathrm{H}))}
$$

$$
\frac{x \notin dom(\mathrm{S}) \quad \text{or} \quad \neg\exists l.\mathrm{S}(x) = l}{(x := \textbf{\textit{new}}(), (\Pi \uplus \{\mathrm{t} \ \square \ \mathrm{S}\}, \mathrm{H})) \longrightarrow_{\mathrm{t}} \textbf{abort}} \qquad \frac{(\mathbb{C}_i, \Sigma) \longrightarrow_{\mathrm{t}_i} \textbf{abort}}{(\mathrm{t_{gc}}.\mathit{AbsGC} ||| \mathrm{t}_1.\mathbb{C}_1 ||| \ldots \mathrm{t}_i.\mathbb{C}_i \ldots ||| \mathrm{t}_n.\mathbb{C}_n, \Sigma) \longrightarrow \textbf{abort}}
$$

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

(b) selected operational semantics rules

Fig. 19. A high-level garbage-collected machine.

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

Fig. 20. Expression evaluation on the low-level machine.

— Variables are transformed to the low level using an extra bit to preserve the high-level type information (0 for nonpointers and 1 for pointers). Usually we use v^{np} and v^{p} short for $(v, 0)$ and $(v, 1)$ respectively.

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

$$\frac{s(x) = (\_, b) \quad [\![E]\!]_{(s,2)} = n \quad s' = s\{x \mapsto (n, b)\}}{(x := E, (\pi \uplus \{\mathrm{t_{gc}} \mapsto s\}, h)) \longrightarrow_{\mathrm{t_{gc}}} (\mathbf{skip}, (\pi \uplus \{\mathrm{t_{gc}} \mapsto s'\}, h))}$$

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

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

$$\frac{s(y) = (\mathrm{t}, 0) \quad s(x) = (\_, 0) \quad \pi(\mathrm{t}) = s_{\mathrm{t}} \quad S = \{n \mid \exists x. s_{\mathrm{t}}(x) = (n, 1)\} \quad s' = s\{x \mapsto (S, 0)\}}{(x := \mathbf{get\_root}(y), (\pi \uplus \{\mathrm{t_{gc}} \mapsto s\}, h)) \longrightarrow_{\mathrm{t_{gc}}} (\mathbf{skip}, (\pi \uplus \{\mathrm{t_{gc}} \mapsto s'\}, h))}$$

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

$$\frac{s(x) = (\_, b) \quad s(y) = (\{n_1, \ldots, n_k\}, 0) \quad s' = s\{x \mapsto (n_1, b)\}}{(\mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, (\pi \uplus \{\mathrm{t_{gc}} \mapsto s\}, h)) \longrightarrow_{\mathrm{t_{gc}}} (C; y := y \backslash \{x\}; \mathbf{foreach}\ x\ \mathbf{in}\ y\ \mathbf{do}\ C, (\pi \uplus \{\mathrm{t_{gc}} \mapsto s'\}, h))}$$

$$\frac{(C, (\pi \uplus \{\mathrm{t} \mapsto s\}, h)) \longrightarrow_{\mathrm{t}}^{*} (\mathbf{skip}, (\pi \uplus \{\mathrm{t} \mapsto s'\}, h'))}{(\mathbf{atomic}\{C\}, (\pi \uplus \{\mathrm{t} \mapsto s\}, h)) \longrightarrow_{\mathrm{t}} (\mathbf{skip}, (\pi \uplus \{\mathrm{t} \mapsto s'\}, h'))} \qquad \frac{(C, (\pi \uplus \{\mathrm{t} \mapsto s\}, h)) \longrightarrow_{\mathrm{t}}^{*} \mathbf{abort}}{(\mathbf{atomic}\{C\}, (\pi \uplus \{\mathrm{t} \mapsto s\}, h)) \longrightarrow_{\mathrm{t}} \mathbf{abort}}$$

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

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

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

$$\frac{s(x) = (n', b) \quad s(y) = (A, 0) \quad s' = s\{y \mapsto (n' :: A, 0)\}}{(\mathbf{push}(x, y), (\pi \uplus \{\mathrm{t_{gc}} \mapsto s\}, h)) \longrightarrow_{\mathrm{t_{gc}}} (\mathbf{skip}, (\pi \uplus \{\mathrm{t_{gc}} \mapsto s'\}, h))}$$

$$\frac{s(x) = (\_, b) \quad s(y) = (n :: A, 0) \quad s' = s\{x \mapsto (n, b), y \mapsto (A, 0)\}}{(x := \mathbf{pop}(y), (\pi \uplus \{\mathrm{t_{gc}} \mapsto s\}, h)) \longrightarrow_{\mathrm{t_{gc}}} (\mathbf{skip}, (\pi \uplus \{\mathrm{t_{gc}} \mapsto s'\}, h))}$$

Fig. 21. Selected operational semantics rules on the low-level machine.

— 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 `colors` are `BLUE` and all the other fields are initialized by 0.

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

where

$$\mathsf{WfState}(\Pi, \mathrm{H}) \triangleq \forall l.\ \mathsf{Reachable}(l)(\Pi, \mathrm{H}) \Longrightarrow l \in dom(\mathrm{H})$$

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

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

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

Fig. 22. The transformation $\mathbf{T}$ on initial states for Boehm et al. GC.

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

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

$$\mathsf{unalloc}(o, \mathrm{H}, l) \triangleq (o = \{\mathrm{pt}_1 \sqsubset \_, \ldots, \mathrm{pt}_m \sqsubset \_, \mathrm{data} \sqsubset \_, \mathtt{color} \sqsubset \mathtt{BLUE}, \mathtt{dirty} \sqsubset \_\}) \wedge l \notin dom(\mathrm{H})$$

$$\mathsf{heap\_map}(h, \mathrm{H}) \triangleq \forall i, l.\ 1 \leq i \leq M \wedge \mathsf{Loc2Int}(l) = i \Longrightarrow \mathsf{obj\_map}(h(i), \mathrm{H}(l)) \vee \mathsf{unalloc}(h(i), \mathrm{H}, l)$$

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

Fig. 23. The α relation for Boehm et al. GC.

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

To prove $\mathsf{Correct}(\mathbf{T})$ in our framework, we apply Theorem 7.1, prove the refinement between low-level and high-level mutators, and verify the GC code using a unary Rely-Guarantee-based logic.

7.3.3. Refinement Proofs for Mutator Instructions. We first define the α and $\zeta(\mathrm{t})$ relations. In α (see Figure 23), the relations between low-level and high-level stores and heaps are enforced by store_map and heap_map respectively. Their definitions reflect the state transformations we describe before, ignoring the values of those high-level-invisible structures (e.g., the GC's local variables, the `color` and `dirty` fields for nonblue objects and all the fields of blue objects). α also requires the well-formedness of high-level states. Here we still use Loc2Int to relate integers and locations.

For each mutator thread t, the $\zeta(\mathrm{t})$ 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 16) be a null pointer (0^{p}).

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

To define the guarantees of the mutator instructions, we first introduce some separation logic assertions in Figure 24 to describe states. Following Parkinson et al. [2006], we treat program variables as resource and use $\mathsf{own}_{\mathrm{p}}(x)$ and $\mathsf{own}_{\mathrm{np}}(x)$ for the current thread's ownerships of pointers and nonpointers respectively. They are interpreted under (π, s, h), where s is the store of the current thread, π consists of the stores of all the other threads and h is the shared heap. We use $E_1.fd \mapsto E_2$ to specify a single-object single-field heap with E_2 stored in the field fd of the object E_1. The separating conjunction $p * q$ means p and q hold on disjoint states. We define the disjoint union of states in Figure 24(c). We use $f_1 \uplus f_2$ as usual to denote the union of two partial functions when their domains are disjoint. Since heaps are curried functions that first map locations to objects, which then map field names to values, they can be transformed to an uncurried form by the uncurry operator. We then use $h_1 \oplus h_2$ to denote the union when their domains of $\mathsf{uncurry}(h_1)$ and $\mathsf{uncurry}(h_2)$ are disjoint. The disjoint union of states is defined based on the disjoint unions of the shared heaps and the stores for each thread. We use $E_1.fd \hookrightarrow E_2$ for $E_1.fd \mapsto E_2 * \mathbf{true}$ and $\circledast_{x \in S}.p(x)$ for iterated separating conjunction over the set S. We overload the notations to the high-level machine and use $\mathbb{E}_1.\mathrm{fd} \mapsto \mathbb{E}_2$ for a single-object single-field heap at the high level.

In Figure 24(d), we define two forms of actions. $p \ltimes_{\mathrm{t}} q$ represents the update over the current thread t's store and the shared heap, which is defined similarly as in Figure 11(b). $p \ltimes_{\mathrm{t}} q$ provided p' ensures that the context p' is not changed by the action.

In Figure 25, we give the guarantees of the high-level mutator instructions and the transformed code, which are defined following their operational semantics. We use $(x^{\mathrm{p}} = n)$ short for $(x = n) \wedge \mathsf{own}_{\mathrm{p}}(x)$ and $(x^{\mathrm{np}} = n)$ for $(x = n) \wedge \mathsf{own}_{\mathrm{np}}(x)$. When the context is clear, we omit the superscript. The predicates blueobj and newobj denote a blue object and a newly allocated object, which are defined in Figure 27. Each action just accesses the local store of the mutator and will not touch the GC store.

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

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

where $\mathcal{G}^{\mathrm{t}}_{\mathrm{write\_barrier}} \triangleq \mathcal{G}^{\mathrm{t}}_{\mathrm{write\_pt}} \cup \mathcal{G}^{\mathrm{t}}_{\mathrm{set\_dirty}}$, that is, the guarantee of the low-level two-step write barrier. $\mathbb{G}^{\mathrm{t}}_{\mathrm{write\_pt}}$ is the guarantee of the high-level atomic write operation. Recall $\mathcal{R}(\mathrm{t})$ and $\mathbb{R}(\mathrm{t})$ are defined in Eq. (7.2) in Section 7.2. Since the transformation of other high-level instructions is identity, the corresponding refinement proofs are simple. For example, we can prove

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

7.3.4. Rely-Guarantee Reasoning about the GC Code. We use a unary logic to verify the GC thread. The proof details here are orthogonal to our simulation-based proof (but it is RGSim that allows us to derive Theorem 7.1, which then links proofs in the unary logic with relational proofs). Thus shortly we only give a sketch of the assertion language, the unary logic, the precondition and the guarantee of the GC thread, the key invariants, and the proof structure.

$$(PVarList) \quad O ::= \bullet \mid x, O$$
$$(StateAssert) \quad p, q \in LThrds \times LStore \times LHeap \to Prop$$

(a) state assertions

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

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

$$f_1 \perp f_2 \triangleq dom(f_1) \cap dom(f_2) = \emptyset$$

$$f_1 \uplus f_2 \triangleq \begin{cases} f_1 \cup f_2 & \text{if } f_1 \perp f_2 \\ \perp & \text{otherwise} \end{cases}$$

$$h_1 \oplus h_2 \triangleq \begin{cases} \mathsf{curry}(\mathsf{uncurry}(h_1) \cup \mathsf{uncurry}(h_2)) & \text{if } \mathsf{uncurry}(h_1) \perp \mathsf{uncurry}(h_2) \\ \perp & \text{otherwise} \end{cases}$$

$$\pi_1 \oplus \pi_2 \triangleq \begin{cases} \{\mathsf{t} \ \square \ (\pi_1(\mathsf{t}) \uplus \pi_2(\mathsf{t})) \mid \mathsf{t} \in dom(\pi_1)\} & \\ \qquad \text{if } dom(\pi_1) = dom(\pi_2) \wedge \ \forall \mathsf{t} \in dom(\pi_1).\ \pi_1(\mathsf{t}) \perp \pi_2(\mathsf{t}) & \\ \perp \qquad \text{otherwise} & \end{cases}$$

$$\sigma_1 \oplus \sigma_2 \triangleq \begin{cases} (\pi, h) & \text{if } \sigma_1 = (\pi_1, h_1) \wedge \sigma_2 = (\pi_2, h_2) \wedge \pi_1 \oplus \pi_2 = \pi \wedge h_1 \oplus h_2 = h \\ \perp & \text{otherwise} \end{cases}$$

(c) disjoint unions

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

(d) actions

Fig. 24. Semantics of basic assertions.

The unary program logic we use to verify the GC thread is a standard rely-guarantee logic adapted to the target language. The assertions are defined in Figure 24 and discussed before. We show the inference rules in Figure 26. Rules on the top half are for

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

(a) high-level guarantees

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

(b) low-level guarantees

Fig. 25. Guarantees of mutator instructions.

sequential reasoning. Most are exactly the same as separation logic [Reynolds 2002] and omitted here. The figure only shows some rules we added for the GC-specific commands (e.g., $x := \mathbf{get\_root}(y)$) and some particular heap manipulation rules adapted to our low-level machine model (e.g., $\mathbf{free}(x)$ just sets the object's color to BLUE). The concurrency rules in the bottom half follow standard rely-guarantee reasoning. The soundness of the logic with respect to the operational semantics is straightforward and we omit the proofs here.

To verify the GC code, we first give the precondition and the guarantee of the GC. The GC starts its executions from a low-level *well-formed* state, that is, $p_{\mathrm{gc}} \triangleq \mathsf{wfstate}$. Just corresponding to the high-level WfState definition (see Figure 22), the low-level wfstate predicate says that none of the reachable objects is BLUE, as

$$\mathsf{wfstate} \triangleq \circledast_{x \in [1..M]}.\mathsf{obj}(x) \wedge (\forall x.\ \mathsf{reachable}(x) \Rightarrow \mathsf{not\_blue}(x)),$$

where $\mathsf{obj}(x)$ means x is a low-level heap location with the $\mathrm{pt}_1, \ldots, \mathrm{pt}_m$, data, color and dirty fields, $\mathsf{reachable}(x)$ is defined similarly to the high-level definition in Figure 19, and $\mathsf{not\_blue}(x)$ is defined in Figure 27. We define $\mathcal{G}_{\mathrm{gc}}$ as follows.

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

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

$$\frac{}{\{x.\mathtt{color} \mapsto \_\} \mathbf{free}(x) \{x.\mathtt{color} \mapsto \mathtt{BLUE}\}} \ (\mathrm{FREE})$$

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

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

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

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

Fig. 26. Selected inference rules for GC verification.

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.

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

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

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

We present the proof of the top-level collection cycle in Figure 28. One of the key invariants used in the proofs is reach_inv (defined in Figure 27). It says, if a reachable BLACK object x points to a WHITE object y, then either x is dirty or a mutator is going to dirty x (the predicate todirty(x, y) holds). The latter occurs when the mutator thread t has done the first step of its write barrier update(x, fd, y). We have t.aux $= x$ and from the mutator's guarantees (Figure 25(b)), we know t must be going to dirty x.

Since each instruction in the GC code is executed atomically, we need to stabilize the pre- and postconditions when verifying it (e.g., see the ATOM-R rule in Figure 26). 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.

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

Here ptfd_sta$(\mathtt{i}.\mathrm{pt}_1, X)$ says either the pt_1 field of i is X, or i is (or is going to be) marked as dirty. Similarly, when reading the color of an object, the postcondition should take

$$
\begin{array}{ll}
\mathsf{obj}(x) & \triangleq x.\mathrm{pt}_1 \mapsto \_ * \ldots * x.\mathrm{pt}_m \mapsto \_ * x.\mathtt{data} \mapsto \_ * x.\mathtt{color} \mapsto \_ * x.\mathtt{dirty} \mapsto \_ * \mathbf{true} \\
\mathsf{blueobj}(x) & \triangleq x.\mathrm{pt}_1 \mapsto \_ * \ldots * x.\mathrm{pt}_m \mapsto \_ * x.\mathtt{data} \mapsto \_ * x.\mathtt{color} \mapsto \mathtt{BLUE} * x.\mathtt{dirty} \mapsto \_ \\
\mathsf{newobj}(x) & \triangleq x.\mathrm{pt}_1 \mapsto 0 * \ldots * x.\mathrm{pt}_m \mapsto 0 * x.\mathtt{data} \mapsto 0 * x.\mathtt{color} \mapsto \mathtt{BLACK} * x.\mathtt{dirty} \mapsto 0 \\
\mathsf{black}(x) & \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{root}(\mathrm{t}, S) & \triangleq \lambda(\pi, s, h).\ \exists s_{\mathrm{t}}.\ s_{\mathrm{t}} = \pi(\mathrm{t}) \wedge S = \{n \mid \exists x.\ s_{\mathrm{t}}(x) = (n, 1) \wedge x \neq \mathtt{aux}\} \\
\mathsf{edge}(x, y) & \triangleq \exists fd \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}.\ (x.fd \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}(\mathrm{t}, x) & \triangleq \exists S, y.\ \mathsf{root}(\mathrm{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) \wedge (\forall x.\ \mathsf{reachable}(x) \implies \mathsf{not\_blue}(x)) \\
\mathsf{white\_edge}(x, fd, y) & \triangleq (x.fd \hookrightarrow y) \wedge \mathsf{white}(y) \wedge fd \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \\
\mathsf{white\_edge}(x, y) & \triangleq \exists fd.\ \mathsf{white\_edge}(x, fd, y) \\
\mathsf{todirty}(x, n) & \triangleq \exists \mathrm{t}, S.\ (\mathrm{t}.\mathtt{aux} = x \wedge \mathsf{root}(\mathrm{t}, S) \wedge n \in S) \\
\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) \implies \mathsf{black}(x) \\
\mathsf{reach\_inv} & \triangleq \forall x, y.\ \mathsf{reachable}(x) \wedge \mathsf{black}(x) \wedge \mathsf{white\_edge}(x, y) \\
 & \quad \implies \mathsf{dirty}(x) \vee \mathsf{todirty}(x, y) \\
\mathsf{reach\_stk}(A) & \triangleq \forall x, y.\ \mathsf{reachable}(x) \wedge \mathsf{black}(x) \wedge \mathsf{white\_edge}(x, y) \\
 & \quad \implies \mathsf{dirty}(x) \vee \mathsf{todirty}(x, y) \vee \mathsf{instk}(x, A) \\
\mathsf{reach\_tomk}(A, x_p, S_f, x_n) & \triangleq \forall x, fd, y.\ \mathsf{reachable}(x) \wedge \mathsf{black}(x) \wedge \mathsf{white\_edge}(x, fd, y) \\
 & \quad \implies \mathsf{dirty}(x) \vee \mathsf{todirty}(x, y) \vee \mathsf{instk}(x, A) \vee (x = x_p \wedge fd \in S_f) \vee (y = x_n) \\
\mathsf{reach\_black} & \triangleq \forall x.\ \mathsf{reachable}(x) \implies \mathsf{black}(x) \\
\mathsf{ptfd\_sta}(x.fd, y) & \triangleq \exists n.\ (x.fd \hookrightarrow n) \wedge (y = n \vee \mathsf{dirty}(x) \vee n = 0 \vee \mathsf{todirty}(x, n)) \\
\mathsf{newobj\_sta}(x) & \triangleq \mathsf{obj}(x) \wedge \mathsf{black}(x) \wedge \forall fd \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}.\ \mathsf{ptfd\_sta}(x.fd, 0) \\
\mathsf{rt\_black}(t) & \triangleq \exists S.\ \mathsf{root}(t, S) \wedge \forall n \in S.\ \mathsf{black}(n) \\
\mathsf{rt\_black} & \triangleq \forall t \in [1..N].\ \mathsf{rt\_black}(t) \\
\mathsf{mark\_rt\_till}(n) & \triangleq \forall t \in [1..n].\ \mathsf{rt\_black}(t) \\
\mathsf{clear\_color\_till}(n) & \triangleq \forall x \in [1..n].\ (x.\mathtt{color} \hookrightarrow \mathtt{BLACK} \implies \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)
\end{array}
$$

Fig. 27. Useful assertions for verifying Boehm et al. GC.

into account the mutators' possible update of the color field in allocation and the updates of pointer fields after allocation.

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

Here $\mathsf{newobj\_sta}(\mathtt{i})$ says i points to a new object whose color field is BLACK, and each pointer field is either 0 or the object is dirty. Both the predicates ptfd_sta and newobj_sta are defined in Figure 27.

{wfstate}
Collection() {
  local mstk: Seq(Int);
  Loop Invariant: {wfstate $*$ (own$_{np}$(mstk) $\wedge$ mstk $= \epsilon$)}
  while (true) {
    Initialize();
    {(wfstate $\wedge$ reach_inv) $*$ (own$_{np}$(mstk) $\wedge$ mstk $= \epsilon$)}
    Trace();
    {(wfstate $\wedge$ reach_inv) $*$ (own$_{np}$(mstk) $\wedge$ mstk $= \epsilon$)}
    CleanCard();
    {(wfstate $\wedge$ reach_inv) $*$ (own$_{np}$(mstk) $\wedge$ mstk $= \epsilon$)}
    atomic{
      ScanRoot();
      {$\exists X$.(wfstate $\wedge$ reach_stk(X) $\wedge$ stk_black(X) $\wedge$ rt_black) $*$ (own$_{np}$(mstk) $\wedge$ mstk $= X$)}
      CleanCard();
    }
    {(wfstate $\wedge$ reach_black) $*$ (own$_{np}$(mstk) $\wedge$ mstk $= \epsilon$)}
    Sweep();
  }
}
{**false**}

Fig. 28. Proof outline of Collection().

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

$$\mathcal{R}_{gc}; \mathcal{G}_{gc} \vdash \begin{array}{l} \left\{ \begin{array}{l} \exists A.\ \mathsf{wfstate} \wedge \mathsf{reach\_tomk}(A, x_p, S_f, \mathtt{i}) \\ \wedge\ \mathsf{stk\_black}(A) \wedge (\mathtt{i} = 0 \vee \mathsf{obj}(\mathtt{i})) \end{array} \right\} \\ \quad \mathtt{MarkAndPush(i)}; \\ \left\{ \begin{array}{l} \exists A.\ \mathsf{wfstate} \wedge \mathsf{reach\_tomk}(A, x_p, S_f, 0) \\ \wedge\ \mathsf{stk\_black}(A) \wedge (\mathtt{i} = 0 \vee \mathsf{not\_white}(\mathtt{i})) \end{array} \right\} \end{array} \tag{7.5}$$

Here as defined in Figure 27, reach_tomk(A, x_p, S_f, x_n) means, if a reachable BLACK object x points to a WHITE object y via the field fd, then one of the following cases holds.

(1) dirty$(x) \vee$todirty(x): x is (or is going to be) marked as dirty, as required in reach_inv;
(2) instk(x, A): x is on the stack A;
(3) $x = x_p \wedge fd \in S_f$: x is x_p, and fd is a field in S_f;
(4) $y = x_n$: y is x_n.

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

Every collection cycle in Figure 28 begins from a well-formed state with an empty mark stack in the GC's local store. Then the GC does the following in order.

(1) Concurrent Initializing (Initialize(), shown in Figure 29). We use clear_color_till(n) to mean that the GC has done color-clearing from locations 1 to n in the heap, but there might still be black objects since the mutators could allocate an black object after the GC's clearing. We could prove reach_inv holds when the GC has cleared the colors of all the objects in the heap, as shown in the following lemma.

{wfstate}
Initialize() {
  local i: [1..M], c: {BLACK, WHITE, BLUE};
  i := 1;
  Loop Invariant: {(wfstate $\wedge$ clear_color_till(i $-$ 1) $\wedge$ 1 $\leq$ i $\leq M+1$) $*$ own$_{\mathrm{np}}$(c)}
  while (i <= M) { ... } // See Figure 15 for the full code
}
{wfstate $\wedge$ reach_inv} ***// using Lemma 7.2***

Fig. 29. Proof outline of `Initialize()`.

{(wfstate $\wedge$ reach_inv) $*$ (own$_{\mathrm{np}}$(mstk) $\wedge$ mstk $= \epsilon$)}
Trace() {
  local t: [1..N], rt: Set(Int), i: [0..M];
  t := 1;
  Loop Invariant: { (wfstate $\wedge$ reach_inv) $*$ (own$_{\mathrm{np}}$(mstk) $\wedge$ mstk $= \epsilon$) $*$ (own$_{\mathrm{np}}$(t) $\wedge$ 1 $\leq$ t $\leq N+1$) $*$ own$_{\mathrm{np}}$(rt) $*$ own$_{\mathrm{p}}$(i) }
  while (t <= N) {
    rt := get_root(t);
    Foreach Invariant: {*FInv*}
    foreach i in rt do {
      {*FInv* $\wedge$ i $\in$ rt} ***// using Lemma 7.3***
      MarkAndPush(i);
      {*FInv* $\wedge$ i $\in$ rt} ***// using Lemma 7.4***
    }
    t := t + 1;
    { $\exists X$. (wfstate $\wedge$ reach_stk(X) $\wedge$ stk_black(X)) $*$ (own$_{\mathrm{np}}$(mstk) $\wedge$ mstk $= X$) $*$ (own$_{\mathrm{np}}$(t) $\wedge$ 1 $\leq$ t $\leq N+1$) $*$ own$_{\mathrm{np}}$(rt) $*$ own$_{\mathrm{p}}$(i) }
    TraceStack();
    { (wfstate $\wedge$ reach_inv) $*$ (own$_{\mathrm{np}}$(mstk) $\wedge$ mstk $= \epsilon$) $*$ (own$_{\mathrm{np}}$(t) $\wedge$ 1 $\leq$ t $\leq N+1$) $*$ own$_{\mathrm{np}}$(rt) $*$ own$_{\mathrm{p}}$(i) }
  }
}
{(wfstate $\wedge$ reach_inv) $*$ (own$_{\mathrm{np}}$(mstk) $\wedge$ mstk $= \epsilon$)}
where $FInv \triangleq \exists X$. (wfstate $\wedge$ reach_stk(X) $\wedge$ stk_black(X)) $*$ (own$_{\mathrm{np}}$(mstk) $\wedge$ mstk $= X$) $*$ (own$_{\mathrm{np}}$(t) $\wedge$ 1 $\leq$ t $\leq N$) $*$ (own$_{\mathrm{np}}$(rt) $\wedge \forall n \in$ rt. $0 \leq n \leq M$) $*$ own$_{\mathrm{p}}$(i)

Fig. 30. Proof outline of `Trace()`.

LEMMA 7.2. wfstate $\wedge$ clear_color_till(M) $\Longrightarrow$ reach_inv.

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

(2) Concurrent mark phase (`Trace()`, shown in Figure 30).
   (a) The GC first calls `MarkAndPush(i)` to mark and push every root object. We need the following two lemmas to relate the unified pre- and postconditions of `MarkAndPush(i)` in (7.5) and the actual pre- and postconditions when calling the module.
      LEMMA 7.3. reach_stk(X) $\Longrightarrow$ reach_tomk($X, 0, \emptyset, i$).
      LEMMA 7.4. reach_tomk($X, 0, \emptyset, 0$) $\Longrightarrow$ reach_stk(X).
   Then by the CONSEQ rule, we can reuse the proof of `MarkAndPush(i)`.

$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reach\_stk}(X) \wedge \mathsf{stk\_black}(X)) * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)\}$
`TraceStack() {`
`  local i: [1..M], j: [0..M];`

Loop Invariant: $\left\{\begin{array}{l}\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reach\_stk}(X) \wedge \mathsf{stk\_black}(X)) \\ * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own_p}(\mathtt{i}) * \mathsf{own_p}(\mathtt{j})\end{array}\right\}$

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

$\left\{\begin{array}{l}\exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reach\_stk}(\mathtt{i} :: X') \wedge \mathsf{stk\_black}(X') \wedge \mathsf{obj}(\mathtt{i})) \\ * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X') * \mathsf{own_p}(\mathtt{j})\end{array}\right\}$

`    j := i.pt1;`

$\left\{\begin{array}{l}\exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reach\_stk}(\mathtt{i} :: X') \wedge \mathsf{stk\_black}(X') \wedge \mathsf{obj}(\mathtt{i}) \\ \qquad \wedge \mathsf{ptfd\_sta}(\mathtt{i.pt}_1, \mathtt{j}) \wedge (\mathtt{j} = 0 \vee \mathsf{obj}(\mathtt{j}))) * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X')\end{array}\right\}$

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

// using Lemma 7.5

`    MarkAndPush(j);`

$\left\{\begin{array}{l}\exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reach\_tomk}(X', \mathtt{i}, \{\mathrm{pt}_2, \ldots, \mathrm{pt}_m\}, 0) \wedge \mathsf{stk\_black}(X') \wedge (\mathtt{j} = 0 \vee \mathsf{not\_white}(\mathtt{j})) \\ \qquad \wedge 1 \leq \mathtt{i} \leq M) * \ (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X')\end{array}\right\}$

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

$\left\{\begin{array}{l}\exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reach\_tomk}(X', \mathtt{i}, \emptyset, 0) \wedge \mathsf{stk\_black}(X') \wedge (\mathtt{j} = 0 \vee \mathsf{not\_white}(\mathtt{j}))) \\ * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X')\end{array}\right\}$

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

Fig. 31. Proof outline of `TraceStack()`.

(b) Then the GC calls the module `TraceStack()` (Figure 31) to perform the depth-first traversal. The loop invariant reach_stk holds at each time before the GC pops an object from the mark stack. Suppose the top object `i` on the mark stack points to a white object x. The GC does the following in order.

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

(ii) Read the pt_1 field of `i` to a local variable `j`. As we explained before, $\mathtt{i.pt}_1$ might not equal `j` since mutators could update this field. We only know that $\mathsf{ptfd\_sta}(\mathtt{i.pt}_1, \mathtt{j})$ holds. Then, x might be either `j`, or pointed to by `i` via fields $\mathrm{pt}_2, \ldots, \mathrm{pt}_m$. Thus we get $\mathsf{reach\_tomk}(\mathtt{mstk}, \mathtt{i}, \{\mathrm{pt}_2, \ldots, \mathrm{pt}_m\}, \mathtt{j})$ holds. Formally, the following lemma holds.

LEMMA 7.5.

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

(*B*) $\mathsf{reach\_tomk}(X, i, S_f, 0) \Longrightarrow \mathsf{reach\_tomk}(X, i, S_f, j)$;

(*C*) $\mathsf{reach\_tomk}(X, i, S_f, j) \wedge \mathsf{ptfd\_sta}(i.fd, j) \wedge fd \in S_f$
$\Longrightarrow \mathsf{reach\_tomk}(X, i, S_f \backslash \{fd\}, j)$.

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

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

$\{(\mathsf{wfstate} \wedge \mathsf{reach\_inv}) * (\mathsf{own_{np}(mstk)} \wedge \mathtt{mstk} = \epsilon)\}$
```
CleanCard() {
  local i: [1..M], c: {BLACK, WHITE, BLUE}, d: {1, 0};
  i := 1;
```
Loop Invariant: $\left\{ \begin{array}{l} \exists X.\ (\mathsf{wfstate} \wedge \mathsf{reach\_stk}(X) \wedge \mathsf{stk\_black}(X)) * (\mathsf{own_{np}(mstk)} \wedge \mathtt{mstk} = X) \\ * (\mathsf{own_p(i)} \wedge 1 \leq \mathtt{i} \leq M + 1) * \mathsf{own_{np}(c)} * \mathsf{own_{np}(d)} \end{array} \right\}$
```
  while (i <= M) { ... } // See Figure 15 for the full code
```
$\left\{ \begin{array}{l} \exists X.\ (\mathsf{wfstate} \wedge \mathsf{reach\_stk}(X) \wedge \mathsf{stk\_black}(X)) * (\mathsf{own_{np}(mstk)} \wedge \mathtt{mstk} = X) \\ * \mathsf{own_p(i)} * \mathsf{own_{np}(c)} * \mathsf{own_{np}(d)} \end{array} \right\}$
```
  TraceStack();
```
$\{(\mathsf{wfstate} \wedge \mathsf{reach\_inv}) * (\mathsf{own_{np}(mstk)} \wedge \mathtt{mstk} = \epsilon) * \mathsf{own_p(i)} * \mathsf{own_{np}(c)} * \mathsf{own_{np}(d)}\}$
```
}
```
$\{(\mathsf{wfstate} \wedge \mathsf{reach\_inv}) * (\mathsf{own_{np}(mstk)} \wedge \mathtt{mstk} = \epsilon)\}$

Fig. 32. Proof outline of `CleanCard()`.

$\{(\mathsf{wfstate} \wedge \mathsf{reach\_inv}) * (\mathsf{own_{np}(mstk)} \wedge \mathtt{mstk} = \epsilon)\}$
```
ScanRoot() {
  local t: [1..N], rt: Set(Int), i: [0..M];
  t := 1;
```
Loop Invariant:
$\left\{ \begin{array}{l} \exists X.\ (\mathsf{wfstate} \wedge \mathsf{reach\_stk}(X) \wedge \mathsf{stk\_black}(X) \wedge \mathsf{mark\_rt\_till}(\mathtt{t} - 1) \wedge 1 \leq \mathtt{t} \leq N + 1) \\ * (\mathsf{own_{np}(mstk)} \wedge \mathtt{mstk} = X) * \mathsf{own_p(i)} * \mathsf{own_{np}(rt)} \end{array} \right\}$
```
  while (t <= N) {
    rt := get_root(t);
```
Foreach Invariant:
$\left\{ \begin{array}{l} \exists X, Y.\ (\mathsf{wfstate} \wedge \mathsf{reach\_stk}(X) \wedge \mathsf{stk\_black}(X) \wedge \mathsf{mark\_rt\_till}(\mathtt{t} - 1) \wedge 1 \leq \mathtt{t} \leq N \wedge \mathsf{root}(\mathtt{t}, Y) \\ \quad \wedge \forall n \in (Y \backslash \mathtt{rt}).\ \mathsf{black}(n) \wedge \mathtt{rt} \subseteq Y) * (\mathsf{own_{np}(mstk)} \wedge \mathtt{mstk} = X) * \mathsf{own_p(i)} \end{array} \right\}$
```
    foreach i in rt do { MarkAndPush(i); }
    t := t + 1;
  }
}
```
$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reach\_stk}(X) \wedge \mathsf{stk\_black}(X) \wedge \mathsf{rt\_black}) * (\mathsf{own_{np}(mstk)} \wedge \mathtt{mstk} = X)\}$

Fig. 33. Proof outline of `ScanRoot()` in an atomic block.

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

(3) Concurrent card-cleaning (`CleanCard()`, as shown in Figure 32). We reuse the proof of `TraceStack()` via the frame rule. We can conclude reach_inv is maintained at the end of this phase.

(4) Stop-the-world card-cleaning.
   (a) The GC first rescans the roots (`ScanRoot()`, shown in Figure 33) as if they were dirty. Then reach_stk and rt_black hold. rt_black says all the root objects are black. Moreover, all the objects on the stack are black (stk_black). The atomic `MarkAndPush(i)` is proved similarly to the concurrent one (7.5) with the same pre- and postconditions.
   (b) Then the GC cleans the cards (the atomic `CleanCard()`, shown in Figure 34) without the interference from the mutators. At the end, the mark stack is empty and all the reachable objects are black (denoted by reach_black). The

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

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

Loop Invariant:

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

```
  while (i <= M) { ... } // See Figure 15 for the full code
```

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

```
  TraceStack();
```

$\{(\mathsf{wfstate} \wedge \mathsf{reach\_inv} \wedge \mathsf{rt\_black} \wedge \mathsf{clear\_dirty\_till}(M)) * (\mathsf{own_{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon) * \mathsf{own_{np}}(\mathtt{c}) * \mathsf{own_{np}}(\mathtt{d})\}$

```
}
```

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

Fig. 34. 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: $\{(\mathsf{wfstate} \wedge \mathsf{reach\_black} \wedge \mathsf{reclaim\_till}(\mathtt{i}-1) \wedge 1 \le \mathtt{i} \le M+1) * \mathsf{own_{np}}(\mathtt{c})\}$

```
  while (i <= M) { ... } // See Figure 15 for the full code
}
```

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

Fig. 35. Proof outline of `Sweep()`.

proof for the atomic `TraceStack()` is similar to the proof of the concurrent one and omitted here.

(5) Concurrent sweep phase (`Sweep()`, shown in Figure 35). No matter how the mutators interleave with the GC, all the white objects remain unreachable. Thus the reclamation is safe that guarantees $\mathcal{G}_{\mathrm{gc}}$. After sweep, the state is still well-formed.

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 article.

Verifying compilation and optimizations of concurrent programs. Compiler verification for concurrent programming languages can date back to work in Wand [1995] and Gladstein and Wand [1996], which is about functional languages using message-passing mechanisms. Recently, Lochbihler [2010] presented a verified compiler for Java threads and proved 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, that is, 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 article.

Burckhardt et al. [2010] 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 [Leroy 2009], Ševčík et al. [2011] 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 [2004] 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 [Benton 2004] and relational separation logic [Yang 2007].

Proving linearizability or atomicity of concurrent objects. Filipović et al. [2010] show linearizability can be characterized in terms of an observational refinement, where the latter is defined similarly to our Correct(**T**). There is no proof method given to verify the linearizability of fine-grained object implementations.

Turon and Wand [2011] propose a refinement-based proof method to verify concurrent objects. They first propose a simple refinement based on Brookes' fully abstract trace semantics [Brookes 1996], 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 should have 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. [2010] 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. Compositionality is not considered in their work.

In his thesis [2008], 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 metatheory (e.g., the soundness).

Verifying concurrent GCs. Vechev et al. [2006] define transformations to generate concurrent GCs from an abstract collector. Afterwards, Pavlovic et al. [2010] 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. [2011] 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. [2007], 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.

The compositionality of RGSim allows us to decompose the refinement for a large program to refinements for basic transformation units (which are usually instructions). However, for those transformation units, we have to refer to the semantics of RGSim (Definition 4.2) rather than syntactic rules to verify them, since Figure 7 provides only compositionality rules, with no rules for primitive instructions. This makes the proofs a bit tedious and complicated. Also, RGSim cannot verify the atomicity of concurrent algorithms with helping mechanism or speculations, such as the RDCSS algorithm [Vafeiadis 2008]. Finally, as we mentioned in Section 4.3, RGSim cannot ensure preservation of termination when establishing refinements. In the future, we would like to extend RGSim with a more complete set of proof rules and with the support of liveness verification. We also hope 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 article; Pierre Castéran and Sandrine Blazy for their generous help on the Coq implementation and understanding co-induction; Aaron Turon for the insightful discussions on comparing their and our work.

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theor. Comput. Sci. 82*, 2, 253–284.

Martín Abadi and Gordon Plotkin. 2009. A model of cooperative threads. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*. ACM Press, New York, 29–40.

Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. 2005. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst. 27*, 6, 1097–1146.

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*. ACM Press, New York, 14–25.

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*. ACM Press, New York, 97–108.

Hans-Juergen Boehm. 2005. Threads cannot be implemented as a library. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM Press, New York, 261–268.

Hans-Juergen Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM Press, New York, 68–78.

Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. 1991. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*. ACM Press, New York, 157–164.

Stephen D. Brookes. 1996. Full abstraction for a shared-variable parallel language. *Inf. Comput. 127*, 2, 145–163.

Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. 2010. Verifying local transformations on relaxed memory models. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software and the International Conference on Compiler Construction (CC'10/ETAPS'10)*. Springer, 104–123.

Coq Development Team. 2010. The Coq proof assistant reference manual. The Coq release v8.3. http://coq.inria.fr/V8.3/refman/

David Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing (DISC'06)*. Springer, 194–208.

Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. 2010. Simplifying linearizability proofs with reduction and abstraction. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*. Springer, 296–311.

Xinyu Feng. 2009. Local rely-guarantee reasoning. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*. ACM Press, New York, 315–327.

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *Theor. Comput. Sci. 411*, 51–52, 4379–4398.

David S. Gladstein and Mitchell Wand. 1996. Compiler correctness for concurrent languages. In *Proceedings of the 1st International Conference on Coordination Languages and Models (COORDINATION'96)*. Lecture Notes in Computer Science, vol. 1061, Springer, 231–248.

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann, San Fransisco.

Charles A. R. Hoare. 1972. Proof of correctness of data representations. *Acta Inf. 1*, 4, 271–281.

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke logical relation between ML and assembly. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM Press, New York, 133–146.

Cliff B. Jones. 1983. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst. 5*, 4, 596–619.

Kalpesh Kapoor, Kamal Lodaya, and Uday Reddy. 2011. Fine-grained concurrency with separation logic. *J. Philos. Logic 40*, 5, 583–632.

Xavier Leroy. 2009. A formally verified compiler back-end. *J. Autom. Reason. 43*, 4, 363–446.

Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM Press, New York, 455–468.

Andreas Lochbihler. 2010. Verifying a compiler for java threads. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP'10)*. Springer, 427–447.

Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. 2007. A general framework for certifying garbage collectors and their mutators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM Press, New York, 468–479.

Matthew Parkinson, Richard Bornat, and Cristiano Calcagno. 2006. Variables as resource in hoare logics. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*. IEEE Computer Society, 137–146.

Dusko Pavlovic, Peter Pepper, and Douglas R. Smith. 2010. Formal derivation of concurrent garbage collectors. In *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC'10)*. 353–376.

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*. IEEE Computer Society, 55–74.

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-memory concurrency and verified compilation. In *Proceedings of the 38th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM Press, New York, 43–54.

R. Kent Treiber. 1986. System programming: Coping with parallelism. Tech. rep. RJ 5118, IBM Almaden Research Center.

Aaron Turon and Mitchell Wand. 2011. A separation logic for refining concurrent objects. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM Press, New York, 247–258.

Viktor Vafeiadis. 2008. Modular fine-grained concurrency verification. Tech. rep. UCAM-CL-TR-726, University of Cambridge, Computer Laboratory.

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A marriage of rely/guarantee and separation logic. In *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR'07)*. Springer, 256–271.

Martin T. Vechev, Eran Yahav, and David F. Bacon. 2006. Correctness-preserving derivation of concurrent garbage collection algorithms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. ACM Press, New York, 341–353.

Mitchell Wand. 1995. Compiler correctness for parallel languages. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*. ACM Press, New York, 120–134.

Hongseok Yang. 2007. Relational separation logic. *Theor. Comput. Sci. 375*, 1–3, 308–334.

Received January 2013; revised September 2013; accepted November 2013