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

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:

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. WK0110000051).

Author's addresses: H. Liang, X. Feng and M. Fu, School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230026, China; Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou, Jiangsu 215123, China.

Emails: H. Liang, lhj1018@mail.ustc.edu.cn; X. Feng (corresponding author), xyfeng@ustc.edu.cn; and M. Fu, fuming@ustc.edu.cn.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© YYYY ACM 0164-0925/XXXX/01-ARTA $15.00

DOI: http://dx.doi.org/10.1145/0000000.0000000

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
1. INTRODUCTION

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

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

— **Atomicity of concurrent objects.** A concurrent object or library provides a set of methods that allow clients to manipulate the shared data structure with abstract atomic behaviors [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 $\text{atomic}\{C\}$, so that programmers can assume the atomicity of the execution of $C$. 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 $T$, 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:

$$\text{Correct}(T) \triangleq \forall C, C = T(C) \Rightarrow C \sqsubseteq C.$$  \hspace{1cm} (1.1)

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

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

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

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

However, existing refinement (or equivalence) relations cannot satisfy all these requirements at the same time. Contextual equivalence, the canonical notion for comparing program behaviors, fails to handle different languages since the contexts of the source and the target will be different. Simulations and logical relations have been used to verify compilation [Leroy 2009; Benton and Hur 2009; Lochbihler 2010; Hur and Dreyer 2011], but they are usually designed for sequential programs (except [Lochbihler 2010; 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 paper, we propose a Rely-Guarantee-based Simulation (RGSim) for compositional verification of concurrent transformations. By addressing the above problems, we make the following contributions:

— RGSim parametrizes the simulation between concurrent programs with rely/guarantee conditions [Jones 1983], which specify the interactions between the programs and their environments. This makes the corresponding refinement relation compositional w.r.t. parallel compositions, allowing us to decompose refinement proofs for multi-threaded programs into proofs for individual threads. On the other hand, the rely/guarantee conditions can incorporate the assumptions about environments made by specific program transformations, so RGSim can be applied to verify many practical transformations.

— Based on the simulation technique, RGSim focuses on comparing externally observable behaviors (e.g., I/O events) only, which gives us considerable leeway in the implementations of related programs. The relation is mostly independent of the language details. It can be used to relate programs in different languages with different views of program states and different granularities of atomic state accesses.

— RGSim makes relational reasoning about optimizations possible in parallel contexts. We present a set of relational reasoning rules to characterize and justify common optimizations in a concurrent setting, including hoisting loop invariants, strength reduction and induction variable elimination, dead code elimination, redundancy introduction, etc.

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

— We reduce the problem of verifying concurrent garbage collectors to verifying transformations, and present a general GC verification framework, which combines unary Rely-Guarantee-based verification [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 online1.

This paper extends the conference paper in POPL 2012 [Liang et al. 2012]. First, we add more examples, including strength reduction and induction variable elimination, the non-blocking 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 run-time systems.

In the rest of this paper, we first analyze the challenges for compositional verification of concurrent program transformations, and explain our approach informally in

1http://kyhcs.ustcez.edu.cn/relconcur/rgsim
A:4 Hongjin Liang et al.

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

Section 2. Then we give the basic technical settings in Section 3 and present the formal
definition of RGSim in Section 4. We show the use of RGSim to reason about optimizations
in Section 5, verify 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, i.e., we should be able to know \( T(C_1) \parallel T(C_2) \sqsubseteq C_1 \parallel C_2 \) if we
have \( T(C_1) \sqsubseteq C_1 \) and \( T(C_2) \sqsubseteq C_2 \).

2.1. Sequential Refinement Loses Parallel Compositionality

Observable behaviors of sequential imperative programs usually refer to their con-
trol effects (e.g., termination and exceptions) and final program states. However, re-
finement relations defined correspondingly cannot be preserved after parallel compo-
sitions. It has been a well-known fact in the compiler community that sound opti-
mizations for sequential programs may change the behaviors of multi-threaded pro-
grams [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. Peo-
ple have given fully abstract semantics to concurrent programs (e.g., [Brookes 1996;
Abadi and Plotkin 2009]). The semantics of a program is modeled as a set of execu-

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
tion 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, e.g., a stack object can only be accessed through push and pop methods, and its internal data cannot be arbitrarily updated.

— Usually the implementation of STM (e.g., TL2 [Dice et al. 2006]) ensures the atomicity of a transaction atomic{C} only when there are no data races. Therefore, the correctness of the transformation from high-level atomic blocks to fine-grained concurrent code assumes data-race-freedom in the source.

— Many garbage-collected languages are type-safe and prohibit operations such as pointer 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 paper we propose a Rely-Guarantee-based Simulation (RGSim) ≤ 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, i.e., the guarantee of each thread is permitted in the rely of every other. Then we can verify their parallel composition by separately verifying each thread, showing its behaviors under the rely condition indeed satisfy its guarantee. After parallel composition, the threads should be executed under their common environment (i.e., the intersection of their relies) and guarantee all the possible transitions made by them (i.e., the union of their guarantees).

Parametrized with rely/guarantee conditions for the two levels, our relation $(C, R, G) ≤ (C, R, G)$ talks about not only the target $C$ and the source $C$, but also the interference $R$ and $G$ between $C$ and its target-level environment, and $R$ and $G$ between $C$ and its environment at the source level. Informally, $(C, R, G) ≤ (C, R, G)$ says the executions of $C$ under the environment $R$ do not exhibit more observable behaviors than the executions of $C$ under the environment $R$, and the state transitions of $C$ and $C$ satisfy $G$ and $G$ respectively. RGSim is now compositional, as long as the threads are composed with well-behaved environments only. The parallel compositionality lemma is in the following form. If we know $(C_1, R_1, G_1) ≤ (C_1, R_1, G_1)$ and $(C_2, R_2, G_2) ≤ (C_2, R_2, G_2)$, and also the interference constraints are satisfied, i.e., $G_2 ⊆ R_1, G_1 ⊆ R_2$, we could get $(C_1 ∥ C_2, R_1 ∩ R_2, G_1 ∪ G_2) ≤ (C_1 ∥ C_2, R_1 ∩ R_2, G_1 ∪ G_2)$.

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

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

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

<table>
<thead>
<tr>
<th>Target Code ($C_1$)</th>
<th>Source Code ($C$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>local $t$; $t := x + 1$; while($i &lt; n$) { $i := i + t$; }</td>
<td>local $t$; while($i &lt; n$) { $t := x + 1$; $i := i + t$; }</td>
</tr>
</tbody>
</table>

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
Rely-Guarantee-Based Simulation

(a) Events and Transition Labels

\[ (\text{Events}) \quad e ::=: \ldots \quad (\text{Labels}) \quad o ::=: e \mid \tau \]

\[ (\text{LState}) \quad \sigma ::=: \ldots \]
\[ (\text{LExpr}) \quad E \in LState \rightarrow \text{Int}_\perp \]
\[ (\text{LBExp}) \quad B \in LState \rightarrow \{\text{true, false}\}_\perp \]
\[ (\text{LInstr}) \quad c \in LState \rightarrow \mathcal{P}((\text{Labels} \times LState) \cup \{\text{abort}\}) \]
\[ (\text{LStmt}) \quad C ::=: \text{skip} \mid c \mid C_1; C_2 \mid \text{if } (B) \ C_1 \ \text{else} \ C_2 \mid \text{while } (B) \ C \mid C_1 \parallel C_2 \]
\[ (\text{LStep}) \quad \rightarrow_L \in \mathcal{P}((\text{LStmt} \setminus \{\text{skip}\} \times LState) \times \text{Labels} \times ((\text{LStmt} \times LState) \cup \{\text{abort}\})) \]

(b) The Low-Level Language

\[ (\text{HState}) \quad \Sigma ::=: \ldots \]
\[ (\text{HEExpr}) \quad E \in HState \rightarrow \text{Int}_\perp \]
\[ (\text{HBExp}) \quad B \in HState \rightarrow \{\text{true, false}\}_\perp \]
\[ (\text{HInstr}) \quad c \in HState \rightarrow \mathcal{P}((\text{Labels} \times HState) \cup \{\text{abort}\}) \]
\[ (\text{HStmt}) \quad C ::=: \text{skip} \mid c \mid C_1; C_2 \mid \text{if } B \ \text{then} \ C_1 \ \text{else} \ C_2 \mid \text{while } B \ \text{do} \ C \mid C_1 \parallel C_2 \]
\[ (\text{HStep}) \quad \rightarrow_H \in \mathcal{P}((\text{HStmt} \setminus \{\text{skip}\} \times HState) \times \text{Labels} \times ((\text{HStmt} \times HState) \cup \{\text{abort}\})) \]

(c) The High-Level Language

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 $R$ at both levels do not update $x$ nor $t$. It requires the programmers to mark $x$ as a volatile variable in practice. The guarantees $G$ of both $C_1$ and $C$ can be specified as arbitrary transitions. Then we can prove the RGSim relation $(C_1, R, G) \preceq (C, R, G)$ 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 can be instantiated by program verifiers, depending on their interest (e.g., input/output events). A label that will be associated with a state transition is either an event or $\tau$, which means the corresponding transition does not generate any event (i.e., a silent step).
The target language, which we also call the low-level language, is shown in Figure 2(b). We abstract away the forms of states, expressions and primitive instructions in the language. An arithmetic expression \( E \) is modeled as a function from states to integers lifted with an undefined value \( \perp \). Boolean expressions \( B \) 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 \( P(\_\_) \) to denote the power set. Unsafe executions lead to \( \text{abort} \). Note that the semantics of an instruction could be non-deterministic. Moreover, it might be undefined on some states, making it possible to model blocking operations such as acquiring a lock.

Statements are either primitive instructions or compositions of them. \( \text{skip} \) 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 \( _\_ \rightarrow \rightarrow_{L..} \), 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 \( \text{skip} \). The step aborts if an unsafe instruction is executed.

The high-level language (source language) is defined similarly in Figure 2(c), but it is important to note that its states and primitive instructions may be different from those in the low-level language. The compound statements are almost the same as their low-level counterparts. \( C_1 :: C_2 \) and \( C_1 || C_2 \) are sequential and parallel compositions of \( C_1 \) and \( C_2 \) respectively. Note that we choose to use the same set of compound statements in the two languages for simplicity only. This is not required by our simulation relation, although the analogous program constructs of the two languages (e.g., parallel compositions \( C_1 || C_2 \) and \( C_1 || C_2 \) ) make it convenient for us to discuss the compositionality later.

\[
\begin{align*}
(a, \Sigma') &\in c \Sigma \\
(c, \Sigma) &\rightarrow (\text{skip}, \Sigma') \\
\text{abort} &\in c \Sigma \\
\Sigma &\notin \text{dom}(c) \\
(c, \Sigma) &\rightarrow \text{abort} \\
(C_1, \Sigma) &\rightarrow (C_1', \Sigma') \\
C_1 || C_2, \Sigma &\rightarrow (C_1 || C_2, \Sigma') \\
(C_1', \Sigma') &\rightarrow (C_1 || C_2, \Sigma') \\
(C_1 || C_2, \Sigma) &\rightarrow \text{abort} \\
(C_1 || C_2, \Sigma) &\rightarrow \text{abort}
\end{align*}
\]

Fig. 3. Selected Operational Semantics Rules of the High-Level Language

Figure 3 shows part of the definition of \( _\_ \rightarrow \rightarrow_{H..} \), which gives the high-level operational semantics of statements. We often omit the subscript \( H \) (or \( L \)) in \( _\_ \rightarrow \rightarrow_{H..} \) (or \( _\_ \rightarrow \rightarrow_{L..} \)) and the label on top of the arrow when it is \( \tau \). 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 \( \Sigma \) (i.e., \( \Sigma \notin \text{dom}(c) \)), we let the program configuration reduce to itself. For example, the instruction \( \text{lock}(1) \) would be blocked when \( 1 \) is not 0, making it be repeated until \( 1 \) becomes 0; whereas \( \text{unlock}(1) \) simply sets \( 1 \) to 0 at any time and would never be blocked. Primitive instructions in the high-level and low-level languages are \( \text{atomic} \) in the interleaving semantics. Below we use \( _\_ \rightarrow \rightarrow_{H..} \) for zero or multiple-step transitions with no events generated, and \( _\_ \rightarrow \rightarrow_{H..}^* \) for multiple-step transitions with only one event \( c \) 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 $\text{done}$ or a fault marker $\text{abort}$.

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

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

1. $\text{ETrSet}_0(C, \sigma) \overset{\Delta}{=} \{\epsilon\}$,
2. $\text{ETrSet}_{n+1}(C, \sigma) \overset{\Delta}{=} \{E \mid (C, \sigma) \xrightarrow{\cdot} (C', \sigma') \land E \in \text{ETrSet}_n(C', \sigma') \}
\lor (C, \sigma) \xrightarrow{e} (C', \sigma') \land E = e::E' 
\lor C = \text{skip} \land E = \text{done} \}.

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

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

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

**Definition 3.2 (Event Trace Refinement).** We say $(C, \sigma)$ is an e-trace refinement of $(\Sigma, \Sigma)$, i.e., $(C, \sigma) \sqsubseteq (\Sigma, \Sigma)$, if and only if

\[
\text{ETrSet}(C, \sigma) \subseteq \text{ETrSet}(C, \Sigma).
\]

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 $T$ should translate states as well as code. We overload the notation and use $T(\Sigma)$ to represent the state transformation, and use $C \sqsubseteq_T C$ for

\[
\forall \sigma, \Sigma. \quad \sigma = T(\Sigma) \implies (C, \sigma) \subseteq (\Sigma, \Sigma),
\]

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

\[
\text{Correct}(T) \quad \overset{\Delta}{=} \quad \forall C, \Sigma. \quad C = T(C) \implies C \sqsubseteq_T C.
\]

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

\[
(C, \sigma) \approx (\Sigma, \Sigma) \quad \overset{\Delta}{=} \quad (C, \sigma) \sqsubseteq (\Sigma, \Sigma) \land (G, \Sigma) \sqsubseteq (C, \sigma),
\]

and use $C \approx_T C$ for $\forall \sigma, \Sigma. \quad \sigma = T(\Sigma) \implies (C, \sigma) \approx (\Sigma, \Sigma)$.

4. THE RGSIM RELATION

The e-trace refinement is defined directly over the externally observable behaviors of programs. It is intuitive, and also abstract in that it is independent of language details. However, as we explained before, it is not compositional w.r.t. parallel compositions. In this section we propose RGSim, which can be viewed as a compositional proof technique that allows us to derive the simple e-trace refinement and then verify the corresponding transformation $T$. 

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
4.1. The Definition

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

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

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

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

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

**Definition 4.1 (\(\alpha\)-Related Transitions).**

$$\langle R, \alpha \rangle \triangleq \{ (\sigma, \sigma') | (\sigma, \sigma') \in R \wedge (\Sigma, \Sigma') \in R \wedge (\sigma, \Sigma) \in \alpha \wedge (\sigma', \Sigma') \in \alpha \}. $$

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

**Definition 4.2 (RGSim).** Whenever $(C, \sigma, R, G) \preceq_{\alpha, \gamma} (C, \Sigma, R, G)$, then $(\sigma, \Sigma) \in \alpha$ and the following are true:

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

Then, $(C, R, G) \preceq_{\alpha, \gamma} (C, \Sigma, R, G)$ iff for all $\sigma$ and $\Sigma$, if $(\sigma, \Sigma) \in \zeta$, then $(C, \sigma, R, G) \preceq_{\alpha, \gamma} (C, \Sigma, R, G)$. Here the precondition $\zeta \in \mathcal{P}(LState \times HState)$ is used to relate the initial states $\sigma$ and $\Sigma$.

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

\begin{align*}
\alpha &\subseteq \mathcal{P}(LState \times HState) \\
\gamma &\subseteq \mathcal{P}(HState) \\
\zeta &\subseteq \mathcal{P}(LState)
\end{align*}
Rely-Guarantee-Based Simulation

\[
(C, \sigma) \xrightarrow{\alpha \prec} (C, \Sigma) \quad \text{and} \quad (C, \sigma) \xrightarrow{\alpha \prec} (C, \Sigma)
\]

\[
(C', \sigma') \xrightarrow{\alpha \prec} (C', \Sigma') \quad \text{and} \quad (C', \sigma') \xrightarrow{\alpha \prec} (C', \Sigma')
\]

(a) Program Steps  \hspace{1cm} (b) Environment Steps

Fig. 5. Simulation Diagrams of RGSim

---

Starting from \(\alpha\)-related states, each step of \(C\) corresponds to zero or multiple steps of \(\mathcal{C}\), and the resulting states are \(\alpha\)-related too. If an external event is produced in the step of \(C\), the same event should be produced by \(\mathcal{C}\). We show the simulation diagram with events generated by the program steps in Figure 5(a), where solid lines denote hypotheses and dashed lines denote conclusions, following Leroy’s notations [Leroy 2009].

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

The corresponding transitions of \(C\) and \(\mathcal{C}\) need to be in \(\langle \mathcal{G}, \mathcal{G}^* \rangle\). 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 \(\mathcal{C}\) should be in the transitive closure of \(\mathcal{G}\). The guarantees are abstractions of the programs’ behaviors. As we will show later in the PAR rule in Figure 7, they will serve as the rely conditions of the sibling threads at the time of parallel compositions. Note that we do not need each step of \(\mathcal{C}\) to be in \(\mathcal{G}\), although we could do so. This is because we only care about the coarse-grained behaviors (with mumbling) of the source that are used to simulate the target. We will explain more by the example (4.1) in Section 4.2.

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

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

Whatever the low-level environment \(R\) and the high-level one \(\mathcal{R}\) do, as long as the state transitions are \(\alpha\)-related, they should not affect the simulation between \(C\) and \(\mathcal{C}\), as shown in Figure 5(b). Here a step in \(\mathcal{R}\) may correspond to zero or multiple steps of \(R\). Note that different from the program steps, some steps of \(\mathcal{R}\) may not correspond to steps of \(R\). On the other hand, only requiring that \(\mathcal{R}\) be simulated by \(R\) (see (4.2) in Section 4.2) is not sufficient for parallel compositionality, which we will explain later in Section 4.2.

Then based on the simulation, we hide the states by the precondition \(\zeta\) and define the RGSim relation between programs only. By the definition we know \(\zeta \subseteq \alpha\) if \((C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha; \zeta \times \gamma} (C, \mathcal{R}, \mathcal{G})\), i.e., the precondition needs to be no weaker than the step invariant. Usually in practice \(\alpha\) is very weak and naturally implied by the pre- and post-conditions \(\zeta\) and \(\gamma\), e.g., \(\zeta\) and \(\gamma\) are the same as \(\alpha\) in examples in Section 6.

RGSim is sound w.r.t. the e-trace refinement (Definition 3.2). That is, \((C, \sigma, R, G) \preceq_{\alpha; \gamma}(C, \Sigma, R, G)\) ensures that \((C, \sigma)\) does not have more observable behaviors than \((C, \Sigma)\).
If there exist $x$ and the unary stable assertion $\text{Sta}(x)$ holds iff for all $\alpha, \beta, \gamma$ and $\Sigma$, if $(\alpha, \beta, \gamma, (\Sigma, \Sigma')) \in \lambda$, then $(\alpha, \beta, \gamma, (\Sigma, \Sigma')) \in \lambda$.

Usually we need $\text{Sta}(\alpha, (\Sigma, \Sigma'))$, which says whenever $\zeta$ holds initially and $\beta$ and $\gamma$ perform related actions, the resulting states still satisfy $\zeta$. By unfolding $(\beta, \gamma, \zeta)$, we could see that $\alpha$ itself is stable $w.r.t.$ any $\alpha$-related transitions, i.e., $\text{Sta}(\alpha, (\beta, \gamma, \zeta))$.

Another simple example is given below, where both environments could increment $x$ and the unary stable assertion $x \geq 0$ is lifted to the relation $\zeta$:

$$
\zeta \triangleq \{(\sigma, \Sigma) \mid \sigma(x) = \Sigma(x) \land x \geq 0\}
$$

$$
\text{Sta}(\zeta) \triangleq \{(\sigma, \Sigma) \mid \sigma(x) = \Sigma(x) \}\}
$$

We can prove $\text{Sta}(\zeta, (\beta, \gamma, \zeta))$. Stability of the pre- and post-conditions under the environments’ interference is assumed as an implicit side condition at every proof rule.
in Figure 7, e.g., we assume $\text{Sta}(\zeta, (R, R^+) \alpha)$ 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, $\gamma$ serves as the postcondition of $C_1$ and $C_1$ and the precondition of $C_2$ and $C_2$ at the same time. The IF rule requires the boolean conditions of both sides to be evaluated to the same value under the precondition $\zeta$. The definitions of the sets $B \equiv B$ and $B \land \bar{B}$ are given in Figure 6. The rule also requires the precondition $\zeta$ to imply the step invariant $\alpha$. In the WHILE rule, the $\gamma$ relation is viewed as a loop invariant preserved at the loop entry point, and needs to ensure $B \equiv B$. 
**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 $C_1 || C_2$ simulate $C_1 || C_2$, we only need a subset of the interleavings of $C_1$ and $C_2$ to simulate those of $C_1$ and $C_2$. Thus the high-level relies and guarantees need to ensure the existence of those interleavings only. Below we give a simple example to explain this subtle issue. We can prove

$$\alpha \triangleq (x:=x+1; x:=x+1, l(R, G)),$$

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

- $R = G \triangleq \{ (\sigma, \sigma') \mid \sigma = \sigma' = \sigma\{x := \sigma(x) + 2\} \};$
- $R = G \triangleq \{ (\Sigma, \Sigma') \mid \Sigma = \Sigma' = \Sigma\{x := \Sigma(x) + 2\} \};$
- $\alpha = \zeta = \gamma \triangleq \{ (\sigma, \Sigma) \mid \sigma(x) = \Sigma(x) \}.$

Note that the high-level program is actually finer-grained than its guarantee, but to prove (4.1) we only need the execution in which it goes two steps to the end without interference from its environment. Also we can prove $(\text{print}(x), R, G) \preceq_{\alpha, \zeta, \gamma} (\text{print}(x), R, G)$, Here we use the instruction $\text{print}(E)$ to observe the value of $x$, which will produce an external event $\text{out}(n)$ if $E$ evaluates to $n$. Then by the PAR rule, we get

$$\alpha \triangleq (x:=x+1; x:=x+1) \parallel \text{print}(x), R, G),$$

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 $R$ be simulated by $R$:

$$\text{If } (\sigma, \sigma') \in R, \text{ then there exists } \Sigma' \text{ such that } (\Sigma, \Sigma') \in R^* \text{ and } (C, \sigma', R, G) \preceq_{\alpha, \zeta, \gamma} (C, \Sigma', R, G).$$

We refer to this modified simulation definition as $\preceq'$. Unfortunately, $\preceq'$ does not have parallel compositionality. As a counterexample, if the invariant $\alpha$ says the left-side $x$ is not greater than the right-side $x$, i.e.,

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

we could prove the following:

$$\alpha \triangleq (x:=x+1, \text{Id}, \text{True}) \preceq'(\alpha, \alpha, \alpha) (x:=x+2, \text{Id}, \text{True});$$

$$\alpha \triangleq (x:=0; \text{print}(x), \text{True}, \text{Id}) \preceq'(\alpha, \alpha, \alpha) (x:=0; \text{print}(x), \text{True}, \text{Id}).$$

Here we use $\text{Id}$ and $\text{True}$ (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:

$$(x:=x+1 \parallel (x:=0; \text{print}(x)), \text{Id}, \text{True}) \preceq'(\alpha, \alpha, \alpha) (x:=x+2 \parallel (x:=0; \text{print}(x)), \text{Id}, \text{True}).$$
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 (4.4) is not provable, because after some $\alpha$-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 \textsc{Stren-$\alpha$} rule allows us to replace the invariant $\alpha$ by a stronger invariant $\alpha'$. We need to check that $\alpha'$ is indeed an invariant preserved by the related program steps, i.e., $\text{Sta}(\alpha', (\mathcal{G}, \mathcal{G'}))$ holds. Symmetrically, the \textsc{Weaken-$\alpha$} rule requires $\alpha$ to be preserved by environment steps related by the weaker invariant $\alpha'$. As usual, the pre- and post-conditions, the relies and the guarantees can be strengthened or weakened by the \textsc{Conseq} rule.

The \textsc{Frame} rule allows us to use local specifications [Reynolds 2002]. When verifying the simulation between $C$ and $\textbf{C}$, we need to only talk about the locally-used resource in $\alpha$, $\zeta$ and $\gamma$, 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 $\eta$ is used, and the accesses of it respect the invariant $\beta$ and $\mathbb{R}_1$, $\mathcal{G}_1$, $\mathbb{R}_1$ and $\mathcal{G}_1$. We give the auxiliary definitions in Figure 6. The disjoint union $\triangledown$ between states is lifted to state pairs. A state relation $\alpha$ is intuitionistic, denoted by $\text{Intui}(\alpha)$, if it is monotone w.r.t. the extension of states. The disjointness $\eta \not\preceq \alpha$ says that any state pair satisfying both $\eta$ and $\alpha$ can be split into two disjoint state pairs satisfying $\eta$ and $\alpha$ respectively. For example, let $\eta \define \{(\sigma, \Sigma) \mid \sigma(y) = \Sigma(y)\}$ and $\alpha \define \{(\sigma, \Sigma) \mid \sigma(x) = \Sigma(x)\}$ where $x$ and $y$ are two distinct variables, then both $\eta$ and $\alpha$ are intuitionistic and $\eta \not\preceq \alpha$ holds. We also require $\eta$ to be stable under interference from the programs (i.e., the programs do not change the extra resource) and the extra environments. We use $\eta \not\preceq \{\zeta, \gamma, \alpha\}$ as a shorthand for $(\eta \not\preceq \zeta) \land (\eta \not\preceq \gamma) \land (\eta \not\preceq \alpha)$. Similar representations are used in this rule.

Finally, the transitivity rule \textsc{Trans} allows us to verify a transformation by using an intermediate level as a bridge. The intermediate environment $\mathbb{R}_3$ should be chosen with caution so that the $(\beta \circ \alpha)$-related transitions can be decomposed into $\beta$-related and $\alpha$-related transitions, as illustrated in Figure 4(b). Here $\circ$ defines the composition of two relations and $\text{isMidOf}$ defines the side condition over the environments, as shown in Figure 6. We use $\theta$ for a middle-level state.

**Soundness.** All the rules in Figure 7 are sound, i.e., for each rule the premises imply the conclusion. We prove their soundness by co-induction, directly following the definition of RGSim. The proofs 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 (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.

\[
(C, \text{Id, True}) \preceq_{\alpha; \zeta; \gamma} (C, \text{Id, True}) \tag{4.5}
\]

The refinement (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.

\[
(C, \text{True, True}) \preceq_{\alpha; \zeta; \gamma} (C, \text{True, True}) \tag{4.6}
\]
4.3. A Simple Example

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

\[
\begin{align*}
\text{print}(x); & \parallel x := x + 2; \\
\downarrow & \text{lock}(1); \text{lock}(1); \\
\text{print}(x); & \parallel x := x + 1; x := x + 1; \\
\text{unlock}(1); & \langle \text{unlock}(1); x := x; \rangle \\
\end{align*}
\]

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 read-only and would not affect the external behaviors of the program [Abadi and Lamport 1991]. Thus below 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 $\alpha$, which only cares about the value of $x$ when the lock is free.

\[
\alpha \triangleq \{ (\sigma, \Sigma) \mid \sigma(x) = \Sigma(x) \land (\sigma(1) = 0 \implies \sigma(x) = \sigma(x)) \}.
\]

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

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

\[
R \triangleq \{ (\sigma, \sigma') \mid (\sigma(1) = \text{cid} \implies \sigma(x) = \sigma'(x) \land (\sigma(1) = \sigma'(1)) \};
\]

\[
G \triangleq \{ (\sigma, \sigma') \mid (\sigma'(1) = 0 \land \sigma' = \sigma[1 \sim \text{cid}] \\
\land (\sigma(1) = \text{cid} \land \sigma' = \sigma[1 \sim \text{cid}] \land (\sigma(1) = 0 \land \sigma(x) = x)) \}.
\]

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

Following the definition, we can prove $(C_1, R, G) \preceq_{\alpha, \alpha, \alpha} (C_1, R, G)$ and $(C_2, R, G) \preceq_{\alpha, \alpha, \alpha} (C_2, R, G)$. By applying the PAR rule and from the soundness of RGSim (Corollary 4.4), we know $C_1 \parallel C_2 \subseteq_T C_1 \parallel C_2$ holds for any $T$ that respects $\alpha$.

Perhaps interestingly, if we omit the lock and unlock operations in $C_1$, then $C_1 \parallel C_2$ would have more externally observable behaviors than $C_1 \parallel C_2$. This does not indicate the unsoundness of our PAR rule (which is sound!). The reason is that $x$ might have different values on the two levels after the environments’ $\alpha$-related transitions, so that we cannot have $(\text{print}(x), R, G) \preceq_{\alpha, \alpha, \alpha} (\text{print}(x), R, G)$ with the current definitions of $\alpha$, $R$ and $G$, 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 \( \alpha \) and do the proof. It is difficult to prove the refinement without this auxiliary variable. One may wish to prove

\[
(C_1, R', G') \preceq_{\alpha', \alpha'^R} (C_1, R, G),
\]

where \( \alpha', R' \) and \( G' \) are defined as follows by eliminating \( x \) from \( \alpha, R \) and \( G \):

\[
\begin{align*}
\alpha' & \triangleq \{(\sigma, \Sigma) \mid \sigma(1) = 0 \implies \sigma(x) = \Sigma(x)\}; \\
R' & \triangleq \{(\sigma, \sigma') \mid \sigma(1) = \text{cid} \implies \sigma(x) = \sigma'(x) \land \sigma(1) = \sigma'(1)\}; \\
G' & \triangleq \{(\sigma, \sigma') \mid \sigma' = \sigma \lor \sigma(1) = 0 \land \sigma' = \sigma [1 \sim \text{cid}] \\
& \lor \sigma(1) = \text{cid} \land \sigma' = \sigma [1 \sim 0]\}.
\end{align*}
\]

But (4.7) does not hold because \( (R', R^*)_{\alpha'} \) (which is used in Definition 4.2(5)) permits unexpected transitions. For instance, we allow \( ((\sigma, \sigma'), (\Sigma, \Sigma')) \in (R', R^*)_{\alpha'} \) for the following \( \sigma, \sigma', \Sigma \) and \( \Sigma' \):

\[
\sigma = \sigma' \triangleq \{x \sim 0, 1 \sim \text{cid}\}; \quad \Sigma \triangleq \{x \sim 0\}; \quad \Sigma' \triangleq \{x \sim 1\}.
\]

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 \( \alpha \) to relate them, we can directly define "relational rely/guarantee" relations \( r, g \in P((LState \times LState) \times (HState \times HState)) \). The new simulation is in the form of \( C \preceq_{\alpha, x, y, z, r, g} C \) and defined by substituting \( r \) and \( g \) for \( (R, R^*)_{\alpha} \) and \( (G, G^*)_{\alpha} \) 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 \( C_1 \preceq_{\alpha', \alpha'^R} C_1 \) and \( C_2 \preceq_{\alpha', \alpha'^R} C_2 \). Here \( C_2' \) results from removing \( x \) from \( C_2 \), \( \alpha' \) is defined as above and \( r \) and \( g \) are as follows:

\[
\begin{align*}
r & \triangleq \{\{(\sigma, \sigma'), (\Sigma, \Sigma')\} \mid \sigma(1) = \text{cid} \implies \sigma(x) = \sigma'(x) \land \sigma(1) = \sigma'(1) \land \Sigma(x) = \Sigma'(x)\}; \\
g & \triangleq \{\{(\sigma, \sigma'), (\Sigma, \Sigma')\} \mid \sigma' = \sigma \land \Sigma' = \Sigma \lor \sigma(1) = 0 \land \sigma' = \sigma [1 \sim \text{cid}] \land \Sigma' = \Sigma \\
& \lor \sigma(1) = \text{cid} \land \sigma' = \sigma [1 \sim 0] \land \Sigma' = \Sigma \{x \sim \sigma(x)\}\}.
\end{align*}
\]

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 \( (R', R^*)_{\alpha'} \), 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 above example, this allows the low-level programs to be blocked forever (e.g., at the time when the lock is held but never released by some other thread). Proving the preservation of the termination behavior would require liveness proofs in a concurrent setting (e.g., proving the absence of deadlock), which we leave as future work.

In the next three sections, we show more serious examples to demonstrate the applicability of RGSim.
5. RELATIONAL REASONING ABOUT OPTIMIZATIONS

As a general correctness notion of concurrent program transformations, RGSim establishes a relational approach to justify compiler optimizations on concurrent programs. Below we adapt Benton’s work [Benton 2004] on sequential optimizations to the concurrent setting.

5.1. Optimization Rules

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

**Sequential Unit Laws**

\[
\begin{align*}
(C_1, R_1, G_1) \preceq_{\alpha, \zeta \wedge \gamma} (C_2, R_2, G_2) & \quad (\text{skip}, C_1, R_1, G_1) \preceq_{\alpha, \zeta \wedge \gamma} (C_2, R_2, G_2) \\
(C_1, R_1, G_1) \preceq_{\alpha, \zeta \wedge \gamma} (C_2, R_2, G_2) & \quad (C_1, R_1, G_1) \preceq_{\alpha, \zeta \wedge \gamma} (\text{skip}, C_2, R_2, G_2)
\end{align*}
\]

Plus the variants with \( \text{skip} \) after the code \( C_1 \) or \( C_2 \). That is, \( \text{skips} \) could be arbitrarily introduced and eliminated.

**Common Branch**

\[
\forall \sigma_1, \sigma_2, (\sigma_1, \sigma_2) \in \zeta \implies B \sigma_2 \neq \bot
\]

\[
(C, R, G) \preceq_{\alpha, \zeta \wedge \gamma} (C_1, R', G') \quad \zeta_1 = (\zeta \cap (\text{true} \wedge B))
\]

\[
(C, R, G) \preceq_{\alpha, \zeta \wedge \gamma} (C_2, R', G') \quad \zeta_2 = (\zeta \cap (\text{true} \wedge \neg B))
\]

\[
(C, R, G) \preceq_{\alpha, \zeta \wedge \gamma} (\text{if} \ B \ C_1; \text{else} \ C_2, R', G')
\]

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**

\[
(C, R, G) \preceq_{\alpha, \zeta \wedge \gamma} (C_1, R', G') \quad \zeta = (\zeta \cap (\text{true} \wedge B))
\]

\[
(C, R, G) \preceq_{\alpha, \zeta \wedge \gamma} (\text{if} \ B \ C_1; \text{else} \ C_2, R', G')
\]

\[
(C, R, G) \preceq_{\alpha, \zeta \wedge \gamma} (C_2, R', G') \quad \zeta = (\zeta \cap (\text{true} \wedge \neg B))
\]

\[
(C, R, G) \preceq_{\alpha, \zeta \wedge \gamma} (\text{if} \ B \ C_1; \text{else} \ C_2, R', G')
\]

Since the if-condition \( B \) is \text{true} (or \text{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**

\[
\zeta = (\zeta \cap (\text{true} \wedge \neg B)) \quad \zeta \subseteq \alpha \quad \text{Sta}(\zeta, (R_1, R_2')_\alpha)
\]

\[
(\text{skip}, R_1, \text{id}) \preceq_{\alpha, \zeta \wedge \zeta} (\text{while} \ (B) \{C\}, R_2, \text{id})
\]

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

**Loop Peeling**

\[
(\text{while} \ (B) \{C\}, R_1, G_1) \preceq_{\alpha, \zeta \wedge \gamma} (\text{while} \ (B) \{C\}, R_2, G_2)
\]

\[
(\text{if} \ (B) \{C; \text{while} \ (B) \{C\} \} \text{else} \text{skip}, R_1, G_1) \preceq_{\alpha, \zeta \wedge \gamma} (\text{while} \ (B) \{C\}, R_2, G_2)
\]
Loop Unrolling

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

Dead Code Elimination

\[
\begin{align*}
\text{(skip, ld, ld)} & \preceq_{\alpha, \zeta, \gamma} (C, ld, G) & \text{Sta}(\{\zeta, \gamma\}, \langle R_1, R_2 \rangle^\alpha) \\
\text{(skip, R_1, ld)} & \preceq_{\alpha, \zeta, \gamma} (C, R_2, G)
\end{align*}
\]

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

Redundancy Introduction

\[
\begin{align*}
(c, ld, G) & \preceq_{\alpha, \zeta, \gamma} \text{(skip, ld, ld)} & \text{Sta}(\{\zeta, \gamma\}, \langle R_1, R_2 \rangle^\alpha) \\
(c, R_1, G) & \preceq_{\alpha, \zeta, \gamma} \text{(skip, R_2, ld)}
\end{align*}
\]

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

5.2. 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 \(t\).

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

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

\[
\alpha \triangleq \{(\sigma_1, \sigma) | \sigma_1(i) = \sigma(i) \land \sigma_1(n) = \sigma(n) \land \sigma_1(x) = \sigma(x)\}
\]

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

\[
(C_1, R, \text{True}) \preceq_{\alpha, \alpha} (C, R, \text{True}) \quad \text{(5.1)}
\]

We could prove (5.1) directly by the RGSim definition and the operational semantics of the code. But below we give a more convenient proof using the optimization rules and the compositionality rules instead. We first prove the following by the Dead-Code-Elimination and Redundancy-Introduction rules:

\[
\begin{align*}
(t := x + 1, R, \text{True}) & \preceq_{\alpha, \zeta, \gamma} \text{(skip, R, True)}; \\
\text{(skip, R, True)} & \preceq_{\alpha, \zeta, \gamma} (t := x + 1, R, \text{True}),
\end{align*}
\]

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

\[
\gamma \triangleq \alpha \cap \{(\sigma_1, \sigma) | \sigma_1(t) = \sigma_1(x) + 1\} \\
\eta \triangleq \gamma \cap \{(\sigma_1, \sigma) | \sigma(t) = \sigma(x) + 1\}
\]
Then by the compositionality rules SEQ and WHILE, we can get \((C'_1, R, \text{True}) \preceq_{\alpha, \alpha \times \alpha} (C', R, \text{True})\) where \(C_1\) and \(C'\) result from adding \texttt{skips} to \(C_1\) and \(C\):

\[
\begin{align*}
C'_1: & \quad \text{while}(i < n) \{ \\
& \quad \text{skip}; \\
& \quad i := i + t;
\}
\end{align*}
\quad \begin{align*}
C': & \quad \text{skip; } \\
& \quad \text{while}(i < n) \{ \\
& \quad \text{t := x + 1; } \\
& \quad \text{t := x + 1; } \\
& \quad \text{i := i + t; }
\}
\end{align*}
\]

Besides, from Sequential-Unit laws and compositionality rules SEQ and WHILE, we can prove \((C_1, R, \text{True}) \preceq_{\alpha, \alpha \times \alpha} (C'_1, R, \text{True})\) and \((C', R, \text{True}) \preceq_{\alpha, \alpha \times \alpha} (C, R, \text{True})\). Finally, by the \texttt{TRANS} rule, we can conclude (5.1), i.e., 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

<table>
<thead>
<tr>
<th>Target-Level (C_2)</th>
<th>Middle-Level (C_1)</th>
<th>Source-Level (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>local (k, r)</td>
<td>local (i, k)</td>
<td>local (i)</td>
</tr>
<tr>
<td>(k := 0)</td>
<td>(i := 0)</td>
<td>(i := 0)</td>
</tr>
<tr>
<td>(r := 6*i)</td>
<td>(k := 0)</td>
<td>(i := 0)</td>
</tr>
<tr>
<td>while((k &lt; n)) {</td>
<td>while((i &lt; n)) {</td>
<td>while((i &lt; n)) {</td>
</tr>
<tr>
<td>(x := x + k)</td>
<td>(x := x + k)</td>
<td>(x := x + 6*i)</td>
</tr>
<tr>
<td>(k := k + 6)</td>
<td>(i := i + 1)</td>
<td>(i := i + 1)</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

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. Below we give the environments \(R, R_1\) and \(R_2\) at the source, intermediate and target levels respectively:

\[
\begin{align*}
R & \triangleq \{ (\sigma, \sigma') \mid \sigma(i) = \sigma'(i) \} \\
R_1 & \triangleq \{ (\sigma_1, \sigma') \mid \sigma_1(i) = \sigma'_1(i) \land \sigma_1(k) = \sigma'_1(k) \} \\
R_2 & \triangleq \{ (\sigma_2, \sigma'_2) \mid \sigma_2(k) = \sigma'_2(k) \land \sigma_2(x) = \sigma'_2(x) \land \sigma_2(n) = \sigma'_2(n) \}
\end{align*}
\]

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 \(\alpha\) (for the transformation from \(C\) to \(C_1\)) and \(\beta\) (for the transformation from \(C_1\) to \(C_2\)) below.

\[
\begin{align*}
\alpha & \triangleq \{ (\sigma_1, \sigma) \mid \sigma_1(i) = \sigma(i) \land \sigma_1(n) = \sigma(n) \land \sigma_1(x) = \sigma(x) \}; \\
\beta & \triangleq \{ (\sigma_2, \sigma_1) \mid \sigma_2(k) = \sigma_1(k) \land \sigma_2(n) = \sigma_1(n) \land \sigma_2(x) = \sigma_1(x) \}.
\end{align*}
\]

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

\[(C_2, R_2, \text{True}) \preceq_{\beta, \beta, \alpha} (C_1, R_1, \text{True}), (C_1, R_1, \text{True}) \preceq_{\alpha, \alpha \times \alpha} (C, R, \text{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:
\[(C_2, R_2, \text{True}) \preceq_{\alpha \circ \beta \circ \delta \circ \kappa \circ \beta} (C, R, \text{True})\]
where \(\alpha \circ \beta = \{(\sigma_2, \sigma) \mid \sigma_2(n) = \sigma(n) \land \sigma_2(x) = \sigma(x)\}\). 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 \(R\) and \(R_2\)).

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 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 non-blocking 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 \(\text{atom}\{\ \}\) to execute \(C\) 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, \(i.e., (C_1 || C_2)\);\(\text{print}(a)\) and \((A_1 || A_2)\);\(\text{print}(a)\) have the same outputs. We use \(\text{print}(a)\) 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 \(A_2\)), \(i.e., C_1^0\) 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 6 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 \(\alpha\) relation as:
\[\alpha = \{(\sigma, \Sigma) \mid \Sigma(a) = \Sigma(a) \land \Sigma(b) = \Sigma(b) \land \sigma(d1) = \Sigma(d1) \land \sigma(d2) = \Sigma(d2)\}\].

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

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
A_1:  A_2:
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

C_1:  C_2:
local d1, t11, t12;  local d2, t21, t22;
d1 := 0;  d2 := 0;
while (d1 = 0) {  while (d2 = 0) {
  0 0
  1 1
  2 2
  3 3
  4 4
  5 5
  t12 := b;  t11 := b;
t11 := a;  t21 := b;
if (t11 = t12)  if (t21 = t22)
  d1 := 1;  d2 := 1;
if (t11 > t12)  if (t21 > t22)
a := t11 - t12;  b := t21 - t22;
}
}

(b) Target Code

Fig. 8. Concurrent GCD

R_1 = G_2 \triangleq \{(\sigma, \sigma') | \sigma'(t11) = \sigma(t11) \land \sigma'(t12) = \sigma(t12) \land \sigma'(d1) = \sigma(d1) \land \sigma'(a) = \sigma(a) \\
\land (\sigma(a) \geq \sigma(b) \Rightarrow \sigma'(b) = \sigma(b))\}
R_2 = G_1 \triangleq \{(\sigma, \sigma') | \sigma'(t21) = \sigma(t21) \land \sigma'(t22) = \sigma(t22) \land \sigma'(d2) = \sigma(d2) \land \sigma'(b) = \sigma(b) \\
\land (\sigma(b) \geq \sigma(a) \Rightarrow \sigma'(a) = \sigma(a))\}
R_1 = G_2 \triangleq \{(\Sigma, \Sigma') | \Sigma'(d1) = \Sigma(d1) \land \Sigma'(a) = \Sigma(a) \land (\Sigma(a) \geq \Sigma(b) \Rightarrow \Sigma'(b) = \Sigma(b))\}
R_2 = G_1 \triangleq \{(\Sigma, \Sigma') | \Sigma'(d2) = \Sigma(d2) \land \Sigma'(b) = \Sigma(b) \land (\Sigma(b) \geq \Sigma(a) \Rightarrow \Sigma'(a) = \Sigma(a))\}

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

(C_1^0, R_1, G_1) \preceq_{a_{\alpha \kappa \alpha}} (A_1^0, R_1, G_1),  (A_1^0, R_1, G_1) \preceq_{a_{\alpha_{1, 1}} \kappa_{\alpha_{1}}^{-1}} (C_1^0, R_1, G_1).

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

(C_1, R_1, G_1) \preceq_{a_{\alpha \kappa \alpha}} (A_1, R_1, G_1),  (A_1, R_1, G_1) \preceq_{a_{\alpha_{1, 1}} \kappa_{\alpha_{1}}^{-1}} (C_1, R_1, G_1).

Similarly, the relations hold between C_2 and A_2:

(C_2, R_2, G_2) \preceq_{a_{\alpha \kappa \alpha}} (A_2, R_2, G_2),  (A_2, R_2, G_2) \preceq_{a_{\alpha_{1, 1}} \kappa_{\alpha_{1}}^{-1}} (C_2, R_1, 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 \text{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), \text{Id}, True) \preceq_{a_{\alpha \kappa \alpha}} (print(a), \text{Id}, True) and the reverse direction hold. Then by the rules PAR and SEQ, we can get:

((C_1 \parallel C_2); \text{print(a), Id, True}) \preceq_{a_{\alpha \kappa \alpha}} ((A_2 \parallel A_2); \text{print(a), Id, True}),
ADD(e):
0 atom {
  S := S ∪ {e};
}

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

(a) An Abstract Set

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

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

(b) The Lock-Coupling List-Based Set

Fig. 9. The Set Object

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

\[(C_1 \parallel C_2); \text{print(a)} \simeq_T (A_1 \parallel A_2); \text{print(a)},\]

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

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

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
and Parkinson 2007; Vafeiadis 2008]. 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.
thread-local memory and the thread pool respectively. We use ping from thread identifiers (ThrdID) $t \in \text{Nat}$

To support dynamically allocated memory and ownership transfers, we split the
stages into shared and thread-local parts. We first take the generic languages in Fig-
ure 2, and instantiate the high-level program states as follows. The state $\Sigma$ consists
of shared memory $M_s$ (where the object resides) and a thread pool $\Pi$, which is a mapping
from thread identifiers ($t \in \text{ThrdID}$) to their memory $M_t$. The low-level state $\sigma$
is defined similarly. We use $m_s$, $m_t$ and $\pi$ 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 $\text{atom}(\{\text{C}\}_A)$ (or $\{\text{C}\}_A$ at the low level) to convert the shared memory to
local and then execute $\text{C}$ (or $\text{C}$) atomically. Following RGSep [Vafeiadis and Parkinson 2007], an abstract transition $A \in \mathcal{P}(\text{HM} \times \text{HM})$ (or $A \in \mathcal{P}(\text{LM} \times \text{LM})$)

(a) The High-Level Language for Abstract Operations

$$\begin{align*}
\text{(HSmts)} & \quad C ::= \text{skip} \mid c \mid \text{atom}(\{\text{C}\}_A) \mid C_1; C_2 \\
& \quad \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{while } B \text{ do } C
\end{align*}$$

(b) The Low-Level Language for Concrete Implementations

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

(c) Selected Operational Semantics Rules of the High-Level Language

Fig. 10. The Languages for Concurrent Objects

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
list(x, A) ≜ \lambda m_s. (m_s = \emptyset \land x = \texttt{null} \land A = \epsilon) \\
\lor (\exists m'_s, v, y, A'. m_s = m'_s \cup \{x \leadsto (v, y)\} \land A = v::A' \land \text{list}(y, A')(m'_s))

\text{sorted}(A) \triangleq \begin{cases} 
\text{true} & \text{if } A = \epsilon \lor A = a::A \\
(a < b) \land \text{sorted}(b::A') & \text{if } A = a::b::A' 
\end{cases}

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

\text{shared_map}(m_s, M_s) \triangleq \exists m'_s, A, x, m_s = m'_s \cup \{\text{head} \leadsto x\} \land \text{list}(x, \text{MIN}\_\text{VAL}::A::\text{MAX}\_\text{VAL})(m'_s) \\
\land \text{sorted}(A) \land (\text{elems}(A) = M_s(S))
\text{local_map}(m_l, M_l) \triangleq m_l(e) = M_l(e) \land \exists m'_l, m_l = m'_l \cup \{x \leadsto \_, y \leadsto \_, z \leadsto \_, u \leadsto \_, v \leadsto \_\}
\alpha \triangleq \{(\pi, m_s), (\Pi, M_s)\} \land \text{shared_map}(m_s, M_s) \land \forall t \in \text{dom}(\Pi). \text{local_map}(\pi(t), \Pi(t))

(a) The α Relation

\begin{align*}
x \mapsto (n, v, y) & \triangleq \lambda (m_1, m_s). (\text{dom}(m_1) = \emptyset) \land (m_s = \{x \leadsto (n, v, y)\}) \\
x \mapsto (n, v, y) & \triangleq \lambda (m_1, m_s). (m_1 = \{x \leadsto (n, v, y)\}) \land (\text{dom}(m_s) = \emptyset)
\end{align*}
\text{Itrue} \triangleq \lambda (m_1, m_s). (\text{dom}(m_1) = \emptyset)
\text{Op} \times \text{Op} \triangleq \lambda (m_1, m_s, m'_1, m'_s, m''_1, m''_s, p(m'_1, m''_1) \land q(m'_s, m''_s) \land (m_1 = m'_1 \cup m''_1) \land (m_1 = m'_s \cup m''_s))
\begin{align*}
x \mapsto v & \triangleq \lambda (M_1, M_s). (\text{dom}(M_1) = \emptyset) \land (M_s = \{x \leadsto v\}) \\
\text{Itrue} & \triangleq \lambda (M_1, M_s). (\text{dom}(M_1) = \emptyset)
\end{align*}

(b) Syntactic Sugar (where p, q \in \text{LMem} \times \text{LMem} \rightarrow \text{Prop})

\begin{align*}
\text{G}_{\text{lock}}(t) & \triangleq \exists x, v, y. (x \mapsto (0, v, y)) \land (x \mapsto (t, v, y)) \\
\text{G}_{\text{unlock}}(t) & \triangleq \exists x, v, y. (x \mapsto (0, v, y)) \land (x \mapsto (t, v, y)) \\
\text{G}_{\text{add}}(t) & \triangleq \exists x, y, z, u, v, w, n, z'. (x \mapsto (t, u, z) \ast y \mapsto (0, v, z) \ast z \mapsto (n, w, z') \land u < v < w) \\
\text{G}_{\text{rmv}}(t) & \triangleq \exists x, y, z, u, v. (x \mapsto (t, u, y) \ast y \mapsto (t, v, z) \land v < \text{MAX} \_\text{VAL}) \land (x \mapsto (t, u, z) \ast y \mapsto (t, v, z)) \\
\text{G}_{\text{local}}(t) & \triangleq \text{Itrue} \land \text{Itrue} \\
\text{G}(t) & \triangleq \text{G}_{\text{lock}}(t) \lor \text{G}_{\text{unlock}}(t) \lor \text{G}_{\text{add}}(t) \lor \text{G}_{\text{rmv}}(t) \lor \text{G}_{\text{local}}(t) \\
\text{R}(t) & \triangleq \cup_{\text{R} \neq t} \text{G}(t')
\end{align*}

(c) Rely and Guarantee Relations

Fig. 11. Auxiliary Definitions and Specifications for the Lock-Coupling List

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. The atomic blocks are instantiations of the generic primitive operations c (or c) in Figure 2. We omit the annotations ∧ and A in Figure 9, which are the same as the corresponding guarantees in Figure 11, as we will explain below.
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 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 → (n, v, y) and x ↦ (n, v, y) for nodes in the low-level shared memory m_s and the local memory m_t of the current thread respectively. Itrue means the thread-local memory m_t is arbitrary. The separating conjunction p × q means p and q hold on disjoint memory. The action p ↛ q represents the update of some memory (m_1, m_2) 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 ⇝ 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 G_lock, G_unlock, G_add, G_rmv and G_local respectively, which are all parametrized with a thread identifier t. For example, G_rmv(t) says that when holding the locks of the node y and its predecessor x, we can transfer the node y from the shared memory to the thread’s local memory. This corresponds to the action performed by the code of line 13 in rmv(e) 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 R(t). Similarly, the high-level G(t) and R(t) are defined according to the abstract ADD(e) and RMV(e). The relies and guarantees are almost the same as those in the proofs in RGSep [Vafeiadis 2008].

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

\[
\begin{align*}
(\text{add}(e), R(t), G(t)) & \triangleleft^t_{\alpha,\alpha} (\text{ADD}(e), R(t), G(t)); \\
(\text{rmv}(e), R(t), G(t)) & \triangleleft^t_{\alpha,\alpha} (\text{RMV}(e), R(t), G(t)).
\end{align*}
\]

Here \(\triangleleft^t_{\alpha,\alpha}\) is the RGSim relation in Definition 4.2 with the transitions \(\rightarrow\) replaced by \(\rightarrow_t\) (defined in Figure 10(c)). The proofs are done operationally based 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 \( R \)) are simulated by the corresponding atomic operations (under the high-level environment \( \mathbb{R} \)), while \( R \) and \( \mathbb{R} \) say all accesses to the set must be done through the add and remove operations. This gives us the atomicity of the concurrent implementation of the set object.

6.3. Non-Blocking 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 incr\((x)\) uses the compare-and-swap (CAS) instruction \( \text{cas}(&x, t_1, t_2) \), 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. Below we use RGSim to prove the atomicity of incr\((x)\).

\[
\text{atom}\{ \quad x := x+1; \quad \}
\]

(a) Source Code INC\((x)\)

\[
\text{local } d, t;
\]
\[
d := 0;
\]
\[
\text{while } (d = 0) \{
\]
\[
\quad <t := x;>
\]
\[
\quad d := \text{cas}(&x, t, t+1);
\]
\[
\}
\]

(b) Target Code incr\((x)\)

Fig. 12. The Atomic and Non-Blocking Counters

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

\[
\alpha \triangleq \{((\pi, m_s), (\Pi, M_s)) \mid m_s(x) = M_s(x)\}.
\]

We let the pre- and post-conditions be the same as the invariant \( \alpha \). Both incr\((x)\) 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.

\[
G(t) \triangleq (\exists n. (x \mapsto n * \text{Itrue}) \land (x \mapsto n + 1 * \text{Itrue})) \lor G_{\text{local}}(t) \quad R(t) \triangleq \bigcup_{t' \neq t} G(t')
\]

Then we can prove the RGSim relation holds:

\[
(\text{incr}(x), R(t), G(t)) \preceq_{t;\alpha;\alpha} (\text{INC}(x), R(t), G(t)).
\]

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

Also we can prove \((\text{INC}(x), R(t), G(t)) \preceq_{t;\alpha;\alpha} (\text{incr}(x), R(t), G(t))\), which says the implementation incr\((x)\) has all the behaviors of INC\((x)\). Thus incr\((x)\) and INC\((x)\) behave just the same.
As a simple illustration of the atomicity, we go on to show that the non-blocking
inc(x) can be used by two threads concurrently without missing any increment,
as if x was updated by the threads one after the other. Formally, we prove that
\(\text{INC}(x) \parallel \text{INC}(y)\) and \(\text{INC}(x) \parallel \text{INC}(y)\) have
the same observable event traces when the initial values of x are the same.

We can prove that \(\text{INC}(x) \parallel \text{INC}(y)\) and the reverse
direction hold. Then by the rules SEQ, PAR and CONSEQ, we can get both
\[
((\text{INC}(x) \parallel \text{INC}(y)) \parallel \text{INC}(x) \parallel \text{INC}(y) \parallel \text{INC}(x) \parallel \text{INC}(y)) \approx T \ (\text{INC}(x) \parallel \text{INC}(x) \parallel \text{INC}(x) \parallel \text{INC}(x))
\]
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:
\[
((\text{INC}(x) \parallel \text{INC}(y)) \parallel \text{INC}(x) \parallel \text{INC}(y) \parallel \text{INC}(x) \parallel \text{INC}(x)) \approx T \ (\text{INC}(x) \parallel \text{INC}(x) \parallel \text{INC}(x) \parallel \text{INC}(x))
\]
for any T that respects \(\alpha\). That is, no matter how the two non-blocking threads inter-
leave, 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: \((\text{INC}(y), R(t), G(t)) \approx(\text{INC}(y), R(t), G(t))\) and \(\text{INC}(y) \parallel \text{INC}(y)\) have
the same observable event traces when the initial values of \(x\) and \(y\) are the same.

By the rules FRAME and SEQ, we can get:
\[
((\text{INC}(x) \parallel \text{INC}(y)) \parallel \text{INC}(x) \parallel \text{INC}(y) \parallel \text{INC}(x) \parallel \text{INC}(y)) \approx T \ (\text{INC}(x) \parallel \text{INC}(x) \parallel \text{INC}(x) \parallel \text{INC}(x))
\]
6.4. Treiber’s Non-Blocking Stack

The last example is to verify the atomicity of Treiber’s non-blocking 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 non-blocking code push(v) and pop() respectively. As shown in Figure 13(b),
the non-blocking implementation uses CAS instructions to obtain fine-grained atomicity.

We use RGSim to prove the atomicity of the non-blocking stack, i.e., push(v) refines
push(v) and pop() refines pop() when they are executed in appropriate environments.

We define the \(\alpha\) 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 sub-memory \( \tilde{m}_s \) 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 in Section 6.2, we use the predicate \( \text{list}(x, A)(\tilde{m}_s) \) to represent a singly-linked list in the shared memory \( \tilde{m}_s \) 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 \( \tilde{m}_s \). For the local memory, \( \text{local}_{\text{map}} \) 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 \( \alpha \) relates the shared memory by \( \text{shared}_{\text{map}} \) and the local memory of each thread \( t \) by \( \text{local}_{\text{map}} \).

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, \( G_{\text{push}}(t) \) 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 non-blocking stack operations are simulated by the corresponding atomic operations:

\[
(p_{\text{push}}(v), R(t), G(t)) \preceq_{\alpha, \alpha \times \alpha} (P_{\text{USH}}(v), R(t), G(t));
(r := p_{\text{op}}(), R(t), G(t)) \preceq_{\alpha, \alpha \times \alpha} (r := P_{\text{OP}}(), R(t), G(t)).
\]

This gives us the atomicity of the non-blocking implementation of the stack object.
shared_map(m_s, M_s) \triangleq \exists \tilde{m}_s. \text{list}(m_s(S), M_s(A))(\tilde{m}_s) \land \tilde{m}_s \subseteq m_s \setminus \{s\}
local_map(m_t, M_t) \triangleq m_t(\{\} = M_t(\{\}) \land \exists m'_t. m_t = m'_t \cup \{d \leadsto \_ \leadsto \_t \leadsto \_x \leadsto \_\}
\alpha \triangleq \{((\pi, m_s), (I, M_s)) \mid \text{shared_map}(m_s, M_s) \land \forall t \in \text{dom}(I). \text{local_map}(\pi(t), I(t))\}
\begin{align*}
\text{G}_{\text{push}}(t) & \triangleq \exists v, x, y. (S \leadsto x \leadsto x \rightarrow (v, y) \ast \text{true}) \land_1 (S \rightarrow x \rightarrow x \rightarrow (v, y) \ast \text{true}) \\
\text{G}_{\text{pop}}(t) & \triangleq \exists x, v, y. (S \rightarrow x \rightarrow x \rightarrow (v, y) \ast \text{true}) \land_1 (S \rightarrow y \rightarrow x \rightarrow (v, y) \ast \text{true}) \\
G(t) & \triangleq \text{G}_{\text{push}}(t) \cup \text{G}_{\text{pop}}(t) \cup \text{G}_{\text{local}}(t) \\
R(t) & \triangleq \bigcup_{t \neq 1} G(t')
\end{align*}
Fig. 14. Auxiliary Definitions and Specifications for the Non-Blocking Stack

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 $T$ 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 $C_{ge}$ running concurrently with the target mutators. That is,

$$T(t_{ge}, AbsGC || t_1.C_1 || \ldots || t_n.C_n) \triangleq t_{ge}.C_{ge} || t_1.T(C_1) || \ldots || t_n.T(C_n),$$

where $T(C)$ simply translates some memory access instructions in $C$ 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

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
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 \( \text{Correct}(T) \), 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 \( \text{Correct}(T) \), 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 \( \text{AbsGCStep} \):

\[
(\Sigma, \Sigma') \in \text{AbsGCStep}
\]

\[
(t_{gc}\text{AbsGC}, \Sigma) \rightarrow (t_{gc}\text{AbsGC}, \Sigma')
\]

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

Thus for the GC thread, we need to show that \( C_{gc} \) is simulated by \( \text{AbsGC} \) when executed in their environments. This can be reduced to unary Rely-Guarantee reasoning about \( C_{gc} \) by proving \( R_{gc} \): \( G_{gc} \leftarrow \{p_{gc}\}C_{gc}\{q_{gc}\} \) in a standard Rely-Guarantee logic with proper \( R_{gc}, G_{gc}, p_{gc} \) and \( q_{gc} \), as long as \( G_{gc} \) is a concrete representation of \( \text{AbsGCStep} \). The judgment says given an initial state satisfying the precondition \( p_{gc} \), if the environment’s behaviors satisfy \( R_{gc} \), then each step of \( C_{gc} \) satisfies \( G_{gc} \), and the postcondition \( q_{gc} \) holds at the end if \( C_{gc} \) terminates. In general, the collector never terminates, thus we can let \( q_{gc} \) be \( \text{false} \). \( G_{gc} \) 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. \( R_{gc} \) encodes the possible behaviors of mutators, which can be derived, as we will show below.

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

\[
G(t) \triangleq \bigcup_{c} G_{T(c)}^{t} ; \quad G(t) \triangleq \bigcup_{c} G_{c}^{t} . \quad (7.1)
\]

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

\[
R(t) \triangleq G_{gc} \cup \bigcup_{t'} G(t') ; \quad R(t) \triangleq \text{AbsGCStep} \cup \bigcup_{t'} G(t') . \quad (7.2)
\]

The \( R_{gc} \) used to verify the GC code can now be defined below:

\[
R_{gc} \triangleq \bigcup_{t} G(t) . \quad (7.3)
\]
The refinement proof also needs definitions of binary relations $\alpha$, $\zeta$ and $\gamma$. The invariant $\alpha$ relates the low-level and the high-level states and needs to be preserved by each low-level step. In general, a high-level state $\Sigma$ can be mapped to a low-level state $\sigma$ by giving a concrete local store for the GC thread, adding additional structures in the heap (to record information for collection), renaming heap cells (for copying GCs), etc. The relations $\zeta$ and $\gamma$ are parametrized over the thread id $t$. For each mutator thread $t$, $\zeta(t)$ and $\gamma(t)$ need to hold at the beginning and the end of each basic transformation unit (every high-level primitive instruction in this case) respectively. We let $\gamma(t)$ be the same as $\zeta(t)$ to support sequential compositions. We require $\text{InitRel}_T(\zeta(t))$ (see Figure 6), i.e., $\zeta(t)$ holds over the initial states. In addition, the target and the source boolean expressions should be evaluated to the same value under $\zeta$-related states, as required in the IF and WHILE rules in Figure 7.

$$\text{Good}_T(\zeta(t)) \triangleq \text{InitRel}_T(\zeta(t)) \land \forall B. \zeta(t) \subseteq (T(B) \iff B) \quad (7.4)$$

**Theorem 7.1 (Verifying Concurrent Garbage Collecting Systems).** If there exist $G^1_c$, $G^2_{T(c)}$, $\zeta(t)$, $\alpha$, $\zeta_{ge}$ and $p_{ge}$ (for any $c$ and $v$) such that the following hold (where $G(t)$, $G(t)$, $R(t)$, $R(t)$ and $R_{ge}$ are defined in (7.1), (7.2) and (7.3), and $\text{Good}_T(\zeta(t))$ defined in (7.4) holds):

1. (Correctness of $T$ on mutator instructions)
   $$\forall t, c. \ (T(c), R(t), G(t)) \subseteq_t \zeta_{T(c)} \quad (c, R(t), G(t));$$
2. (Verification of the GC code)
   $$R_{ge} : G_{ge} \vdash \{p_{ge}\} C_{ge}\{\text{false}\};$$
3. (Side conditions)
   $$\zeta_{ge} \circ \alpha^{-1} \subseteq \alpha^{-1} \circ (\text{AbsGCStep})^{*} \quad \text{and} \quad \forall \sigma, \Sigma. \sigma = T(\Sigma) \implies p_{ge} \sigma;$$

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

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

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

— Find some proper $\zeta_{ge}$ and $p_{ge}$, and verify the GC code by R-G reasoning. We require the GC’s guarantee $\zeta_{ge}$ should not contain more behaviors than $\text{AbsGCStep}$ (the first side condition), and $\zeta_{ge}$ can start its execution from any state $\sigma$ transformed from a high-level one (the second side condition).

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

$$(C_{ge}, R_{ge}, G_{ge}) \subseteq_{\alpha, \zeta_{ge} \times \zeta_{ge}} (\text{AbsGC}, \text{True}, \text{AbsGCStep})$$

Here $\zeta_{ge} \triangleq \{(\sigma, \Sigma) \mid \sigma = T(\Sigma)\}$. 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:

$$\forall C_1, \ldots, C_n. \ (t_{ge}. C_{ge} || t_1. T(C_1) || \ldots || t_n. T(C_n), \text{Id}, \text{True})$$

$$\subseteq_{\alpha, \zeta_{ge} \times \zeta_{ge}} (t_{ge}. \text{AbsGC} || t_1. C_1 || \ldots || t_n. C_n, \text{Id}, \text{True}).$$

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

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
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 re-traces from the dirty objects which have been marked (called card-cleaning). Thus all reachable objects have been marked before the sweep phase, ensuring the correctness of the GC.

We show the code of the GC thread in Figure 15. We assume each object contains $m$ pointer fields $pt_1, \ldots, pt_m$, 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 $[1..N]$.

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 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 rescanned 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
constant int WHITE, BLACK, BLUE; // colors
constant int N; // total number of threads
constant int M; // size of heap

Collection() {
  local mstk;
  while (true) {
    Initialize();
    Trace();
    CleanCard();
    atomic{ ScanRoot(); CleanCard(); }
    Sweep();
  }
}

Initialize() {
  local i, c;
  i := 1;
  while (i <= M) {
    Initialize();
    c := i.color;
    i := i + 1;
  }
}

Trace() {
  local t, rt, i;
  t := 1;
  while (t <= N) {
    rt := get_root(t);
    foreach i in rt do {
      MarkAndPush(i);
      rt := get_root(t);
      foreach i in rt do {
        t := t + 1;
        TraceStack();
      }
    }
    t := t + 1;
  }
}

TraceStack() {
  local i, j;
  while (!is_empty(mstk)) {
    i := pop(mstk);
    j := i.pt[1]; MarkAndPush(j);
    ...
    j := i.pt[m]; MarkAndPush(j);
  }
}

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

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

Sweep() {
  while (!is_empty(mstk)) {
    local i, c;
    i := pop(mstk);
    i := 1;
    j := i.pt[1]; MarkAndPush(j);
    while (i <= M) {
      ...
      j := i.pt[m]; MarkAndPush(j);
    }
    if (c = WHITE) { free(i); }
    i := i + 1;
  }
}

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

(HExpr) E ::= x | n | nil | E+E | E-E | ...  
(HBExpr) B ::= true | false | E=E | !B | ...  
(HInstr) c ::= print(E) | x:=E | x:=y.fd | x.fd := E | x := new()  
(HStmts) C ::= skip | c | C1; C2 | if B then C1 else C2 | while B do C  
(HProg) W ::= t0.AbsGC||t1.C1||...||tn.Cn  
(HField) fd ∈ {pt1,...,ptm,data}  
(MutID) t ∈ [1..N]

(a) The Language

(b) Program States

Fig. 17. The High-level Language and State Model

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 (Gt 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.
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 is regarded as distinct kinds (or types) of values. We present the high-level operational semantics for the language. For the power set and for pointers, so that the GC can easily follow a path to the pointers field of an object, and then it will be transformed to the write barrier in Figure 16. Note here \( E \) is restricted to be either nil (null pointers) or pointer variables. That is, to write an expression \( E \) to the pointer field \( x.\text{fd} \), the high-level mutator has to first assign \( E \) to a pointer variable \( y \) and then perform \( x.\text{fd} := y \).

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 \( \text{SameType}(V, V') \) 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 \( \text{Seq}(\_ \_) \) for the set of sequences. Every low-level variable is given an extra bit to preserve its high-level type information (0 for non-pointers and 1 for pointers), so that the GC can easily get the roots. The low-level mutators are still prohibited from pointer arithmetic. An expression \( E \) is evaluated (shown in Figure 20) under the store with an extra tag \( \text{tag} \) to indicate whether it is used as an object location in the heap (\( \text{tag} = 1 \) if \( E \) is used as a heap location; and \( \text{tag} = 0 \) otherwise). When \( \text{tag} = 2 \), 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 \texttt{foreach $x$ in $y$ do $C$}, we assume $x$ and $y$ are temporary variables and not updated by $C$. At the beginning of each iteration, we set $x$ to an arbitrary item in the set $y$, and after executing $C$ we remove that item from $y$. The \texttt{foreach} loop terminates when $y$ becomes empty.
\[
\begin{align*}
&\text{t} \in [1..N] \quad s(x) = (\_b) \quad [E]_{(s,b)} = n \quad s' = s \{ x \leadsto (n, b) \} \\
&(x := E, (\pi \uplus \{ t \leadsto s \}, h)) \rightarrow_1 (skip, (\pi \uplus \{ t \leadsto s' \}, h)) \\
&s(x) = (\_b) \quad [E]_{(s,b)} = n \quad s' = s \{ x \leadsto (n, b) \} \\
&(x := E, (\pi \uplus \{ t_{ge} \leadsto s \}, h)) \rightarrow_1 (\pi \uplus \{ t_{ge} \leadsto s' \}, h)) \\
&s(y) = (n, 1) \quad h(n) = o \quad fd \in \{ pt_1, \ldots, pt_m \} \Rightarrow n' \quad fd \in \{ data \} \Rightarrow \Rightarrow n' \quad fd \in \{ color, dirty \} \Rightarrow \Rightarrow n' \\
&(x, fd := E, (\pi \uplus \{ t \leadsto s \}, h)) \rightarrow_1 (\pi \uplus \{ t \leadsto s \}, h(n \leadsto o(fd \leadsto n')))) \\
&s(y) = (t, 0) \quad s(x) = (\_0) \quad \pi(t) = s_1 \quad S = \{ n \mid \exists x.s(x) = (n, 1) \} \quad s' = s \{ x \leadsto (S, 0) \} \\
&(x := get_root(y), (\pi \uplus \{ t_{ge} \leadsto s \}, h)) \rightarrow_1 (\pi \uplus \{ t_{ge} \leadsto s' \}, h)) \\
&x \in \text{dom}(s) \quad s(y) = (\emptyset, 0) \\
&(\text{foreach } x \in y \text{ do } C, (\pi \uplus \{ t_{ge} \leadsto s \}, h)) \rightarrow_1 (\pi \uplus \{ t_{ge} \leadsto s \}, h)) \\
&s(x) = (\_b) \quad s(y) = (\{ n_1, \ldots, n_k \}, 0) \quad s' = s \{ x \leadsto (n_1, b) \} \\
&(\text{foreach } x \in y \text{ do } C, (\pi \uplus \{ t_{ge} \leadsto s \}, h)) \rightarrow_1 (\pi \uplus \{ t_{ge} \leadsto s \}, h)) \\
&(C, (\pi \uplus \{ t \leadsto s \}, h)) \Rightarrow_1 \Rightarrow (\pi \uplus \{ t \leadsto s' \}, h')) \\
&(C, (\pi \uplus \{ t \leadsto s \}, h)) \Rightarrow_1 \Rightarrow \text{abort} \\
&(\text{atomic } C, (\pi \uplus \{ t \leadsto s \}, h)) \Rightarrow_1 (\text{atomic } C, (\pi \uplus \{ t \leadsto s \}, h)) \\
&(\text{free } x, (\pi \uplus \{ t \leadsto s \}, h)) \rightarrow_1 (\pi \uplus \{ t \leadsto s \}, h)) \\
&s(x) = (n, 1) \quad h(n) = o \\
&(x := \text{new}, (\pi \uplus \{ t \leadsto s \}, h)) \rightarrow_1 (\pi \uplus \{ t \leadsto s \}, h)) \\
&s(x) = (\_1, 1) \quad h(n)(\text{color}) = \text{BLUE} \quad s' = s \{ x \leadsto (n, 1) \} \\
&h' = h(n \leadsto \{ pt_1 \leadsto 0, \ldots, pt_m \leadsto 0, \text{data} \leadsto 0, \text{color} \leadsto \text{BLACK}, \text{dirty} \leadsto 0 \}) \\
&(x := \text{new}, (\pi \uplus \{ t \leadsto s \}, h)) \rightarrow_1 (\pi \uplus \{ t \leadsto s \}, h)) \\
&s(x) = (\_1, 1) \quad h(n) = o \\
&\Rightarrow_1 \Rightarrow (\pi \uplus \{ t \leadsto s \}, h)) \\
&s(x) = (\_n', b) \quad s(y) = (A, 0) \quad s' = s \{ y \leadsto (n' : A, 0) \} \\
&(\text{push } x, (\pi \uplus \{ t \leadsto s \}, h)) \rightarrow_1 (\pi \uplus \{ t \leadsto s \}, h)) \\
&s(x) = (\_b) \quad s(y) = (n : A, 0) \quad s' = s \{ x \leadsto (n, b), y \leadsto (A, 0) \} \\
&(x := \text{pop } y, (\pi \uplus \{ t \leadsto s \}, h)) \rightarrow_1 (\pi \uplus \{ t \leadsto s \}, h))
\end{align*}
\]

Fig. 21. Selected Operational Semantics Rules on the Low-Level Machine
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 \texttt{nil} for null pointers and it will be translated to 0 on the low-level machine. We assume there is a bijective function from high-level locations to low-level integers:

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

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

The transformation \( T \) is defined as follows. For \textit{code}, the high-level abstract GC thread is transformed to the GC thread shown in Figure 15. Each instruction \( x.\texttt{fd} := E \) in mutators is transformed to the write barrier \( \text{update}(x.\texttt{fd}, T(E)) \), where \( \texttt{fd} \) is a pointer field of \( x \). \( T \) over expressions \( E \) returns 0 if \( E \) is \texttt{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 \( T(\Sigma) \) is defined in Figure 22.

First we require the high-level initial state to be \textit{well-formed} (\( \text{WfState}(\Sigma) \)), i.e., reachable locations cannot be dangling pointers.

High-level locations are transformed to integers by the bijective function \( \text{Loc2Int} \).

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

High-level objects are transformed to the low level by adding the \texttt{color} and \texttt{dirty} fields with initial values \texttt{WHITE} and 0 respectively. Other addresses in the low-level heap domain \([1..M]\) are filled out using unallocated objects whose \texttt{colors} are \texttt{BLUE} and all the other fields are initialized by 0.

The concrete GC thread is given an initial store \( s_{gc,init} \) where its local variables are initialized by 0 (for integer and pointer variables), \( \epsilon \) (for the mark stack \texttt{mstk}) or \( \emptyset \) (for the root set \texttt{rt}).
store_map(s, S) \triangleq \forall x \neq \text{aux}. (\forall m. s(x) = n^p \iff S(x) = n) \\
\land (\forall m. s(x) = n^p \iff \exists l. \text{Loc2Int}(l) = n \land S(x) = l)

\text{obj_map}(\alpha, O) \triangleq \exists n_1, \ldots, n_m, n. c, l_1, \ldots, l_m. \text{Loc2Int}(l_i) = n_i \ldots \land \text{Loc2Int}(l_m) = n_m \\
\land \alpha = \{\{\pi_1 \sim n_1, \ldots, \pi_n \sim n_m, \text{data} \sim n, \text{color} \sim c, \text{dirty} \sim \}_{\bot} \land c \neq \text{BLUE} \land O = \{\{\pi_1 \sim l_1, \ldots, \pi_n \sim l_m, \text{data} \sim n\}\}

\text{unalloc}(\alpha, H, l) \triangleq \{o = \{\pi_1 \sim n_1, \ldots, \pi_n \sim n_m, \text{data} \sim n, \text{color} \sim \text{BLUE}, \text{dirty} \sim \}_{\bot} \land l \not\in \text{dom}(H)

\text{heap_map}(h, H) \triangleq \forall i, l. 1 \leq i \leq M \land \text{Loc2Int}(l_i) = i \implies \text{obj_map}(h(i), H(l_i)) \lor \text{unalloc}(h(i), H, l)

\alpha \triangleq \{((\pi \cup \{\text{gc} \sim \}_{\bot}), h), (\Pi, H)) \mid \forall t. \text{store_map}(\pi(t), \Pi(t)) \land \text{heap_map}(h, H) \land \text{WfState}(\Pi, H)\}

\text{Fig. 23. The } \alpha \text{ Relation for Boehm et al. GC}

To prove \text{Correct}(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 \( \alpha \) and \( \zeta(t) \) relations. In \( \alpha \) (see Figure 23), the relations between low-level and high-level stores and heaps are enforced by \text{store_map} and \text{heap_map} respectively. Their definitions reflect the state transformations we describe above, ignoring the values of those high-level-invisible structures (e.g., the GC’s local variables, the \text{color} and \text{dirty} fields for non-blue objects and all the fields of blue objects). \( \alpha \) also requires the well-formedness of high-level states. Here we still use \text{Loc2Int} to relate integers and locations.

For each mutator thread \( t \), the \( \zeta(t) \) relation enforced at the beginning and the end of each transformation unit (each high-level instruction) is stronger than \( \alpha \). It requires that the value of the auxiliary variable \( \text{aux} \) (see Figure 16) be a null pointer (0^p):

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

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 \( \text{own}_p(x) \) and \( \text{own}_n(x) \) for the current thread’s ownerships of pointers and non-pointers respectively. They are interpreted under \( (\pi, s, h) \), where \( s \) is the store of the current thread, \( \pi \) consists of the stores of all the other threads and \( h \) is the shared heap. We use \( E_1.\text{fd} \rightarrow E_2 \) to specify a single-object single-field heap with \( E_2 \) stored in the field \( \text{fd} \) of the object \( E_1 \). The separating conjunction \( p \land q \) means \( p \) and \( q \) hold on disjoint states. We define the disjoint union of states in Figure 15(c). We use \( E_1 \cup E_2 \) 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 \text{uncurry} operator. We then use \( h_1 \oplus h_2 \) to denote the union when their domains of \text{uncurry}(h_1) and \text{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 \( E_1.\text{fd} \rightarrow E_2 \) for \( E_1.\text{fd} \rightarrow E_2 \) \( \ast \text{true} \) and \( \otimes \subseteq \ast \text{true} \) \( p(x) \) for iterated separating conjunction over the set \( S \). We overload the notations to the high-level machine and use \( E_1.\text{fd} \Rightarrow E_2 \) for a single-object single-field heap at the high level.

In Figure 24(d), we define two forms of actions. \( p \prec q \) represents the update over the current thread t’s store and the shared heap, which is defined similarly as in Figure 11(b). \( p \prec q \) 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 \((x^p = n)\) short for \((x = n) \land \text{own}_p(x)\) and \((x^m = n)\) for \((x = n) \land \text{own}_n(x)\). When the context is clear, we omit the superscript. The predicates \text{blueobj} and \text{newobj} denote a
(PVarList) \[O ::= \bullet \mid x, O\]
(StateAssert) \[p, q \in LThrs \times LStore \times LHeap \rightarrow Prop\]

(a) State Assertions

\[
\begin{align*}
B & \triangleq \lambda (\pi, s, h). [E]_{E_{(s, 2)}} = \text{true} \\
\text{emp}_h & \triangleq \lambda (\pi, s, h). \text{dom}(h) = \emptyset \\
\text{own}_{np}(x) & \triangleq \lambda (\pi, s, h). \text{dom}(s) = \{x\} \land s(x) = (\cdot, 0) \\
\text{own}_{p}(x) & \triangleq \lambda (\pi, s, h). \text{dom}(s) = \{x\} \land s(x) = (\cdot, 1) \\
\text{own}(x) & \triangleq \lambda (\pi, s, h). \text{dom}(s) = \{x\} \\
p \times q & \triangleq \lambda (\pi, s, h). \text{dom}(s) = \{x\} \\
t \times E & \triangleq \lambda (\pi, s, h). \exists n, b. \pi(t)(x) = (n, b) \land [E]_{E_{(s, 2)}} = n \\
E_1, f_d \rightarrow E_2 & \triangleq \lambda (\pi, s, h). \exists n, n'. [E]_{E_{(s, 2)}} = n' \land \text{dom}(h) = \{n'\} \\
& \quad \land h(n')(f_d) = n \land \text{dom}(h(n')) = \{f_d\} \land [E]_{E_{(s, 2)}} = n \\
E_1, f_d \rightarrow E_2 & \triangleq (E_1, f_d \rightarrow E_2) \times \text{true} \\
o_{np}, o_p \vdash p & \triangleq (\text{own}_{np}(x_1) \times \ldots \times \text{own}_{np}(x_i) \times \text{own}_p(y_1) \times \ldots \times \text{own}_p(y_j)) \land p \\
& \quad \text{where } o_{np} = x_1, \ldots, x_i, \bullet \text{ and } o_p = y_1, \ldots, y_j, \bullet \\
x \in S & \triangleq \exists X, S. X = S \sqcup \{x\} \\
\oplus_{\in S} p(x) & \triangleq (S = \phi \land \text{emp}) \lor (\exists z, S'. (S = \{z\} \sqcup S') \land (\oplus_{\in S'} p(x)) \times p(z))
\end{align*}
\]

(b) Shorthand Notations for Some State Assertions (\(\oplus \) defined below)

\[
\begin{align*}
f_1 \sqcup f_2 & \triangleq \text{dom}(f_1) \cap \text{dom}(f_2) = \emptyset \\
f_1 \sqcup f_2 & \triangleq \{f_1 \cup f_2 \text{ if } f_1 \sqcup f_2 \}
\quad \text{otherwise} \\
h_1 \sqcap h_2 & \triangleq \{\text{curry}(\text{uncurry}(h_1) \sqcup \text{uncurry}(h_2)) \text{ if } \text{uncurry}(h_1) \sqcup \text{uncurry}(h_2) \}
\quad \text{otherwise} \\
\pi_1 \sqcup \pi_2 & \triangleq \{t \leadsto (\pi_1(t) \sqcup \pi_2(t)) \mid t \in \text{dom}(\pi_1)\}
\quad \text{if } \text{dom}(\pi_1) = \text{dom}(\pi_2) \land \forall t \in \text{dom}(\pi_1). \pi_1(t) \sqcup \pi_2(t) \}
\quad \text{otherwise} \\
\sigma_1 \sqcup \sigma_2 & \triangleq \{\pi, h\} \text{ if } \sigma_1 = (\pi_1, h_1) \land \sigma_2 = (\pi_2, h_2) \land \pi_1 \sqcup \pi_2 = \pi \land h_1 \sqcap h_2 = h
\quad \text{otherwise}
\end{align*}
\]

(c) Disjoint Unions

\[
p \sqcup q \triangleq \{(\pi \sqcup \{t \leadsto s\}, h), (\pi \sqcup \{t \leadsto s', h'\}) \mid \exists s_1, h_1, s_2, h_2, s'_1, h'_1, p(\pi, s_1, h_1) \land q(\pi, s'_1, h'_1) \land (s = s_1 \sqcup s_2) \land (h = h_1 \sqcap h_2) \land (s' = s'_1 \sqcup s_2) \land (h' = h'_1 \sqcap h'_2)\}
\]

(d) Actions

Fig. 24. Semantics of Basic Assertions

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:

\[
\text{(update}(x, f_d, E), \mathcal{R}(t), \mathcal{G}_{\text{write}}^{t}) \triangleright_{\alpha(\zeta(t))} (x.f_d := E, \mathbb{R}(t), \mathcal{G}_{\text{write}, \text{pt}}^{t}),
\]

where \(\mathcal{G}_{\text{write}, \text{barrier}}^{t} \triangleq \mathcal{G}_{\text{write}, \text{pt}}^{t} \cup \mathcal{G}_{\text{set, dirty}}^{t}\), i.e., the guarantee of the low-level two-step write barrier. \(\mathcal{G}_{\text{write}, \text{pt}}^{t}\) is the guarantee of the high-level atomic write operation. Recall \(\mathcal{R}(t)\)
G_{\text{assign_int}} \triangleq \exists x, n, n'. (x^n = n \land \text{emp}_q) \Rightarrow (x^n = n' \land \text{emp}_q) \quad \text{provided} (\text{aux}^p = 0)

G_{\text{assign_pt}} \triangleq \exists x, l, l'. (x^l = l \land \text{emp}_p) \Rightarrow (x^{l'} = l' \land \text{emp}_p) \quad \text{provided} (l' = \text{nil} \lor \exists y. y = l' \lor \exists y, \text{fd}. \text{y}.\text{fd} \Rightarrow l')

G_{\text{write_data}} \triangleq \exists x, n, n'. (x.\text{data} \Rightarrow n) \Rightarrow (x.\text{data} \Rightarrow n')

G_{\text{write_pt}} \triangleq \exists x, \text{fd}, l, l'. (x.\text{fd} \Rightarrow l) \Rightarrow (x.\text{fd} \Rightarrow l') \quad \text{provided} (l' = \text{nil} \lor \exists y. y = l')

G_{\text{new}} \triangleq \exists x. (x = \_ \land \text{emp}_n) \Rightarrow (x = l \land \text{pt}_1 \Rightarrow \text{nil} * \ldots * \text{lp}_m \Rightarrow \text{nil} * l.\text{data} \Rightarrow 0)

G(t) \triangleq G_{\text{assign_int}} \cup G_{\text{assign_pt}} \cup G_{\text{write_data}} \cup G_{\text{write_pt}} \cup G_{\text{new}}

(a) High-Level Guarantees

G_{\text{assign_int}} \triangleq \exists x, n, n'. (x^n = n \land \text{emp}_n) \Rightarrow (x^n = n' \land \text{emp}_n) \quad \text{provided} (\text{aux}^p = 0)

G_{\text{assign_pt}} \triangleq \exists x, n, n'. (x^p = n \land \text{emp}_p) \Rightarrow (x^p = n' \land \text{emp}_p) \quad \text{provided}

G_{\text{write_data}} \triangleq \exists x, n, n'. (x.\text{data} \Rightarrow n) \Rightarrow (x.\text{data} \Rightarrow n') \quad \text{provided} (\text{aux}^p = 0)

G_{\text{write_pt}} \triangleq \exists x, \text{fd}, n, n'. (\text{aux}^p = 0 \lor x.\text{fd} \Rightarrow n) \Rightarrow (\text{aux}^p = x \times x.\text{fd} \Rightarrow n')

G_{\text{set_dirty}} \triangleq \exists n. (\text{aux}^p = 0 \land n.\text{dirty} \Rightarrow \_ \Rightarrow (\text{aux}^p = 0 \land n.\text{dirty} \Rightarrow 1)

G_{\text{new}} \triangleq \exists x, n, n'. (x^n = n \land \text{blueobj}(n')) \Rightarrow (x^n = n' \land \text{newobj}(n')) \quad \text{provided} (\text{aux}^p = 0)

G(t) \triangleq G_{\text{assign_int}} \cup G_{\text{assign_pt}} \cup G_{\text{write_data}} \cup G_{\text{write_pt}} \cup G_{\text{new}}

(b) Low-Level Guarantees

\{ (x^n = X') \ast (1 \leq y^n \leq N) \} x := \text{get_root}(y) \{(x^n = X) \ast (1 \leq y^n \leq N \land \text{root}(y, X)) \}

\{(x.\text{color} \Rightarrow \_ \text{free}(x) \Rightarrow \text{BLUE}) \}

\{ y, \text{O}_n; \text{O}_p \vdash x = X \land y = Y \} \text{push}(x, y) \{ y, \text{O}_n; \text{O}_p \vdash x = X \land y = X : Y \}

\{ y, \text{O}_n; \text{O}_p \vdash x = X \land y = X' : Y \} x := \text{pop}(y) \{ y, \text{O}_n; \text{O}_p \vdash x = X' \land y = Y \}

\{ p \}_{C} \{ q \} \quad (\text{ATOM})

\text{ld}; G \vdash \{ p \}_{\text{atomic}} \{ q \} \quad (\text{ATOM-R})

\text{ld}; G \vdash \{ p \}_{\text{atomic}} \{ q \}

\text{ld}; G \vdash \{ p \}_{\text{atomic}} \{ q \}

p \Rightarrow \text{own}_n(y) \Rightarrow \text{true} \quad R; G \vdash \{ p \ast \text{own}(x) \land x \in y \} \quad C; y := y' \{ x \ast p \ast \text{own}(x)\}

\text{R}; G \vdash \{ p \ast \text{own}(x) \}_{\text{foreach}} x \in y \text{ do } C \{ p \ast \text{own}(x) \land y = \emptyset \}

\text{P-FOREACH}

Fig. 25. Guarantees of Mutator Instructions

Fig. 26. Selected Inference Rules for GC Verification

\text{and } R(t) \text{ are defined in Equation (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:}

\( (x := \text{new}(), R(t), G_{\text{new}} \cup G_{\text{assign_pt}}) \Rightarrow (x := \text{new}(), R(t), G_{\text{new}} \cup G_{\text{assign_pt}}) \).
obj(x) \triangleq x.pt_1 \mapsto \ldots x.pt_n \mapsto x.data \mapsto x.color \mapsto x.dirty \mapsto \text{true}
blueobj(x) \triangleq x.pt_1 \mapsto \ldots x.pt_n \mapsto x.data \mapsto x.color \mapsto \text{BLUE} \mapsto x.dirty \mapsto \text{false}
newobj(x) \triangleq x.pt_1 \mapsto 0 \mapsto \ldots x.pt_n \mapsto 0 \mapsto x.data \mapsto 0 \mapsto x.color \mapsto \text{BLACK} \mapsto x.dirty \mapsto 0
black(x) \triangleq x.color \mapsto \text{BLACK}
white(x) \triangleq x.color \mapsto \text{WHITE}
dirty(x) \triangleq x.dirty \mapsto 1
not_blue(x) \triangleq \exists c. (x.color \mapsto c \land c \neq \text{BLUE})
not_white(x) \triangleq \exists c. (x.color \mapsto c \land c \neq \text{WHITE})
not_dirty(x) \triangleq x.dirty \mapsto 0
root(t,S) \triangleq \lambda(x,s,h). \exists n. s_n = \pi(t) \land S = \{n \mid \exists x. s(x) = (n,1) \land x \neq \text{aux}\}
edge(x,y) \triangleq \exists fd \in \{pt_1, \ldots, pt_n\}. (x,fd \mapsto y)
path_k(x,y) \triangleq \begin{cases} x = y & \text{if } k = 0 \\ \exists z. \text{edge}(x,z) \land \text{path}_{k-1}(z,y) & \text{if } k > 0 \end{cases}
reachable(t,x) \triangleq \exists y. \text{root}(t,S) \land y \in S \land \text{path}(y,x) \land x \neq 0
reachable(x) \triangleq \exists t \in [1..N]. \text{reachable}(t,x)
wfstate \triangleq \forall_{x \in [1..M]\cdot \text{obj}(x) \land (\forall x. \text{reachable}(x) \implies \text{not_blue}(x))}
white_edge(x,fd,y) \triangleq (x,fd \mapsto y) \land \text{white}(y) \land fd \in \{pt_1, \ldots, pt_n\}
white_edge(x,y) \triangleq \exists fd. \text{white_edge}(x,fd,y)
todirty(x,x) \triangleq \exists S. (x,\text{aux} = x \land \text{root}(t,S) \land n \in S)
instk(n,A) \triangleq \exists n'. A = n' : A' \land (n = n' \lor \text{instk}(n,A'))
stk_black(A) \triangleq \forall x. \text{instk}(x,A) \implies \text{black}(x)
reach_inv \triangleq \forall x,y. \text{reachable}(x) \land \text{black}(x) \land \text{white_edge}(x,y) \implies \text{dirty}(x) \lor \text{todirty}(x,y)
reach_stk(A) \triangleq \forall x,y. \text{reachable}(x) \land \text{black}(x) \land \text{white_edge}(x,y) \implies \text{dirty}(x) \lor \text{todirty}(x,y) \lor \text{instk}(x,A)
reach_termk(A,x_p,S_f,x_n) \triangleq \forall x,fd,y. \text{reachable}(x) \land \text{black}(x) \land \text{white_edge}(x,fd,y) \implies \text{dirty}(x) \lor \text{todirty}(x,y) \lor \text{instk}(x,A) \lor (x = x_p \land fd \in S_f) \lor (y = x_n)
reach_black \triangleq \forall x. \text{reachable}(x) \implies \text{black}(x)
ptfd_sta(x,fd,y) \triangleq \exists n. (x,fd \mapsto n) \land (y = n \lor \text{dirty}(x,y) \lor n = 0 \lor \text{todirty}(x,y))
newobj_sta(x) \triangleq \exists \text{obj}(x) \land \text{black}(x) \land \forall fd \in \{pt_1, \ldots, pt_n\}. \text{ptfd_sta}(x,fd,0)
rt_black(t) \triangleq \exists S. \text{root}(t,S) \land \forall n \in S. \text{black}(n)
rt_black \triangleq \forall t \in [1..N]. \text{rt_black}(t)
mark_rt(till(n)) \triangleq \forall t \in [1..n]. \text{rt_black}(t)
clear_color(till(n)) \triangleq \forall x \in [1..n]. (x.color \mapsto \text{BLACK} \implies \text{newobj_sta}(x))
clear_dirty(till(n)) \triangleq \forall x \in [1..n]. \text{not_dirty}(x)
reclaim(till(n)) \triangleq \forall x \in [1..n]. \text{not_white}(x)

NOTE: Here we use _ for an unspecified integer n that 0 \leq n \leq M.

Fig. 27. Useful Assertions for Verifying Boehm et al. GC

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 below we only give a sketch of the assertion language, the unary logic, the preconditions and the guarantee of the GC thread, the key invariants and the proof structure.

The unary program logic we use to verify the GC thread is a standard Rely-Guarantee logic adapted to the target language. The assertions are defined in Figure 24 and discussed before. We show the inference rules in Figure 26. Rules on the top half are for 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., \( x := \text{get\_root}(y) \)) and some particular heap manipulation rules adapted to our low-level machine model (e.g., \( \text{free}(x) \) 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 w.r.t. 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, i.e., \( \text{p}_{\text{gc}} \triangleq \text{wfstate} \). Just corresponding to the high-level WState definition (see Figure 22), the low-level \text{wfstate} predicate says that none of the reachable objects are BLUE, as follows:

\[
\text{wfstate} \triangleq \bigcap_{i \in [1..M]} \text{obj}(i) \land (\forall x. \text{reachable}(x) \Rightarrow \text{not\_blue}(x)),
\]

where \( \text{obj}(x) \) means \( x \) is a low-level heap location with the \( \text{pt}_{1}, \ldots, \text{pt}_{m} \), data, color and dirty fields, \( \text{reachable}(x) \) is defined similarly to the high-level definition in Figure 19, and \( \text{not\_blue}(x) \) is defined in Figure 27. It’s easy to see that any low-level initial state is well-formed. We define \( G_{\text{gc}} \) as follows:

\[
G_{\text{gc}} \triangleq \{ ((\pi \cup \{ \text{gc} \mapsto s \}), h), (\pi \cup \{ \text{gc} \mapsto s' \}, h') : \forall n. \\text{reachable}(n)(\pi, h) \implies [h(n)] = [h'(n)] \land h(n).\text{color} \neq \text{BLUE} \land h'(n).\text{color} \neq \text{BLUE} \}.
\]

The GC guarantees not modifying the mutator stores. For any mutator-reachable object, the GC does not update its fields coming from the high-level mutator, nor does it reclaim the object. Here \([\cdot]\) lifts a low-level object to a new one that contains mutator data only:

\[
[\cdot] \triangleq \{ \text{pt}_{1} \mapsto o(\text{pt}_{1}), \ldots, \text{pt}_{m} \mapsto o(\text{pt}_{m}), \text{data} \mapsto o(\text{data}) \}.
\]

We could prove that \( G_{\text{gc}} \) does not contain more behaviors than \( \text{AbsGCStep} \):

\[
G_{\text{gc}} \circ \alpha^{-1} \subseteq \alpha^{-1} \circ \text{AbsGCStep}.
\]

We present the proof of the top-level collection cycle in Figure 28. One of the key invariants used in the proofs is \( \text{reach\_jnv} \) (defined in Figure 27). It says, if a reachable \( \text{BLACK} \) object \( x \) points to a \( \text{WHITE} \) object \( y \), then either \( x \) is dirty or a mutator is going to dirty \( x \) (the predicate \( \text{old\_dirty}(x, y) \) holds). The latter occurs when the mutator thread \( t \) has done the first step of its write barrier \( \text{update}(x, fd, y) \). We have \( t.\text{aux} = x \) 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 post-conditions 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.

\[
R_{\text{gc}}; G_{\text{gc}} \vdash \{ \exists X. (j = Y) * (i.\text{pt}_{1} \leftarrow X) \}
\]

Here \( \text{ptfd}\_\text{sta}(i.\text{pt}_{1}, X) \) says either the \( \text{pt}_{1} \) field of \( i \) is \( X \), or \( i \) is (or is going to be) marked as dirty. Similarly, when reading the color of an object, the postcondition should take into account the mutators’ possible update of the color field in allocation and the updates of pointer fields after allocation.

\[
R_{\text{gc}}; G_{\text{gc}} \vdash \{ \exists X. (c = X) * (i.\text{color} \leftarrow Y) \}
\]

\[
\text{c} := i.\text{color};
\]

\[
\{ \exists X. (c = X) * (i.\text{color} \leftarrow Y) \land (X = Y \lor X = \text{BLUE} \land \text{newobj}\_\text{sta}(i)) \}
\]
Here newobj(sta(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_stk and newobj_stk are defined in Figure 27.

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.

Here as defined in Figure 27, reach_stk(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) ∨ 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 ∧ fd ∈ 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...
The GC first calls \texttt{inv} \texttt{stk} to \texttt{reach} \texttt{inv} \texttt{black} \texttt{stk} Concurrent mark-phase (\texttt{stk} \texttt{stk} \texttt{till} \texttt{inv} \texttt{black} \texttt{inv} \texttt{till} (2)) where \texttt{FInv} \texttt{A:46 Hongjin Liang et al.}

{ \texttt{wfstate}\}
\texttt{Initialize()} { \texttt{local i: [1..M], c: \{BLACK, WHITE, BLUE\}; i := 1; \texttt{Loop Invariant: \{ \texttt{wfstate \& clear_color_fill(i - 1) \& 1 \leq i \leq M + 1 \& own_op(c) \} \}}
while (i <= M) \{ ... \} // See Figure 15 for the full code
}

{ \texttt{wfstate \& reach_inv} } // using \texttt{Lemma 7.2}

\begin{verbatim}
{ (wfstate \& reach_inv) \& (own_op(mstk) \& mstk = \epsilon) }
\texttt{Trace()} { \texttt{local t: [1..N], rt: Set(Int), i: [0..M]; t := 1; \texttt{Loop Invariant: \{ \texttt{(wfstate \& reach_inv) \& (own_op(mstk) \& mstk = \epsilon) \} \}}
while (t <= N) { \{ ... \} // using Lemma 7.3
rt := get_root(t);
\texttt{Foreach Invariant: \{ FInv \}}
foreach i in rt do {
\{ FInv \& i \in rt \} // using \texttt{Lemma 7.3}
MarkAndPush(i);
\{ FInv \& i \in rt \} // using \texttt{Lemma 7.4}
}
}
\texttt{t := t + 1; \{ \exists X. (wfstate \& reach_stk(X) \& stk_black(X)) \& (own_op(mstk) \& mstk = X) \}
\{ \& (own_op(t) \& 1 \leq t \leq N + 1) \& own_op(rt) \& own_op(i) \}
\texttt{TraceStack();}
\{ \texttt{(wfstate \& reach_inv) \& (own_op(mstk) \& mstk = \epsilon) \& (own_op(t) \& 1 \leq t \leq N + 1) \}
\{ \& own_op(rt) \& own_op(i) \}
}
\}
\}
\{ (wfstate \& reach_inv) \& (own_op(mstk) \& mstk = \epsilon) \}
where \texttt{FInv} \triangleq \exists X. \texttt{(wfstate \& reach_stk(X) \& stk_black(X)) \& (own_op(mstk) \& mstk = X)} 
\& (own_op(t) \& 1 \leq t \leq N) \& (own_op(rt) \& \forall n \in rt, 0 \leq n \leq M) \& own_op(i)
\end{verbatim}

Fig. 29. Proof Outline of \texttt{Initialize()}

\begin{verbatim}
\begin{align*}
\{ (wfstate \& reach_inv) \& (own_op(mstk) \& mstk = \epsilon) \}
\end{align*}
\end{verbatim}

when the GC has cleared the colors of all the objects in the heap, as shown in the following lemma.

\textbf{Lemma 7.2.} \texttt{wfstate \& clear_color_fill(M) \implies reach_inv.}

That is, after initialization, if a \texttt{BLACK} reachable object \texttt{x} points to a \texttt{WHITE} object \texttt{y}, then \texttt{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 (\texttt{Trace()}, shown in Figure 30).
(a) The GC first calls \texttt{MarkAndPush(i)} to mark and push every root object. We need the following two lemmas to relate the unified pre- and post-conditions of \texttt{MarkAndPush(i)} in (7.5) and the actual pre- and post-conditions when calling the module.

\textbf{Lemma 7.3.} \texttt{reach_stk(X) \implies reach_tomk(X, 0, \emptyset, i)}.

\textbf{Lemma 7.4.} \texttt{reach_tomk(X, 0, \emptyset, 0) \implies reach_stk(X)}.

Then by the CONSEQ rule, we can reuse the proof of \texttt{MarkAndPush(i)}.
\{ \exists X. (\text{wfstate} \land \text{reach\_stk}(X) \land \text{stk\_black}(X)) \land (\text{own}\_stk(\text{mstk}) \land \text{mstk} = X) \}\}

```c
TraceStack() {
    local i: [1..M], j: [0..M];
    \textbf{Loop Invariant:} \{ \exists X. (\text{wfstate} \land \text{reach\_stk}(X) \land \text{stk\_black}(X)) \land (\text{own}\_stk(\text{mstk}) \land \text{mstk} = X) \land \text{own}_p(i) \land \text{own}_p(j) \}\n    while (!\text{is\_empty}(\text{mstk})) {
        i := \text{pop}(\text{mstk});
        \{ \exists X'. (\text{wfstate} \land \text{reach\_stk}(i :: X') \land \text{stk\_black}(X') \land \text{obj}(i)) \land (\text{own}\_stk(\text{mstk}) \land \text{mstk} = X') \land \text{own}_p(i) \}\n        j := i.pt1;
        \{ \exists X'. (\text{wfstate} \land \text{reach\_stk}(i :: X') \land \text{stk\_black}(X') \land \text{obj}(i) \land \text{ptfd}_{\text{sta}}(i.pt1, j) \land (j = 0 \lor \text{obj}(j))) \land (\text{own}\_stk(\text{mstk}) \land \text{mstk} = X') \land \text{own}_p(i) \land \text{own}_p(j) \}\n        \{ \exists X'. (\text{wfstate} \land \text{reach\_tomk}(X', i, \{pt_2, \ldots, pt_m\}, j) \land \text{stk\_black}(X') \land (j = 0 \lor \text{not\_white}(j))) \land (\text{own}\_stk(\text{mstk}) \land \text{mstk} = X') \land 1 \leq i \leq M \land \text{own}_p(i) \land \text{own}_p(j) \land \text{own}_p(j) \}\n        MarkAndPush(j);
        \{ \exists X'. (\text{wfstate} \land \text{reach\_tomk}(X', i, \{pt_2, \ldots, pt_m\}, 0) \land \text{stk\_black}(X') \land (j = 0 \lor \text{not\_white}(j))) \land 1 \leq i \leq M \land (\text{own}\_stk(\text{mstk}) \land \text{mstk} = X') \land \text{own}_p(i) \land \text{own}_p(j) \land \text{own}_p(j) \}\n        \ldots
    }
    j := i.ptm; MarkAndPush(j);
    \{ \exists X'. (\text{wfstate} \land \text{reach\_tomk}(X', i, 0, 0) \land \text{stk\_black}(X') \land (j = 0 \lor \text{not\_white}(j))) \land (\text{own}\_stk(\text{mstk}) \land \text{mstk} = X') \land \text{own}_p(i) \land \text{own}_p(j) \land \text{own}_p(j) \}\n    \{ \text{MarkAndPush}(j); \}
}
```

\{(\text{wfstate} \land \text{reach\_inv}) \land (\text{own}\_stk(\text{mstk}) \land \text{mstk} = \epsilon)\}

(b) Then the GC calls the module \texttt{TraceStack()} (Figure 31) to perform the depth-first traversal. The loop invariant \texttt{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, \(i.pt_1\) might not equal \(j\) since mutators could update this field. We only know that \(\text{ptfd}_{\text{sta}}(i.pt_1, j)\) holds. Then, \(x\) might be either \(j\), or pointed to by \(i\) via fields \(pt_2, \ldots, pt_m\). Thus we get \(\text{reach\_tomk}(\text{mstk}, i, \{pt_2, \ldots, pt_m\}, j)\) holds. Formally, the following lemma holds:

\textbf{LEMMA 7.5.}

\begin{enumerate}
    \item \texttt{reach\_stk}(i :: X) \iff \texttt{reach\_tomk}(X, i, \{pt_1, \ldots, pt_m\}, 0);
    \item \texttt{reach\_tomk}(X, i, S_f, 0) \implies \texttt{reach\_tomk}(X, i, S_f, j);
    \item \texttt{reach\_tomk}(X, i, S_f, j) \land \texttt{ptfd}_{\text{sta}}(i, fd, j) \land fd \in S_f 
    \implies \texttt{reach\_tomk}(X, i, S_f \setminus \{fd\}, j).
\end{enumerate}

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

iv. Mark and push other children. The proof is similar to the above two steps, so we omit the discussions. Finally, \texttt{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).

After tracing, we can ensure \texttt{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 rule frame. We can conclude reach_inv is maintained at the end of this phase.

(4) Stop-the-world card-cleaning.
   (a) The GC first re-scans the roots (ScanRoot(), shown in Figure 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 post-conditions.
   (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 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 $g_{gc}$. After sweep, the state is still well-formed.
\[
\exists X. (\text{wfstate} \land \text{reach stk}(X) \land \text{stk black}(X) \land \text{rt black} \land \text{clear dirty till}(i - 1)) \land 1 \leq i \leq M + 1 \land (\text{own np}(\text{mstk}) \land \text{mstk} = X) \land \text{own np}(c) \land \text{own np}(d)
\]

\[
\text{Loop Invariant:
\{ \exists X. (\text{wfstate} \land \text{reach stk}(X) \land \text{stk black}(X) \land \text{rt black} \land \text{clear dirty till}(i - 1)) \land 1 \leq i \leq M + 1 \land (\text{own np}(\text{mstk}) \land \text{mstk} = X) \land \text{own np}(c) \land \text{own np}(d) \}
\]

\[
\text{while (i <= M) { ... } // See Figure 15 for the full code}
\]

\[
\{ \exists X. (\text{wfstate} \land \text{reach stk}(X) \land \text{stk black}(X) \land \text{rt black} \land \text{clear dirty till}(M)) \}
\]

\[
\{ (\text{wfstate} \land \text{reach inv} \land \text{rt black} \land \text{clear dirty till}(M)) \}
\]

\[
\{ (\text{wfstate} \land \text{reach black} \land \text{reclaim till}(i - 1) \land 1 \leq i \leq M + 1) \land \text{own np}(c) \}
\]

\[
\text{while (i <= M) { ... } // See Figure 15 for the full code}
\]

\[
\{ (\text{wfstate} \land \text{reach black} \land \text{reclaim till}(M)) \}
\]

Fig. 34. Verification of CleanCard() in an Atomic Block

\[
\{ \text{wfstate} \land \text{reach black} \}
\]

\[
\text{Sweep()}
\]

\[
\{ \text{wfstate} \land \text{reach black} \land \text{reclaim till}(M) \}
\]

Fig. 35. Proof Outline of Sweep()

8. RELATED WORK AND CONCLUSION

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

Verifying compilation and optimizations of concurrent programs. Compiler verification for concurrent programming languages can date back to work in [Wand 1995; Gladstein and Wand 1996], which is about functional languages using message-passing mechanisms. Recently, Lochbihler [2010] presents a verified compiler for Java threads and prove semantics preservation by a weak bisimulation. He views every heap update as an observable move, thus does not allow the target and the source to have different granularities of atomic updates. To achieve parallel compositional-ity, he requires the relation to be preserved by any transitions of shared states, i.e., the environments are assumed arbitrary. As we explained in Section 2.2, this is a too strong requirement in general for many transformations, including the examples in this paper.

Burckhardt et al. [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.

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
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 meta-theory (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 paper; 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


Rely-Guarantee-Based Simulation


Received; revised; accepted