# Reasoning about Optimistic Concurrency Using a Program Logic for History

Ming Fu[1], Yong Li[1], Xinyu Feng[1,2], Zhong Shao[3], and Yu Zhang[1]

[1] University of Science and Technology of China
[2] Toyota Technological Institute at Chicago
[3] Yale University

**Abstract.** Optimistic concurrency algorithms provide good performance for parallel programs but they are extremely hard to reason about. Program logics such as concurrent separation logic and rely-guarantee reasoning can be used to verify these algorithms, but they make heavy uses of history variables which may obscure the high-level intuition underlying the design of these algorithms. In this paper, we propose a novel program logic that uses invariants on history traces to reason about optimistic concurrency algorithms. We use past tense temporal operators in our assertions to specify execution histories. Our logic supports modular program specifications with history information by providing separation over both space (program states) and time. We verify Michael's non-blocking stack algorithm and show that the intuition behind such algorithm can be naturally captured using trace invariants.

## 1 Introduction

Optimistic concurrency algorithms [6, 7] allow concurrent access to shared data and ensure data consistency by performing dynamic conflict detection. These algorithms can be more efficient than coarse-grained lock-based synchronization if there is sufficient data independence. However, the design of the algorithms has to consider many more thread-interleaving scenarios than coarse-grained synchronization. The algorithms are usually complex and error-prone. Their correctness is usually far from obvious and is hard to verify too.

As an example, Fig. 1 shows a non-blocking stack algorithm, where a stack is implemented as a linked list pointed by the `Top` pointer. It allows simultaneous read (line 4 and 13) and write (7, 15) of `Top`, and the conflict detection is done by the CAS (compare-and-swap) command. This algorithm has two subtle bugs. One is that `t` might be a dangling pointer when the dereference occurs in line `6`. The other is the notorious ABA problem: suppose the top three nodes on the stack are A, B and C; Thread 1 calls `pop` and reaches the end of line `6`; so `t` points to A and `next` points to B; then Thread 2 comes, pops A and B, and pushes A onto the stack; Thread 1 continues to execute line `7`, where the comparison succeeds and `Top` is set to point to B, which is no longer on the stack.

Here, we have to refer to the historical events to explain the problems above. It is not surprising that temporal reasoning is needed to argue for the correctness of such highly concurrent algorithms.

```
pop(){
01  local done, next, t;
02  done := false;
03  while (!done){
04    t := Top;
05    if (t == null) return null;
06    next := t.Next;
07    done := CAS(&Top, t, next);
08  }
09  return t;
}
```

```
push(x){
10 local done, t;
11  done := false;
12  while (!done){
13    t := Top;
14    x.Next := t;
15    done := CAS(&Top, t, x);
16  }
17  return true;
}
```

**Fig. 1.** A Buggy Implementation of Non-Blocking Stacks

Concurrent separation logic (CSL [13]) and rely-guarantee (R-G) based reasoning [8] are two well-studied approaches to concurrency verification. Previous work [14, 18] has shown that they can be used to verify fine-grained and optimistic algorithms. However, since assertions in these logics only specify program states (or state transitions in R-G reasoning), it is difficult to use them to express directly the temporal properties about the subtle interaction between threads. Instead, we have to introduce history variables to record the occurrence of certain events. This indirect approach to specifying historical events makes specifications and proofs complex, and in many cases fails to demonstrate the high-level intuition behind the design of the algorithms.

In this paper, we propose a new program logic that uses invariants on historical execution traces to reason about optimistic concurrency algorithms. The logic extends previous work on R-G reasoning by introducing past tense temporal operators in the assertion language. It allows us to specify historical events directly without using history variables, which makes the verification process more modular and intuitive.

Although it has also been observed before that past tense operators in temporal logic can be used to eliminate the need of history variables [10], developing a modular logic with temporal reasoning that is able to verify modern concurrent algorithms has so far been an open problem. Our logic inherits previous work on combining R-G reasoning with separation logic [3, 19, 2] to support modular verification. Separating conjunction in separation logic is now defined over assertions on execution histories instead of state assertions. The frame rule and the hide rule in the Local Rely-Guarantee (LRG) logic [2]—the keys for modular verification—are supported naturally in this new setting.

We apply our new logic to reason about Michael's non-blocking stack algorithm [11] which uses hazard pointers to fix the buggy algorithm in Fig. 1. We use trace invariants to capture the main intuition underlying the algorithm. The program specifications and proofs used in our logic are more intuitive than those from previous work [14]. They do not require history variables. Our logic

$$
\begin{array}{lll}
\text{(Expr)} & E ::= x \mid n \mid E \texttt{+} E \mid E \texttt{-} E \mid \ \ldots & \\
\text{(Bexp)} & B ::= \mathsf{true} \mid \mathsf{false} \mid E \texttt{=} E \mid E \neq E \mid \ \ldots & \\
\text{(Stmts)} & C ::= x := E \mid x := \texttt{[}E\texttt{]} \mid \texttt{[}E\texttt{]} := E' \mid \mathbf{skip} \mid x := \mathbf{cons}(E, \ldots, E) & \\
 & \quad \mid \mathbf{dispose}(E) \mid \mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ B\ \mathbf{do}\ C \mid \langle C \rangle \mid C;C & \\
\text{(Prog)} & W ::= \ \mathsf{t}_1.C_1 \parallel \ldots \parallel \mathsf{t}_n.C_n & \text{(ThrdID)} \quad \mathsf{t} \ \in \ \mathsf{Nat}
\end{array}
$$

**Fig. 2.** A Concurrent Programming Language

$$
\begin{array}{lll}
\text{(Store)} & s \ \in \ \mathsf{PVar} \rightharpoonup_{\mathrm{fin}} \mathsf{Int} & \text{(Heap)} \quad h \ \in \ \mathsf{Nat} \rightharpoonup_{\mathrm{fin}} \mathsf{Int} \\
\text{(State)} & \sigma \ \in \ \mathsf{Store} \times \mathsf{Heap} & \\
\text{(Trace)} & \mathcal{T} \ ::= \ (\sigma_0, \mathsf{t}_0) :: (\sigma_1, \mathsf{t}_1) :: \cdots :: (\sigma_n, \mathsf{t}_n) & \text{(Trans)} \quad \mathcal{R}, \mathcal{G} \ \in \ \mathcal{P}(\mathsf{Trace})
\end{array}
$$

**Fig. 3.** Program States and Execution Traces

$$
\frac{\mathcal{T}.last = (\sigma, \_) \quad (C, \sigma) \longrightarrow (C', \sigma')}{(\mathsf{t}.C, \mathcal{T}) \hookrightarrow (\mathsf{t}.C', \mathcal{T} :: (\sigma', \mathsf{t}))} \qquad \frac{\mathcal{T}.last = (\sigma, \_) \quad (C, \sigma) \longrightarrow \mathbf{abort}}{(\mathsf{t}.C, \mathcal{T}) \hookrightarrow \mathbf{abort}}
$$

$$
\frac{(\mathsf{t}_i.C_i, \mathcal{T}) \hookrightarrow (\mathsf{t}_i.C_i{}', \mathcal{T}')}{(\mathsf{t}_1.C_1 \parallel \ldots \mathsf{t}_i.C_i \ldots \parallel \mathsf{t}_n.C_n, \mathcal{T}) \overset{\mathcal{R}}{\longmapsto} (\mathsf{t}_1.C_1 \parallel \ldots \mathsf{t}_i.C_i{}' \ldots \parallel \mathsf{t}_n.C_n, \mathcal{T}')}
$$

$$
\frac{(\mathsf{t}_i.C_i, \mathcal{T}) \hookrightarrow \mathbf{abort}}{(\mathsf{t}_1.C_1 \parallel \ldots \mathsf{t}_i.C_i \ldots \parallel \mathsf{t}_n.C_n, \mathcal{T}) \overset{\mathcal{R}}{\longmapsto} \mathbf{abort}} \qquad \frac{(\mathcal{T} :: (\sigma, \mathsf{t})) \in \mathcal{R}}{(W, \mathcal{T}) \overset{\mathcal{R}}{\longmapsto} (W, \mathcal{T} :: (\sigma, \mathsf{t}))}
$$

**Fig. 4.** Selected Rules of Operational Semantics

also supports a new frame rule that further simplifies the proofs (e.g., for the `retireNode` function in Fig. 9) and makes the verification more modular.

## 2 A Concurrent Programming Language

Figure 2 shows a simple concurrent language. The statements $x := \texttt{[}E\texttt{]}$ and $\texttt{[}E\texttt{]} := E$ are memory-load and store operations respectively; **cons** allocates fresh memory cells, and **dispose** frees a cell. The atomic block $\langle C \rangle$ executes $C$ atomically. Other statements have standard meanings. A program $W$ contains $n$ parallel threads, each marked with a unique thread ID (e.g., $\mathsf{t}_i$ for $C_i$).

Figure 3 defines program states and execution traces. The store $s$ is a finite partial mapping from program variables to integers; the heap $h$ maps memory locations (natural numbers) to integers. A program state $\sigma$ is a pair $(s, h)$. An execution trace $\mathcal{T}$ is a (nonempty) finite sequence $(\sigma_0, \mathsf{t}_0) :: (\sigma_1, \mathsf{t}_1) :: \cdots :: (\sigma_n, \mathsf{t}_n)$. A pair $(\sigma_i, \mathsf{t}_i)$ in a trace $\mathcal{T}$ means that a thread with ID $\mathsf{t}_i$ reached the state $\sigma_i$ after executing one step from the state $\sigma_{i-1}$. Thread ID $\mathsf{t}_0$ can be any

$$\begin{array}{ll} \text{(StateAssert)} & P, Q ::= B \mid \mathsf{emp}_h \mid \mathsf{emp}_s \mid \mathsf{own}(x) \mid E \mapsto E \mid P * Q \mid \ldots \\ \text{(TraceAssert)} & p, q, R, G, I ::= P \mid \mathsf{Id} \mid [p]_{\mathtt{t}} \mid p \rhd q \mid \ominus p \mid p * q \mid \exists X.p \mid \neg p \mid p \vee q \mid \ldots \end{array}$$

**Fig. 5.** The Assertion Language

$$\begin{array}{lll} (s,h) \models_{\text{SL}} \mathsf{emp}_s & \text{iff} & s = \emptyset \qquad\qquad (s,h) \models_{\text{SL}} \mathsf{emp}_h \quad \text{iff} \quad h = \emptyset \\ (s,h) \models_{\text{SL}} \mathsf{own}(x) & \text{iff} & \mathsf{dom}(s) = \{x\} \\ (s,h) \models_{\text{SL}} E_1 \mapsto E_2 & \text{iff} & \text{there exist } \ell \text{ and } n \text{ such that} \\ & & \qquad [\![E_1]\!]_s = \ell, [\![E_2]\!]_s = n, \mathsf{dom}(h) = \{\ell\} \text{ and } h(\ell) = n \\ \sigma \models_{\text{SL}} P * Q & \text{iff} & \text{there exist } \sigma_1 \text{ and } \sigma_2 \\ & & \qquad \text{such that } \sigma_1 \uplus \sigma_2 = \sigma, \sigma_1 \models_{\text{SL}} P \text{ and } \sigma_2 \models_{\text{SL}} Q \\ \mathsf{emp} & \stackrel{\text{def}}{=} & \mathsf{emp}_s \wedge \mathsf{emp}_h \end{array}$$

**Fig. 6.** Semantics of Selected Separation Logic Assertions

value. We use $\mathcal{T}.last$ to denote the last element in $\mathcal{T}$. A trace $(\mathcal{T} :: (\sigma, \mathtt{t}))$ in the trace sets $\mathcal{R}$ and $\mathcal{G}$ is also used to model a single-step transition by the thread $\mathtt{t}$ that starts from $\mathcal{T}$ and reaches a new state $\sigma$.

Figure 4 gives selected rules modeling the operational semantics (see the technical report [4] for the complete rules). The binary relation $\_ \longrightarrow \_$ models a transition over states made by a primitive statement. The definition is standard and is omitted here. The relation $\_ \hookrightarrow \_$ lifts state transitions to transitions over traces. The thread ID of the executing thread $\mathtt{t}$ is recorded on the trace at the end of the transition. Finally, the relation $\_ \overset{\mathcal{R}}{\longmapsto} \_$ models the transitions over traces made by programs in an environment characterized by $\mathcal{R}$. Here $\mathcal{R}$ contains all the possible transitions of the environment.

## 3 The Assertion Language

Our assertion language is defined in Fig. 5. We use separation logic assertions ($P$ and $Q$) to specify program states. Following Parkinson et al. [15], we also treat program variables as resources. Semantics of some separation logic assertions are shown in Fig. 6. We use $\sigma_1 \uplus \sigma_2$ to represent the union of the two disjoint states.

*Trace assertions.* Trace assertions $p$, $q$, $R$, $G$ and $I$ specify historical execution traces. Semantics of trace assertions are defined in Fig. 7. Here we use $|\mathcal{T}|$ to represent the length of $\mathcal{T}$, and use $\mathcal{T}_{k-}$ to represent the subsequence resulting from truncating the last $k$ elements from $\mathcal{T}$ $(0 \leq k < |\mathcal{T}|)$.

A state assertion $P$ is viewed as a trace assertion that specifies only the last state. Assertion $\mathsf{Id}$ says that the last two states on the trace are the same (i.e. the last state transition is an identity transition). Assertion $[p]_{\mathtt{t}}$ means that the

$$\mathcal{T}_{k-} \quad \stackrel{\text{def}}{=} \quad (\sigma_0, \mathsf{t}_0) :: \cdots :: (\sigma_{n-k}, \mathsf{t}_{n-k}) \quad \text{if } \mathcal{T} = (\sigma_0, \mathsf{t}_0) :: \cdots :: (\sigma_n, \mathsf{t}_n) \text{ and } 0 \le k \le n$$

$$\mathcal{T} \models P \quad \text{iff} \quad \text{there exists } \sigma \text{ such that } \mathcal{T}.last = (\sigma, \_) \text{ and } \sigma \models_{\text{SL}} P$$

$$\mathcal{T} \models \mathsf{Id} \quad \text{iff} \quad \text{there exist } \mathcal{T}' \text{ and } \sigma \text{ such that } \mathcal{T} = \mathcal{T}' :: (\sigma, \_) :: (\sigma, \_)$$

$$\mathcal{T} \models [p]_{\mathsf{t}} \quad \text{iff} \quad \mathcal{T}.last = (\_, \mathsf{t}) \text{ and } \mathcal{T} \models p$$

$$\mathcal{T} \models p \rhd q \quad \text{iff} \quad \text{there exists } 0 < i < |\mathcal{T}| \text{ such that } \mathcal{T}_{i-} \models p \text{ and } \forall j < i.\, \mathcal{T}_{j-} \models q$$

$$\mathcal{T} \models \ominus p \quad \text{iff} \quad 1 < |\mathcal{T}| \text{ and } \mathcal{T}_{1-} \models p \qquad\qquad [\![p]\!] \stackrel{\text{def}}{=} \{\mathcal{T} \mid \mathcal{T} \models p\}$$

$$((\sigma_0, \mathsf{t}_0) :: \ldots :: (\sigma_n, \mathsf{t}_n)) \oplus ((\sigma'_0, \mathsf{t}'_0) :: \ldots :: (\sigma'_m, \mathsf{t}'_m))$$
$$\stackrel{\text{def}}{=} \begin{cases} ((\sigma_0 \uplus \sigma'_0, \mathsf{t}_0) :: \ldots :: (\sigma_n \uplus \sigma'_m, \mathsf{t}_n)) & \text{if } n = m \wedge \forall 0 \le i \le n.\mathsf{t}_i = \mathsf{t}'_i \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{T} \models p * q \quad \text{iff} \quad \text{there exist } \mathcal{T}_1 \text{ and } \mathcal{T}_2 \text{ such that } \mathcal{T} = \mathcal{T}_1 \oplus \mathcal{T}_2, \mathcal{T}_1 \models p \text{ and } \mathcal{T}_2 \models q$$

$$p \trianglerighteq q \quad \stackrel{\text{def}}{=} \quad (p \rhd q) \vee p \qquad\qquad \diamondsuit\!\!\!-\, p \quad \stackrel{\text{def}}{=} \quad p \trianglerighteq \mathsf{true} \qquad\qquad \boxminus p \quad \stackrel{\text{def}}{=} \quad \neg \diamondsuit\!\!\!-\, (\neg p)$$

$$p \blacktriangleright q \quad \stackrel{\text{def}}{=} \quad \diamondsuit\!\!\!-\, (\diamondsuit\!\!\!-\, p \wedge q) \qquad\qquad p \ltimes_{\mathsf{t}} q \quad \stackrel{\text{def}}{=} \quad \ominus p \wedge [q]_{\mathsf{t}} \qquad\qquad p \ltimes q \quad \stackrel{\text{def}}{=} \quad \exists \mathsf{t}.\, p \ltimes_{\mathsf{t}} q$$

We assume unary operators ($\diamondsuit\!\!\!-$ and $\ominus$) have higher precedence than other operators.

**Fig. 7.** Semantics of Trace Assertions

trace satisfies $p$ and the last state transition is made by the thread $\mathsf{t}$. Assertion $p \rhd q$ holds over $\mathcal{T}$ if and only if $p$ holds over the trace $\mathcal{T}_{i-}$ for some $i$ and $q$ holds ever since. It is also represented as $q \mathcal{S} p$ ($q$ since $p$) in the literature of temporal logic [10]. Assertion $\ominus p$ holds if and only if the trace prior to the last transition satisfies $p$. $[\![p]\!]$ is the set of traces that satisfy $p$.

Assertion $p * q$ lifts separating conjunction to traces; it specifies a program trace consisting of two *disjoint* parts: one satisfies $p$ and another $q$. Traces $\mathcal{T}_1$ and $\mathcal{T}_2$ are disjoint if they have the same length, and for each $i$ such that $0 \le i < |\mathcal{T}|$ the states in $\mathcal{T}_1[i]$ and $\mathcal{T}_2[i]$ are disjoint (see the definition of $\mathcal{T}_1 \oplus \mathcal{T}_2$ in Fig. 7).

Other useful connectors can be defined using these primitive operators. Assertion $p \trianglerighteq q$ is a weaker version of $p \rhd q$. Assertion $\diamondsuit\!\!\!-\, p$ says that $p$ was once true in the history. Assertion $\boxminus p$ holds if and only if $p$ holds at every step in the history. Assertion $p \blacktriangleright q$ says that $p$ first came true in the history, and then $q$ came true later. Assertion $p \ltimes_{\mathsf{t}} q$ means that the last transition is made by thread $\mathsf{t}$, and assertion $p$ holds prior to the transition, and $q$ holds after it. This allows us to define the built-in $\ltimes$ operator in LRG [2].

*Example 3.1.* In the TL2 transactional memory protocol [1], before updating a shared memory cell, we must first acquire the corresponding lock and then increase the global version clock. This requirement (among many others in the protocol) can be defined as the following guarantee:

$$G_{\mathtt{tid}}(\mathtt{x}) \quad \stackrel{\text{def}}{=} \quad \exists D, D', T, T'.\, \left( \begin{array}{c} ((\mathtt{x} \mapsto 0, D * \mathtt{gt} \mapsto T) \rhd (\mathtt{x} \mapsto \mathtt{tid}, D * \mathtt{gt} \mapsto \_)) \\ \wedge (\mathtt{x} \mapsto \mathtt{tid}, D * \mathtt{gt} \mapsto T') \wedge (T' > T) \end{array} \right) \\ \ltimes_{\mathtt{tid}} (\mathtt{x} \mapsto \mathtt{tid}, D' * \mathtt{gt} \mapsto T')$$

Here x points to two fields, its lock and its value. The first line above says that, before the transition, the lock was acquired (it was changed from 0 to tid) when the global version clock gt was $T$. Then the lock and the value have been preserved ever since, but gt might have been changed. The second line says gt is greater than $T$ right before the transition. The third line says the value of x is updated by the transition. This definition also implies that the increment of gt is done after the lock is acquired.

The guarantee above refers to two events before the specified transition. In traditional R-G reasoning, the guarantee condition can only specify two states, so we have to introduce history variables to describe such historical events.

As in separation logic, a class of trace assertions that are of special interest to us are those that are precise about the last state on the trace.

**Definition 3.2 (Last-State-Precise Trace Assertions).** *$p$ is last state precise,* i.e. $\mathsf{LPrec}(p)$ *holds, if and only if for all $\mathcal{T}$, $\mathtt{t}$, $s$, $h$, $s_1$, $s_2$, $h_1$, $h_2$, if $s_1 \subseteq s$, $s_2 \subseteq s$, $h_1 \subseteq h$, $h_2 \subseteq h$, $\mathcal{T} :: ((s_1, h_1), \mathtt{t}) \models p$ and $\mathcal{T} :: ((s_2, h_2), \mathtt{t}) \models p$, then $s_1 = s_2$ and $h_1 = h_2$.*

The example below shows a last-state-precise assertion $p$ can specify a precise state domain that is determined dynamically by historical events. It is more powerful than a precise state assertion $P$ in separation logic [17]. This can also be seen in our hazard-pointer-based stack example.

*Example 3.3.* Let $I = \exists X. \ominus(\ell \mapsto X * \mathsf{true}) \wedge (\ell \mapsto \_ * (X = 0 \wedge r \vee X \neq 0 \wedge \mathsf{emp}))$ where $r = x \mapsto \_ * y \mapsto \_$, then $I$ is a last-state-precise trace assertion. It specifies traces where the domain of the last state depends on the historical value $X$ of $\ell$.

## 4 A Program Logic for History

Now we present our program logic for history, named HLRG, which extends the LRG logic [2] with trace assertions for reasoning about historical traces.

As in LRG, we use the judgments $R; G \vdash \{p\} W \{q\}$ and $R; G; I \vdash_{\mathtt{t}} \{p\} C \{q\}$ for well-formed programs and well-formed thread t respectively. The rely condition $R$ and the guarantee $G$ specify the interference between the environment and the thread. The judgments say informally that starting from a trace satisfying both $\boxminus(R \vee G) * \mathsf{true}$ and $p$, if the environment's transitions satisfy $R$, then $W$ (or $C$) would not abort, its transitions satisfy $G$, and $q$ holds at the end if $W$ (or $C$) terminates. The invariant $I$ specifies the well-formedness of the shared resource. Unlike in the LRG logic, $R$, $G$, $I$, $p$ and $q$ are all trace assertions now.

Figure 8 gives the main inference rules. The PROG rule allows us to verify the whole program by verifying the $n$ parallel threads $\mathtt{t}_1.C_1$,$\mathtt{t}_2.C_2$,...,$\mathtt{t}_n.C_n$ separately. Each thread $\mathtt{t}_i$ has exclusive access to its own private resource specified by $p_i$ and $q_i$. All threads can access the shared resource specified by $r$, $r_1$ ... $r_n$. To verify each thread, we need to find an invariant $I$ specifying the basic well-formedness of the shared resource.

$$\frac{\begin{array}{c} R\vee G_1\vee\ldots\vee G_{i-1}\vee G_{i+1}\vee\ldots\vee G_n; G_i; I\vdash_{\mathsf{t}_i}\{p_i * r\}C_i\{q_i * r_i\} \quad \forall i\in\{1,\ldots,n\} \\ r\vee r_1\vee\ldots\vee r_n \Rightarrow I \end{array}}{R; G_1\vee\ldots\vee G_n\vdash\{p_1 *\ldots * p_n * r\}\mathsf{t}_1.C_1\parallel\ldots\parallel\mathsf{t}_n.C_n\{q_1 *\ldots * q_n * (r_1\wedge\ldots\wedge r_n)\}}\ (\textsc{prog})$$

$$\frac{p\Rightarrow P \quad \{P\}C\{Q\} \quad p\ltimes_{\mathsf{t}} Q\Rightarrow G * \mathsf{true}}{\mathsf{Id}_I; G; I\vdash_{\mathsf{t}}\{p\}\,\langle C\rangle\,\{Q\}}\ (\textsc{atom})$$

where $\mathsf{Id}_I$ is defined as $\mathsf{Id}\wedge(I\ltimes I)$.

$$\frac{p\Rightarrow p' \quad \mathsf{Id}_I; G; I\vdash_{\mathsf{t}}\{p'\}\langle C\rangle\{Q'\} \quad \ominus p\wedge Q'\Rightarrow q \quad \mathtt{Sta}(\{p,q\}, R{*}\mathsf{Id})}{R; G; I\vdash_{\mathsf{t}}\{p\}\,\langle C\rangle\,\{q\}}\ (\textsc{atom-r})$$

$$\frac{\begin{array}{c} R; G; I\vdash_{\mathsf{t}}\{p\}C\{q\} \\ \mathtt{Sta}(r, R'{*}\mathsf{Id}) \quad r\Rightarrow I'*\mathsf{true} \end{array}}{R{*}R'; G{*}G'; I{*}I'\vdash_{\mathsf{t}}\{p{*}r\}C\{q{*}r\}}\ (\textsc{frame}) \quad \frac{R; G; I\vdash_{\mathsf{t}}\{p\}C\{q\}}{R; G; I\vdash_{\mathsf{t}}\{p\wedge\diamondsuit r\}C\{q\wedge\diamondsuit r\}}\ (\textsc{frame}\mathrm{T})$$

$$\frac{R; G; I\vdash_{\mathsf{t}}\{p\wedge(I'{*}\mathsf{true})\}C\{q\} \quad \boxminus(R\vee G)\Rightarrow(I'\wedge I'')}{R; G; I\vdash_{\mathsf{t}}\{p\}C\{q\wedge(I''{*}\mathsf{true})\}}\ (\textsc{inv})$$

**Fig. 8.** Selected Inference Rules of the HLRG Program Logic

The ATOM rule says that we can treat $C$ in the atomic block as sequential code since its execution cannot be interrupted. Here the judgment $\{P\}C\{Q\}$ can be derived following the standard sequential separation logic rules [17], which we do not show here. This rule allows us to strengthen $P$ into a trace assertion $p$ so that we can carry the historical information. The transition from $p$ to $Q$ needs to satisfy the guarantee $G$, which may have some constraints over the history traces (examples about $G$ can be found in Fig. 10 in Sec. 5).

The ATOM rule uses a strong rely condition about the environment, which is an identity transition preserving the invariant $I$ of the shared resource. To relax it, we can apply the next ATOM-R rule borrowed from RGSep [18]. It allows us to adjust the pre- and post-conditions so that they are both *stable* with respect to the rely condition $R$.

**Definition 4.1 (Stability).** *We say a trace assertion $p$ is stable with respect to a trace assertion $q$,* i.e. $\mathtt{Sta}(p, q)$ *holds, if and only if $\ominus p\wedge q\Rightarrow p$.*

That is, if $p$ holds before the most recent transition, and the transition satisfies $q$, then $p$ holds after it. This is the standard requirement in R-G reasoning. With temporal operators, it can now be encoded as a temporal assertion. We use $\mathtt{Sta}(\{p, q\}, R)$ as a shorthand for $\mathtt{Sta}(p, R)\wedge\mathtt{Sta}(q, R)$.

The interesting (and *new*) part of this ATOM-R rule is the post condition $q$, which is weakened from the trace assertion $\ominus p\wedge Q'$. This allows us to carry information about historical events happened before this atomic transition.

The FRAME rule comes from LRG. It supports local reasoning and allows us to write "small" specifications about resources that are indeed accessed in

$C$. Invariants about other resources are preserved and can be added into the specifications later. We also introduce a new FRAMET rule to show the frame property over "time". Since historical traces would not affect the execution of programs, knowledge about history can be added when necessary.

The new INV rule is also very useful. It is like the reverse of the standard consequence rule in Hoare logic, since it allows us to strengthen the pre-condition, and prove a post-condition weaker than we wanted. This rule is sound because the invariants $I'$ and $I''$ can be derived from the fact that each step of the transition satisfies $R \vee G$, so that they can be used anywhere in the proof for free. We will demonstrate the use of FRAME, FRAMET and INV in our example in Sec. 5.

The rest of the rules are the same as those in LRG, and are not shown here. Note that in each rule we implicitly require the following properties hold.

- $\mathsf{fence}(I, R)$ and $\mathsf{fence}(I, G)$;
- $p \vee q \Rightarrow I * \mathsf{true}$;

where $\mathsf{fence}(I, p)$ is defined below:

$$\mathsf{fence}(I, p) \stackrel{\text{def}}{=} (\mathsf{Id} \wedge (I \ltimes I) \Rightarrow p) \wedge (p \Rightarrow I \ltimes I) \wedge \mathsf{LPrec}(I)\,.$$

Informally, it requires that the most recent transition is confined in a precise domain enforced by the last-state-precise assertion $I$. This constraint is inherited from LRG. Interested readers can refer to our previous work [2] to see the technical discussions about this requirement.

*Semantics and soundness.* The semantics of our logic and its soundness proof are similar to those of LRG. We first define the non-interference property below.

**Definition 4.2 (Non-Interference).** *Let* $W = \mathsf{t}_1.C_1 \parallel \ldots \parallel \mathsf{t}_n.C_n$. $(W, \mathcal{T}, \mathcal{R}) \Longrightarrow^0 \mathcal{G}$ *always holds.* $(W, \mathcal{T}, \mathcal{R}) \Longrightarrow^{m+1} \mathcal{G}$ *holds iff* $\neg(W, \mathcal{T}) \overset{\mathcal{R}}{\longmapsto}$ **abort** *and the following are true:*

1. *for all* $\mathsf{t}$ *and* $\sigma$, *if* $(\mathcal{T} :: (\sigma, \mathsf{t})) \in \mathcal{R}$, *then for all* $k \leq m$, $(W, \mathcal{T} :: (\sigma, \mathsf{t}), \mathcal{R}) \Longrightarrow^k \mathcal{G}$;
2. *for all* $\sigma$ *and* $i \in \{1, \ldots, n\}$, *if* $(\mathsf{t}_i.C_i, \mathcal{T}) \hookrightarrow (\mathsf{t}_i.C_i', \mathcal{T} :: (\sigma, \mathsf{t}_i))$, *then* $(\mathcal{T} :: (\sigma, \mathsf{t}_i)) \in \mathcal{G}$ *and* $(\mathsf{t}_1.C_1 \parallel \ldots \mathsf{t}_i.C_i' \ldots \parallel \mathsf{t}_n.C_n, \mathcal{T} :: (\sigma, \mathsf{t}_i), \mathcal{R}) \Longrightarrow^k \mathcal{G}$ *holds for all* $k \leq m$.

Then the semantics of $R; G \vdash \{p\} W \{q\}$ is defined below. Theorem 4.4 shows the soundness theorem of the logic.

**Definition 4.3.** $R; G \models \{p\} W \{q\}$ *iff, for all* $\mathcal{T}$ *such that* $\mathcal{T} \models p \wedge (\boxminus(R \vee G) * \mathsf{true})$, *the following are true (where* $\mathcal{R} = [\![R * \mathsf{Id}]\!]$ *and* $\mathcal{G} = [\![G * \mathsf{true}]\!]$*):*

1. *if* $(W, \mathcal{T}) \overset{\mathcal{R}}{\longmapsto}^* (\mathbf{skip}, \mathcal{T}')$, *then* $\mathcal{T}' \models q$;
2. *for all* $m$, $(W, \mathcal{T}, \mathcal{R}) \Longrightarrow^m \mathcal{G}$.

**Theorem 4.4 (Soundness).** *If* $R; G \vdash \{p\} W \{q\}$ *then* $R; G \models \{p\} W \{q\}$.

We show the proof in the extended technical report [4].

```
pop(){
01  local done, next, t, t1;
02  done := false;
03  while (!done){
04    t := Top;
05    if (t == null) return null;
06    HP[tid] := t;
07    t1 := Top;
08    if (t == t1){
09      next := t.Next;
10      done := CAS(&Top, t, next);
11    }
12  }
13  retireNode(t);
14  HP[tid] := null;
15  return t;
}
```

```
retireNode(t){
16  local i, t';
17  i := 1;
18  while(i<=th_num){
19    if (i != tid){
20      t' := HP[i];
21      if (t'!= t){
22        i:= i+1;
23      }
24    }else i:= i+1;
25  }
}
```

**Fig. 9.** Optimistic Lock-Free Stacks with Hazard Pointers

## 5 Verification of Lock-Free Stacks with Hazard Pointers

We now apply HLRG to verify Michael's lock-free stacks, which use hazard pointers [11] to address the problems with the algorithm in Fig. 1. In Fig. 9 we show the new `pop` function. The `push` function is the same as in Fig. 1 and is omitted here. We use $\mathsf{stack}(\mathtt{Top}, A)$ below to specify the shared stack, which is implemented as a linked list pointed by `Top`. The set $A$ records the memory locations of the nodes on the list. It is kept to simplify our proofs. Below we use $E \mapsto E_1, E_2$ as a shorthand for $E \mapsto E_1 * E{+}1 \mapsto E_2$, and $E \mapsto \_$ for $\exists n.\, E \mapsto n$.

$$\begin{array}{rl} \mathsf{List}(\ell, \emptyset, \mathsf{nil}) & \stackrel{\text{def}}{=} \mathsf{emp} \wedge \ell = \mathtt{null} \\ \mathsf{List}(\ell, A, n :: L) & \stackrel{\text{def}}{=} \ell \in A \wedge \exists \ell'.\, (\ell \mapsto n, \ell') * \mathsf{List}(\ell', A - \{\ell\}, L) \\ \mathsf{stack}(\mathtt{Top}, A) & \stackrel{\text{def}}{=} \exists \ell, L.\, (\mathtt{Top} \mapsto \ell) * \mathsf{List}(\ell, A, L) \end{array} \quad (1)$$

The algorithm fixes the ABA problem by using a global array `HP`, which contains a "hazard" pointer for each thread. The array is specified by $I_{\text{hp}}(\mathtt{HP})$. Here `HP+tid` is the location of `HP[tid]`, and `th_num` is the number of threads.

$$I_{\text{hp}}(\mathtt{HP}) \stackrel{\text{def}}{=} \circledast_{\mathtt{tid} \in [1..\mathtt{th\_num}]}.\, \mathtt{HP}{+}\mathtt{tid} \mapsto \_ \quad (2)$$

where $\circledast_{x \in s}.p(x) \stackrel{\text{def}}{=} s = \emptyset \wedge \mathsf{emp} \vee \exists z.\, (s = \{z\} \uplus s') \wedge (\circledast_{x \in s'}.p(x)) * p(z)$
and $\uplus$ is the union of disjoint sets.

Before a racy access to the top node on the stack, a thread stores the node's memory location into its `HP` entry (lines 06-08). This announces to other threads that the node is being accessed and should not be reclaimed. When a node is successfully removed from the stack (line 10), the remover thread calls `retireNode` (line 13) and waits till after this node is no longer being accessed by any other

threads (i.e., not pointed by their HP entries). Finally, it clears its own HP entry (line 14) before it obtains the full ownership of the node (line 15).

We use $\mathsf{remove}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid})$ in (3) to specify that the thread tid is in the *remove* phase: it has popped the node at $\ell$ from the stack, but has not reached line 14 yet. The part in front of $\trianglerighteq$ says that there was a primitive operation in history, which popped the node from the stack. The $\trianglerighteq$ operator and the assertion following it require that the removed node be pointed by the remover's own HP entry ever since. Here $E_1 \rightsquigarrow E_2$ is a shorthand for $(E_1 \mapsto E_2) * \mathsf{true}$. The predicate $\mathsf{not\_rem}(\mathtt{Top}, \mathtt{HP}, \mathtt{tid})$ in (4) says that tid is currently not in the remove phase.

$$\begin{aligned}\mathsf{remove}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \stackrel{\text{def}}{=}\ & \big(((\mathtt{HP}{+}\mathtt{tid} \mapsto \ell * \mathtt{Top} \rightsquigarrow \ell) \ltimes_{\mathtt{tid}} (\mathtt{HP}{+}\mathtt{tid} \mapsto \ell * \exists \ell'.\mathtt{Top} \rightsquigarrow \ell' \wedge \ell \neq \ell')) \\ & \trianglerighteq \mathtt{HP}{+}\mathtt{tid} \rightsquigarrow \ell\big) \wedge \ell \neq \mathtt{null}\end{aligned} \tag{3}$$

$$\mathsf{not\_rem}(\mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \stackrel{\text{def}}{=} \neg\exists \ell.\, \mathsf{remove}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \tag{4}$$

In addition to the stack and the HP array, the popped nodes that are accessible from the hazard pointers should be viewed as shared resources as well. We use $\mathsf{opset}(\mathtt{Top}, \mathtt{HP}, S, O)$ in (5) to iterate these shared nodes, where $O$ is the set of pointers pointing to these nodes and $S$ is the set of threads.

$$\mathsf{opset}(\mathtt{Top}, \mathtt{HP}, \emptyset, \emptyset) \stackrel{\text{def}}{=} \mathsf{true} \tag{5}$$

$$\begin{aligned}&\mathsf{opset}(\mathtt{Top}, \mathtt{HP}, \{\mathtt{tid}\} \uplus S, O) \stackrel{\text{def}}{=} \\ &\quad (\exists \ell.\, \mathsf{remove}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \wedge \ell \in O \wedge \mathsf{opset}(\mathtt{Top}, \mathtt{HP}, S, O - \{\ell\})) \\ &\vee\ (\mathsf{not\_rem}(\mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \wedge \mathsf{opset}(\mathtt{Top}, \mathtt{HP}, S, O))\end{aligned}$$

The invariant $I$ below specifies all three parts of the shared resources. $I$ is a last-state-precise assertion. The domain of the shared resource depends on the historical information whether nodes are popped before or not.

$$\begin{aligned}I \stackrel{\text{def}}{=}\ & \exists O.\, \mathsf{opset}(\mathtt{Top}, \mathtt{HP}, [1..\mathtt{th\_num}], O) \\ & \wedge\ (I_{\mathrm{hp}}(\mathtt{HP}) * \exists A.\mathsf{stack}(\mathtt{Top}, A) * (\circledast_{\ell \in O}.\, \ell \mapsto \_, \_))\end{aligned} \tag{6}$$

Below we characterize the meaning of hazard pointers. $\mathsf{ishazard}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid})$ says $\mathtt{HP}[\mathtt{tid}]$ contains a "confirmed" hazard pointer $\ell$, i.e. $\ell$ was once the top of the stack in history and the thread tid has not updated the Top pointer ever since (though Top might have been updated by other threads). When the remover thread invokes retireNode on the top node t, it scans the hazard pointers of all other threads and make sure that $\mathsf{ishazard}(\mathtt{t}, \mathtt{Top}, \mathtt{HP}, \mathtt{tid})$ does not hold for each non-remover thread tid. This is specified by $\mathsf{hazfree}(\mathtt{t}, \mathtt{Top}, \mathtt{HP}, \mathtt{tid})$, which says that the node t has been popped by the thread tid and other threads no longer treat it as a hazard node.

$$\mathsf{upd\_top}(\mathtt{tid}) \stackrel{\text{def}}{=} \exists \ell, \ell'.\, (\mathtt{Top} \rightsquigarrow \ell \ltimes_{\mathtt{tid}} \mathtt{Top} \rightsquigarrow \ell') \wedge \ell \neq \ell'$$

$$\begin{aligned}&\mathsf{ishazard}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \stackrel{\text{def}}{=} \\ &\quad (\mathtt{HP}{+}\mathtt{tid} \mapsto \ell * \mathtt{Top} \rightsquigarrow \ell) \rhd \big((\mathtt{HP}{+}\mathtt{tid} \rightsquigarrow \ell) \wedge \neg \mathsf{upd\_top}(\mathtt{tid})\big)\end{aligned} \tag{7}$$

$$\begin{aligned}&\mathsf{hazfree}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \stackrel{\text{def}}{=} \\ &\quad \mathsf{remove}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \wedge \forall \mathtt{tid}' \in [1..\mathtt{th\_num}].\, \mathtt{tid}' = \mathtt{tid} \vee \neg \mathsf{ishazard}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}')\end{aligned} \tag{8}$$

$$
\begin{array}{lll}
\mathsf{Pop}_{\mathtt{tid}} & \stackrel{\text{def}}{=} \exists \ell, \ell'.\ \ominus\left((\mathtt{Top} \rightsquigarrow \ell) * (\mathtt{HP}{+}\mathtt{tid} \mapsto \ell) * (\ell \mapsto \_, \ell') * \mathsf{List}(\ell', \_, \_)\right) & \\
 & \quad \wedge((\mathtt{Top} \mapsto \ell \ltimes_{\mathtt{tid}} \mathtt{Top} \mapsto \ell') * \mathsf{Id}) & \text{(line 10)} \\
\mathsf{Retire}_{\mathtt{tid}} & \stackrel{\text{def}}{=} \exists \ell.\ \ominus \mathsf{hazfree}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) & \\
 & \quad \wedge (((\mathtt{HP}{+}\mathtt{tid} \mapsto \ell * \ell \mapsto \_, \_) \ltimes_{\mathtt{tid}} \mathtt{HP}{+}\mathtt{tid} \mapsto \mathtt{null}) * \mathsf{Id}) & \text{(line 14)} \\
\mathsf{Reset\_HP}_{\mathtt{tid}} & \stackrel{\text{def}}{=} \ominus \mathsf{not\_rem}(\mathtt{Top}, \mathtt{HP}, \mathtt{tid}) & \\
 & \quad \wedge ((\mathtt{HP}{+}\mathtt{tid} \mapsto \_ \ltimes_{\mathtt{tid}} \mathtt{HP}{+}\mathtt{tid} \mapsto \_) * \mathsf{Id}) & \text{(line 06)} \\
\mathsf{Push}_{\mathtt{tid}} & \stackrel{\text{def}}{=} (\mathtt{Top} \mapsto \ell \ltimes_{\mathtt{tid}} (\mathtt{Top} \mapsto \ell' * \ell' \mapsto \_, \ell)) * \mathsf{Id} & \text{(line 15 in Fig. 1)}
\end{array}
$$

$$
\begin{array}{lll}
G_{\mathtt{tid}} & \stackrel{\text{def}}{=} & (\mathsf{Retire}_{\mathtt{tid}} \vee \mathsf{Pop}_{\mathtt{tid}} \vee \mathsf{Push}_{\mathtt{tid}} \vee \mathsf{Reset\_HP}_{\mathtt{tid}} \vee \mathsf{Id}) \wedge (I \ltimes I) \\
R_{\mathtt{tid}} & \stackrel{\text{def}}{=} & \bigvee_{\mathtt{tid}' \in [1..\mathtt{th\_num}] \wedge \mathtt{tid} \neq \mathtt{tid}'} G_{\mathtt{tid}'}
\end{array}
$$

**Fig. 10.** Transitions over Shared Resources, and R-G Specifications

The call to `retireNode` is crucial. As we will show below, it ensures that a confirmed hazard pointer cannot be a dangling pointer, and a popped node pointed by any confirmed hazard pointers cannot show up on the stack again (thus the ABA problem is avoided).

**Verification of the Algorithm.** We first define in Fig. 10 all the operations over the shared data structure, and show which line of the code makes the corresponding transition (read-only operations are simply $\mathsf{Id}$ transitions and are omitted). $\mathsf{Pop}_{\mathtt{tid}}$ pops the top node from the stack. It requires that the hazard pointer point to the top of the stack. $\mathsf{Retire}_{\mathtt{tid}}$ sets the value of $\mathtt{HP}[\mathtt{tid}]$ into `null`, knowing that the popped node is no longer a hazard node. Then the node $\ell$ is converted logically from *shared* resource to *private*. $\mathsf{Reset\_HP}_{\mathtt{tid}}$ resets the hazard pointer when the thread `tid` fails to pop a node. $\mathsf{Push}_{\mathtt{tid}}$ pushes a *private node* onto the stack.

We also define the rely ($R_{\mathtt{tid}}$) and guarantee ($G_{\mathtt{tid}}$) of the thread `tid`. Here $I$ (defined in (6)) is used to fence the domain of all possible actions. It is easy to see $\mathsf{fence}(I, R_{\mathtt{tid}})$ and $\mathsf{fence}(I, G_{\mathtt{tid}})$ are satisfied. Next we show some key trace invariants derivable from $\boxminus(R_{\mathtt{tid}} \vee G_{\mathtt{tid}})$. They are used when the INV rule is applied. Also they show the key intuition of the algorithm.

*Invariant 1.* This invariant ensures that a node pointed by a hazard pointer is either on the stack or in the set $O$, so it is safe to dereference a hazard pointer.

$$
\begin{array}{l}
\forall \ell, \mathtt{tid}, A, O.\ \mathsf{ishazard}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \wedge \mathsf{opset}(\mathtt{Top}, \mathtt{HP}, [1..\mathtt{th\_num}], O) \\
\qquad\qquad \wedge (\mathsf{stack}(\mathtt{Top}, A) * \mathsf{true}) \wedge \ell \neq \mathtt{null} \Rightarrow \ell \in A \vee \ell \in O
\end{array}
$$

*Invariant 2.* If a thread `tid` once held a hazard pointer pointing to the top of the stack, and the top node on the stack was popped by other threads, then the node will not be on the stack again as long as `tid`'s `HP` entry is not changed. This invariant ensures that there are no ABA problems.

$$
\begin{array}{l}
\forall \ell, A, A', \mathtt{tid}. \\
\quad \big((\mathsf{ishazard}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \wedge (\mathsf{stack}(\mathtt{Top}, A) * \mathsf{true})) \rhd \mathtt{HP}{+}\mathtt{tid} \rightsquigarrow \ell\big) \\
\qquad\qquad \wedge (\mathsf{stack}(\mathtt{Top}, A') * \mathsf{true}) \wedge \ell \neq \mathtt{null} \wedge \ell \notin A \Rightarrow \ell \notin A'
\end{array}
$$

$\mathsf{POP}_{\mathtt{tid}}(\ell, n, \ell') \stackrel{\text{def}}{=} \ominus((\mathtt{Top} \rightsquigarrow \ell) * (\ell \mapsto n, \ell') * \mathsf{List}(\ell', \_, \_))$
$\qquad\qquad \wedge ((\mathtt{Top} \mapsto \ell \ltimes_{\mathtt{tid}} \mathtt{Top} \mapsto \ell') * \mathsf{Id})$

```
pop(){
```
$\{\boxed{\mathtt{HP}+\mathtt{tid} \mapsto \mathtt{null}}_I\}$
```
01  local done, next, t, t1;
02  done := false;
```
$\{\boxed{\mathtt{HP}+\mathtt{tid} \mapsto \mathtt{null}}_I\} \wedge \neg\mathtt{done}\}$

$\{\boxed{\mathsf{not\_rem}(\mathtt{Top}, \mathtt{HP}, \mathtt{tid})}_I \wedge \neg\mathtt{done}\}$

loop invariant:

$\{\boxed{\mathsf{not\_rem}(\mathtt{Top}, \mathtt{HP}, \mathtt{tid})}_I \wedge \neg\mathtt{done}$
$\qquad \vee \boxed{\exists n, \ell'. \diamondsuit\, \mathsf{POP}_{\mathtt{tid}}(\mathtt{t}, n, \ell') \wedge (\mathsf{remove}(\mathtt{t}, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) * (\mathtt{t} \mapsto n, \ell'))}_I \wedge \mathtt{done}\}$

```
03  while (!done){
```
$\{\boxed{\mathsf{not\_rem}(\mathtt{Top}, \mathtt{HP}, \mathtt{tid})}_I\}$
```
04    <t := [Top]>;
05    if (t == null) return null;
06    <HP[tid] := t>;
```
$\{\exists \ell. \boxed{\mathsf{not\_rem}(\mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \wedge \mathtt{HP}+\mathtt{tid} \mapsto \ell}_I \wedge \mathtt{t} = \ell \wedge \ell \neq \mathtt{null}\}$
```
07    <t1 := [Top]>;      Apply ATOM-R and ATOM
```
$\{\exists \ell, \ell'. \boxed{\mathsf{not\_rem}(\mathtt{Top}, \mathtt{HP}, \mathtt{tid}) \wedge (\mathtt{HP}+\mathtt{tid} \mapsto \ell * \mathtt{Top} \mapsto \ell')}_I \wedge \mathtt{t} = \ell \wedge \ell \neq \mathtt{null} \wedge \mathtt{t1} = \ell'\}$

$\{\exists \ell, \ell'. \boxed{\ell = \ell' \Rightarrow \mathsf{ishazard}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid})}_I \wedge \mathtt{t} = \ell \wedge \ell \neq \mathtt{null} \wedge \mathtt{t1} = \ell'\}$
```
08    if (t == t1){
```
$\{\exists \ell. \boxed{\mathsf{ishazard}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid})}_I \wedge \mathtt{t} = \ell \wedge \ell \neq \mathtt{null}\}$

Apply INV with *Invariant 1*

$\{\exists \ell, n, \ell'. \boxed{\mathsf{ishazard}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) * \ell \mapsto n, \ell'}_I \wedge \mathtt{t} = \ell\}$
```
09      <next := t.Next>;
```
$\{\exists \ell, n, \ell'. \boxed{\mathsf{ishazard}(\ell, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) * \ell \mapsto n, \ell'}_I \wedge \mathtt{t} = \ell \wedge \mathtt{next} = \ell'\}$
```
10      <done := CAS(&Top, t, next)>;      Apply INV with Invariant 2
11    }
12  }
```
$\{\boxed{\exists n, \ell'. \diamondsuit\, \mathsf{POP}_{\mathtt{tid}}(\mathtt{t}, n, \ell') \wedge (\mathsf{remove}(\mathtt{t}, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) * (\mathtt{t} \mapsto n, \ell'))}_I\}$
```
13  retireNode(t);      Apply FRAME and FRAMET
```
$\{\boxed{\exists n, \ell'. \diamondsuit\, \mathsf{POP}_{\mathtt{tid}}(\mathtt{t}, n, \ell') \wedge (\mathsf{hazfree}(\mathtt{t}, \mathtt{Top}, \mathtt{HP}, \mathtt{tid}) * (\mathtt{t} \mapsto n, \ell'))}_I\}$
```
14  <HP[tid] := null>;      Apply ATOM-R and ATOM
```
$\{\exists n, \ell'. \boxed{\diamondsuit\, \mathsf{POP}_{\mathtt{tid}}(\mathtt{t}, n, \ell')}_I * (\mathtt{t} \mapsto n, \ell')\}$
```
15  return t;
```
$\{\boxed{\diamondsuit\, \mathsf{List}(\mathtt{null}, \mathsf{nil})}_I \wedge \mathtt{t} = \mathtt{null} \vee \exists n, \ell'. \boxed{\diamondsuit\, \mathsf{POP}_{\mathtt{tid}}(\mathtt{t}, n, \ell')}_I * (\mathtt{t} \mapsto n, \ell')\}$
```
}
```

**Fig. 11.** Verification of pop

$I_{\text{thp}} \stackrel{\text{def}}{=} (\texttt{Top} \mapsto \_) * I_{\text{hp}}(\texttt{HP})$ $\quad$ $I_{\text{rN}} \stackrel{\text{def}}{=} \mathsf{remove}(\texttt{t}, \texttt{Top}, \texttt{HP}, \texttt{tid}) \wedge I_{\text{thp}}$

$\mathsf{selfornothazard}(\texttt{i}) \stackrel{\text{def}}{=} \boxed{\forall \texttt{tid}' \in [1..\texttt{i}-1].\, \texttt{tid}' = \texttt{tid} \vee \neg \mathsf{ishazard}(\texttt{t}, \texttt{Top}, \texttt{HP}, \texttt{tid}')}_{I_{\text{rN}}}$

```
retireNode(t){
```
$\{I_{\text{rN}}\}$
```
16  local i, t';
17  i := 1;
```
loop invariant: $\{\texttt{i} \leq \texttt{th\_num+1} \wedge \mathsf{selfornothazard}(\texttt{i-1})\}$
```
18  while(i<=th_num){
19    if (i != tid){
20      <t' := HP[i]>;
21      if (t '!=  t) {
```
$\{\diamondsuit\,(\exists \ell.\texttt{HP}+\texttt{i} \mapsto \ell \wedge (\texttt{i} \neq \texttt{tid} \wedge \ell \neq \texttt{t})) \wedge \mathsf{selfornothazard}(\texttt{i-1}) \wedge \texttt{i} \leq \texttt{th\_num}\}$

Apply INV with *Invariant 3*

$\{\mathsf{selfornothazard}(\texttt{i}) \wedge \texttt{i} \leq \texttt{th\_num}\}$
```
22          i := i+1;
23       }
24     }else i:= i+1;
25  }
```
$\{\mathsf{selfornothazard}(\texttt{th\_num+1})\}$

$\{\boxed{\mathsf{hazfree}(\texttt{t}, \texttt{Top}, \texttt{HP}, \texttt{tid})}_{I_{\text{thp}}}\}$

```
}
```

**Fig. 12.** Verification of `retireNode`

*Invariant 3.* This invariant justifies the `retireNode` procedure. If the thread `tid` popped a node $\ell$ and its `HP` entry points to the node, then for all other thread `tid'` its hazard pointer cannot point to the node and becomes a confirmed hazard pointer again if it was set to point to a different node ($\ell'$) in history.

$$\forall \ell, \ell', \texttt{tid}, \texttt{tid}'.\ \left(\begin{array}{l}\mathsf{remove}(\ell, \texttt{Top}, \texttt{HP}, \texttt{tid}) \wedge (\texttt{HP}+\texttt{tid}' \rightsquigarrow \ell') \wedge \ell' \neq \ell \\ \quad \rhd\ \texttt{HP}+\texttt{tid} \rightsquigarrow \ell\end{array}\right) \Rightarrow \neg \mathsf{ishazard}(\ell, \texttt{Top}, \texttt{HP}, \texttt{tid}')$$

We show the proof sketch for `pop` in Fig. 11. Here $\boxed{p}_I$ is used as a shorthand for $(p * \mathsf{true}) \wedge I$. The precondition of `pop` requires that the invariant $I$ hold over the shared resources, and that the calling thread's `HP` entry be initialized to `null`. The post-condition says $I$ holds at the end; the stack was either empty or the node `t` was popped out of the stack and is now part of the local resource of the calling thread. The proof sketch for `retireNode` is given in Fig. 12.

Most part of the proof simply follows the rules of the logic. The interesting part is that the specification of `retireNode` does not mention the linked list and the nodes in opset. Neither does it need to know that the pop operation has been done ($\exists n, \ell'.\, \diamondsuit\, \mathsf{POP}_{\texttt{tid}}(\texttt{t}, n, \ell')$). The knowledge can be added back by applying the FRAME and FRAMET rules respectively before we compose `retireNode` with `pop` (see Fig. 11). The `push` procedure has nothing to do with hazard pointers, thus the proof is trivial and can be seen in the extended TR [4].

## 6 Related Work and Conclusions

Assume-Guarantee (A-G) reasoning [12, 16] often views a concurrent program as an "invariant maintainer" instead of a "predicate transformer" [9], especially for safety verification. With this view, sequential composition of programs seems always trivial and is rarely discussed in previous work on A-G reasoning.

R-G reasoning [8], on the other hand, decomposes specifications into program invariants ($R$ and $G$) and Hoare-style pre- and post-conditions, which gives us a "predicate transformer" view of programs. With this view, sequential composition of programs ($C_1; C_2$) is no longer trivial. For instance, to use the post condition $q$ of $C_1$ as the pre-condition of $C_2$ as in Hoare Logic, we need to ensure the stability of $q$. Our logic is an extension of R-G reasoning. We take this "predicate transformer" view of programs, and try to spell out the details of the verification step associated with each program construct. Also, our logic successfully combines separation logic and temporal reasoning, which gives us better modularity. The two different frame rules in our logic reflect the frame properties over space (program states) and time respectively.

Gotsman et al. [5] introduced temporal operators in RGSep [19] to reason about liveness properties of non-blocking algorithms. They do not use any past tense operators. Their temporal operators were only used in their rely and guarantee conditions, but not in the pre- and post-conditions. Since the frame rule over $R$ and $G$ was not used there, the interoperability between temporal operators and separation logic operators was not discussed.

Parkinson et al. [14] used CSL to verify safety of the same stack algorithm we verified here. The specifications makes heavy uses of history variables. We believe that our specifications reflect the intuition of the algorithm more directly. Vafeiadis [18] applied RGSep to verify several non-blocking stack algorithms, which all rely on garbage collectors to avoid the ABA problem. It is unclear how the specification of Michael's stacks would look like in RGSep.

*Conclusion.* In this paper we have proposed a new program logic HLRG, which combines R-G reasoning with past tense temporal assertions to reason about optimistic concurrency algorithms. Our new logic supports modular verification, including the frame rules over both the separation logic and temporal operators. We have verified Michael's lock-free stack with hazard pointers and show that our history logic can directly capture the high-level intuition about the algorithm.

*Acknowledgments.* We thank anonymous referees for their comments on this paper. Ming Fu and Yong Li are supported in part by China Scholarship Council and by National Natural Science Foundation of China under grant No. 90718026. Much of this work was done during their visits to Yale University in 2009-2010. Zhong Shao is supported in part by NSF grants CNS-0915888, CNS-0910670, and CCF-0811665. Xinyu Feng is supported in part by National Natural Science Foundation of China under grant No. 90818019.

## References


[1] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. 20th Int'l Symp. on Distributed Computing (DISC'06)*, pages 194–208, 2006.

[2] X. Feng. Local rely-guarantee reasoning. In *Proc. 36th ACM Symp. on Principles of Prog. Lang.*, pages 315–327. ACM Press, Jan. 2009.

[3] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 173–188. Springer-Verlag, March 2007.

[4] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about optimistic concurrency using a program logic for history. Technical Report YALEU/DCS/TR-1428, Dept. of Computer Science, Yale University, New Haven, CT, June 2010. `http://flint.cs.yale.edu/publications/roch.html`.

[5] A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *Proc. 36th ACM Symp. on Principles of Prog. Lang.*, pages 16–28. ACM, 2009.

[6] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Annual Int'l Symp. on Computer Architecture (ISCA)*, pages 289–300, 1993.

[8] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. on Programming Languages and Systems*, 5(4):596–619, 1983.

[9] L. Lamport and F. B. Schneider. The "Hoare Logic" of CSP, and all that. *ACM Trans. Program. Lang. Syst.*, 6(2):281–296, 1984.

[10] O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *Proc. Conf. on Logic of Programs*, volume 193 of *LNCS*, pages 196–218. Springer, 1985.

[11] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[12] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

[13] P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, pages 49–67, 2004.

[14] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *Proc. 34th ACM Symp. on Principles of Prog. Lang.*, pages 297–302. ACM Press, Jan. 2007.

[15] M. J. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in hoare logics. In *Proc. 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 137–146. IEEE Computer Society, August 2006.

[16] A. Pnueli. In transition from global to modular temporal resoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO ASI Series, pages 123–144. Springer-Verlag, 1984.

[17] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74. IEEE Computer Society, July 2002.

[18] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, UK, July 2007.

[19] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. 18th Int'l Conf. on Concurrency Theory*, pages 256–271, 2007.