Verification of Lock-free *Scalable Synchronous Queue* (Technical Report)

Lei Jinjiang and Qiu Zongyan

LMAM and Department of Informatics, School of Mathematics
Peking University, Beijing 100871, CHINA

Abstract. Lock-free algorithms are extremely hard to be built correct due to their fine-grained concurrency natures. Formal frameworks for verifying them are crucial. In this paper we present a framework for verification of CAS-based lock-free algorithms, based on RGSep [25]. As an example, we prove a nontrivial lock-free algorithm that is practically adopted in Java 6. The strength of our method lies on that it completely rules out auxiliary variables/commands, thus is relatively easier to conduct and comprehend, comparing to the existing work.

Key words: Rely-guarantee, Separation Logic, Lock-Free Algorithms, Verification, Scalable Synchronous Queue

1 Introduction

Implementations of concurrent programs usually rely on locks, semaphores, or similar mechanisms. Although lock-based synchronization approaches are effective for supporting mutual exclusion and information sharing, they also incur many defects, including deadlock, livelock, starvation, priority inversion, convoy effect, etc.

Lock-free (as a class of non-blocking) algorithms are immune to deadlock and achieve progress even if some threads are descheduled or fail. Various primitives for these algorithms have been proposed and developed, e.g., CAS, DCAS, LL/SC, etc. Take CAS as an example. A CAS instruction, with the form of `cas`(loc, old, new), takes three arguments: a location (memory address) loc, an old value, and a new value. In execution of the `cas`, if the location holds value old as expected, it will be replaced by new and returns **`true`**, otherwise the `cas` returns **false** while nothing is changed. To develop a lock-free algorithm, designers need to arrange in the algorithm these instructions, especially the CASs (or other primitives), skillfully to achieve the goal.

Existing theoretical weapons for reasoning concurrent programs include Concurrent Separation Logic (CSL) [16, 2], Rely-guarantee Reasoning [14, 15], etc. CSL adapts a form of local reasoning in which one needs only to focus on the footprint of (the part of memory actually touched by) the thread. This facilitates modular reasoning based on the resource invariant. Rely-guarantee reasoning is

super in tackling finer grained interferences among threads and effects of intertwined atomic commands. Every specification in rely-guarantee is accompanied with a pair of rely ($\mathcal{R}$) and guarantee ($\mathcal{G}$) conditions. $\mathcal{R}$ is the thread's expectations for the state transitions made by its environment, and $\mathcal{G}$ denotes the promise of the transitions made by the thread itself.

So far, based on either CSL or rely guarantee, most proofs of lock-free algorithms heavily rely on auxiliary variables/commands, e.g. [20, 23]. However, on one side, auxiliary variables/commands would usually obscure the proof; on the other side, the layout of auxiliary variables/commands itself is also a tricky work.

Our aim is also to develop useful foundation and techniques for proving lock-free algorithms. We propose new methods for verifying lock-free algorithms. Most importantly, by packaging CAS into control flow commands, the dependency of auxiliary variables and commands is relieved, and make proofs more explicit. The main contributions of our work are as follows:

- We propose new method for reasoning under garbage collection (GC). It allow primitives touch the shared state of concurrent programs, and also capture the semantics that is implicitly guaranteed by GC.
- We package CAS primitive into control flow commands. This choice is reasonable for proving lock-free algorithms. A set of rules for packaged-CAS commands are developed, and the soundness of rules are proved.
- We generalize rules for control flow commands; and the new rules are more expressive due to the relaxed premise. This and above techniques are important for practically proving concurrent programs, and also relatively intuitive because they don't require for any auxiliary variables and commands.
- We prove a synchronous queue algorithm that has been adopted by Java 6. By the proving, we find some semantically equivalent variations of it. We also have an investigation on its fairness and liveness properties.

The rest of the paper is organized as follows. In Section 2, we propose and discuss reasoning with garbage collection, rules for packaged-CAS commands, and generalized rules for control flow commands. The scalable synchronous queue algorithm is introduced in Section 3; with its proof in Section 4. We discuss some related work in Section 5 then conclude.

2 The Inference Rules

In this section we build our foundations for reasoning lock-free algorithms. Basic concepts and definitions in RGSep are introduced first. Then we give rules for assignments under garbage collection, and rules for packaged-CAS commands, at last the improved rules for control flow commands.

2.1 Notations and Definitions

Now we introduce some notations and concepts in this paper, which are developed from RGSep. The techniques are general for most heap-stack models.

The assertion language is a variant of SL proposed by Vafeiadis [25], where P stands for any SL assertion:

$$\begin{array}{rll} p, q, r ::= & P & \text{Local assertion} \\ | & \boxed{P} & \text{Shared assertion} \\ | & p * q & \text{Separating conjunction} \\ | & p \wedge q & \text{Conjunction} \\ | & p \vee q & \text{Disjunction} \\ | & \forall x.p & \text{Universal quant.} \\ | & \exists x.p & \text{Existential quant.} \end{array}$$

In the model of the Logic, heap is partitioned into one part shared by all threads in the program, and several disjunct local portions where each is privately owned by a thread. Local assertions specify states of private portions (namely, *private states*), while the shared assertions specify the state of the shared part (*shared state*). However, since there is only one shared state at a time, comparing with standard Separation Logic, $*$ here takes the little different meaning that $*$ splits only the local states, but not the shared state. In this case, every shared assertion targets to the whole shared part, in particular, $\boxed{P} * \boxed{Q} \Leftrightarrow \boxed{P \wedge Q}$.

By shared state, we mean all the resources that are accessible to all threads. Variables in Separation Logic are viewed, originally, as shared resource. This is not suitable for concurrent programs, thus comes the *variables-as-resource* [19, 1]. For simplicity, we don't explicitly specify variables ownership, but still distinguish shared/private variables. Finally, we take "shared state = shared heap + shared variables".

In rely-guarantee reasoning, a thread expresses its interaction with the environment by a pair of rely/guarantee conditions. The rely condition $\mathcal{R}$ specifies all possibly mutation on shared state caused by the environment, and the guarantee condition $\mathcal{G}$ specifies what caused by the thread itself. The thread ensures that its atomic transitions abide by its guarantee condition as long as its environment respects its rely condition. Here $\mathcal{R}/\mathcal{G}$ specifies only the mutations of shared state. It is a sanity requirement that they should not access the private state of any thread. Rely/guarantee conditions are important for reasoning concurrent programs, because they can be viewed as a contract for the interferences between a thread and its environment.

We use *action*, $P \rightsquigarrow Q$, to depict changes performed on the shared state. $P \rightsquigarrow Q$ means replacing a portion of the shared state satisfying P to a portion satisfying Q. It's worth noting that P and Q are not required to depict the whole shared state. The semantics of an action, and a set of actions, are defined as state transitions, i.e., a set of pre- and post-states pairs:

$$\begin{aligned} [\![P \rightsquigarrow Q]\!] \mathrel{\widehat{=}} \{(\sigma, \sigma') | & (\sigma = \sigma_1 * \sigma_0) \wedge \\ & (\sigma' = \sigma_2 * \sigma_0) \wedge \\ & \sigma_1 \models P \wedge \sigma_2 \models Q\} \\ [\![\{P_i \rightsquigarrow Q_i \mid i \in 1..n\}]\!] \mathrel{\widehat{=}} & (\cup_{i=1}^{n} [\![P_i \rightsquigarrow Q_i]\!])^* \end{aligned}$$

Here σ denotes a state, $\sigma \models P$ means P holds in state σ, and t^* is the reflective transitive closure of transition t.

Because we use SL to express rely/guarantee conditions, we require each actions in $\mathcal{R}/\mathcal{G}$ conditions to be *precise* [21]. Assertion P is *precise*, if for any state there is at most one sub-state satisfying P. Action $P \rightsquigarrow Q$ is *precise* if both P and Q are precise. Precise action is reasonable because it ensures a precise state transition.

A predicate u is *stable* under an action if it is valid both before and after the state transition based on the action:

Definition 1. *(Stable) A predicate u is stable under $P \rightsquigarrow Q$, if for any $(\sigma, \sigma') \in [\![P \rightsquigarrow Q]\!]$ and $\sigma \models u$, then $\sigma' \models u$.*

A predicate u is *stable* under a set of actions $\mathcal{S}$, if it is stable under every element of $\mathcal{S}$.

Definition 2. *(Compatible) Assume for a concurrent program we have n threads and n pairs $\{(\mathcal{R}_i, \mathcal{G}_i)\}_i$ accordingly. We say that the system is compatible in* RGL*,* iff *for any thread $i \in 1 \ldots n$, $[\![\mathcal{G}_i]\!] \subseteq [\![\mathcal{R}_j]\!]$ for each $j \neq i$.*

That is, a *compatible* concurrent program requires that the guarantee of a thread is what other threads rely on.

Hoare rules in rely-guarantee reasoning take the form of $\mathcal{R}, \mathcal{G} \models \{P\}\, C\, \{Q\}$, where both pre- and post-conditions are required to be stable under $\mathcal{R}$, and $[\![P \rightsquigarrow Q]\!] \subseteq [\![\mathcal{G}]\!]$. In addition, we introduce a function Fd such that $Fd(x)$ returns the set of fields according to the type of x, and then the rule takes the form of $\mathcal{R}, \mathcal{G}, Fd \models \{P\}\ C\ \{Q\}$. For example, if x denotes a node of binary trees, $Fd(x) = \{\texttt{lchild}, \texttt{rchild}\}$. Usage of Fd can be seen in Section 2.2. We may write $Fd \models \bullet$, or $\mathcal{R}, \mathcal{G} \models \bullet$, when the other components are not important.

In order to prove algorithms involving mutable data structures (as the one in Section 3), we propose some new rules for assignments in the environment with *garbage collection* (GC). The existence of GC and no explicit memory release guarantee, as long as some thread holds a reference to a dynamic node, the node will not be reclaimed.

For most algorithms under GC, we need to specify the reachable part from an object in heap. For convenient, we use $\mathsf{rset}(x, s)$ to represent *reachable set* of nodes from x wrt. a given set s of field names. When a node is in $\mathsf{rset}(x, s)$, we say it is *reachable* from x via s:

$$\mathsf{rset}(x, s) \stackrel{def}{=} \begin{cases} \mathbf{emp} & x \text{ is non-pointer or null} \\ \mathit{cell}(x) * (\circledast_{\forall n \in s} \mathsf{rset}(x.n, s)) & s \subseteq \mathit{Fd}(x) \\ \mathbf{false} & \text{otherwise} \end{cases}$$

Here $\circledast$ is the iterative version of "$*$", and we use $\mathit{cell}(x)$ to denote the object x points to, where *cell* is some object type. For example, if x is `node` which has a link field `next`, then $\mathsf{rset}(x, \{\texttt{next}\})$ represents $\texttt{node}(x) * \mathsf{rset}(x.\texttt{next}, \{\texttt{next}\})$; if x is a `tnode` for binary trees and let $s = \{\texttt{lchild}, \texttt{rchild}\}$, $\mathsf{rset}(x, s)$ is $\texttt{tnode}(x) *$

$\mathsf{rset}(x.\mathtt{lchild}, s) * \mathsf{rset}(x.\mathtt{rchild}, s)$. We allow s contain some (but not all) fields of object x. If x refers to a `tnode`, $\mathsf{rset}(x, \{\mathtt{lchild}\})$ is fine but only depicts part of reachable set from x.

Two aspects of rset need to be declared. First, the definition of rset is valid when the reference relation is Directed Acyclic Diagram (DAG), while for general diagram, rset can be trivially defined as the fixpoint of the expansion from a node. Second, rset can also defined without its second argument, and $\mathsf{rset}(x)$ simply means the full expansion from cell x. However, a full expansion usually is not needed in really proof, and the second argument will make proofs concise.

2.2 Reasoning under Garbage Collection

The algorithm we will prove in section 4 in running under the effect of garbage collection. There are two levels when reasoning with GC. One is low-level, where the behavior and effects of GC is taken into account. For example, in the low-level, GC can be viewed as a thread that is concurrently running with other threads, and GC would affect the state of a system. The other is high-level, the effect of GC is not explicitly depict this level. In high level, all assertions are required to be GC-insensitive. Intuitively, assertion P is GC-insensitive if $\{P\}\,\mathtt{gc()}\,\{P\}$.

Hur *et al.* [12] discussed separation logic in the presence of GC. They formally constructed the relation between low- and high-levels. In this paper, we only reason programs in the high-level, where all assertions are GC-insensitive. As in [12], in order to guarantee the soundness, both the heap and all program variables only store GC-safe values, that is either non-pointer values or pointers to heads of allocated blocks.

The idea for reasoning under GC is simple: because GC will not reclaim the nodes that is accessible to any thread, $\mathsf{rset}(x, s)$ is a GC-insensitive assertion. We can add $\mathsf{rset}(x, s)$ in the pre- and post-conditions to extend the expressiveness of a specification.

Taken as a example, we define a rule for assignments under GC in SL as follows, where s is any given set, x and y are program variables:

$$Fd \vdash_{\mathrm{SL}} \{\mathsf{rset}(y, s)\}\ x := y\ \{x = y \wedge \mathsf{rset}(x, s)\}$$

This rule is valid for any given type of x. It says if all nodes reachable from y via s are in the heap before the assignment, then afterward, $x = y$ and all those nodes from x via s are still in the heap.

In concurrent programs, the situation becomes more complicated: $\mathsf{rset}(x, s)$ is volatile if x is a shared variable, because it may be modified by other threads. Thus, if x or y is in shared state, $x = y$ will not be stable under rely condition. It seems some restrictions are necessary. To solve the problem, we require that all assignments to shared variables are implemented by `cas` commands, then we only need to consider assignment to private variables. When x is private variable and Y is shared, we define:

$$\begin{array}{rl} \mathcal{R}, \mathcal{G}, Fd \vdash & \{\boxed{\mathsf{rset}(Y, s)}\}\ x := Y \\ & \{\boxed{\mathsf{rset}(x, s) * \mathbf{true}} * \boxed{\mathsf{rset}(Y, s) * \mathbf{true}}\} \end{array} \qquad \text{(ASS-GC)}$$

It says if Y is in shared state and nodes in $\mathsf{rset}(Y, s)$ are all in shared state, after the assignment, x points to some node in shared state and the $\mathsf{rset}(x, s)$ are all in shared state. Note, since Y is a shared variable, $x = Y$ is not guaranteed, so the post-assertion is stable under $\mathcal{R}$.

Besides, (ASS-GC) is intuitively sound for any $\mathcal{R}, \mathcal{G}$ and Fd under GC. The validity of its pre- and post-conditions are guaranteed by GC. Since reading a shared variable into a private variable does not change the shared state, it satisfies any $\mathcal{G}$. In addition, this rule is also sound when x and Y are both private variables; the explanation is similar.

2.3 Rules for Packaged-CAS Commands

One crux of this work is to formalize the `cas` primitive. A `cas` gives a boolean value, thus is often used as guards in **if** or **while** statements. However, a `cas` may also change the state as a side effect. If we formalized it as an expression, that would force us to introduce side-effect feature into all expressions, and make the approach more complicated. Some previous work treats `cas` as a command. However, hardly can a method found to capture the boolean result at the same time, various auxiliary variables/commands are introduced to remember the extra semantic information.

Based on an analysis of various lock-free algorithms, we decide to package `cas` primitive into control statements, that is, always considering **if** $\texttt{cas}(l, o, n)$ **then** `C1` **else** `C2` or **do** `C` **while** $\texttt{cas}(l, o, n)$ as whole structures, and define inference rules directly for them. Consequently, we can code a single `cas` into a semantically equivalent form **if** $\texttt{cas}(l, o, n)$ **then** `skip` **else** `skip`. Comparing with RGSep, although packaging of `cas` command don't extend its capability, it helps to rule out auxiliary variables and commands completely from our approach.

Rule for packaged-CAS `if` commands is:

$$\frac{\begin{array}{c}\varnothing, \mathcal{G}, Fd \vdash \{p * \boxed{l \mapsto o * \mathbf{true}}\}\ [l] := n\ \{r_1\} \\ \mathcal{R}, \mathcal{G}, Fd \vdash \{r_1\}\ C_1\ \{q\} \quad p * \boxed{l \mapsto o' * \mathbf{true}} \wedge o' \neq o \Rightarrow r_2 \\ \mathcal{R}, \mathcal{G}, Fd \vdash \{r_2\}\ C_2\ \{q\} \quad p \Rightarrow \boxed{l \mapsto - * \mathbf{true}} * \mathbf{true} \\ \mathbf{stable}(\{p, q, r_1, r_2\}, \mathcal{R})\end{array}}{\mathcal{R}, \mathcal{G}, Fd \vdash \{p\}\ \mathbf{if}\ \texttt{cas}(l, o, n)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \{q\}} \quad \text{(IF-CAS)}$$

Note that $[l] := n$ means the assignment of value n to location l, which is called *mutation* in Separation Logic.

Because the location accessed by $\texttt{cas}(l, o, n)$ must be in the shared state (which may be modified by other threads), any CAS-based command must rely on the nondeterministic value at location l. In another word, $\mathcal{R}$ should ensure the invariability of $[l]$ by the environment if the pre-condition of a packaged-CAS command specifies the value of $[l]$. Specifically, in rule (IF-CAS), we require $\mathcal{R}$ to be $\varnothing$ to ensure the stability of $p * \boxed{l \mapsto o * \mathbf{true}}$. Another difference between ordinary **if** B **then** C_1 **else** C_2 and **if** $\texttt{cas}(l, o, n)$ **then** C_1 **else** C_2 is that, the

guard of the latter one will access the heap location l, while in ordinary **if-then-else**, guard B is usually *pure*, without any effect to/from the heap. Therefore, here premise $p \Rightarrow \boxed{l \mapsto - * \mathbf{true}} * \mathbf{true}$ is necessary.

Similarly, there is a symmetrical rule:

$$\frac{\begin{array}{c}\varnothing, \mathcal{G}, Fd \vdash \{p * \boxed{l \mapsto o * \mathbf{true}}\}\ [l] := n\ \{r_2\} \\ \mathcal{R}, \mathcal{G}, Fd \vdash \{r_2\}\ C_2\ \{q\} \quad p * \boxed{l \mapsto o' * \mathbf{true}} \wedge o' \neq o \Rightarrow r_1 \\ \mathcal{R}, \mathcal{G}, Fd \vdash \{r_1\}\ C_1\ \{q\} \quad p \Rightarrow \boxed{l \mapsto - * \mathbf{true}} * \mathbf{true} \\ \mathbf{stable}(\{p, q, r_1, r_2\}, \mathcal{R})\end{array}}{\mathcal{R}, \mathcal{G}, Fd \vdash \{p\}\ \mathbf{if}\ !\mathtt{cas}(\mathrm{l}, \mathrm{o}, \mathrm{n})\ \mathbf{then}\ \mathrm{C}_1\ \mathbf{else}\ \mathrm{C}_2\ \{q\}} \quad \text{(IF-N-CAS)}$$

We also define the packaged `cas` command with **do-while** statement that are commonly used in lock-free algorithms. Taking here **do-while** but not **while-do**, since the former is used more frequently in lock-free algorithms. We can define rules for **while-do** without any problem.

$$\frac{\begin{array}{c}\mathcal{R}, \mathcal{G}, Fd \vdash \{p\}\ C\ \{r\} \quad \mathbf{stable}(\{p, q, r\}, \mathcal{R}) \\ \varnothing, \mathcal{G}, Fd \vdash \{r * \boxed{l \mapsto o * \mathbf{true}}\}\ [l] := n\ \{q\} \\ r * \boxed{l \mapsto o' * \mathbf{true}} \wedge o' \neq o \Rightarrow p \quad p \Rightarrow \boxed{l \mapsto - * \mathbf{true}} * \mathbf{true}\end{array}}{\mathcal{R}, \mathcal{G}, Fd \vdash \{p\}\ \mathbf{do}\ C\ \mathbf{while}\ !\mathtt{cas}(l, o, n)\ \{q\}} \quad \text{(WHILE-N-CAS)}$$

$$\frac{\begin{array}{c}\mathcal{R}, \mathcal{G}, Fd \vdash \{p\}\ C\ \{r\} \quad \mathbf{stable}(\{p, q, r\}, \mathcal{R}) \\ \varnothing, \mathcal{G}, Fd \vdash \{r * \boxed{l \mapsto o * \mathbf{true}}\}\ [l] := n\ \{p\} \\ r * \boxed{l \mapsto o' * \mathbf{true}} \wedge o' \neq o \Rightarrow q \quad p \Rightarrow \boxed{l \mapsto - * \mathbf{true}} * \mathbf{true}\end{array}}{\mathcal{R}, \mathcal{G}, Fd \vdash \{p\}\ \mathbf{do}\ C\ \mathbf{while}\ \mathtt{cas}(l, o, n)\ \{q\}} \quad \text{(WHILE-CAS)}$$

2.4 Improved Rules for Control Flow Commands

Besides rules for packaged-CAS commands, we also propose improved rules of control statements. The naive rules given below are not useful in many practical cases:

$$\frac{\begin{array}{c}p \Rightarrow (B = B) * \mathbf{true} \\ \mathcal{R}, \mathcal{G} \vdash \{p \wedge B\}\ C_1\ \{q\} \quad \mathcal{R}, \mathcal{G} \vdash \{p \wedge \neg B\}\ C_2\ \{q\}\end{array}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \{q\}} \quad \text{(N-IF)}$$

$$\frac{p \Rightarrow (B = B) * \mathbf{true} \quad \mathcal{R}, \mathcal{G} \vdash \{p \wedge B\}\ C\ \{q\}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{while}\ B\ \mathbf{do}\ C\ \{p \wedge \neg B\}} \quad \text{(N-WHILE)}$$

These rules are sound, but not usable when value of B relies on shared variables. Take (N-IF) for example, RGL requires the pre-condition of the specification to be stable under $\mathcal{R}$. However, if B accesses shared variables which may be modified by other threads, $\{p \wedge B\}$ is not stable under $\mathcal{R}$, thus the rule cannot be used. Consider a possible scenario in executing **if** B **then** C_1 **else** C_2: B holds initially and branch C_1 is chosen, but before executing C_1, the thread is preempted by another one, and B may not hold still when C_1 begins to run; for another branch, situation is the same.

To cope with the scenarios like above, we propose the following improved rules of control flow commands.

$$\frac{\begin{array}{c} p \wedge B \Rightarrow r_1 \quad p \wedge \neg B \Rightarrow r_2 \quad \mathbf{stable}(\{r_1, r_2, p\}, \mathcal{R}) \\ \mathcal{R}, \mathcal{G}, Fd \vdash \{r_1\}\ C_1\ \{q\} \quad \mathcal{R}, \mathcal{G}, Fd \vdash \{r_2\}\ C_2\ \{q\} \end{array}}{\mathcal{R}, \mathcal{G}, Fd \vdash \{p\}\ \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \{q\}} \quad \text{(IF)}$$

$$\frac{\begin{array}{c} p \wedge B \Rightarrow r \quad p \wedge \neg B \Rightarrow q \\ \mathcal{R}, \mathcal{G}, Fd \vdash \{r\}\ C\ \{p\} \quad \mathbf{stable}(\{p, q\}, \mathcal{R}) \end{array}}{\mathcal{R}, \mathcal{G}, Fd \vdash \{p\}\ \mathbf{while}\ B\ \mathbf{do}\ C\ \{q\}} \quad \text{(WHILE)}$$

These rules still keep the information of guard B, but do not require B to be stable under $\mathcal{R}$. Besides, (N-IF) and (N-WHILE) can be seen as special cases of (IF) and (WHILE).

For the soundness of the inference rules, we have the following theorem. Some of their proofs are given in Appendix A.

Theorem 1. (IF-CAS), (IF-N-CAS), (WHILE-CAS), (WHILE-N-CAS), (IF), (WHILE) *are sound.*

3 Scalable Synchronous Queue

The algorithm we are going to prove implements a CAS-based scalable synchronous queues (SSQ) [13]. It has been adopted by Java 6 concurency libraries because it remarkably outperforms Java SE 5.0 *SynchronousQueue* class. However, interferences among threads in SSQ are complicated, so hardly can one have an intuition about its correctness.

Paper [13] gives two modes of the implementation: the (LIFO) stack and the (FIFO) queue. As mentioned in [13], `enqueue` for queue mode and `push` for stack mode are symmetric with `dequeue` and `pop` respectively, except for the direction of data transfer, so we only prove `enqueue` and `push` in this paper. We copy the codes from [13] and list them in Table 1 and Table 2.

The data structure used in the queue mode is a single linked list held by global variables `Head` and `Tail` (Figure 1). The first node in the list is dummy; and the list is empty if there is only a dummy node (state S1). A node contains three fields: `data`, `state` (for storing tag `DATA` or `REQ`), `next`. In Figure 1, we use `N` to denote a null field, and `!N` for not null. We use `D` to denote that the `state` field is `DATA`, while `R` says it is `REQ`. A *matched data node* is a node where `state` field is `DATA` but its `data` field is null in the same time. A *matched request node* is a node whose `state` is `REQ` and its `data` field is not null. We say a *data node* x is *matched*, when another thread "withdraw" the value stored at x.`data` and set x.`data` `null`; for a request node, vice versa.

Now we look at the code at Table 1 . The first step is to read shared variables `Head` and `Tail` into local variables `h` and `t` (line 04-05). Then there are two branches: if the list is empty or has at least one *data node*, it tries to attach the newly created *data node* pointed by local variable `offer`, to the end of the list

```
01  void enqueue(e) {
02     offer = new Node(e);
03     while true {
04        t = Tail;
05        h = Head;
06        if (h == t || t.state == DATA) {
07           n = t.next;
08           if (t == Tail){
09              if (n != null) {
10                 casTail(t, n);
11              }
12              else
13                 if(t.casNext(n, offer)) {
14                    casTail(t, offer);
15                    while (offer.data != null));
16                       /* spin */;
17                    h = Head;
18                    if (offer == h.next)
19                       casHead(h, offer);
20                    return;
21                 }
22        } else{
23           n = h.next;
24           if(t != Tail || h != Head || n == null)
25              continue;
26           if(n.casData(null, e)) {
27              casHead(h, n);
28              return;
29           }else casHead(h, n);
30        }
31     }
32  }
```

Table 1. Scalable Dual Queue — Enqueue

(line 06-21); otherwise, when the list has at least one *request node*, it tries to match the data to the *unmatched request node* that behinds `Head` (line 22-30). In addition, the call to enqueue will not return until the `offer` is matched (line 15-16) or a *request node* is matched (line 26-28).

Algorithm for the stack mode is listed in Table 2. Unlike the queue mode, there is only one shared variable `Head` that refers to the top of the stack and no dummy node. The first step of `push` is also to read `Head` into local variable h (line 04). Based on the state of h, there are three branches: if the stack is empty or its top is a *data node*, it tries to attach the newly created *data node* to the top of the stack (line 06-14); if h refers to an *request node*, it tries to match it with the value that intended to be pushed (line 16-23); otherwise, the stack is fulfilling, it helps to match the top two nodes in the stack (line 25-28).

```
01  void push(e) {
02     d = new Node(e, DATA);
03     while true {
04        h = Head;
05        if (h == null || h.state == DATA) {
06           d.next = h;
07           if (!casHead(h, d))
08              continue;
09           while (d.match == null)
10              /* spin */;
11           h = Head;
12           if(h != null && h.next = d)
13              casHead(h, d.next);
14           return;
15        } else if (h.state = REQ){
16           f = new Node(e, DATA_F, h);
17           if (!casHead(h, f))
18              continue;
19           h = f.next;
20           n = h.next;
21           h.casMatch(null, f);
22           casHead(f, n);
23           return;
24        } else {
25           n = h.next;
26           nn = n.next;
27           n.casMatch(null, h)
28           casHead(h, nn);
29        }
30     }
31  }
```

Table 2. Scalable Dual Queue — Push

Can these algorithms work in every circumstance, no matter how interleaving of the invitations come from different threads? It is really not easy to conclude.

4 Proof for Synchronous Queue

Now we prove the SSQ algorithm. We give here the fundamental work, and put proof listing in Appendix B.

4.1 Adapted Rules for `casHead`, `casTail`, n.`casData`, and n.`casNext`

Primitives `casHead`, `casTail`, n.`casData` and n.`casNext` are used in SSQ. Their semantics is a bit different from plain `cas` discussed before: `casHead`(o, n) (alt., `casTail`(o, n)) compares the value of `Head` (or `Tail`) with o, and probably assigns

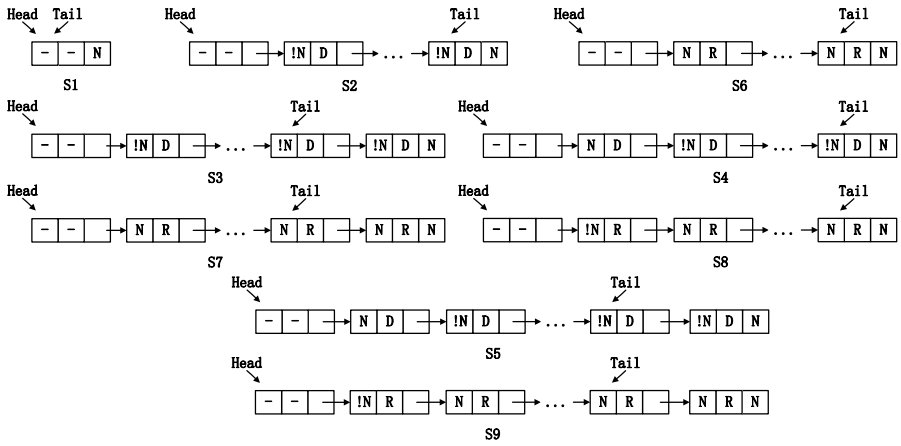

Fig. 1. The Link List's States

the new value n to it. On the other hand, $v.\texttt{casData}(o, n)$ (alt., $v.\texttt{casNext}(o, n)$) compares (and probably assigns) the value stored in the `data` (`next`) field of v with o. However, the essence of these CAS-based primitives is identical, and the adapted rules are similar with rules introduced in Section 2.3:

$$\frac{\begin{array}{c}\varnothing, \mathcal{G} \vdash \{p * \boxed{\texttt{Head} = o * \mathbf{true}}\}\ \texttt{Head} := n\ \{r_1\} \\ \mathcal{R}, \mathcal{G} \vdash \{r_1\}\ C_1\ \{q\} \quad \mathcal{R}, \mathcal{G} \vdash \{r_2\}\ C_2\ \{q\} \\ p * \boxed{\texttt{Head} = o' * \mathbf{true}} \wedge o' \neq o \Rightarrow r_2 \quad \mathbf{stable}(\{p, q, r_1, r_2\}, \mathcal{R})\end{array}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{if}\ \texttt{casHead}(o, n)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \{q\}} \quad \text{(IF-casHead)}$$

$$\frac{\begin{array}{c}\varnothing, \mathcal{G} \vdash \{p * \boxed{v.\texttt{data} \mapsto o * \mathbf{true}}\}\ v.\texttt{data} := n\ \{r_1\} \\ \mathcal{R}, \mathcal{G} \vdash \{r_1\}\ C_1\ \{q\} \quad \mathcal{R}, \mathcal{G} \vdash \{r_2\}\ C_2\ \{q\} \\ p * \boxed{v.\texttt{data} \mapsto o' * \mathbf{true}} \wedge o' \neq o \Rightarrow r_2 \\ p \Rightarrow \boxed{v.\texttt{data} \mapsto - * \mathbf{true}} * \mathbf{true} \quad \mathbf{stable}(\{p, q, r_1, r_2\}, \mathcal{R})\end{array}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{if}\ v.\texttt{casData}(o, n)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \{q\}} \quad \text{(IF-casData)}$$

Rules for `casTail`, $v.\texttt{casMatch}(o, n)$ and $v.\texttt{casNext}(o, n)$ are similar with rules for `casHead` and $v.\texttt{casData}(o, n)$.

4.2 Proof of the Queue Mode

We give the proof and discuss properties of `enqueue` in this section. Proof of stack mode is given in section 4.3.

Predicates for State Assertions (Queue Mode)

We specify the states using Separation Logic assertions for the queue mode. To facilitate the specification, we define some predicates in the first. According to our analysis of the possible shared states depicted in Figure 1, we need some predicates for specifying nodes and lists.

$$
\begin{aligned}
S_1 &= \mathsf{QNd}(\mathtt{Head}, -, -, \mathtt{null}) \wedge \mathtt{Head} = \mathtt{Tail} \\
S_2 &= \mathsf{QNd}(\mathtt{Head}, -, -, \mathtt{Head.next}) * \mathsf{DIs}(\mathtt{Head.next}, \mathtt{Tail.next}) \\
&\quad \wedge \mathtt{Tail.next} = \mathtt{null} \\
S_3 &= \exists x.\mathsf{QNd}(\mathtt{Head}, -, -, -) * \mathsf{DIs}(\mathtt{Head.next}, \mathtt{Tail.next}) \\
&\quad * \mathsf{QNd}(x, !\mathtt{null}, \mathtt{DATA}, \mathtt{null}) \wedge \mathtt{Tail.next} = x \\
S_4 &= \exists x, y.\mathsf{QNd}(\mathtt{Head}, -, -, x) * \mathsf{QNd}(x, \mathtt{null}, \mathtt{DATA}, y) \\
&\quad * \mathsf{DIs}(y, \mathtt{Tail.next}) \wedge \mathtt{Tail.next} = \mathtt{null} \\
S_5 &= \exists x, y.\mathsf{QNd}(\mathtt{Head}, -, -, x) * \mathsf{QNd}(x, \mathtt{null}, \mathtt{DATA}, -) \\
&\quad * \mathsf{DIs}(x.\mathtt{next}, \mathtt{Tail.next}) * \mathsf{QNd}(y, !\mathtt{null}, \mathtt{DATA}, \mathtt{null}) \\
&\quad \wedge \mathtt{Tail.next} = y \\
S_6 &= \mathsf{QNd}(\mathtt{Head}, -, -, \mathtt{Head.next}) * \mathsf{RIs}(\mathtt{Head.next}, \mathtt{Tail.next}) \\
&\quad \wedge \mathtt{Tail.next} = \mathtt{null} \\
S_7 &= \exists x.\mathsf{QNd}(\mathtt{Head}, -, -, -) * \mathsf{RIs}(\mathtt{Head.next}, \mathtt{Tail.next}) \\
&\quad * \mathsf{QNd}(x, \mathtt{null}, \mathtt{REQ}, \mathtt{null}) \wedge \mathtt{Tail.next} = x \\
S_8 &= \exists x, y.\mathsf{QNd}(\mathtt{Head}, -, -, x) * \mathsf{QNd}(x, !\mathtt{null}, \mathtt{REQ}, y) \\
&\quad * \mathsf{RIs}(y, \mathtt{Tail.next}) \wedge \mathtt{Tail.next} = \mathtt{null} \\
S_9 &= \exists x, y.\mathsf{QNd}(\mathtt{Head}, -, -, x) * \mathsf{QNd}(x, !\mathtt{null}, \mathtt{REQ}, -) \\
&\quad * \mathsf{RIs}(x.\mathtt{next}, \mathtt{Tail.next}) * \mathsf{QNd}(y, \mathtt{null}, \mathtt{REQ}, \mathtt{null}) \\
&\quad \wedge \mathtt{Tail.next} = y
\end{aligned}
$$

Fig. 2. Share States of the Queue Mode

Predicate QNd asserts a node with specific contents:

$$\mathsf{QNd}(x, v, s, y) \mathrel{\widehat{=}} x \mapsto \{\mathtt{data} \mapsto v, \mathtt{state} \mapsto s, \mathtt{next} \mapsto y\}$$

where `state` could be `DATA`, or `REQ`.

We define some predicates for asserting list segments, or special types of list segments:

$$
\begin{aligned}
\mathsf{Is}(x, y) &\mathrel{\widehat{=}} (\mathbf{emp} \wedge x = y) \vee (\mathsf{QNd}(x, -, -, -) * \mathsf{Is}(x.\mathtt{next}, y)) \\
\mathsf{DIs}(x, y) &\mathrel{\widehat{=}} (\mathbf{emp} \wedge x = y) \vee (\mathsf{QNd}(x, !\mathtt{null}, \mathtt{DATA}, -) * \mathsf{DIs}(x.\mathtt{next}, y)) \\
\mathsf{RIs}(x, y) &\mathrel{\widehat{=}} (\mathbf{emp} \wedge x = y) \vee (\mathsf{QNd}(x, \mathtt{null}, \mathtt{REQ}, -) * \mathsf{RIs}(x.\mathtt{next}, y))
\end{aligned}
$$

Here $\mathsf{Is}(x, y)$ asserts a list segment that begins with x and ends with a node whose `next` field is y. $\mathsf{DIs}(x, y)$ asserts additional that each node in the list is *unmatched data node*, i.e., whose `state` field is `DATA` and `data` field is not null; $\mathsf{RIs}(x, y)$ asserts that each node in the list is *unmatched request node*, whose `state` field is `REQ` and `data` field is null.

With these predicates, we specify the possible states of the linked list in the queue, totally nine sets of them as given in Figure 1. We list the specifications in Figure 2. Two properties are worth to mention: $S_1 = S_2 \cap S_6$; and there is at most one *matched node* in each of the states. These states are crucial for our proof, because they cover and partition all possible shared states of the algorithm.

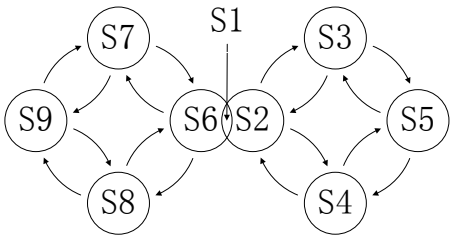

Fig. 3. Transition Graph of Shared States – Queue Mode

The Rely/Guarantee Conditions (Queue Mode)

The essence of rely-guarantee logic is using rely/guarantee conditions, $\mathcal{R}$ and $\mathcal{G}$, to describe the interferences between a thread and its environment. We construct these conditions for the algorithm in this subsection.

1. **The Rely Condition (Queue Mode)**
   After a careful analysis, we dig out the relationship between all the possible shared states. Figure 3 depicts all possible transitions among them, it shows out a very beautiful graph: regular, and symmetric. The regularity should be the real foundation of the correctness of the algorithm. It seems that no one knows this graph before, even for the designers of the algorithm.
   Please note that we do not depict state set S_1 as other sets in the graph, because S_1 consists of states belonging to both S_2 and S_6. The transition between S_2 and S_6 takes the different manner from the other transitions, via some common regression state of both sets.
   We specify the state transitions of Figure 3 by a set of actions, as given in Figure 4. In order to distinguish values of shared variables in pre- and post-states of an *action*, we use the hat form to denote the pre-values. We substitute all shared variables in the pre-state with their hat form, and use additional boolean expressions to depict the relation.
   Take the 7th action in Figure 4 as an example, which makes a transition from $S4$ to $S2$. After substitution, the left hand side becomes $\exists x, y.\mathsf{QNd}(\widehat{\texttt{Head}}, -, -, x) * \mathsf{QNd}(x, \texttt{null}, \texttt{DATA}, y) * \mathsf{Dls}(y, \widehat{\texttt{Tail}}.\texttt{next}) \wedge \widehat{\texttt{Tail}}.\texttt{next} = \texttt{null}$, depicting that the head of the queue is followed by a matched data node and then some unmatched data nodes. The right hand side says the head node is followed by some unmatched data node and there exists a node, say $\mathsf{QNd}(z_0, -, \texttt{DATA}, \texttt{Head})$ (we use z_0 instead of z to remove the $\exists$), which is the pre-node of `Head`. In order to precisely confine the state transition to an atomic action, $\widehat{\texttt{Head}} = z_0$ and $\widehat{\texttt{Tail}} = \texttt{Tail}$ are indispensable, which confine the footprint transition caused by the action: $\widehat{\texttt{Head}} = z_0$ and $\mathsf{QNd}(z_0, -, \texttt{DATA}, \texttt{Head})$ implies `Head` moves one step ahead; $\widehat{\texttt{Tail}} = \texttt{Tail}$ implies `Tail` does not move.
   The state transitions among $S6, S7, S8, S9$ are not given, because they are symmetric with the ones listed in Figure 4. We define our rely condition, $\mathcal{R}$, as the reflexive transitive closure of all the actions above.

$S2[\widehat{\texttt{Head}}/\texttt{Head}, \widehat{\texttt{Tail}}/\texttt{Tail}] \rightsquigarrow S3 \wedge \widehat{\texttt{Head}} = \texttt{Head} \wedge \widehat{\texttt{Tail}} = \texttt{Tail}$
Attach a *data node* to the tail of a list when the list is in $S2$

$S3[\widehat{\texttt{Head}}/\texttt{Head}, \widehat{\texttt{Tail}}/\texttt{Tail}] \rightsquigarrow$
$S2 \wedge \widehat{\texttt{Head}} = \texttt{Head} \wedge \widehat{\texttt{Tail}}.\texttt{next} = \texttt{Tail}$
Move `Tail` to its next node when the list is in $S3$

$S3[\widehat{\texttt{Head}}/\texttt{Head}, \widehat{\texttt{Tail}}/\texttt{Tail}] \rightsquigarrow S5 \wedge \widehat{\texttt{Head}} = \texttt{Head} \wedge \widehat{\texttt{Tail}} = \texttt{Tail}$
Match the first *unmatched data node* when the list is in $S3$

$S5[\widehat{\texttt{Head}}/\texttt{Head}, \widehat{\texttt{Tail}}/\texttt{Tail}] \rightsquigarrow$
$\exists z.\mathsf{QNd}(z, -, D, \texttt{Head}) * S3 \wedge \widehat{\texttt{Head}} = z \wedge \widehat{\texttt{Tail}} = \texttt{Tail}$
Move `Head` to its next node when the list is in $S5$

$S5[\widehat{\texttt{Head}}/\texttt{Head}, \widehat{\texttt{Tail}}/\texttt{Tail}] \rightsquigarrow$
$S4 \wedge \widehat{\texttt{Head}} = \texttt{Head} \wedge \widehat{\texttt{Tail}}.\texttt{next} = \texttt{Tail}$
Move *Tail* to its next node when the list is in $S5$

$S4[\widehat{\texttt{Head}}/\texttt{Head}, \widehat{\texttt{Tail}}/\texttt{Tail}] \rightsquigarrow S5 \wedge \widehat{\texttt{Head}} = \texttt{Head} \wedge \widehat{\texttt{Tail}} = \texttt{Tail}$
Attach a *data node* to the tail of a list when the list is in $S4$

$S4[\widehat{\texttt{Head}}/\texttt{Head}, \widehat{\texttt{Tail}}/\texttt{Tail}] \rightsquigarrow$
$\exists z.\mathsf{QNd}(z, -, D, \texttt{Head}) * S2 \wedge \widehat{\texttt{Head}} = z \wedge \widehat{\texttt{Tail}} = \texttt{Tail}$
Move `Head` to its next node when the list is in $S4$

$S2[\widehat{\texttt{Head}}/\texttt{Head}, \widehat{\texttt{Tail}}/\texttt{Tail}] \rightsquigarrow$
$S4 \wedge \widehat{\texttt{Head}}.\texttt{next} = \texttt{Head} \wedge \widehat{\texttt{Tail}} = \texttt{Tail}$
Match the first *unmatched data node* when the list is in $S3$

Fig. 4. State Transitions Rules for the Queue Mode

In addition, all the actions in $\mathcal{R}/\mathcal{G}$ defined in Figure 2 are *precise*. The portion of state that remain unchanged is not depicted in the actions, which satisfy our requirement at Section 2.1. We use an invariant I to describe the whole shared state: $I = \boxed{S_2 \cup S_3 \ldots \cup S_9 * \mathbf{true}}$.

In fact, the detailed state of the shared memory may be affected by GC, because GC may periodically (or by other manners) reclaim nodes that are not accessible to any thread. However, this makes no trouble, since we do not introduce any auxiliary variable, GC will not affect the validity of our assertions. In fact, GC may shrink some part of the heap originally covered by the **true** part in invariant I, but this makes no effect to our $\mathcal{R}$ condition. For this algorithm, GC may only reclaim nodes that previously in the queue but not still. This discussion tells us, no thread will notice the state modification caused by GC, and the definition of $\mathcal{R}$ is thus reasonable for SSQ.

2. **Deduced Properties (Queue Mode)**
Since the rely condition depicts all *actions* in the shared state (except for node reclaimed by GC), the rest of the shared state will remain the same on the assertion level. The following facts are trivially true:

Property 1. Under GC and the rely condition $\mathcal{R}$ defined in section 4.2, in the SSQ, the `state` field of a node will not change until it is reclaimed by GC.

Property 2. Under GC and the rely condition $\mathcal{R}$ defined in section 4.2, in the SSQ, if the ***next*** field of a node is not null, it will not change until it is reclaimed by GC .

Property 3. Under GC and the rely condition $\mathcal{R}$ defined in section 4.2, in the SSQ, the ***value*** field of a matched data or request node will not change until it is reclaimed by GC.

There are many interesting properties that can deduced from $\mathcal{R}$, but we do not want to mention the others here. We only list the properties used in our proof. For example, as mentioned in Section 2.2, $\mathsf{rset}(x, s)$ is volatile when x refers to some shared variable; however, because of Property 2, the `next` field will not redirect to other node in this algorithm. Two rules for assignment in the algorithm are proposed based on that consideration.

Theorem 2 (Assignment Rules for SSQ). *Under GC and the rely condition $\mathcal{R}$ defined in* Section 4.2*, the following tuples hold for* SSQ algorithm*:*

$$\mathcal{R}, \mathcal{G}, Fd \vdash \{\boxed{\mathsf{ls}(\mathtt{Head}, \mathtt{Tail}) * \mathbf{true}} * \mathbf{true}\}\ h := \mathtt{Head}$$
$$\{\boxed{\mathsf{ls}(h, \mathtt{Head}) * \mathbf{true}} * \boxed{\mathsf{ls}(\mathtt{Head}, \mathtt{Tail}) * \mathbf{true}} * \mathbf{true}\}$$
$$\mathcal{R}, \mathcal{G}, Fd \vdash \{\boxed{\mathsf{ls}(\mathtt{Head}, \mathtt{Tail}) * \mathbf{true}} * \mathbf{true}\}\ t := \mathtt{Tail}$$
$$\{\boxed{\mathsf{ls}(t, \mathtt{Tail}) * \mathbf{true}} * \boxed{\mathsf{ls}(\mathtt{Head}, \mathtt{Tail}) * \mathbf{true}} * \mathbf{true}\}$$

Proof. This theorem is sound based on the assignment rule under GC in Section 2.2 and property 2.

3. **The Guarantee Condition (Queue Mode)**
   The guarantee condition specifies the shared state transitions caused by current thread. As introduced in Section 2.1, the rely and guarantee conditions should be *compatible* for a concurrent system.
   The behavior of the `enqueue` method includes: attaching a *data node* to `Tail` when `Tail.next` = `null`; moving `Tail` to its next node when `Tail.next` ≠ `null`; matching an *unmatched request node* that referred by `Head.next`; ....
   More importantly, each of these *actions* caused by `enqueue` is also an action included in $\mathcal{R}$ defined in Section 4.2. Based on this recognition, we define $\mathcal{G} = \mathcal{R}$, which are sure *compatible*.

Thread's Shared and Private States (Queue Mode)
In RGL, each thread in a system has a private state and can access the shared state. As discussed in Section 4.2, now the shared state is the list of nodes in queue, which is depicted by invariant I. Particularly, the state is composed with three parts: list segment $\mathsf{ls}(\mathtt{Head}, \mathtt{Tail})$; the node that is about to be taken into the queue when `Tail.next` refers to it; the nodes that once in the queue but do not in the segment $\mathsf{ls}(\mathtt{Head}, \mathtt{Tail})$ as `Head` and `Tail` move and not reclaimed by GC yet. The first two parts are depicted by $S_2 \cup S_3 \ldots \cup S_9$ in I, and the third part is depicted by **true** in I.

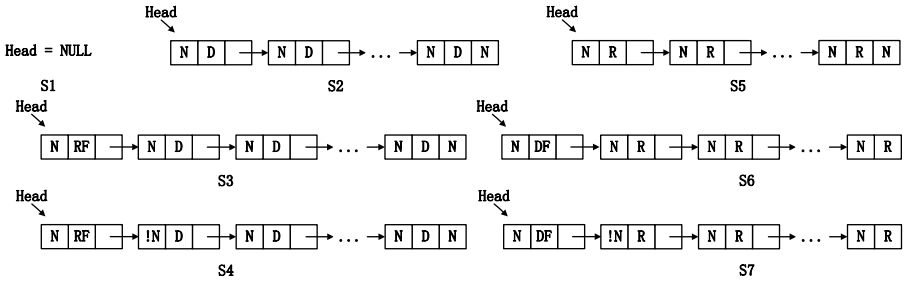

Fig. 5. The Stack's States

The private part of the thread is the node that referred by `offer` before it is attached to the queue in the shared part. Note the 1st and 6th actions in Figure 4, the post assertion's footprint of them is one node larger than their pre counterpart. This is because the node that originally belongs to the private part, is attached to the queue and belongs to the shared part after these transitions.

4.3 Proof of the Stack Mode

Predicates for State Assertions (Stack Mode)

As proof of the queue mode, predicates for states of the stack mode are defined first. Node for the stack mode has an additional `match` field to indicate which node is matched. A *matched node* is a node whose `match` filed is not null. Therefore, we define assertion for a stack node as follows:

$$\mathsf{SNd}(x, n, s, y) \mathrel{\widehat{=}} x \mapsto \{\texttt{match} \mapsto n, \texttt{state} \mapsto s, \texttt{next} \mapsto y\}$$

where `state` could be `DATA`, `REQ`, `DATA_F`, or `REQ_F`. Note that we omit `data` field because it will not affect our proof.

There are totally seven states of the shared stack, all of which are illustrated in figure 5. Note that in that figure, DF or RF additionally represent the state of a node is `DATA_F` or `REQ_F`, which means that a *data/request node* is trying to match with a *request/data node*, while `F` means fulfilling for short. Unlike figure 1, the fist field of a node is `match`, but not `data`.

We define some predicates that depict stack segments, or special types of stack segments:

$$\begin{aligned}
\mathsf{sk}(x) &\mathrel{\widehat{=}} (\mathbf{emp} \wedge x = \texttt{null}) \vee (\mathsf{SNd}(x, -, -, x.\texttt{next}) * \mathsf{sk}(x.\texttt{next})) \\
\mathsf{Dsk}(x) &\mathrel{\widehat{=}} (\mathbf{emp} \wedge x = \texttt{null}) \vee (\mathsf{SNd}(x, \texttt{null}, \texttt{DATA}, x.\texttt{next}) * \mathsf{Dsk}(x.\texttt{next})) \\
\mathsf{Rsk}(x) &\mathrel{\widehat{=}} (\mathbf{emp} \wedge x = \texttt{null}) \vee (\mathsf{SNd}(x, \texttt{null}, \texttt{REQ}, x.\texttt{next}) * \mathsf{Rsk}(x.\texttt{next}))
\end{aligned}$$

Where $\mathsf{sk}(x)$ asserts an empty stack or a stack with its top node pointed by x; $\mathsf{Dsk}(x)$ additionally asserts that each node in the stack is *unmatched data node*; $\mathsf{Rsk}(x)$ asserts that each node in the stack is *unmatched request node*. Specifications of the seven states are given in Figure 6.

$S_1 = \mathbf{emp} \wedge \mathtt{Head} = \mathtt{null}$
$S_2 = \mathsf{Dsk}(\mathtt{Head})$
$S_3 = \exists p.\mathsf{SNd}(\mathtt{Head}, \mathtt{null}, \mathtt{REQ\_F}, p) * \mathsf{Dsk}(p)$
$S_4 = \exists p, pn.\mathsf{SNd}(\mathtt{Head}, \mathtt{null}, \mathtt{REQ\_F}, p) * \mathsf{SNd}(p, !\mathtt{null}, \mathtt{DATA}, pn) * \mathsf{Dsk}(pn)$
$S_5 = \mathsf{Rsk}(\mathtt{Head})$
$S_6 = \exists p.\mathsf{SNd}(\mathtt{Head}, \mathtt{null}, \mathtt{DATA\_F}, p) * \mathsf{Rsk}(p)$
$S_7 = \exists p, pn.\mathsf{SNd}(\mathtt{Head}, \mathtt{null}, \mathtt{DATA\_F}, p) * \mathsf{SNd}(p, !\mathtt{null}, \mathtt{REQ}, p) * \mathsf{Rsk}(pn)$

Fig. 6. Share States the Stack Mode

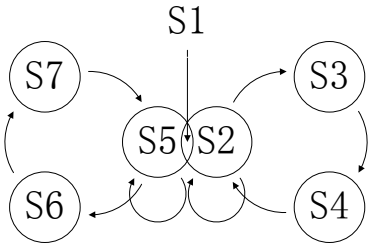

Fig. 7. Transition Graph of Shared States – Stack Mode

$S2[\widehat{\mathtt{Head}}/\mathtt{Head}, \widehat{\mathtt{Tail}}/\mathtt{Tail}] \rightsquigarrow S2 \wedge \widehat{\mathtt{Head}} = \mathtt{Head.next}$
Attach a *data node* to the top of a stack when the list is in $S2$
$S2[\widehat{\mathtt{Head}}/\mathtt{Head}, \widehat{\mathtt{Tail}}/\mathtt{Tail}] \rightsquigarrow S3 \wedge \widehat{\mathtt{Head}} = \mathtt{Head.next}$
Attach a *request node* to the top of a stack when the list is in $S2$
$S3[\widehat{\mathtt{Head}}/\mathtt{Head}, \widehat{\mathtt{Tail}}/\mathtt{Tail}] \rightsquigarrow S4 \wedge \widehat{\mathtt{Head}} = \mathtt{Head}$
Match the top two nodes in a stack when the list is in $S3$
$S4[\widehat{\mathtt{Head}}/\mathtt{Head}, \widehat{\mathtt{Tail}}/\mathtt{Tail}] \rightsquigarrow$
$S2 \wedge \exists p.\widehat{\mathtt{Head}}.\mathtt{next} = p \wedge p.\mathtt{next} = \mathtt{Head}$
Pop the top two nodes in a stack when the list is in $S5$

Fig. 8. State Transitions Rules for the Stack Mode

The Rely/Guarantee Conditions (Stack Mode)

As proof of the queue mode, we dig out the rely/guarantee conditions for the stack mode in this section. In the stack mode, the shared state of this system is the stack which can only be accessed by the shared variable `Head`. The shared state transitions are shown in Figure 7.

Figure 8 shows actions that depict the state transitions between $S2$, $S3$, and $S4$ in figure 7, while state transitions between $S5$, $S6$, and $S7$ are symmetric. We define the rely condition for the stack mode, $\mathcal{R}$, as the reflexive transitive closure of all the actions above. Moreover, we define the guarantee condition, $\mathcal{G} = \mathcal{R}$.

It is worth to mention that the shared state can be depicted by the invariant, $I = \boxed{S2 \cup S3 \cup \ldots S7 * \mathbf{true}}$. In a possible scenario, many threads are contending to attach new nodes on the top of a stack, and the `next` fields of the new nodes

all point to the `Head` of the stack. However, only the thread who first set `Head` to its new node can win the contention, others will retry the attachment again. The linearisation point that a node transform from a thread's private state to the stack shared state, is only at the moment when `Head` is set to point to that node.

4.4 The Proof, and What We Find

Now we are ready to carry on the proof. We list it in Appendix B. In the proof, we need to use Hoare rules for restricted forms of jumps. Specifically, for a loop with invariant I and exit condition q, the pre-condition and postcondition for `break` are q and `false` respectively; and for `continue`, they are I and `false`, respectively.

Semantically equivalent variance. It seems that many formal verifications become, at the end, a work to re-ensure a recognized fact: the program is correct. As a work targeting an industry-respected algorithm, could we expect more? However, after this work, we can not only say the algorithm is correct, but also report some interesting features of it that are hardly detected without a formal work.

In the proof, we find that the pre/post conditions of lines 8 and 17 in Table 1, i.e., lines marked by (†) (††) in Appendix B, are the same. These mean that, if we deleted either or both of these lines, we would have an semantically equivalent algorithm. More specifically, we take line 17 in Table 1 as example. This is the second read for the shared variable `Head`. Clearly, `Head` may be modified between line 5 and line 17 by other threads, but it may also be modified after line 17 and before line 18. As a result, semantically, the read primitive in line 17 provides no useful information, neither for our proof, nor for real execution. It is the same case for line 20 in table 2.

Via emails, one of the algorithm's designer, Doug Lea, agrees with us that the algorithm is still correct if the two lines are deleted, but he thinks the deletions will sacrifice performance. However, in our mind, this fact can only be confirmed by extensive test running on benchmarks, and may depend on the circumstance. Anyway, Lea's feedback demonstrates the power of our techniques from another side.

Now we discuss some issues related to the algorithms.

Fairness. For SSQ, *fairness* can be defined as, every call to `enqueue`/`dequeue` and `push`/`pop` will eventually return when there exist corresponding calls. Unfortunately, SSQ is sure not fair. Suppose a thread calls `enqueue`, but each time before it executes line 8 in Table 1, it is preempted by other one with a success `enqueue` call. When the thread is rescheduled the guard of line 8 is not true. In this scenario, the thread will loop forever, and its call to `enqueue` will never return.

Liveness. Since SSQ is not fair, we cannot discuss its liveness from a single thread's view. Take `enqueue` as an example, for the whole system, we define a *liveness* condition as: if $\mathsf{ls}(\texttt{Head}, \texttt{Tail})$ has at least one unmatched data node, there will be one call to `dequeue` will return; for calls to `enqueue`, we

can have a similar definition. Note, since SSQ implements a synchronous queue, for any return from `dequeue`/`enqueue`, its corresponding `enqueue`/`dequeue` returns too. Based on our rely/guarantee conditions, this property holds for SSQ. However, strict proof may need temporal logic, and a transition model of the rely/guarantee conditions [3].

Modularity. Our proof is modular in the sense that it has no restriction on the number of threads. The shared state can only be accessed by `enqueue`/`dequeue`. All calls to these procedures share the same pair of rely/guarantee conditions. The compatibility thus remains and is independent of the number of threads.

5 Related Work

People have worked on reasoning concurrent systems for decades. [17] is a pioneer paper in this area, and many work follows. [18] defined semantics of concurrent programs as a set of *execution sequences*, and a version of temporal logic for reasoning properties; Hailpern [11] proposed a way for modular verification of concurrent programs, and viewed a program as a set of modules that interact via procedure calls.

A new story in this area began around the turn of the century, after the seminal work of John C. Reynolds *et al.* on the Separation Logic (SL) [21]. Many variants of SL and relating theories were blossomed since then. One remarkable work is a model of Concurrent SL (CSL) proposed by S. Brookes [2], in which many crucial concepts related to concurrency – resource, states, semantics, etc. – were treated deeply and formally. The view of *variables-as-resource* was proposed by Bornat *et al.* [1] and Parkinson *et al.* [19], where the ownership of variables in every predicate is identified.

As a most important tool for reasoning concurrent programs, the rely-guarantee method [14] has a longer history. It is one of the best studied techniques for compositional concurrent program verification. Rely guarantee has been combined with SL by X. Feng *et al.* [8] and V. Vafeiadis *et al.* [25]. Recent theories intend to improve rely guarantee reasoning in aspects of modularity and expressiveness, as concurrent abstract predicate [6] and theories of refinement [22].

There are other works on proving lock-free algorithms. Doherty *et al.* [7] proposed verification of lock-free queue algorithms using automata model. Colvin *et al.* [4] found and fixed bugs of a lock-free algorithm with formal methods.

One character of our work is that it requires one to possess a deep understanding about how the algorithm operates, as "explore the algorithm thoroughly, and then construct a proof correspondingly". Some recent works take the similar way, e.g., Colvin [5], Parkinson [20]. There are also some works on automate verification of lock-free algorithms, e.g., Yahav [26] and Vafeiadis [24]. We would like to think about the combination of both approaches in the future.

On the another side, here we strictly proved the safety properties, and discussed its fairness and liveness. Strict proof of them seems need new techniques. A. Gotsman *et al.* [10] and M. Fu *et al.* [9] incorporated rely guarantee logic

with temporal logic and facilitate reasoning of liveness property to some extend. It is possible to incorporate their ideas with our approach too.

6 Conclusion and Future Work

In this paper, we developed an approach based on reply-guarantee reasoning, for verifying CAS-based lock-free algorithms. The contributions of the work lie in some aspects.

First, we illustrate an approach for verifying with garbage collection (GC). Second, by packaging CAS commands into control commands, we ruled out use of auxiliaries, and made the inference based on our rules more direct and clear. The soundness of our inference rules were all proved. Third, new rules for control flow commands are proposed to adapt to more scenarios. Finally, We verified an important nontrivial industry-respected algorithm which has been adopted in *java.util.concurrent*. We found some semantical equivalent variations of the algorithm in the proof, which shows further the value of this work.

As the future work, we plan to enhance our approach in the first and for reasoning CAS-based lock-free algorithms, plan to exercise our framework by proving more lock-free algorithms, and compare further it with other works. We also plan an implementation of the framework on some existing proof-assistant systems, to support further the development of CAS-based lock-free algorithms.

References

1. Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in separation logic. *Electr. Notes Theor. Comput. Sci.*, 155:247–276, 2006.
2. Stephen D. Brookes. A semantics for concurrent separation logic. In *CONCUR*, pages 16–34, 2004.
3. Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *POPL*, pages 117–126, 1983.
4. Robert Colvin and Lindsay Groves. Formal verification of an array-based nonblocking queue. In *ICECCS*, pages 507–516, 2005.
5. Robert Colvin and Lindsay Groves. A scalable lock-free stack algorithm and its verification. In *SEFM*, pages 339–348, 2007.
6. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.
7. Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, pages 97–114, 2004.
8. Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, pages 173–188, 2007.
9. Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, pages 388–402, 2010.

10. Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, pages 16–28, 2009.
11. Brent Hailpern and Susan S. Owicki. Modular verification of concurrent programs. In *POPL*, pages 322–336, 1982.
12. Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Separation logic in the presence of garbage collection. In *LICS*, 2011.
13. William N. Scherer III, Doug Lea, and Michael L. Scott. Scalable synchronous queues. *Commun. ACM*, 52(5):100–111, 2009.
14. Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
15. Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
16. Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
17. Susan S. Owicki and Divid Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
18. Susan S. Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.
19. Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in hoare logics. In *LICS*, pages 137–146, 2006.
20. Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, pages 297–302, 2007.
21. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
22. Aaron Joseph Turon and Mitchell Wand. A separation logic for refining concurrent objects. In *POPL*, pages 247–258, 2011.
23. Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.
24. Viktor Vafeiadis. Automatically proving linearizability. In *CAV*, pages 450–464, 2010.
25. Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.
26. Eran Yahav and Shmuel Sagiv. Automatically verifying concurrent queue algorithms. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.

A Proving Packaged-CAS Rules

We prove soundness of packaged-CAS commands by translating them into Generic Parallel Programming Language [23] (GPPL) and using GPPL's rules. The syntax and inference rules of GPPL are given in Figure 9. GPPL is suitable for describing parallel programs built from basic atomic actions. Command **assume**(B) checks whether B holds, and reduces to **skip** if B holds, otherwise loops forever. Now we give detailed proofs for some rules.

Theorem 3 (IF-CAS Sound). *Rule* (IF-CAS) *is sound based on the semantics of GPPL.*

$$
\begin{array}{llll}
C ::= \mathbf{skip} & \text{Skip} & \mid c & \text{Basic} \\
\quad \mid C_1; C_2 & \text{Sequential} & \mid C_1 + C_2 & \text{Choice} \\
\quad \mid C^* & \text{Looping} & \mid \langle C \rangle & \text{Atomic} \\
\quad \mid C_1 \parallel C_2 & \text{Parallel} & & \\
c ::= \mathbf{assume}(B) & \text{Assumption} & \mid \mathtt{x} := e & \text{Assignment} \\
\quad \mid \mathtt{x} := [e] & \text{Lookup} & \mid [e] := e & \text{Mutation} \\
\quad \mid \mathtt{x} := \mathtt{new}() & \text{Allocation} & \mid \mathtt{dispose}(e) & \text{Deallocation}
\end{array}
$$

$$
\frac{\mathcal{R}', \mathcal{G}' \vdash \{p'\}\, C\, \{q'\} \quad \mathcal{R} \subseteq \mathcal{R}' \quad p \Rightarrow p' \quad \mathcal{G}' \subseteq \mathcal{G} \quad q' \Rightarrow q}{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C\, \{q\}}\ \texttt{(WEAKEN)}
$$

$$
\frac{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C\, \{q\} \quad \mathbf{stable}(r, \mathcal{R} \cup \mathcal{G})}{\mathcal{R}, \mathcal{G} \vdash \{p * r\}\, C\, \{q * r\}}\ \texttt{(FRAME)}
$$

$$
\frac{\mathcal{R}, \mathcal{G} \vdash \{p_1\}\, C\, \{q\} \quad \mathcal{R}, \mathcal{G} \vdash \{p_2\}\, C\, \{q\}}{\mathcal{R}, \mathcal{G} \vdash \{p_1 \vee p_2\}\, C\, \{q\}}\ \texttt{(DISJ)}
$$

$$
\frac{x \notin \mathrm{fv}(q, C, \mathcal{R}, \mathcal{G}) \quad \mathcal{R}, \mathcal{G} \vdash \{p\}\, C\, \{q\}}{\mathcal{R}, \mathcal{G} \vdash \{\exists x.p\}\, C\, \{q\}}\ \texttt{(EX)}
$$

$$
\frac{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C\, \{q_1\} \quad \mathcal{R}, \mathcal{G} \vdash \{p\}\, C\, \{q_2\}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C\, \{q_1 \wedge q_2\}}\ \texttt{(CONJ)}
$$

$$
\frac{x \notin \mathrm{fv}(p, C, \mathcal{R}, \mathcal{G}) \quad \mathcal{R}, \mathcal{G} \vdash \{p\}\, C\, \{q\}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C\, \{\forall x.q\}}\ \texttt{(ALL)}
$$

$$
\frac{\mathbf{stable}(p, \mathcal{R})}{\mathcal{R}, \mathcal{G} \vdash \{p\}\, \mathbf{skip}\, \{q\}}\ \texttt{(SKIP)}
$$

$$
\frac{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C\, \{p\}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C^*\, \{p\}}\ \texttt{(LOOP)}
$$

$$
\frac{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C_1\, \{r\} \quad \mathcal{R}, \mathcal{G} \vdash \{r\}\, C_2\, \{q\}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C_1; C_2\, \{q\}}\ \texttt{(SEQ)}
$$

$$
\frac{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C_1\, \{q\} \quad \mathcal{R}, \mathcal{G} \vdash \{p\}\, C_2\, \{q\}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C_1 + C_2\, \{q\}}\ \texttt{(CHOICE)}
$$

$$
\frac{\vdash_{\mathrm{SL}} \{p\}\, C\, \{q\}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\, C\, \{q\}}\ \texttt{(PRIM)}
$$

$$
\frac{\varnothing, \mathcal{G} \vdash \{p\}\, \langle C \rangle\, \{q\} \quad \mathbf{stable}(p, q, \mathcal{R})}{\mathcal{R}, \mathcal{G} \vdash \{p\}\, \langle C \rangle\, \{q\}}\ \texttt{(ATOMR)}
$$

$$
\frac{\mathbf{precise}(P, Q) \quad \mathcal{R}, \mathcal{G} \vdash \{P * P'\}\, C\, \{Q * Q'\} \quad P \rightsquigarrow Q \subseteq \mathcal{G}}{\varnothing, \mathcal{G} \vdash \{\boxed{P * F} * P'\}\, \langle C \rangle\, \{\boxed{Q * F} * Q'\}}\ \texttt{(ATOM)}
$$

Fig. 9. Syntax/Inference Rules of GPPL [23]

Proof. Firstly, we translate the packaged-CAS command into a piece of GPPL code which preserves the semantics:

$$
\begin{aligned}
&\mathbf{if}\ \mathtt{cas}(l, o, n)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \\
\equiv\ &\langle \mathbf{assume}([l] = o); [l] := n \rangle; C_1 + \mathbf{assume}([l] \neq o); C_2
\end{aligned}
$$

1. $p \Rightarrow \boxed{l \mapsto - * \mathbf{true}} * \mathbf{true}$ premise
2. $\vdash_{\text{SL}} \{p\}\, \mathbf{assume}([l] = o)\, \{p * \boxed{l \mapsto o * \mathbf{true}}\}$ 1, Separation logic
3. $\varnothing, \mathcal{G} \vdash \{p\}\, \mathbf{assume}([l] = o)\, \{p * \boxed{l \mapsto o * true}\}$ 2, PRIM
4. $\varnothing, \mathcal{G} \vdash \{p * \boxed{l \mapsto o * \mathbf{true}}\}\ [\mathrm{l}] := \mathrm{n}\, \{r_1\}$ premise
5. $\varnothing, \mathcal{G} \vdash \{p\}\, \mathbf{assume}([l] = o); [l] := n\, \{r_1\}$ 3,4, SEQ
6. $\varnothing, \mathcal{G} \vdash \{p\}\, \langle \mathbf{assume}([l] = o); [l] := n \rangle\, \{r_1\}$ 5, ATOM
7. $\mathbf{stable}(\{p, r_1, r_2\}, \mathcal{R})$ premise
8. $\mathcal{R}, \mathcal{G} \vdash \{p\}\, \langle \mathbf{assume}([l] = o); [l] := n \rangle\, \{r_1\}$ 6,7, ATOMR
9. $\mathcal{R}, \mathcal{G} \vdash \{r_1\}\, C_1\, \{q\}$ premise
10. $\mathcal{R}, \mathcal{G} \vdash \{p\}\, \langle \mathbf{assume}([l] = o); [l] := n \rangle; C_1\, \{q\}$ 8,9, SEQ
11. $\vdash_{\text{SL}} \{p\}\, \mathbf{assume}([l] \neq o)\, \{p * \boxed{l \mapsto o' * \mathbf{true} \wedge o \neq o'}\}$ 1, Separation logic
12. $p * \boxed{l \mapsto o' * \mathbf{true}} \wedge o \neq o' \Rightarrow r_2$ premise
13. $\vdash_{\text{SL}} \{p\}\, \mathbf{assume}([l] \neq o)\, \{r_2\}$ 11,12, SL-WEAKEN
14. $\varnothing, \mathcal{G} \vdash \{p\}\, \mathbf{assume}([l] \neq o)\, \{r_2\}$ 13, PRIM
15. $\mathcal{R}, \mathcal{G} \vdash \{p\}\, \mathbf{assume}([l] \neq o)\, \{r_2\}$ 7,14, ATOMR
16. $\mathcal{R}, \mathcal{G} \vdash \{r_2\}\, C_2\, \{q\}$ premise
17. $\mathcal{R}, \mathcal{G} \vdash \{p\}\, \mathbf{assume}([l] \neq o); C_2\, \{q\}$ 15,16, SEQ
18. $\mathcal{R}, \mathcal{G} \vdash \{p\}\, \langle \mathbf{assume}([l] = o); [l] := n \rangle; C_1 + \mathbf{assume}([l] \neq o); C_2\, \{q\}$ 10,17, CHOICE

Fig. 10. Proof of Rule (IF-CAS)

The packaged-IF-CAS can be viewed as syntactic sugared form of the GPPL codes. Now we need to prove the soundness of following rule:

$$\frac{\begin{array}{c}\varnothing, \mathcal{G} \vdash \{p * \boxed{l \mapsto o * \mathbf{true}}\}\ [l] := n\ \{r_1\} \\ p * \boxed{l \mapsto o' * \mathbf{true}} \wedge o' \neq o \Rightarrow r_2 \\ \mathcal{R}, \mathcal{G} \vdash \{r_1\}\ C_1\ \{q\} \quad \mathcal{R}, \mathcal{G} \vdash \{r_2\}\ C_2\ \{q\} \\ p \Rightarrow \boxed{l \mapsto - * \mathbf{true}} * \mathbf{true} \quad \mathbf{stable}(\{p, q, r_1, r_2\}, \mathcal{R})\end{array}}{\begin{array}{c}\mathcal{R}, \mathcal{G} \vdash \{p\}\ \langle \mathbf{assume}([l] = o); [l] := n \rangle; C_1 \\ + \mathbf{assume}([l] \neq o); C_2\ \{q\}\end{array}}$$

To begin with, we know that if l is a shared location in the footprint of p, $\{p\}\, \mathbf{assume}([l] = o)\, \{p * \boxed{l \mapsto o * \mathbf{true}}\}$ gives the axiomatic semantics of **assume** in Separation Logic. It is intuitively sound based on the operational semantics of **assume**: if the initial state, say $\sigma \not\models \boxed{l \mapsto o * \mathbf{true}} * \mathbf{true}$, the **assume** primitive loops forever, and the triple holds trivially; otherwise, the **assume** terminates without changes the state, $\sigma \models p * \boxed{l \mapsto o * \mathbf{true}}$ holds because $\sigma \models p$ and $\sigma \models \boxed{l \mapsto o * \mathbf{true}} * \mathbf{true}$. Symmetrically, $\{p\}\, \mathbf{assume}([l] \neq o)\, \{p * \boxed{l \mapsto o' * \mathbf{true} \wedge o \neq o'}\}$.

We list the proof in Figure 10. For the rule PRIM used in line 3 and 14, we give some explanation in footnote[1].

Theorem 4 (WHILE-N-CAS Sound). *Rule* (WHILE-N-CAS) *is sound based on the semantics of GPPL.*

[1] The PRIM rule in [23] requires primitive commands do not access the shared state, which is not suitable in many occasions. To preserve soundness, we restrict $\mathcal{R}$ to be $\varnothing$ when the primitive command access the shared state.

1.	$\vdash_{\mathrm{SL}} \{p\}\ \mathbf{assume}(B)\ \{p \wedge B\}$	Separation Logic
2.	$p \wedge B \Rightarrow r_1$	premise
3.	$\vdash_{\mathrm{SL}} \{p\}\ \mathbf{assume}(B)\ \{r_1\}$	1,2, Separation Logic
4.	$\varnothing, \mathcal{G} \vdash \{p\}\ \mathbf{assume}(B)\ \{r_1\}$	3, PRIM
5.	$\mathbf{stable}(\{p, r_1, r_2\}, \mathcal{R})$	premise
6.	$\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{assume}(B)\ \{r_1\}$	4,5, ATOMR
7.	$\mathcal{R}, \mathcal{G} \vdash \{r_1\}\, C_1\, \{q\}$	premise
8.	$\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{assume}(B); C_1\ \{q\}$	6,7, SEQ
9.	$\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{assume}(\neg B); C_2\ \{q\}$	similarly proved as 8
10.	$\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{assume}(B); C_1 + \mathbf{assume}(\neg B); C_2\ \{q\}$	8,9, CHOICE

Fig. 11. Proof of Rule (IF)

Proof. (Sketch) The proof of (WHILE-N-CAS) is the same as proof for (IF-CAS) in general. The packaged command is translated as follows:

$$\begin{aligned} &\mathbf{do}\ C\ \mathbf{while}\ !\mathtt{cas}(l, o, n) \\ \equiv\ &(C; \mathbf{assume}([l] \neq o))^*; C; \langle \mathbf{assume}([l] = o); [l] := n \rangle \end{aligned}$$

Here we need to prove:

$$\forall n \geq 0. \frac{\begin{array}{c} \mathcal{R}, \mathcal{G} \vdash \{p\}\, C\, \{r\} \quad p \Rightarrow \boxed{l \mapsto - * \mathbf{true}} * \mathbf{true} \\ \varnothing, \mathcal{G} \vdash \{r * \boxed{l \mapsto o * \mathbf{true}}\}\ [\mathrm{l}] := \mathrm{n}\ \{q\} \\ r * \boxed{l \mapsto o' * \mathbf{true}} \wedge o' \neq o \Rightarrow p \\ \mathbf{stable}(\{p, q, r\}, \mathcal{R}) \end{array}}{\begin{array}{r} \mathcal{R}, \mathcal{G} \vdash \{p\}\, (C; \mathbf{assume}([l] \neq o))^n; C; \\ \langle \mathbf{assume}([l] = o); [l] := n \rangle\ \{q\} \end{array}}$$

This can be proved by induction.

Theorem 5 (IF Soundness). *Rule* (IF) *is sound based on the semantics of GPPL.*

Proof. The packaged command is translated as follows:

$$\begin{aligned} &\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \\ \equiv\ &\mathbf{assume}(B); C_1 + \mathbf{assume}(\neg B); C_2 \end{aligned}$$

The task is to prove:

$$\frac{\begin{array}{l} p \wedge B \Rightarrow r_1 \quad p \wedge \neg B \Rightarrow r_2 \quad \mathbf{stable}(\{r_1, r_2, p\}, \mathcal{R}) \\ \mathcal{R}, \mathcal{G} \vdash \{r_1\}\ C_1\ \{q\} \quad \mathcal{R}, \mathcal{G} \vdash \{r_2\}\ C_2\ \{q\} \end{array}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{assume}(B); C_1 + \mathbf{assume}(\neg B); C_2\ \{q\}}$$

As previous proofs, we assume $\vdash_{\mathrm{SL}} \{p\}\ \mathbf{assume}(B)\ \{p \wedge B\}$ defines the axiomatic semantics of the primitive **assume** in separation logic.

Proof are listed in Figure 11.

Theorem 6 (WHILE Soundness). *Rule* (WHILE) *is sound based on the semantics of GPPL.*

Base Case: $n = 0$, trivially holds
Inductive Step:

1.	$\{p\}\ (\mathbf{assume}(B); C)^n\ \{p\}$	Inductive Assumption
2.	$\vdash_{\mathrm{SL}} \{p\}\ \mathbf{assume}(B)\ \{p \wedge B\}$	Separation Logic
3.	$p \wedge B \Rightarrow r$	premise
4.	$\vdash_{\mathrm{SL}} \{p\}\ \mathbf{assume}(B)\ \{r\}$	2,3, Separation Logic
5.	$\varnothing, \mathcal{G} \vdash \{p\}\ \mathbf{assume}(B)\ \{r\}$	4, PRIM
6.	$\mathbf{stable}(\{p, r\}, \mathcal{R})$	premise
7.	$\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{assume}(B)\ \{r\}$	5,6, ATOMR
8.	$\mathcal{R}, \mathcal{G} \vdash \{r\}\ C\ \{p\}$	premise
9.	$\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{assume(B)}; C\ \{p\}$	7,8, SEQ
10.	$\mathcal{R}, \mathcal{G} \vdash \{p\}\ (\mathbf{assume(B)}; C)^{n+1}\ \{p\}$	1,9, SEQ

Fig. 12. Proof of ($\star$)

Proof. The packaged command is translated as follows:

$$\begin{aligned} &\mathbf{while}\ B\ \mathbf{do}\ C \\ \equiv\ &(\mathbf{assume}(B); C)^*; \mathbf{assume}(\neg B) \end{aligned}$$

The task is to prove:

$$\frac{\begin{array}{c} p \wedge B \Rightarrow r \quad p \wedge \neg B \Rightarrow q \\ \mathcal{R}, \mathcal{G} \vdash \{r\}\ C\ \{p\} \quad \mathbf{stable}(\{p, q\}, \mathcal{R}) \end{array}}{\mathcal{R}, \mathcal{G} \vdash \{p\}\ (\mathbf{assume}(B); C)^*; \mathbf{assume}(\neg B)\ \{q\}}$$

Further, the task is decomposed as:

$$\forall n.\ \frac{\cdots}{\mathcal{R}, \mathcal{G} \vdash \{p\}\ (\mathbf{assume}(B); C)^n\ \{p\}} \qquad (\star)$$

$$\frac{\cdots}{\mathcal{R}, \mathcal{G} \vdash \{p\}\ \mathbf{assume}(\neg B)\ \{q\}} \qquad (\star\star)$$

Proof of ($\star\star$) is trivial, and inductive proof of ($\star$) is listed in Figure 12.

B Proof Listing

Proof of the Queue Mode (Enqueue)

Based on $\mathcal{R}$ and $\mathcal{G}$ defined in Section 4.2, and rules for packaged-CAS-command and assignment under GC, we list the whole proof for synchronous queue of queue mode, where the invariant is $I = \boxed{S_2 \cup S_3 \ldots \cup S_9 * \mathbf{true}}$.

$\{I\}$
`offer = new Node(`e`, DATA);`
$\{I * \mathsf{Nd}(\texttt{offer}, e, \texttt{DATA}, \texttt{null})\}$
`while true {`
  $\{I * \mathsf{Nd}(\texttt{offer}, e, \texttt{DATA}, \texttt{null})\}$
  `t = Tail;`
  `h = Head;`
  $\{I * \mathsf{Nd}(\texttt{offer}, e, \texttt{DATA}, \texttt{null})$

```
   * ⌈ls(t, Tail) * true⌉ * ⌈ls(h, Head) * true⌉}                    ≡ G
if (h == t || t.state == DATA) {
   {G ∧ (h = t ∨ t.state = DATA)}                                    ≡ H
   n = t.next;
   {H * (n = null ∨ ⌈Nd(t, −, −, n) * Nd(n, −, −, null) * true⌉)}
                                                                     (1)
   if (t == Tail){                                                   (†)
      if (n != null) {
         {H * ⌈Nd(t, −, −, n) * Nd(n, −, −, null) * true⌉}
         casTail(t, n);                                              (2)
         {I * Nd(offer, e, DATA, null)}
      }
      else
         {H ∧ n = null}
         if(t.casNext(n, offer)) {
            {I * ⌈ls(h, Head) * true⌉
               * ⌈Nd(t, −, −, offer)*
                  Nd(offer, −, DATA, −) * true⌉}                     (3)
            casTail(t, offer);
            {I * ⌈Nd(offer, −, DATA, −) * true⌉
               * ⌈ls(h, Head) * true⌉}
            while (offer.date != null));
               /* spin */;
            {I * ⌈Nd(offer, null, DATA, −) * true⌉
               * ⌈Nd(h, Head) * true⌉}                              (4)
            h = head;                                                (††)
            {I * ⌈Nd(offer, null, DATA, −) * true⌉
               * ⌈ls(h, Head) * true⌉}
            if (offer == h.next)
               {I * ⌈Nd(Head, −, −, offer) *
                     Nd(offer, null, DATA, −) * true⌉}
               casHead(h, offer);
            {I}
            return;
         }
   }
} else{
   {G ∧ h! = t ∧ t.state = REQ}                                      ≡ J
   n = h.next;
   {J * (n = null ∨ ⌈Nd(h, −, −, n) * Nd(n, −, −, −) * true⌉)}
   if(t != Tail || h != Head || n == null)                           (5)
      continue;
   {J * ⌈Nd(h, −, −, n) * Nd(n, −, REQ, null) * true⌉}               (6)
   if(n.casData(null, e)) {
      {J * ⌈Nd(h, −, −, n) * Nd(n, e, REQ, null) * true⌉}            (7)
      casHead(h, n);
      {I}
      return;
   }else
```

$\{J * \boxed{\mathsf{Nd}(\mathtt{t}, -, -, \mathtt{n}) * \mathsf{Nd}(\mathtt{n}, !\mathtt{null}, \mathtt{REQ}, \mathtt{null}) * \mathbf{true}}\}$
`casHead(h, n);`
$\{I * \mathsf{Nd}(\mathtt{offer}, e, \mathtt{DATA}, \mathtt{null})\}$
}
}

Discussions are given to lines marked with numbers:

(1) $\mathtt{t.next} = \mathtt{n}$ is stable under $\mathcal{R}$ (Property 2).

(2) `casTail(t, n)` is translated to `if(casTail(t,n)) then skip; else skip;` and rule `IF-casTail` is applied. Note:
$\{H * \boxed{\mathsf{Nd}(\mathtt{t}, -, -, \mathtt{n}) * \mathsf{Nd}(\mathtt{n}, -, -, \mathtt{null}) * \mathbf{true}} \wedge$
$\mathtt{Tail} = \mathtt{t}\}$
$\mathtt{Tail} := \mathtt{n};$
$\{I * \mathsf{Nd}(\mathtt{offer}, e, \mathtt{DATA}, \mathtt{null})\}$

(3) Here, the node referred by `offer` was transformed from private state to shared state. Note that:
$\{H \wedge \mathtt{n} = \mathtt{null} \wedge \mathtt{t.next} = \mathtt{n}\}$
$\{H \wedge \mathtt{t} = \mathtt{Tail} \wedge \mathtt{Tail.next} = \mathtt{null}\}$
$\mathtt{t.next} = \mathtt{offer};$
$\{I * \boxed{\mathsf{ls}(\mathtt{h}, \mathtt{Head}) * \mathbf{true}} * \boxed{\begin{array}{l}\mathsf{Nd}(\mathtt{t}, -, -, \mathtt{offer}) * \\ \mathsf{Nd}(\mathtt{offer}, -, \mathtt{DATA}, -) * \mathbf{true}\end{array}}\}$
$\mathtt{offer.state} = \mathtt{DATA}$ is stable under $\mathcal{R}$ (Property 1); $\mathtt{t.next} = \mathtt{offer}$ is stable under $\mathcal{R}$ (Property 2).

(4) The $\mathtt{offer.data} = \mathtt{null}$ is stable under $\mathcal{R}$ (Property 3).

(5) Only $\mathtt{n} = \mathtt{null}$ is stable under $\mathcal{R}$ because `n` is private. While the other $\mathtt{t}! = \mathtt{Tail}$ and $\mathtt{h}! = \mathtt{Head}$ are not stable under $\mathcal{R}$ because `Head` and `Tail` are shared.

(6) This assertion is deduced from one branch of the upper if-clause other than `continue`. For `continue` branch, note that post-condition of `continue` is `false`.

(7) The $\mathtt{n.data} = e$ is stable under $\mathcal{R}$ (Property 3).

Besides, the algorithm is semantically equivalent if we delete the two lines that marked (†) (with corresponding close-brace) and (††). We find this fact because the pre- and post-conditions of these lines are the same. The fact has been confirmed with the algorithm's designer.

Proof of the Stack Mode (Push)

Based on the rely/guarantee conditions that defined in section 4.3, we list proof of synchronous queue for the stack mode in here, where the invariant $I = \boxed{S2 \cup S3 \cup \ldots S7 * \mathbf{true}}$.

$\{I\}$
`d = new Node(e, DATA);`
$\{I * \mathsf{SNd}(d, \mathtt{null}, \mathtt{DATA}, \mathtt{null})\}$
`while true {`
$\{I * \mathsf{SNd}(d, \mathtt{null}, \mathtt{DATA}, -)\}$
`h = Head;`

$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * (h = \texttt{null} \vee \boxed{\mathsf{SNd}(h, -, -, -) * \mathbf{true}})\}$
`if (h == null || h.state == DATA) {`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * (h = \texttt{null} \vee \boxed{\mathsf{SNd}(h, -, \texttt{DATA}, -) * \mathbf{true}})\}$
`d.next = h;`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, h) * (h = \texttt{null} \vee \boxed{\mathsf{SNd}(h, -, \texttt{DATA}, -) * \mathbf{true}})\}$
`if (!casHead(h, d))`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -)\}$
`continue;`
$\{I * \boxed{\mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, h) * \mathbf{true}}\}$
`while (d.match == null)`
`/* spin */;`
$\{I * \boxed{\mathsf{SNd}(d, !\texttt{null}, \texttt{DATA}, h) * \mathbf{true}}\}$
`h = Head;`
$\{I * \boxed{\mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * \mathbf{true}} * (h = \texttt{null} \vee \boxed{\mathsf{SNd}(h, -, \texttt{DATA}, -) * \mathbf{true}})\}$
`if(h != null && h.next = d)`
$\{I * \boxed{\mathsf{SNd}(h, -, \texttt{DATA}, d) * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * \mathbf{true}})\}$
`casHead(h, d.next);`
$\{I\}$
$\{I\}$
`return;`
`} else if (h.state = REQ){`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * \boxed{\mathsf{SNd}(h, -, \texttt{REQ}, -) * \mathbf{true}}\}$
`f = new Node(e, DATA_F, h);`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * \mathsf{SNd}(f, \texttt{null}, \texttt{DATA\_F}, h) * \boxed{\mathsf{SNd}(h, -, \texttt{REQ}, -) * \mathbf{true}}\}$
`if (!casHead(h, f))`
`continue;`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * \boxed{\mathsf{SNd}(f, \texttt{null}, \texttt{DATA\_F}, h) * \mathsf{SNd}(h, -, \texttt{REQ}, -) * \mathbf{true}}\}$
`h = f.next;` (†††)
`n = h.next;`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * \boxed{\mathsf{SNd}(f, \texttt{null}, \texttt{DATA\_F}, h) * \mathsf{SNd}(h, -, \texttt{REQ}, n) * \mathbf{true}}\}$
`h.casMatch(null, f);`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * \boxed{\mathsf{SNd}(f, \texttt{null}, \texttt{DATA\_F}, h) * \mathsf{SNd}(h, !\texttt{null}, \texttt{REQ}, n) * \mathbf{true}}\}$
`casHead(f, n);`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -)\}$
`return;`
`} else {`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * \boxed{\mathsf{SNd}(h, -, \texttt{DATA\_F} \vee \texttt{REQ\_F}, -) * \mathbf{true}}\}$
`n = h.next;`
`nn = n.next;`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * \boxed{\mathsf{SNd}(h, -, \texttt{DATA\_F} \vee \texttt{REQ\_F}, n) * \mathsf{SNd}(n, -, -, nn) * \mathbf{true}}\}$
`n.casMatch(null, h)`
$\{I * \mathsf{SNd}(d, \texttt{null}, \texttt{DATA}, -) * \boxed{\mathsf{SNd}(h, -, \texttt{DATA\_F} \vee \texttt{REQ\_F}, n) * \mathsf{SNd}(n, !\texttt{null}, -, nn) * \mathbf{true}}\}$
`casHead(h, nn);`
$\{I * \mathsf{SNd}(d, e, \texttt{DATA}, -)\}$
`}`
`}`

Note that the algorithm is also semantically equivalent if the line that marked by (†††) is deleted. The reason is that the pre- and post-conditions of that line are the same.