# Verifying Concurrent Memory Reclamation Algorithms with Grace

Alexey Gotsman, Noam Rinetzky, and Hongseok Yang

[1] IMDEA Software Institute
[2] Tel-Aviv University
[3] University of Oxford

**Abstract.** Memory management is one of the most complex aspects of modern concurrent algorithms, and various techniques proposed for it—such as hazard pointers, read-copy-update and epoch-based reclamation—have proved very challenging for formal reasoning. In this paper, we show that different memory reclamation techniques actually rely on the same implicit synchronisation pattern, not clearly reflected in the code, but only in the form of assertions used to argue its correctness. The pattern is based on the key concept of a *grace period*, during which a thread can access certain shared memory cells without fear that they get deallocated. We propose a modular reasoning method, motivated by the pattern, that handles all three of the above memory reclamation techniques in a uniform way. By explicating their fundamental core, our method achieves clean and simple proofs, scaling even to realistic implementations of the algorithms without a significant increase in proof complexity. We formalise the method using a combination of separation logic and temporal logic and use it to verify example instantiations of the three approaches to memory reclamation.

## 1 Introduction

Non-blocking synchronisation is a style of concurrent programming that avoids the blocking inherent to lock-based mutual exclusion. Instead, it uses low-level synchronisation techniques, such as compare-and-swap operations, that lead to more complex algorithms, but provide a better performance in the presence of high contention among threads. Non-blocking synchronisation is primarily employed by concurrent implementations of data structures, such as stacks, queues, linked lists and hash tables.

Reasoning about concurrent programs is generally difficult, because of the need to consider all possible interactions between concurrently executing threads. This is especially true for non-blocking algorithms, where threads interact in subtle ways through dynamically-allocated data structures. In the last few years, great progress has been made in addressing this challenge. We now have a number of logics and automatic tools that combat the complexity of non-blocking algorithms by verifying them *thread-modularly*, i.e., by considering every thread in an algorithm in isolation under some assumptions on its environment and thus avoiding explicit reasoning about all thread interactions. Not only have such efforts increased our confidence in the correctness of the algorithms, but they have often resulted in human-understandable proofs that elucidated the core design principles behind these algorithms.

However, one area of non-blocking concurrency has so far resisted attempts to give proofs with such characteristics—that of *memory management*. By their very nature, non-blocking algorithms allow access to memory cells while they are being updated by concurrent threads. Such optimistic access makes memory management one of the most complex aspects of the algorithms, as it becomes very difficult to decide when it is safe to reclaim a memory cell. Incorrect decisions can lead to errors such as memory access violations, corruption of shared data and return of incorrect results. To avoid this, an algorithm needs to include a protocol for coordinating between threads accessing the shared data structure and those trying to reclaim its nodes. Relying on garbage collection is not always an option, since non-blocking algorithms are often implemented in languages without it, such as C/C++.

In recent years, several different methods for explicit memory reclamation in non-blocking algorithms have been proposed:

– ***Hazard pointers*** [11] let a thread publish the address of a node it is accessing as a special global pointer. Another thread wishing to reclaim the node first checks the hazard pointers of all threads.
– ***Read-copy-update (RCU)*** [10] lets a thread mark a series of operations it is performing on a data structure as an RCU critical section, and provides a command that waits for all threads currently in critical sections to exit them. A thread typically accesses a given node inside the same critical section, and a reclaimer waits for all threads to finish their critical sections before deallocating the node.
– ***Epoch-based reclamation*** [5] uses a special counter of epochs, approximating the global time, for quantifying how long ago a given node has been removed from the data structure. A node that has been out of the data structure for a sufficiently long time can be safely deallocated.

Despite the conceptual simplicity of the above methods, their implementations in non-blocking algorithms are extremely subtle. For example, as we explain in §2, the protocol for setting a hazard pointer is more involved than just assigning the address of the node being accessed to a global variable. Reasoning naturally about protocols so subtle is very challenging. Out of the above algorithms, only restricted implementations of hazard pointers have been verified [13, 6, 3, 15], and even in this case, the resulting proofs were very complicated (see §6 for discussion).

The memory reclamation algorithms achieve the same goal by intuitively similar means, yet are very different in details. In this paper, we show that, despite these differences, the algorithms actually rely on the same synchronisation pattern that is *implicit*—not clearly reflected in the code, but only in the form of assertions used to argue its correctness. We propose a modular reasoning method, formalising this pattern, that handles all three of the above approaches to memory reclamation in a uniform way. By explicating their fundamental core, we achieve clean and simple proofs, scaling even to realistic implementations of the algorithms without a significant increase in proof complexity.

In more detail, we reason about memory reclamation algorithms by formalising the concept of a *grace period*—the period of time during which a given thread can access certain nodes of a data structure without fear that they get deallocated. Before deallocating a node, a reclaimer needs to wait until the grace periods of all threads that could have had access to the node pass. Different approaches to memory reclamation define the grace period in a different way. However, we show that, for the three approaches above, the duration of a grace period can be characterised by a temporal formula of a fixed form "$\eta$ $\mathsf{since}$ $\mu$", e.g., "the hazard pointer has pointed to the node since the node was present in the shared data structure". This allows us to express the contract between threads accessing nodes and those trying to reclaim them by an invariant stating that a

```
int *C = new int(0);
int inc() {
  int v, *s, *n;
  n = new int;
  do {
    s = C;
    v = *s;
    *n = v+1;
  } while
    (!CAS(&C,s,n));
  // free(s);
  return v;
}
```

(a)

```
int *C = new int(0);
int *HP[N] = {0};
Set detached[N] = {∅};
int inc() {
  int v, *n, *s, *s2;
  n = new int;
  do {
    do {
      s = C;
      HP[tid-1] = s;
      s2 = C;
    } while (s != s2);
    v = *s;
    *n = v+1;
  } while(!CAS(&C,s,n));
  reclaim(s);
  return v; }
```

(b)

```
void reclaim(int *s) {
  insert(detached[tid-1],s);
  if (nondet()) return;
  Set in_use = ∅;
  while (!isEmpty(
      detached[tid-1])) {
    bool my = true;
    int *n =
      pop(detached[tid-1]);
    for (int i = 0;
        i < N && my; i++)
      if (HP[i] == n)
        my = false;
    if (my) free(n);
    else insert(in_use, n);
  }
  moveAll(detached[tid-1],
    in_use); }
```

(c)

**Fig. 1.** A shared counter: (a) an implementation leaking memory; (b)-(c) an implementation based on hazard pointers. Here `tid` gives the identifier of the current thread.

node cannot be deallocated during the corresponding grace period for any thread. The invariant enables modular reasoning: to prove the whole algorithm correct, we just need to check that separate threads respect it. Thus, a thread accessing the data structure has to establish the assertion "$\eta$ since $\mu$", ensuring that it is inside a grace period; a thread wishing to reclaim a node has to establish the *negation* of such assertions for all threads, thus showing that all grace periods for the node have passed. Different algorithms just implement code that establishes assertions of the same form in different ways.

We formalise such correctness arguments in a modular program logic, combining one of the concurrent versions of separation logic [16, 4] with temporal logic (§3). We demonstrate our reasoning method by verifying example instantiations of the three approaches to memory reclamation—hazard pointers (§4), RCU (§5) and epoch-based reclamation (§D). In particular, for RCU we provide the first specification of its interface that can be effectively used to verify common RCU-based algorithms. Due to space constraints, the development for epochs is deferred to §D. As far as we know, the only other algorithm that allows explicitly returning memory to the OS in non-blocking algorithms is the Repeat-Offender algorithm [7]. Our preliminary investigations show that our method is applicable to it as well; we leave formalisation for future work.

## 2 Informal Development

We start by presenting our reasoning method informally for hazard pointers and RCU, and illustrating the similarities between the two.

### 2.1 Running Example

As our running example, we use a counter with an increment operation `inc` that can be called concurrently by multiple threads. Despite its simplicity, the example is representative of the challenges that arise when reasoning about more complex algorithms.

The implementation shown in Figure 1a follows a typical pattern of non-blocking algorithms. The current value of the counter is kept in a heap-allocated node pointed to by the global variable `C`. To increment the counter, we allocate a new memory cell `n` (line 4), atomically read the value of `C` into a local pointer variable `s` (line 6), dereference `s` to get the value `v` of the counter (line 7), and then store `v`'s successor into `n` (line 8). At that point, we try to change `C` so that it points to `n` using an atomic *compare-and-swap* (CAS) command (line 10). A CAS takes three arguments: a memory address (e.g., `&C`), an expected value (`s`) and a new value (`n`). It atomically reads the memory address and updates it with the new value if the address contains the expected value; otherwise, it does nothing. The CAS thus succeeds only if the value of `C` is the same as it was when we read it at line 6, thus ensuring that the counter is updated correctly. If the CAS fails, we repeat the above steps all over again. The algorithm is *memory safe*, i.e., it never accesses unallocated memory cells. It is also functionally correct in the sense that every increment operation appears to take effect atomically. More formally, the counter is *linearizable* with respect to the expected sequential counter specification [8]. Unfortunately, the algorithm leaks memory, as the node replaced by the CAS is never reclaimed. It is thus not appropriate for environments without garbage collection.

**A naive fix.** One can try to prevent memory leakage by uncommenting the `free` command in line 11 of Figure 1a, so that the node previously pointed to by `C` is deallocated by the thread that changed `C`'s value (in this case we say that the thread *detached* the node). However, this violates both memory safety and linearizability. To see the former, consider two concurrent threads, one of which has just read the value $x$ of `C` at line 6, when the other executed `inc` to completion and reclaimed the node at the address $x$. When the first thread resumes at line 7 it will access an unallocated memory cell.

The algorithm also has executions where a memory fault does not happen, but `inc` just returns an incorrect value. Consider the following scenario: a thread $t_1$ running `inc` gets preempted after executing line 7 and, at that time, `C` points to a node $x$ storing $v$; a thread $t_2$ executes `inc`, deallocating the node $x$ and incrementing the counter to $v+1$; a thread $t_3$ calls `inc` and allocates $x$, recycled by the memory system; $t_3$ stores $v+2$ into $x$ and makes `C` point to it; $t_1$ wakes up, its CAS succeeds, and it sets the counter value to $v+1$, thereby decrementing it! This is a particular instance of the well-known *ABA problem*: if we read the value $A$ of a global variable and later check that it has the value $A$, we cannot conclude, in general, that in the meantime it did not change to another value $B$ and then back to $A$. The version of the algorithm without `free` in line 11 does not suffer from this problem, as it always allocates a fresh cell. This algorithm is also correct when executed in a garbage-collected environment, as in this case the node $x$ in the above scenario will not be recycled as long as $t_1$ keeps the pointer `s` to it.

## 2.2 Reasoning about Hazard Pointers

Figure 1b shows a correct implementation of `inc` with explicit memory reclamation based on *hazard pointers* [11]. We assume a fixed number of threads with identifiers from 1 to $N$. As before, the thread that detaches a node is in charge of reclaiming it. However, it delays the reclamation until it is assured that no other thread requested that the node be protected from reclamation. A thread announces a request for a node to be protected using the array `HP` of shared hazard pointers indexed by thread identifiers. Every thread is allowed to write to the entry in the array corresponding to its identifier and read all entries. To protect the location `s`, a thread writes `s` into its entry of the hazard array (line 23) and then checks that the announcement was not too late by validating that `C` still points to `s` (line 25). Once the validation succeeds, the thread is assured that the node `s` will not be deallocated as long as it keeps its hazard pointer equal to `s`. In particular, it is guaranteed that the node `s` remains allocated when executing lines 26–28, which ensures that the algorithm is memory safe. This also guarantees that, if the CAS in line 28 is successful, then `C` has not changed its value since the thread read it at line 24. This prevents the ABA problem and makes the algorithm linearizable.

The protection of a node pointed to by a hazard pointer is ensured by the behaviour of the thread that detaches it. Instead of invoking `free` directly, the latter uses the `reclaim` procedure in Figure 1c. This stores the node in a thread-local `detached` set (line 32) and occasionally performs a batched reclamation from this set (for clarity, we implemented `detached` as an abstract set, rather than a low-level data structure). To this end, the thread considers every node `n` from the set and checks that no hazard pointer points to it (lines 40–43). If the check succeeds, the node gets deallocated (line 44).

**Reasoning challenges.** The main idea of hazard pointers is simple: threads accessing the shared data structure set hazard pointers to its nodes, and threads reclaiming memory check these pointers before deallocating nodes. However, the mechanics of implementing this protocol in a non-blocking way is very subtle.

For example, when a thread $t_1$ deallocates a node $x$ at line 44, we may actually have a hazard pointer of another thread $t_2$ pointing to $x$. This can occur in the following scenario: $t_2$ reads the address $x$ from `C` at line 22 and gets preempted; $t_1$'s CAS detaches $x$ and successfully passes the check in lines 40–43; $t_2$ wakes up and sets its hazard pointer to $x$; $t_1$ deallocates $x$ at line 44. However, such situations do not violate the correctness, as the next thing $t_2$ will do is to check that `C` still points to $x$ at line 25. Provided $x$ has not yet been recycled by the memory system, this check will fail and the hazard pointer of $t_2$ will have no force. This shows that the additional check in line 25 is indispensable for the algorithm to be correct.

It is also possible that, before $t_2$ performs the check in line 25, $x$ is recycled, allocated at line 19 by another thread $t_3$ and inserted into the shared data structure at line 28. In this case, the check by $t_2$ succeeds, and the element can safely be accessed. This highlights a subtle point: when $t_3$ executes the CAS at line 28 to insert $x$, we might already have a hazard pointer pointing to $x$. This, however, does not violate correctness.

**Our approach.** We achieve a natural reasoning about hazard pointers and similar patterns by formalising the main intuitive concept in their design—that of a *grace period*. As follows from the above explanation, a thread $t$ can only be sure that a node $x$ its hazard pointer points to is not deallocated after a moment of time when both the hazard pointer was set and the node was pointed to by `C`. The grace period for the node $x$ and thread $t$ starts from this moment and lasts for as long as the thread keeps its hazard pointer pointing to $x$. Informally, this is described by the following temporal judgement:

"the hazard pointer of thread $t$ has pointed to $x$ $\underline{\text{since}}$ `C` pointed to $x$", (1)

where $\underline{\text{since}}$ is a temporal connective with the expected interpretation: both of the facts connected were true at some point, and since then, the first fact has stayed true. We can thus specify the contract between threads accessing nodes and those trying to reclaim them by the following invariant that all threads have to respect:

"for all $t$ and $x$, if the hazard pointer of thread $t$ has pointed to $x$ $\underline{\text{since}}$ `C` pointed to $x$, then $x$ is allocated." (2)

It is this invariant that justifies the safety of the access to a shared node at line 26. On the other hand, a thread that wants to deallocate $x$ when executing `reclaim` checks that the hazard pointers of other threads do not point to $x$ (lines 40–43) only after detaching the node from the shared data structure, and it keeps the node in the `detached` set until its deallocation. Thus, even though threads can set their hazard pointers to $x$ after the reclaimer executes the check in lines 40–43, they cannot do this at the same time as `C` points to $x$. Hence, when the reclaimer deallocates $x$ at line 44, we know that

> "for all $t$, `C` has not pointed to $x$ <u>since</u> the hazard pointer of $t$ did not point to $x$." (3)

Clearly, (3) is inconsistent with (1). Therefore, no thread is inside a grace period for $x$ at the time of its deallocation, and the command in line 44 does not violate invariant (2).

More formally, let us denote the property "the hazard pointer of thread $t$ points to node $x$" by $\eta_{t,x}$, "`C` points to node $x$" by $\mu_x$, and "$x$ is allocated" by $\lambda_x$. Then (1) is $(\eta_{t,x} \text{ since } \mu_x)$, (2) is $(\forall t, x.\, (\eta_{t,x} \text{ since } \mu_x) \implies \lambda_x)$, and (3) is $(\forall t.\, \neg\mu_x \text{ since } \neg\eta_{t,x})$. The combination of (1) and (3) is inconsistent due to the following tautology:

$$\forall \eta, \mu.\, (\eta \text{ since } \mu) \wedge (\neg\mu \text{ since } \neg\eta) \implies \text{false}. \tag{4}$$

The above argument justifies the memory safety of the algorithm, and (as we show in §4) the absence of memory leaks. Moreover, (2) guarantees to a thread executing `inc` that, when the CAS in line 28 succeeds, the node `s` has not been reallocated, and so the ABA problem does not occur.

We have achieved a simple reasoning about the algorithm by defining the duration of a grace period (1), the protocol all threads follow (2), and the fact a reclaimer establishes before deallocating a node (3) as temporal formulas of particular forms. We find that the above reasoning with temporal facts of these forms is applicable not only to our example, but also to uses of hazard pointers in other data structures (§B), and in fact, to completely different approaches to memory reclamation, as we now illustrate.

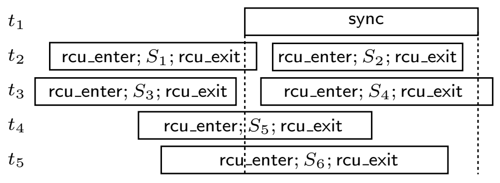
### 2.3 Reasoning about Read-Copy-Update

Read-Copy-Update (RCU) [10] is a non-standard synchronisation mechanism used in Linux to ensure safe memory deallocation in data structures with concurrent access. So far, there have been no methods for reasoning about programs with RCU. We now show that we can use our temporal reasoning principle based on grace periods to this end.

**RCU primer.** RCU provides three commands: rcu_enter, rcu_exit and sync. The rcu_enter and rcu_exit commands delimit an RCU *critical section*. They do *not* ensure mutual exclusion, so multiple threads can be in their critical sections simultaneously. Instead of enforcing mutual exclusion, RCU provides the sync command, which records the identifiers of the threads currently in critical sections and waits until all of them exit the sections. Note that if a new thread enters a critical section while sync is waiting, the command does not wait for the completion of its section. For example, when $t_1$ calls sync in the execution in Figure 2, it has to wait for critical sections $S_1$, $S_5$ and $S_6$ to finish. However, it does not wait for $S_2$ or $S_4$, as they start after sync was called.

Figure 2 shows an *abstract* implementation of the RCU primitives, formalising the above description of their semantics (for now, the reader should disregard the annotations in the figure). A *concrete* optimised RCU implementation would simulate the abstract one. Whether every thread is inside or outside an RCU critical section is determined by its entry in the `rcu` array.

```
bool rcu[N] = {0};
rcu_enter() {⟨rcu[tid-1]=1⟩RCU_tid;}
rcu_exit() {⟨rcu[tid-1]=0⟩RCU_tid;}
sync() {
  bool r[N] = {0};
  for(int i = 0; i < N; i++)
    ⟨r[i] = rcu[i]⟩Id;
  for(int i = 0; i < N; i++)
    if(r[i]) {while(⟨rcu[i]⟩Id);}
}
```



$t_1$
sync
$t_2$
rcu_enter; $S_1$; rcu_exit
rcu_enter; $S_2$; rcu_exit
$t_3$
rcu_enter; $S_3$; rcu_exit
rcu_enter; $S_4$; rcu_exit
$t_4$
rcu_enter; $S_5$; rcu_exit
$t_5$
rcu_enter; $S_6$; rcu_exit

**Fig. 2.** An abstract RCU implementation and an illustration of the semantics of sync. Blocks represent the time spans of RCU critical sections or an execution of sync.

```
1  int *C = new int(0);
2  bool rcu[N] = {0};
3  Set detached[N] = {∅};
4
5  void reclaim(int* s) {
6    insert(detached[tid-1], s);
7    if (nondet()) return;
8    sync();
9    while (!isEmpty(detached[tid]))
10      free(pop(detached[tid])); }

15 int inc() {
16   int v, *n, *s;
17   n = new int; rcu_enter();
18   do {
19     rcu_exit(); rcu_enter();
20     s = C; v = *s; *n = v+1;
21   } while (!CAS(&C,s,n));
22   rcu_exit();
23   reclaim(s);
24   return v; }
```

**Fig. 3.** Counter with RCU-based memory management

**RCU-based counter.** Figure 3 gives the implementation of the running example using RCU. Its overall structure is similar to the implementation using hazard pointers. In `inc`, we wrap an RCU critical section around the commands starting from the read of the global variable `C` at line 20 and including all memory accesses involving the value read up to the CAS at line 21. The correctness of the algorithm is ensured by having `reclaim` call sync at line 8, before deallocating the detached nodes. This blocks the thread until all critical sections that existed at the time of the call to sync finish. Since, when sync is called, the nodes to be deallocated have already been moved to the thread-local `detached` set, newly arriving `inc` operations have no way of gaining a reference to one of these nodes, which guarantees the safety of their deallocation. We can similarly argue that an ABA problem does not occur, and thus, the algorithm is linearizable. We can formulate the contract among threads as follows:

"for all $t$ and $x$, if thread $t$ has stayed in a critical section $\underline{\text{since}}$ it saw `C` pointing to $x$, then $x$ is allocated," (5)

which is of the same form as (2). Here, a grace period for a thread, specified by the 'since' clause, lasts for as long as the thread stays in its critical section. During the time span of sync, every thread passes through a point when it is not in a critical section. Hence, after executing line 8, for every node $x$ to be deallocated we know:

"for all $t$, `C` has not pointed to $x$ $\underline{\text{since}}$ $t$ was not in a critical section," (6)

which is of the same form as (3). As before, this is inconsistent with the 'since' clause of (5), which guarantees that deallocating $x$ will not violate (5).

**Pattern.** The algorithms using hazard pointers and read-copy-update fundamentally rely on the same synchronisation pattern, where a potentially harmful race between threads accessing nodes and those trying to reclaim them is avoided by establishing an assertion of the form $(\eta_{t,x}\ \mathsf{since}\ \mu_x)$ before every access, and $(\neg\mu_x\ \mathsf{since}\ \neg\eta_{t,x})$ before every deallocation. This *implicit* pattern is highlighted not by examining the syntactic structure of different memory management implementations, but by observing that the arguments about their correctness have the same form, as can be seen in our proofs.

## 3 Abstract Logic

Reasoning about highly concurrent algorithms, such as the example in §2, is convenient in logics based on rely-guarantee [9, 14], which avoids direct reasoning about all possible thread interactions in a concurrent program by specifying a relation (the *guarantee* condition) for every thread restricting how it can change the program state. For any given thread, the union of the guarantee conditions of all the other threads in the program (its *rely* condition) restricts how those threads can interfere with it, and hence, allows reasoning about this thread in isolation.

The logic we use to formalise our verification method for memory reclamation algorithms uses a variant of rely-guarantee reasoning proposed in SAGL [4] and RGSep [16]—logics for reasoning about concurrent programs that combine rely-guarantee reasoning with separation logic. These partition the program heap into several *thread-local* parts (each of which can only be accessed by a given thread) and the *shared* part (which can be accessed by all threads). The partitioning is defined by proofs in the logic: an assertion in the code of a thread restricts its local state and the shared state. Thus, while reasoning about a thread, we do not have to consider local states of other threads. Additionally, the partitioning is dynamic, meaning that we can use ownership transfer to move some part of the local state into the shared state and vice versa. Rely and guarantee conditions are then specified as relations *on the shared state* determining how the threads change it. This is in contrast with the original rely-guarantee method, in which rely and guarantee conditions are relations *on the whole program state*. We use RGSep [16] as the basis for the logic presented in this section. Our logic adds just enough temporal reasoning to RGSep to formalise the verification method for algorithms based on grace periods that we explained in §2.

### 3.1 Preliminaries

**Programming language.** We formalise our results for a simple language:

$$C\ ::=\ \alpha \mid C;C \mid C+C \mid C^* \mid \langle C\rangle \qquad \mathcal{P}\ ::=\ C_1 \parallel \ldots \parallel C_N$$

A program $\mathcal{P}$ is a parallel composition of $N$ threads, which can contain primitive commands $\alpha \in \mathsf{PComm}$, sequential composition $C;C'$, nondeterministic choice $C + C'$, iteration $C^*$ and atomic execution $\langle C\rangle$ of $C$. We forbid nested atomic blocks. Even though we present our logic for programs in the above language, for readability we use a C-like notation in our examples, which can be easily desugared (§A).

**Separation algebras.** To reason about concurrent algorithms, we often use *permissions* [1], describing ways in which threads can operate on an area of memory. We present our logic in an abstract form [2] that is parametric in the kind of permissions used. A *separation algebra* is a set $\Sigma$, together with a partial commutative, associative and cancellative operation $*$ on $\Sigma$ and a unit element $\varepsilon \in \Sigma$. The property of cancellativity says that for each $\theta \in \Sigma$, the function $\theta * \cdot : \Sigma \rightharpoonup \Sigma$ is injective. In the rest of the paper we assume a separation algebra $\mathsf{State}$ with the operation $*$. We think of elements $\theta \in \mathsf{State}$ as *portions* of program states and the $*$ operation as combining such portions.

**Primitive commands.** We assume that the semantics of every primitive command $\alpha \in \mathsf{PComm}$, executed by thread $t$, is given by a transformer $f^t_\alpha : \mathsf{State} \to \mathcal{P}(\mathsf{State})^\top$. Here $\mathcal{P}(\mathsf{State})^\top$ is the set of subsets of $\mathsf{State}$ with a special element $\top$ used to denote an error state, resulting, e.g., from dereferencing an invalid pointer. For our logic to be sound, we need to place certain standard restrictions on $f^t_\alpha$, deferred to §A.

**Notation.** We write $g(x)\downarrow$ to mean that the function $g$ is defined on $x$, and $g(x)\uparrow$ that it is undefined on $x$. We also write $\_$ for an expression whose value is irrelevant.

## 3.2 Assertion Language

Assertions in the logic describe sets of *worlds*, comprised of the *local state* of a thread and a *history* of the *shared* state. Local states are represented by elements of a separation algebra (§3.1), and histories, by sequences of those. Our assertion language thus includes three syntactic categories, for assertions describing states, histories and worlds.

**Logical variables.** Our logic includes logical variables from a set $\mathsf{LVar} = \mathsf{LIVar} \uplus \mathsf{LSVar}$; variables from $\mathsf{LIVar} = \{x, y, \dots\}$ range over integers, and those from $\mathsf{LSVar} = \{X, Y, \dots\}$, over memory states. Let $\mathsf{LVal} = \mathsf{State} \cup \mathbb{Z}$ be the set of values of logical variables, and $\mathsf{LInt} \subseteq \mathsf{LVar} \to \mathsf{LVal}$, the set of their type-respecting interpretations.

**Assertions for states.** We assume a language for denoting subsets of $\mathsf{State} \times \mathsf{LInt}$:

$$p, q \ ::= \ \mathsf{true} \mid \neg p \mid p \Rightarrow q \mid X \mid \exists x.\, p \mid \exists X.\, p \mid \mathsf{emp} \mid p * q \mid \dots$$

The interpretation of interesting connectives is as follows:

$$\begin{array}{lll} \theta, \mathbf{i} \models \mathsf{emp} & \iff \theta = \varepsilon & \qquad \theta, \mathbf{i} \models X \iff \theta = \mathbf{i}(X) \\ \theta, \mathbf{i} \models p * q & \iff \exists \theta', \theta''.\, (\theta' * \theta'' = \theta) \wedge (\theta', \mathbf{i} \models p) \wedge (\theta'', \mathbf{i} \models q) & \end{array}$$

The assertion emp denotes an empty state; $X$, the state given by its interpretation; and $p * q$, states that can be split into two pieces such that one of them satisfies $p$ and the other, $q$. We assume that $*$ binds stronger than the other connectives.

**Assertions for histories.** A history is a non-empty sequence recording all shared states that arise during the execution of a program: $\xi \in \mathsf{History} = \mathsf{State}^+$. We denote the length of a history $\xi$ by $|\xi|$, its $i$-th element by $\xi_i$, and its $i$-th prefix, by $\xi|_i$ (so that $\big|\xi|_i\big| = i$.) We refer to the last state $\xi_{|\xi|}$ in a history $\xi$ as the *current* state. We define assertions denoting subsets of $\mathsf{History} \times \mathsf{LInt}$:

$$\tau, \Upsilon \ ::= \ \mathsf{true} \mid \neg\tau \mid \tau_1 \Rightarrow \tau_2 \mid \exists x.\, \tau \mid \exists X.\, \tau \mid \boxed{p} \mid \tau_1 \ \mathsf{since}\ \tau_2 \mid \tau \lhd \boxed{p}$$

$$\begin{array}{ll} \xi, \mathbf{i} \models \boxed{p} & \iff \xi_{|\xi|}, \mathbf{i} \models p \\ \xi, \mathbf{i} \models \tau_1 \ \mathsf{since}\ \tau_2 & \iff \exists i \in \{1, ..., |\xi|\}.\, (\xi|_i, \mathbf{i} \models \tau_2) \wedge \forall j \in \{i, ..., |\xi|\}.\, (\xi|_j, \mathbf{i} \models \tau_1) \\ \xi, \mathbf{i} \models \tau \lhd \boxed{p} & \iff \exists \xi', \theta.\, (\xi = \xi'\theta) \wedge (\xi', \mathbf{i} \models \tau) \wedge (\theta, \mathbf{i} \models p) \end{array}$$

The assertion $\boxed{p}$ denotes the set of histories of shared states, whose last state satisfies $p$; the box signifies that the assertion describes a shared state, as opposed to a thread-local one. The assertion ($\tau_1$ since $\tau_2$) describes those histories where both $\tau_1$ and $\tau_2$ held at some point in the past, and since then, $\tau_1$ has held continuously. The assertion $\tau \lhd \boxed{p}$ ($\tau$ *extended with* $\boxed{p}$) describes histories obtained by appending a state satisfying $p$ to the end of a history satisfying $\tau$. It is easy to check that (4) from §2 is indeed a tautology.

**Assertions for worlds.** A world consists of a thread-local state and a history of shared states such that the combination of the local state and the current shared state is defined:

$$\omega \in \mathsf{World} = \{(\theta, \xi) \in \mathsf{State} \times \mathsf{History} \mid (\theta * \xi_{|\xi|})\downarrow\}. \tag{7}$$

We define assertions denoting subsets of $\mathsf{World} \times \mathsf{LInt}$:

$$P, Q \ ::= \ p \mid \tau \mid \mathsf{true} \mid \neg P \mid P \Rightarrow Q \mid \exists x.\, P \mid \exists X.\, P \mid P * Q$$

$$\begin{array}{lll} \theta, \xi, \mathbf{i} \models p & \iff \theta, \mathbf{i} \models p & \qquad \theta, \xi, \mathbf{i} \models \tau \iff \xi, \mathbf{i} \models \tau \\ \theta, \xi, \mathbf{i} \models P * Q & \iff \exists \theta', \theta''.\, (\theta = \theta' * \theta'') \wedge (\theta', \xi, \mathbf{i} \models P) \wedge (\theta'', \xi, \mathbf{i} \models Q) & \end{array}$$

An assertion $P * Q$ denotes worlds in which the local state can be divided into two parts such that one of them, together with the history of the shared partition, satisfies $P$ and the other, together with the same history, satisfies $Q$. Note that $*$ does not split the shared partition, $p$ does not restrict the shared state, and $\tau$ does not restrict the thread-local one.

## 3.3 Rely/Guarantee Conditions and the Temporal Invariant

**Actions.** The judgements of our logic include *guarantee* $G$ and *rely* $R$ conditions, determining how a thread or its environment can change the shared state, respectively. Similarly to RGSep [16], these are sets of *actions* of the form $l \mid p * X \rightsquigarrow q * X$, where $l$, $p$ and $q$ are assertions over states, and $X$ is a logical variable over states. An action denotes a relation in $\mathcal{P}(\mathsf{State} \times \mathsf{State} \times \mathsf{State})$:

$$[\![ l \mid p * X \rightsquigarrow q * X ]\!] = \{(\theta_l, \theta_p, \theta_q) \mid \exists \mathbf{i}.\, (\theta_l, \mathbf{i} \models l) \wedge (\theta_p, \mathbf{i} \models p * X) \wedge (\theta_q, \mathbf{i} \models q * X)\},$$

and a rely or a guarantee denotes the union of their action denotations. We write $R \Rightarrow R'$ for $[\![R]\!] \subseteq [\![R']\!]$. Informally, the action $l \mid p * X \rightsquigarrow q * X$ allows a thread to change the part of the shared state that satisfies $p$ into one that satisfies $q$, while leaving the rest of the shared state $X$ unchanged. The assertion $l$ is called a *guard*: it describes a piece of state that has to be in the local partition of the thread for it to be able to perform the action. We omit $l$ when it is $\mathsf{emp}$. Our actions refer explicitly to the unchanged part $X$ of the shared state, as we often need to check that a command performing an action preserves global constraints on it (see §4.3). We require that $p$ and $q$ in $l \mid p * X \rightsquigarrow q * X$ be precise. An assertion $r$ for states is *precise* [12], if for every state $\theta$ and interpretation $\mathbf{i}$, there exists at most one substate $\theta_1$ satisfying $r$, i.e., such that $\theta_1, \mathbf{i} \models r$ and $\theta = \theta_1 * \theta_2$ for some $\theta_2$. Informally, a precise assertion carves out a unique piece of the heap.

**Temporal invariant.** Rely/guarantee conditions describe the set of actions that threads can perform at any point, but do not say anything about temporal protocols that the actions follow. We describe such protocols using a *temporal invariant*, which is an assertion $\Upsilon$ over histories of the shared state. Every change to the shared state that a thread performs using one of the actions in its guarantee has to preserve $\Upsilon$; in return, a thread can rely on the environment not violating the invariant. We require that $\Upsilon$ be insensitive to logical variables, i.e., $\forall \xi, \mathbf{i}, \mathbf{i}'.\, (\xi, \mathbf{i} \models \Upsilon) \iff (\xi, \mathbf{i}' \models \Upsilon)$.

**Stability.** When reasoning about the code of a thread in our logic, we take into account the interference from the other threads in the program, specified by the rely $R$ and the temporal invariant $\Upsilon$, using the concept of stability. An assertion over worlds $P$ is *stable* under an action $l \mid p_s \rightsquigarrow q_s$ and a temporal invariant $\Upsilon$, if it is insensitive to changes to the shared state permitted by the action that preserve the invariant:

$$\begin{array}{c} \forall \theta, \theta_s, \theta'_s, \theta_l, \mathbf{i}, \xi.\, ((\theta, \xi\theta_s, \mathbf{i} \models P) \wedge (\xi\theta_s, \mathbf{i} \models \Upsilon) \wedge (\xi\theta_s\theta'_s, \mathbf{i} \models \Upsilon) \wedge \\ (\theta_l, \theta_s, \theta'_s) \in [\![ l \mid p_s \rightsquigarrow q_s ]\!] \wedge (\theta * \theta_l * \theta_s)\downarrow \wedge (\theta * \theta'_s)\downarrow) \implies (\theta, \xi\theta_s\theta'_s, \mathbf{i} \models P). \end{array} \tag{8}$$

$$\frac{f_\alpha^{\mathsf{tid}}([\![p]\!]) \subseteq [\![q]\!]}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{p\}\,\alpha\,\{q\}}\ \textsc{Local} \qquad \frac{\begin{array}{c} P \wedge \Upsilon \Rightarrow P' \quad R \Rightarrow R' \quad G' \Rightarrow G \quad Q' \wedge \Upsilon \Rightarrow Q \\ R', G', \Upsilon \vdash_{\mathsf{tid}} \{P'\}\,C\,\{Q'\} \end{array}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\,C\,\{Q\}}\ \textsc{Conseq}$$

$$\frac{\begin{array}{c} R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\,C\,\{Q\} \\ F \text{ is stable under } R \cup G \text{ and } \Upsilon \end{array}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P * F\}\,C\,\{Q * F\}}\ \textsc{Frame} \qquad \frac{\begin{array}{c} Q \Rightarrow \Upsilon \qquad P, Q \text{ are stable under } R \text{ and } \Upsilon \\ \emptyset, G, \mathsf{true} \vdash_{\mathsf{tid}} \{P\}\,\langle C \rangle_a\,\{Q\} \end{array}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\,\langle C \rangle_a\,\{Q\}}\ \textsc{Shared-R}$$

$$\frac{p \Rightarrow l * \mathsf{true} \qquad \{l \mid p_s \rightsquigarrow q_s\} \Rightarrow \{a\} \qquad a \in G \qquad \emptyset, \emptyset, \mathsf{true} \vdash_{\mathsf{tid}} \{p * p_s\}\,C\,\{q * q_s\}}{\emptyset, G, \mathsf{true} \vdash_{\mathsf{tid}} \{p \wedge \tau \wedge \boxed{p_s}\}\,\langle C \rangle_a\,\{q \wedge ((\tau \wedge \boxed{p_s}) \triangleleft \boxed{q_s})\}}\ \textsc{Shared}$$

$$\frac{\begin{array}{c} R_1, G_1, \Upsilon \vdash_1 \{P_1\}\,C_1\,\{Q_1\} \quad \ldots \quad R_n, G_n, \Upsilon \vdash_n \{P_n\}\,C_n\,\{Q_n\} \\ R_{\mathsf{tid}} = \bigcup\{G_k \mid 1 \leq k \leq n \wedge k \neq \mathsf{tid}\} \qquad P_1 * \ldots * P_n \Rightarrow \Upsilon \qquad P_k, Q_k \text{ stable under } R_k \text{ and } \Upsilon \end{array}}{\vdash \{P_1 * \ldots * P_n\}\,C_1 \parallel \ldots \parallel C_n\,\{Q_1 * \ldots * Q_n\}}\ \textsc{Par}$$

**Fig. 4.** Proof rules of the logic

This makes use of the guard $l$: we do not take into account environment transitions when the latter cannot possibly own the guard, i.e., when $\theta_l$ is inconsistent with the current thread-local state $\theta$ and the current shared state $\theta_s$. An assertion is stable under $R$ and $\Upsilon$, when it is stable under every action in $R$ together with $\Upsilon$. We only consider assertions that are closed under stuttering on histories: $(\theta, \xi\theta_s\xi', \mathbf{i} \models P) \Rightarrow (\theta, \xi\theta_s\theta_s\xi', \mathbf{i} \models P)$.

## 3.4 Proof System

The judgements of the logic are of the form $R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\,C\,\{Q\}$. Here $P$ and $Q$ are the pre- and postcondition of $C$, denoting sets of worlds; $G$ describes the set of atomic changes that the thread tid executing $C$ can make to the shared state; $R$, the changes to the shared state that its environment can make; and $\Upsilon$, the temporal invariant that both have to preserve. The judgement guarantees that the command $C$ is safe, i.e., it does not dereference any invalid pointers when executed in an environment respecting $R$ and $\Upsilon$.

The proof rules of our logic are given in Figure 4. We have omitted the more standard rules (§A). We have a single axiom for primitive commands executing on the local state (LOCAL), which allows any pre- and postconditions consistent with their semantics. The axiom uses the expected pointwise lifting of the transformers $f_\alpha^t$ from §3.1 to assertion denotations, preserving the interpretation of logical variables. The CONSEQ rule looks as usual in rely/guarantee, except it allows strengthening the pre- and postcondition with the information provided by the temporal invariant $\Upsilon$.

By convention, the only commands that can operate on the shared state are atomic blocks, handled by the rules SHARED-R and SHARED. The SHARED-R rule checks that the atomic block meets its specification in an empty environment, and then checks that the pre- and postcondition are stable with respect to the actual environment $R$, and that the postcondition implies the invariant $\Upsilon$. Note that to establish the latter in practice, we can always add $\Upsilon$ to the precondition of the atomic block using CONSEQ.

SHARED handles the case of an empty rely condition, left by SHARED-R. It is the key rule in the proof system, allowing an atomic command $C$ to make a change to the shared state according to an action $l \mid p_s \rightsquigarrow q_s$. The action has to be included into the *annotation* $a$ of the atomic block, which in its turn, has to be permitted by the guarantee $G$. The annotations are part of proofs in our logic. For the logic to be sound, we require that every atomic command in the program be annotated with the same action throughout the proof. SHARED also requires the thread to have a piece of state satisfying the guard $l$ in its local state $p$. It combines the local state $p$ with the shared state $p_s$, and runs $C$ as if this combination were in the thread's local state. The rule then splits the resulting state into local $q$ and shared $q_s$ parts. Note that SHARED allows the postcondition of the atomic block to record how the shared state looked like before its execution: the previous view $\boxed{p_s}$ of the shared state and the assertion $\tau$ about its history are extended with the new shared state $\boxed{q_s}$ with the aid of $\triangleleft$ (§3.1).

The FRAME rule ensures that if a command $C$ is safe when run from states in $P$, then it does not touch an extra piece of state described by $F$. Since $F$ can contain assertions constraining the shared state, we require it to be stable under $R \cup G$ and $\Upsilon$.

PAR combines judgements about several threads. Their pre- and postconditions in the premisses of the rule are $*$-conjoined in the conclusion, which composes the local states of the threads and enforces that they have the same view of the shared state.

### 3.5 Soundness

Let us denote by $\mathsf{Prog}$ the set of programs $\mathcal{P}$ with an additional command $\mathsf{done}$, describing a completed computation. The language of §3.1 has a standard small-step operational semantics, defined by a relation $\longrightarrow$: $\mathsf{Config} \times \mathsf{Config}$, which transforms configurations from the set $\mathsf{Config} = (\mathsf{Prog} \times \mathsf{State}) \cup \{\top\}$. (Note that this semantics ignores the effects of weak memory consistency models, which are left for future work.) We defer the definition of $\longrightarrow$ to §A. The following theorem is proved in §E.

**Theorem 1 (Soundness).** *Assume* $\vdash \{P\}\,\mathcal{P}\,\{Q\}$ *and take* $\theta_l$, $\theta_s$ *and* $\mathbf{i}$ *such that* $\theta_l, \theta_s, \mathbf{i} \models P$. *Then* $(\mathcal{P}, \theta_l * \theta_s) \not\longrightarrow^* \top$ *and, whenever* $(\mathcal{P}, \theta_l * \theta_s) \longrightarrow^* (\mathsf{done} \parallel \ldots \parallel \mathsf{done}, \theta')$, *for some* $\theta'_l$, $\theta'_s$ *and* $\xi$ *we have* $\theta' = \theta'_l * \theta'_s$ *and* $\theta'_l, \xi\theta'_s, \mathbf{i} \models Q$.

## 4 Logic Instantiation and Hazard Pointers

As explained in §2, proofs of algorithms based on grace periods, use only a restricted form of temporal reasoning. In this section, we describe an instantiation of the abstract logic of §3 tailored to such algorithms. This includes a particular form of the temporal invariant (§4.2) and a specialised version of the SHARED rule (SHARED-I below) that allows us to establish that the *temporal* invariant is preserved using standard *state-based* reasoning. We present the instantiation by the example of verifying the concurrent counter algorithm with hazard pointers from §2.

### 4.1 Assertion Language

**Permissions.** We instantiate $\mathsf{State}$ to $\mathsf{RAM_e} = \mathbb{N} \rightharpoonup_{\mathit{fin}} ((\mathbb{Z} \times \{1, \mathsf{m}\}) \cup \{\mathsf{e}\})$. A state thus consists of a finite partial function from memory locations allocated in the heap to the values they store and/or permissions. The permission $1$ is a *full* permission, which allows a thread to perform any action on the cell; the permission $\mathsf{m}$ is a *master* permission, which allows reading and writing the cell, but not deallocating it; and $\mathsf{e}$ is an *existential* permission, which only allows reading the cell and does not give any guarantees regarding its contents. The transformers $f^t_\alpha$ over $\mathsf{RAM_e}$ are given in §A.

We define $*$ on cell contents as follows: $(u, \mathsf{m}) * \mathsf{e} = (u, 1)$; undefined in all other cases. This only allows a full permission to be split into a master and an existential one,

$$
\begin{array}{llrl}
X \rightsquigarrow X & (\mathsf{Id}) & x \mapsto_{\mathsf{m}} \_ \mid x \mapsto_{\mathsf{e}} \_ * X \rightsquigarrow X & (\mathsf{Take}) \\
\multicolumn{3}{l}{\mathtt{HP}[\mathsf{tid}-1] \mapsto \_ * X \rightsquigarrow \mathtt{HP}[\mathsf{tid}-1] \mapsto \_ * X} & (\mathsf{HP}_{\mathsf{tid}}) \\
\multicolumn{3}{l}{\mathtt{C} \mapsto x * x \mapsto \_ * X \rightsquigarrow \mathtt{C} \mapsto y * y \mapsto \_ * x \mapsto_{\mathsf{e}} \_ * X} & (\mathsf{Inc})
\end{array}
$$

$$G_{\mathsf{tid}} = \{\mathsf{HP}_{\mathsf{tid}}, \mathsf{Inc}, \mathsf{Take}, \mathsf{Id}\}; \qquad R_{\mathsf{tid}} = \bigcup\{G_k \mid 1 \le k \le N \wedge k \ne \mathsf{tid}\}$$

$$\Upsilon_{\mathsf{HP}} \iff \forall x, t.\, ((\boxed{\mathtt{HP}[t-1] \mapsto x * \mathsf{true}} \text{ since } \boxed{\mathtt{C} \mapsto x * x \mapsto \_ * \mathsf{true}}) \Rightarrow \boxed{x \mapsto_{\mathsf{e}} \_ * \mathsf{true}})$$

**Fig. 5.** Rely/guarantee conditions and the temporal invariant used in the proof of the counter algorithm with hazard pointers

which is enough for our purposes. For $\theta_1, \theta_2 \in \mathsf{RAM}_{\mathsf{e}}$, $\theta_1 * \theta_2$ is undefined, if for some $x$, we have $\theta_1(x)\downarrow$, $\theta_2(x)\downarrow$, but $(\theta_1(x) * \theta_2(x))\uparrow$. Otherwise,

$$\theta_1 * \theta_2 = \{(x, w) \mid (\theta_1(x) = w \wedge \theta_2(x)\uparrow) \vee (\theta_2(x) = w \wedge \theta_1(x)\uparrow) \vee (w = \theta_1(x) * \theta_2(x))\}.$$

**State assertions.** To denote elements of $\mathsf{RAM}_{\mathsf{e}}$, we extend the assertion language for predicates over states given in §3.2: $p ::= \ldots \mid E \mapsto F \mid E \mapsto_{\mathsf{m}} F \mid E \mapsto_{\mathsf{e}} \_$, where $E, F$ range over expressions over integer-valued logical variables. The semantics is as expected; e.g., $[\llbracket E \rrbracket_{\mathbf{i}} : (\llbracket F \rrbracket_{\mathbf{i}}, 1)], \mathbf{i} \models E \mapsto F$ and $x \mapsto u \Leftrightarrow x \mapsto_{\mathsf{m}} u * x \mapsto_{\mathsf{e}} \_$.

**Conventions.** We assume that logical variables $t, t', \ldots$ range over thread identifiers in $\{1, \ldots, N\}$. We write $\mathtt{A}[k]$ for $\mathtt{A} + k$, and $\mathsf{true}_{\mathsf{e}}$ for $\exists A.\, \circledast_{x \in A}\, x \mapsto_{\mathsf{e}} \_$, where $\circledast$ is the iterated version of $*$. We adopt the convention that global variables are constants, and local variables are allocated at fixed addresses in memory. For a local variable `var` of thread tid, we write $var \Vdash P$ for $\exists var.\, (\&\mathtt{var} + \mathsf{tid} - 1) \mapsto var * P$, where $\&\mathtt{var}$ is the address of the variable. Note that here `var` is a program variable, whereas $var$ is a logical one. We use a similar notation for lists of variables $V$.

## 4.2 Actions and the Temporal Invariant

The actions used in the proof of the running example and the rely/guarantee conditions constructed from them are given in Figure 5. Id allows reading the contents of the shared state, but not modifying it, and $\mathsf{HP}_{\mathsf{tid}}$ allows modifying the contents of the $t$-th entry in the hazard pointer array. The rely $R_{\mathsf{tid}}$ and the guarantee $G_{\mathsf{tid}}$ are set up in such a way that only thread tid can execute $\mathsf{HP}_{\mathsf{tid}}$.

Inc allows a thread to change the node pointed to by `C` from $x$ to $y$, thus detaching the old node $x$. Note that $y \mapsto \_$ occurs on the right-hand side of Inc, but not on its left-hand side. Hence, the thread executing the action transfers the ownership of the node $y$ (in our example, initially allocated in its local state) into the shared state. Since $x \mapsto \_$ occurs on the left-hand side of Inc, but only $x \mapsto_{\mathsf{e}} \_$ occurs on its right-hand side, the thread gets the ownership of $x \mapsto_{\mathsf{m}} \_$. This is used to express the protocol that the thread detaching the node will be the one to deallocate it. Namely, Take allows a thread to take the remaining existential permission from the shared state only when it has the corresponding master permission in its local state. The existential permission left in the shared state after a thread executes Inc lets concurrently running threads access the detached node until it is deallocated.

Threads can only execute Take and other actions when these do not violate the temporal invariant $\Upsilon_{\mathsf{HP}}$ in Figure 5. Temporal invariants used for proofs of algorithms based on grace periods are of the form "$\forall x, t.\, (\boxed{g} \text{ since } \boxed{r}) \Rightarrow \boxed{c}$", where "$\boxed{g} \text{ since } \boxed{r}$"

```
1  int *C = new int(0), *HP[N] = {0};
2  Set detached[N] = {∅};
3  int inc() {
4    int v, *n, *s, *s2;
5    $\{V \Vdash F_{\mathsf{tid}} \wedge I\}$
6    n = new int;
7    do {
8      $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \wedge I\}$
9      do {
10       $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \wedge I\}$
11       ⟨s = C⟩$_{\mathsf{Id}}$;
12       $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \wedge I\}$
13       ⟨HP[tid-1] = s⟩$_{\mathsf{HP}_{\mathsf{tid}}}$;
14       $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \wedge I \wedge$
15         $\boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}\}$
16       ⟨s2 = C⟩$_{\mathsf{Id}}$;
17       $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \wedge I \wedge$
18         $(\boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}$
19          since $\boxed{\mathtt{C} \mapsto s2 * s2 \mapsto \_ * \mathsf{true}})\}$
20     } while (s != s2);
21     $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \wedge I \wedge \boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}} \wedge$
22      $(\boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}$
23       since $\boxed{\mathtt{C} \mapsto s * s \mapsto \_ * \mathsf{true}})\}$
24     ⟨v = *s⟩$_{\mathsf{Id}}$;
25     *n = v+1;
26     $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \wedge I \wedge \boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}} \wedge$
27      $(\boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}$
28       since $\boxed{\mathtt{C} \mapsto s * s \mapsto \_ * \mathsf{true}})\}$
29   } while (!CAS$_{\mathsf{Inc},\mathsf{Id}}$(&C, s, n));
30   $\{V \Vdash s \mapsto_{\mathsf{m}} \_ * F_{\mathsf{tid}} \wedge I \wedge \boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}}\}$
31   reclaim(s);
32   $\{V \Vdash F_{\mathsf{tid}} \wedge I\}$
33   return v; }
```

**Fig. 6.** Proof outline for `inc` with hazard pointers. Here $V$ is $v$, $n$, $s$, $s2$, $my$, $in\_use$, $i$.

defines the duration of the grace period for a thread $t$ and a location $x$, and $\boxed{c}$ gives the property that has to be maintained during the grace period. In our example, the invariant formalises (2): if a hazard pointer of $t$ has pointed to a node $x$ continuously since `C` pointed to $x$, then an existential permission for $x$ is present in the shared state.

## 4.3 Proof Outlines and a Derived Rule for Grace Periods

The proof outline for the running example is shown in Figures 6 and 7. In the figure, we write `CAS`$_{a,b}$`(addr,v1,v2)` as a shorthand for the following, where the `assume` command "assumes" its parameter to be non-zero (§A):

```
if (nondet()) {⟨assume(*addr == v1); *addr = v2⟩$_a$; return 1; }
else { ⟨assume(*addr != v1)⟩$_b$; return 0; }
```

The bulk of the proof employs standard state-based reasoning of the kind performed in RGSep [16]. Temporal reasoning is needed, e.g., to check that every command changing the shared state preserves the temporal invariant $\Upsilon_{\mathsf{HP}}$ (the premiss $Q \Rightarrow \Upsilon$ in SHARED-R). We start by discussing the proof outline of `inc` in Figure 6 in general terms; we then describe the handling of commands changing the shared state in detail.

**Verifying** `inc`**.** Let $H \Leftrightarrow (\circledast_t \, \mathtt{HP}[t-1] \mapsto \_)$ and $I \Leftrightarrow \boxed{H * \exists y. \, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}_{\mathsf{e}}}$. The pre- and postcondition of `inc` in Figure 6 thus state that the shared state always contains the hazard pointer array, the pointer at the address `C` and the node it identifies. Additionally, we can have an arbitrary number of existential permissions for nodes that threads leave in the shared state in between executing Inc and Take. We also have an assertion $F_{\mathsf{tid}}$, defined later, which describes the thread-local `detached` set.

At line 11 of `inc`, the current thread reads the value of `C` into the local variable `s`. For the postcondition of this command to be stable, we do not maintain any correlation between the values of `C` and `s`, as other threads might change `C` using Inc at any time. The thread sets its hazard pointer to `s` at line 13. The postcondition includes $\boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}$, which is stable, as $R_{\mathsf{tid}}$ and $G_{\mathsf{tid}}$ (Figure 5) allow only the current thread to execute $\mathsf{HP}_{\mathsf{tid}}$.

At line 16, the thread reads the value of `C` into `s2`. Right after executing the command, we have $\boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}} \wedge \boxed{\mathtt{C} \mapsto s2 * s2 \mapsto \_ * \mathsf{true}}$. This assertion is unstable, as other threads may change C at any time using Inc. We therefore weaken it to the postcondition shown by using the tautology $(\eta \wedge \mu) \Rightarrow (\eta \text{ since } \mu)$. It is easy to check that an assertion $(\eta \text{ since } \mu)$ is stable if $\eta$ is. Since $\boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}$ is stable, so is the postcondition of the command in line 16. After the test `s != s2` in line 20 fails, the since clause in this assertion characterises the grace period of the thread tid for the location `s`, as stated by $\Upsilon_{\mathsf{HP}}$. This allows us to exploit $\Upsilon_{\mathsf{HP}}$ at line 23 using CONSEQ, establishing $\boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}}$. This assertion allows us to access the node at the address $s$ safely at line 24.

If the CAS in line 29 is successful, then the thread transfers the ownership of the newly allocated node `n` to the shared state, and takes the ownership of the master permission for the node $s$; the existential permission for $s$ stays in the shared state. The resulting assertion $s \mapsto_{\mathsf{m}} \_ \wedge \boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}}$ is stable, because the only action that can remove $s \mapsto_{\mathsf{e}} \_$ from the shared state, Take, is guarded by $s \mapsto_{\mathsf{m}} \_$. Since the current thread has the ownership of $s \mapsto_{\mathsf{m}} \_$ and $s \mapsto_{\mathsf{m}} \_ * s \mapsto_{\mathsf{m}} \_$ is inconsistent, the condition $(\theta * \theta_l * \theta_s)\!\downarrow$ in (8), checking that the guard is consistent with the local state, implies that the action cannot be executed by the environment, and thus, the assertion is stable.

**Derived rule for grace periods.** To check that the commands in lines 13 and 29 of `inc` preserve $\Upsilon_{\mathsf{HP}}$, we use the following rule SHARED-I, derived from SHARED (§A):

$$\frac{\begin{array}{c} p \Rightarrow l * \mathsf{true} \qquad a = (l \mid p'_s \rightsquigarrow q'_s) \in G \qquad p_s \Rightarrow p'_s \qquad q_s \Rightarrow q'_s \\ \emptyset, \emptyset, \mathsf{true} \vdash_{\mathsf{tid}} \{p * (p_s \wedge \neg(g \wedge r))\}\, C\, \{q * (q_s \wedge (g \wedge r \Rightarrow c))\} \\ \emptyset, \emptyset, \mathsf{true} \vdash_{\mathsf{tid}} \{p * (p_s \wedge g \wedge c)\}\, C\, \{q * (q_s \wedge (g \Rightarrow c))\} \end{array}}{\emptyset, G, \mathsf{true} \vdash_{\mathsf{tid}} \{p \wedge \boxed{p_s} \wedge ((\boxed{g} \text{ since } \boxed{r}) \Rightarrow \boxed{c})\} \langle C \rangle_a \{q \wedge \boxed{q_s} \wedge ((\boxed{g} \text{ since } \boxed{r}) \Rightarrow \boxed{c})\}}$$

This gives conditions under which $\langle C \rangle$ preserves the validity of an assertion of the form

$$(\boxed{g} \text{ since } \boxed{r}) \Rightarrow \boxed{c} \qquad (9)$$

and thus allows us to prove the preservation of a temporal invariant of the form (9) using standard Hoare-style reasoning. In the rule, $p_s$ describes the view of the shared partition that the current thread has before executing $C$, and $q_s$, the state in which $C$ leaves it. The rule requires that the change from $p_s$ to $q_s$ be allowed by the annotation $a = (l \mid p'_s \rightsquigarrow q'_s)$, i.e., that $p_s \Rightarrow p'_s$ and $q_s \Rightarrow q'_s$. It further provides two Hoare triples to be checked of $C$, which correspond, respectively, to the two cases for why $(\boxed{g} \text{ since } \boxed{r}) \Rightarrow \boxed{c}$ may hold before the execution of $C$: $\neg(\boxed{g} \text{ since } \boxed{r})$ or $(\boxed{g} \text{ since } \boxed{r}) \wedge \boxed{c}$.

As in SHARED, the two Hoare triples in the premiss allow the command inside the atomic block to access both local and shared state. Consider the first one. We can assume $\neg(g \wedge r)$ in the precondition, as it is implied by $\neg(\boxed{g} \text{ since } \boxed{r})$. Since $\boxed{g} \text{ since } \boxed{r}$ does not hold before the execution of $C$, the only way to establish it afterwards is by obtaining $g \wedge r$. In this case, to preserve (9), we have to establish $c$, which motivates the postcondition. Formally: $((\neg(\boxed{g} \text{ since } \boxed{r})) \lhd \boxed{g \wedge r \Rightarrow c}) \Rightarrow ((\boxed{g} \text{ since } \boxed{r}) \Rightarrow \boxed{c})$.

Consider now the second Hoare triple. Its precondition comes from the tautology $((\boxed{g} \text{ since } \boxed{r}) \wedge \boxed{c}) \Rightarrow \boxed{g \wedge c}$. We only need to establish $c$ in the postcondition when $\boxed{g}$ since $\boxed{r}$ holds there, which will only be the case if $g$ continues to hold after $C$ executes: $(((\boxed{g} \text{ since } \boxed{r}) \wedge \boxed{c}) \lhd \boxed{g \Rightarrow c}) \Rightarrow ((\boxed{g} \text{ since } \boxed{r}) \Rightarrow \boxed{c})$.

**Preserving the temporal invariant.** We illustrate the use of SHARED-I on the command in line 29 of Figure 6; the one in line 13 is handled analogously. We consider the case when the CAS succeeds, i.e., $C$ is `{assume(C == s); C = n;}`. Let $P$

```
void reclaim(int *s) { $\{V \Vdash s \mapsto_{\mathsf{m}} \_ * F_{\mathsf{tid}} \wedge \boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}} \wedge I\}$
  insert(detached[tid-1], s);
  if (nondet()) return;
  Set in_use = ∅;
  while (!isEmpty(detached[tid-1])) {
    $\{V \Vdash \exists A.\, \mathtt{detached}[\mathsf{tid}-1] \mapsto A * D(A) * D(\mathit{in\_use}) \wedge A \neq \emptyset \wedge I\}$
    bool my = true;
    Node *n = pop(detached[tid-1]);
    $\{V \Vdash \mathit{my} \wedge \exists A.\, \mathtt{detached}[\mathsf{tid}-1] \mapsto A * D(A) * D(\mathit{in\_use}) * n \mapsto_{\mathsf{m}} \_ \wedge \boxed{n \mapsto_{\mathsf{e}} \_ * \mathsf{true}} \wedge I\}$
    for (int i = 0; i < N && my; i++) {
      $\{V \Vdash \mathit{my} \wedge \exists A.\, \mathtt{detached}[\mathsf{tid}-1] \mapsto A * D(A) * D(\mathit{in\_use}) * n \mapsto_{\mathsf{m}} \_ * \boxed{n \mapsto_{\mathsf{e}} \_ * \mathsf{true}} \wedge$
       $0 \leq i < N \wedge I \wedge \boxed{H * \exists y.\, y \neq n \wedge \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}_{\mathsf{e}}} \wedge$
       $\forall 0 \leq j < i.\, (\boxed{\exists y.\, y \neq n \wedge \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}_{\mathsf{e}}}$ since $\boxed{\exists x.\, x \neq n \wedge \mathtt{HP}[j] \mapsto x * \mathsf{true}})\}$
      if ($\langle$HP[i] == n$\rangle_{\mathsf{Id}}$) my = false;
    }
    if (my) {
      $\{V \Vdash \exists A.\, \mathtt{detached}[\mathsf{tid}-1] \mapsto A * D(A) * D(\mathit{in\_use}) * n \mapsto_{\mathsf{m}} \_ \wedge \boxed{n \mapsto_{\mathsf{e}} \_ * \mathsf{true}} \wedge I \wedge$
       $\forall t.\, \neg \boxed{\mathtt{C} \mapsto n * \mathsf{true}}$ since $\neg \boxed{\mathtt{HP}[t-1] \mapsto n * \mathsf{true}}\}$
      $\langle$ ; $\rangle_{\mathsf{Take}}$
      $\{V \Vdash \exists A.\, \mathtt{detached}[\mathsf{tid}-1] \mapsto A * D(A) * D(\mathit{in\_use}) * n \mapsto \_ \wedge I\}$
      free(n);
    } else { insert(in_use, n); }
  } $\{V \Vdash \mathtt{detached}[\mathsf{tid}-1] \mapsto \emptyset * D(\mathit{in\_use}) \wedge I\}$
  moveAll(in_use, detached[tid-1]); $\{V \Vdash F_{\mathsf{tid}} \wedge I\}$
}
```

**Fig. 7.** Proof outline for `reclaim` with hazard pointers. $V$ is $v$, $n$, $s$, $s2$, $my$, $in\_use$, $i$.

and $Q$ be the pre- and postconditions of this command in lines 26 and 30, respectively. We thus need to prove $R_{\mathsf{tid}}, G_{\mathsf{tid}}, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, \langle C \rangle_{\mathsf{Inc}}\, \{Q\}$. We first apply CONSEQ to strengthen the precondition of the CAS with $\Upsilon$, and then apply SHARED-R. This rule, in particular, requires us to show that the temporal invariant is preserved: $\emptyset, G_{\mathsf{tid}}, \mathsf{true} \vdash_{\mathsf{tid}} \{P \wedge \Upsilon\}\ C\ \{Q \wedge \Upsilon\}$. Let us first strip the quantifiers over $x$ and $t$ in $\Upsilon$ using a standard rule of Hoare logic. We then apply SHARED-I with

$$
\begin{array}{lll}
g = (\mathtt{HP}[t-1] \mapsto x * \mathsf{true}); & r = (\mathtt{C} \mapsto x * x \mapsto \_ * \mathsf{true}); & c = (x \mapsto_{\mathsf{e}} \_ * \mathsf{true}); \\
p_s = (H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}_{\mathsf{e}}); & & p = n \mapsto \_; \\
q_s = (H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * s \mapsto_{\mathsf{e}} \_ * \mathsf{true}_{\mathsf{e}}); & & q = s \mapsto_{\mathsf{m}} \_.
\end{array}
$$

We consider only the first Hoare triple in the premiss of SHARED-I, which corresponds to $\boxed{g}$ since $\boxed{r}$ being false before the atomic block. The triple instantiates to

$$
\begin{array}{l}
\{n \mapsto \_ * ((H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}_{\mathsf{e}}) \wedge \neg(\mathtt{HP}[t-1] \mapsto x * \mathsf{true} \wedge \mathtt{C} \mapsto x * x \mapsto \_ * \mathsf{true}))\} \\
\quad \mathtt{assume(C\ ==\ s);\ C\ =\ n;}\ \{s \mapsto_{\mathsf{m}} \_ * (H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * s \mapsto_{\mathsf{e}} \_ * \mathsf{true}_{\mathsf{e}}) \wedge \\
\qquad\qquad\qquad (((\mathtt{HP}[t-1] \mapsto x * \mathsf{true}) \wedge (\mathtt{C} \mapsto x * x \mapsto \_ * \mathsf{true})) \Rightarrow (x \mapsto_{\mathsf{e}} \_ * \mathsf{true}))\}
\end{array}
$$

Recall that when the CAS at line 29 inserts a node into the shared data structure, we already might have a hazard pointer set to the node (§2). The postcondition of the above triple states that, in this case, we need to establish the conclusion of the temporal invariant. This is satisfied, as $x \mapsto \_ \Leftrightarrow x \mapsto_{\mathsf{m}} \_ * x \mapsto_{\mathsf{e}} \_$.

**Verifying `reclaim`.** We now explain the proof outline in Figure 7. The predicate $F_{\mathsf{tid}}$ describes the `detached` set of thread tid:

$$
\begin{array}{rcl}
D(A) & \iff & \circledast_{x \in A}(x \mapsto_{\mathsf{m}} \_ \wedge \boxed{x \mapsto_{\mathsf{e}} \_ * \mathsf{true}}); \\
F_{\mathsf{tid}} & \iff & \exists A.\, \mathtt{detached}[\mathsf{tid}-1] \mapsto A * D(A).
\end{array}
\tag{10}
$$

$F_{\mathsf{tid}}$ asserts that thread tid owns the tid-th entry of the `detached` array, which stores the set $A$ of addresses of detached nodes; $D(A)$ asserts that, for every $x \in A$, the thread has the master permission for $x$ in its local state, and the shared state contains the existential permission for $x$. The assertion $F_{\mathsf{tid}}$ is stable, since, as we explained above, so is $x \mapsto_{\mathsf{m}} \_ \wedge \boxed{x \mapsto_{\mathsf{e}} \_ * \mathsf{true}}$. We assume the expected specifications for set operations.

The core of `reclaim` is the loop following the `pop` operation in line 8, which checks that the hazard pointers do not point to the node that we want to deallocate. The assertion in line 13 formalises (3) and is established as follows. If the condition on the pointer $\mathtt{HP}[i]$ in line 14 fails, then we know that $\boxed{\exists x.\, x \neq n \wedge \mathtt{HP}[i] \mapsto x * \mathsf{true}}$. Recall that, according to (7), §3.2, the combination of the local and the shared states has to be consistent. Then, since we have $n \mapsto_{\mathsf{m}} \_$ in our local state, we cannot have `C` pointing to $n$: in this case the full permission $n \mapsto \_$ would be in the shared state, and $n \mapsto \_ * n \mapsto_{\mathsf{m}} \_$ is inconsistent. Hence, $\boxed{\exists y.\, y \neq n \wedge \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}$. By the tautology $(\eta \wedge \mu) \Rightarrow (\eta \text{ since } \mu)$, we obtain the desired assertion:

$$\boxed{\exists y.\, y \neq n \wedge \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}} \text{ since } \boxed{\exists x.\, x \neq n \wedge \mathtt{HP}[i] \mapsto x * \mathsf{true}}. \qquad (11)$$

Since $n \mapsto_{\mathsf{m}} \_ \wedge \boxed{\exists y.\, y \neq n \wedge \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}$ is stable, so is the loop invariant. At line 19, we use (11) and (4) to show that the existential permission for the node $n$ can be safely removed from the shared state. After this, we recombine it with the local master permission to obtain $n \mapsto \_$, which allows deallocating the node.

**Absence of memory leaks.** According to Theorem 1, the above proof establishes that the algorithm is memory safe. In fact, it also implies that the algorithm does not leak memory. Indeed, let $\mathcal{P}$ be the program consisting of any number of `inc` operations running in parallel. From our proof, we get that $\mathcal{P}$ satisfies the following triple:

$$\vdash \{L * (\circledast_t\, \mathtt{detached}[t-1] \mapsto \emptyset) \wedge \boxed{(\circledast_t\, \mathtt{HP}[t-1] \mapsto 0) * \exists y.\, \mathtt{C} \mapsto y * y \mapsto 0}\}$$
$$\mathcal{P}\ \{L * (\circledast_t\, F_t) \wedge \boxed{(\circledast_t\, \mathtt{HP}[t-1] \mapsto \_) * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}_{\mathsf{e}}}\},$$

where $L$ includes the local variables of all threads. The assertion $\mathsf{true}_{\mathsf{e}}$ in the postcondition describes an arbitrary number of existential permissions for memory cells. However, physical memory cells are denoted by full permissions; an existential permission can correspond to one of these only when the corresponding master permission is available. Every such master permission comes from some $F_t$, and hence, the corresponding cell belongs to $\mathtt{detached}[t-1]$. Thus, at the end of the program, any allocated cell is reachable from either `C` or one of the `detached` sets.

**Extensions.** Even though we illustrated our proof technique using the idealistic example of a counter, the technique is also applicable both to other algorithms based on hazard pointers and to different ways of optimising hazard pointer implementations. In §B, we demonstrate this on the example of a non-blocking stack with several optimisations of hazard pointers used in practice [11]: e.g., the pointers are dynamically allocated, `reclaim` scans the hazard list only once, and the `detached` sets are represented by lists with links stored inside the detached elements themselves. The required proof is not significantly more complex than the one presented in this section.

In §C, we also present an adaptation of the above proof to establish the linearizability of the algorithm following the approach in [16] (we leave a formal integration of the two methods for future work). The main challenge of proving linearizability of this and similar algorithms lies in establishing that the ABA problem described in §2 does not

Let $S(\mathsf{tid}, k) = \boxed{\mathtt{rcu}[\mathsf{tid}-1] \mapsto k * \mathsf{true}}$ and

$$X \rightsquigarrow X \quad (\mathsf{Id}) \qquad\qquad \mathtt{rcu}[\mathsf{tid}-1] \mapsto \_ * X \rightsquigarrow \mathtt{rcu}[\mathsf{tid}-1] \mapsto \_ * X \quad (\mathsf{RCU}_{\mathsf{tid}})$$

Then, $R, \{\mathsf{RCU}_{\mathsf{tid}}\}, \Upsilon \vdash_{\mathsf{tid}} \{S(\mathsf{tid}, 0) \land \mathsf{emp}\}\, \mathtt{rcu\_enter}()\, \{S(\mathsf{tid}, 1) \land \mathsf{emp}\}$;
$R, \{\mathsf{RCU}_{\mathsf{tid}}\}, \Upsilon \vdash_{\mathsf{tid}} \{S(\mathsf{tid}, 1) \land \mathsf{emp}\}\, \mathtt{rcu\_exit}()\, \{S(\mathsf{tid}, 0) \land \mathsf{emp}\}$;
$R, \{\mathsf{Id}\}, \Upsilon \vdash_{\mathsf{tid}} \{p \land \tau\}\, \mathsf{sync}()\, \{p \land \forall t.\, \tau \text{ since } S(t, 0)\}$,

where 1. $R \Rightarrow \{(\mathtt{rcu}[\mathsf{tid}-1] \mapsto x * \mathsf{true}) \rightsquigarrow (\mathtt{rcu}[\mathsf{tid}-1] \mapsto x * \mathsf{true})\}$;
2. $\Upsilon$ is stable under $\{\mathsf{Id}, \mathsf{RCU}_{\mathsf{tid}}\}$ and true; and
3. $p \land \tau$ is stable under $R \cup \{\mathsf{Id}\}$ and $\Upsilon$.

**Fig. 8.** Specification of RCU commands

occur, i.e., when the CAS in line 29 of Figure 6 is successful, we can be sure that the value of C has not changed since we read it at line 16. In our proof this is easy to establish, as between lines 16 and 29, all assertions are stable and contain $\boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}}$, which guarantees that $s$ cannot be recycled.

## 5 Formalising Read-Copy-Update

**RCU specification.** We start by deriving specifications for RCU commands in our logic from the abstract RCU implementation in Figure 2; see Figure 8. The formula $S(\mathsf{tid}, 1)$ states that the thread tid is in a critical section, and $S(\mathsf{tid}, 0)$, that it is outside one. We use the identity action Id and an action $\mathsf{RCU}_{\mathsf{tid}}$ allowing a thread tid to enter or exit a critical section. The latter is used to derive the specification for rcu_enter and rcu_exit (see Figure 2). To satisfy the premisses of the SHARED-R rule in these derivations, we require certain conditions ensuring that the RCU client will not corrupt the rcu array. First, we require that the rely $R$ does not change the element of the rcu array for the thread tid executing the RCU function (condition 1). In practice, $R$ includes the actions $\mathsf{RCU}_k$ for $k \neq \mathsf{tid}$ and actions that do not access the rcu array. Second, we require that $\Upsilon$ be preserved under the actions that RCU functions execute (condition 2).

The specification for sync is the most interesting one. The precondition $p \land \tau$ is required to be stable (condition 3), and thus holds for the whole of sync's duration. Since, while sync is executing, every thread passes through a point when it is not in a critical section, we obtain $\forall t.\, \tau$ since $S(t, 0)$ in the postcondition. (We mention the local state $p$ in the specification, as it helps in checking stability; see below.) The derivation of the specification from Figure 2 is straightforward: e.g., the invariant of the loop in line 8 is $r, i \Vdash p \land \forall t.\, (t < i + 1 \lor r[t-1] = 0) \Rightarrow (\tau \text{ since } S(t, 0))$. As usual, here we obtain the since clause by weakening: $(\tau \land S(\mathsf{tid}, 0)) \Rightarrow (\tau \text{ since } S(\mathsf{tid}, 0))$.

**Verification of the RCU-based counter.** Since this RCU-based algorithm is similar to the one using hazard pointers, most actions in relies and guarantees are reused from that proof (Figure 5): we let $G_{\mathsf{tid}} = \{\mathsf{Id}, \mathsf{Inc}, \mathsf{Take}, \mathsf{RCU}_{\mathsf{tid}}\}$ and $R_{\mathsf{tid}} = \bigcup\{G_k \mid 1 \le k \le N \land k \neq \mathsf{tid}\}$. The following invariant formalises (5):

$$\Upsilon_{\mathsf{RCU}} \iff \forall x, t.\, (S(t, 1) \text{ since } \boxed{\mathsf{C} \mapsto x * x \mapsto \_ * \mathsf{true}}) \Rightarrow \boxed{x \mapsto_{\mathsf{e}} \_ * \mathsf{true}}.$$

The proof outline for the RCU-based counter is given in Figure 9. The assertion $F_{\mathsf{tid}}$ is the same as for hazard pointers and is defined by (10) in §4. The assertion $I$ describes the state invariant of the algorithm:

$$I \iff \boxed{(\circledast_t\, \mathtt{rcu}[t-1] \mapsto \_) * \exists y.\, \mathsf{C} \mapsto y * y \mapsto \_ * \mathsf{true}_{\mathsf{e}}}.$$

```
1  int *C=new int(0); bool rcu[N]={0};
2  Set detached[N]={∅};
3  int inc() {
4    int v, *n, *s;
5    {$V \Vdash F_{\mathsf{tid}} \land I \land S(\mathsf{tid}, 0)\}$}
6    n = new int;
7    {$V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \land I \land S(\mathsf{tid}, 0)\}$}
8    rcu_enter();
9    do { {$V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \land I \land S(\mathsf{tid}, 1)$}
10     rcu_exit();
11     rcu_enter();
12     ⟨s = C⟩$_{\mathsf{Id}}$;
13     {$V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \land I \land \boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}} \land$
14       $(S(\mathsf{tid}, 1)$ since $\boxed{\mathsf{C} \mapsto s * s \mapsto \_ * \mathsf{true}})$}
15     ⟨v = *s⟩$_{\mathsf{Id}}$;
16     *n = v+1;
17     {$V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \land I \land \boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}}$
18       $(S(\mathsf{tid}, 1)$ since $\boxed{\mathsf{C} \mapsto s * s \mapsto \_ * \mathsf{true}})$}
19   } while (!CAS$_{\mathsf{Inc},\mathsf{Id}}$(&C, s, n));
20   rcu_exit();
21   {$V \Vdash s \mapsto_{\mathsf{m}} \_ * F_{\mathsf{tid}} \land I \land S(\mathsf{tid}, 0) \land$
22     $\boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}}$}
23   reclaim(s);
24   {$V \Vdash F_{\mathsf{tid}} \land I \land S(\mathsf{tid}, 0)$}
25   return v; }

26 void reclaim(int* s) {
27   {$V \Vdash s \mapsto_{\mathsf{m}} \_ * F_{\mathsf{tid}} \land I \land S(\mathsf{tid}, 0) \land$
28     $\boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}}$}
29   insert(detached[tid-1], s);
30   if (nondet()) return;
31   {$V \Vdash I \land S(\mathsf{tid}, 0) \land$
32   $\exists A.\, \mathtt{detached}[\mathsf{tid}-1] \mapsto A \,*$
33   $(\circledast_{x \in A}\, x \mapsto_{\mathsf{m}} \_) \land$
34   $\boxed{(\circledast_{x \in A}\, x \mapsto_{\mathsf{e}} \_) * \mathsf{true}} \land$
35   $(\circledast_{x \in A} \neg\boxed{\mathsf{C} \mapsto x * x \mapsto \_ * \mathsf{true}})$}
36   sync();
37   {$V \Vdash I \land S(\mathsf{tid}, 0) \land$
38   $\exists A.\, \mathtt{detached}[\mathsf{tid}-1] \mapsto A \,*$
39   $\circledast_{x \in A}((x \mapsto_{\mathsf{m}} \_ \land \boxed{x \mapsto_{\mathsf{e}} \_ * \mathsf{true}}) \land \forall t.$
40   $\neg\boxed{\mathsf{C} \mapsto x * x \mapsto \_ * \mathsf{true}}$ since $S(t, 0))$}
41   ⟨ ; ⟩$_{\mathsf{Take}}$
42   {$V \Vdash I \land S(\mathsf{tid}, 0) \land$
43   $\exists A.\, \mathtt{detached}[\mathsf{tid}-1] \mapsto A \,*$
44   $(\circledast_{x \in A}\, x \mapsto \_)$}
45   while (!isEmpty(detached[tid]))
46     free(pop(detached[tid]));
47   {$V \Vdash F_{\mathsf{tid}} \land I \land S(\mathsf{tid}, 0)$}
48 }
```

**Fig. 9.** Counter with an RCU-based memory management. Here $V$ is $v, n, s$.

The key points are as follows. After reading C at line 12, we obtain an unstable assertion $S(\mathsf{tid}, 1) \land \boxed{\mathsf{C} \mapsto s * s \mapsto \_ * \mathsf{true}}$, which we weaken to a stable one $(S(\mathsf{tid}, 1)$ since $\boxed{\mathsf{C} \mapsto s * s \mapsto \_ * \mathsf{true}})$. Then $\Upsilon_{\mathsf{RCU}}$ yields $\boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}}$, which justifies the safety of dereferencing s at line 15. The same assertion in line 17 would let us rule out the ABA problem in a linearizability proof. We get the assertion in line 35 from the tautology $x \mapsto_{\mathsf{m}} \_ \Rightarrow \neg\boxed{\mathsf{C} \mapsto x * x \mapsto \_ * \mathsf{true}}$. At line 36, we apply the specification of sync with $\tau = \boxed{(\circledast_{x \in A}\, x \mapsto_{\mathsf{e}} \_) * \mathsf{true}} \land (\circledast_{x \in A} \neg\boxed{\mathsf{C} \mapsto x * x \mapsto \_ * \mathsf{true}})$ and $p = (\circledast_{x \in A}\, x \mapsto_{\mathsf{m}} \_)$. The resulting since clause formalises (6) and allows us to justify that the Take action in line 41 does not violate $\Upsilon_{\mathsf{RCU}}$.

Like for hazard pointers, this proof implies that the algorithm does not leak memory, and that the ABA problem does not occur.

## 6 Related Work

Out of the three techniques for memory reclamation that we consider in this paper, only restricted versions of the non-blocking stack with hazard pointers that we handle in §B have been verified: in concurrent separation logic [13], a combination of separation logic and temporal logic [6], a reduction-based tool [3] and interval temporal logic [15]. These papers use different reasoning methods from the one we propose, none of which has been grounded in a pattern common to different algorithms.

Among the above-mentioned verification efforts, the closest to us technically is the work by Fu et al. [6], which proposed a combination of separation logic and temporal logic very similar to the one we use for formalising our method. We emphasise that we do not consider the logic we present in §3 as the main contribution of this paper, but merely as a *tool* for formalising our reasoning method. It is this method that is the main difference of our work in comparison to Fu et al. The method used by Fu et al. to verify a non-blocking stack with hazard pointers leads to a complicated proof that embeds a lot of implementation detail into its invariants and rely/guarantee conditions. In contrast, our proofs are conceptually simple and technically straightforward, due to the use of a strategy that captures the essence of the algorithms considered. Fu et al. also handle only an idealistic implementation of hazard pointers, where deallocations are not batched, and many assertions in the proof inherently rely on this simplification. We do not think that their proof would scale easily to the implementation that batches deallocations (§2), let alone other extensions we consider §B.

Having said that, we fully acknowledge the influence of the work by Fu et al. In particular, we agree that a combination of temporal and separation logics provides a useful means of reasoning about non-blocking algorithms. We hope that our formalisation of powerful proof patterns in such a combined logic will motivate verification researchers to adopt the pattern-based approach in verifying other complex concurrent algorithms.

**Acknowledgements.** We thank the following people for discussions and comments: Richard Bornat, Sungkeun Cho, Byron Cook, Wonchan Lee, Paul McKenney, Peter O'Hearn, Matthew Parkinson, Mooly Sagiv, Viktor Vafeiadis, Jonathan Walpole, Eran Yahav and Kwangkeun Yi.

## References

1. J. Boyland. Checking interference with fractional permissions. In *SAS*, 2003.
2. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.
3. T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, 2009.
4. X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.
5. K. Fraser. Practical lock-freedom. PhD Thesis. University of Cambridge, 2004.
6. M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, 2010.
7. M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *DISC*, 2002.
8. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 1990.
9. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.
10. P. McKenney. Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels. PhD Thesis. OGI, 2004.
11. M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 2004.
12. P. O'Hearn. Resources, concurrency and local reasoning. *TCS*, 2007.
13. M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, 2007.
14. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, 1985.
15. B. Tofan, G. Schellhorn, and W. Reif. Formal verification of a lock-free stack with hazard pointers. In *ICTAC*, 2011.
16. V. Vafeiadis. Modular fine-grained concurrency verification. PhD Thesis. University of Cambridge, 2008.

## A Additional Definitions

### A.1 Operational Semantics

We give a standard small-step operational semantics for our programming language, omitted from §3.5. Let us denote the set of commands $C$ by $\mathsf{Comm}$. The semantics is defined by in Figure 10. The definition uses an auxiliary relation $\longrightarrow_{\mathsf{tid}}\colon \mathsf{TConfig} \times \mathsf{TConfig}$ for $\mathsf{tid} \in \mathsf{ThreadID}$ describing a single step by thread $\mathsf{tid}$. Here $\mathsf{TConfig}$ is defined like $\mathsf{Config}$ in §3.5, but with $\mathsf{Comm}$ instead of $\mathsf{Prog}$. We assume that atomic blocks in programs are annotated with the actions used in their proofs. We annotate the arrows in transitions pertaining to execution of atomic blocks with the actions annotating the block. We annotate all other arrows with the special symbol $\ell$. We use $e$ as a meta variable ranging over transitions labels.

### A.2 Restrictions on Transformers for Primitive Commands

For the logic to be sound, we have to require that the transformers $f_\alpha^{\mathsf{tid}}$ defining the semantics of primitive commands $\alpha \in \mathsf{PComm}$ satisfy the standard condition of *locality* [2]. Let us lift $*$ to $\mathsf{State} \cup \{\top\}$ as follows:

$$\forall \theta \in \mathsf{State}.\, \theta * \top = \top * \theta = \top * \top = \top.$$

Then $f_\alpha^{\mathsf{tid}}$ is local when for any $\theta_1, \theta_2 \in \mathsf{State}$

$$(\theta_1 * \theta_2)\!\downarrow \wedge\, f_\alpha^{\mathsf{tid}}(\theta_1) \neq \top \Rightarrow f_\alpha^{\mathsf{tid}}(\theta_1 * \theta_2) = f_\alpha^{\mathsf{tid}}(\theta_1) * \{\theta_2\}.$$

Informally the condition states that:

– *Safety monotonicity:* if executing $\alpha$ from a state $\theta_1 * \theta_2$ results in an error, then so does executing $\alpha$ from a smaller state $\theta_1$: $f_\alpha^{\mathsf{tid}}(\theta_1 * \theta_2) = \top$ implies $f_\alpha^{\mathsf{tid}}(\theta_1) = \top$;
– *Frame property:* if executing $\alpha$ from a state $\theta_1$ does not produce an error, then executing $\alpha$ from a larger state $\theta_1 * \theta_2$, has the same effect and leaves $\theta_2$ unchanged.

### A.3 Additional definitions for $\mathsf{RAM}_\mathsf{e}$

We give transformers and axioms for standard primitive commands:

$$\alpha \ :=\ [\mathrm{x}] = \mathsf{y} \mid [\mathrm{y}] = [\mathrm{x}] \mid \mathsf{new}([\mathrm{x}]) \mid \mathsf{delete}([\mathrm{x}]) \mid \mathsf{assume}([\mathrm{x}])$$

where $\mathsf{x}$ and $\mathsf{y}$ are constants. Here square brackets denote pointer dereferencing. We give only simple cases of commands with constants as expressions[1]. The $\mathsf{assume}$ command acts as a filter on the state space of programs and is used to implement conditionals and loops. Its parameter is assumed to be non-zero after the command is executed. Using $\mathsf{assume}$, we can implement conditionals and loops as follows:

$$(\mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) = (\mathsf{assume}(E); C_1) + (\mathsf{assume}(!E); C_2);$$
$$(\mathsf{while}\ E\ \mathsf{do}\ C) = (\mathsf{assume}(E); C)^*;\ \mathsf{assume}(!E).$$

For a function $g$, we denote by $g[x : y]$ the function that has the same value as $g$ everywhere, except for $x$, where it has the value $y$. Transformers $f_\alpha^{\mathsf{tid}} : \mathsf{RAM}_\mathsf{e} \to$

[1] For more complex cases see: A. Appel and S. Blazy. Separation logic for small-step Cminor, *TPHOLs*, 2007.

$$\frac{\alpha \in \mathsf{PComm} \quad f^{\mathsf{tid}}_{\alpha}(\theta) \neq \top \quad \theta' \in f^{\mathsf{tid}}_{\alpha}(\theta)}{\alpha, \theta \xrightarrow{\ell}_{\mathsf{tid}} \mathsf{done}, \theta'}$$

$$\frac{\alpha \in \mathsf{PComm} \quad f^{\mathsf{tid}}_{\alpha}(\theta) = \top}{\alpha, \theta \xrightarrow{\ell}_{\mathsf{tid}} \top}$$

$$\frac{}{\mathsf{done}; C_2, \theta \xrightarrow{\ell}_{\mathsf{tid}} C_2, \theta}$$

$$\frac{C_1, \theta \xrightarrow{e}_{\mathsf{tid}} C_1', \theta'}{C_1; C_2, \theta \xrightarrow{e}_{\mathsf{tid}} C_1'; C_2, \theta'}$$

$$\frac{C_1, \theta \xrightarrow{e}_{\mathsf{tid}} \top}{C_1; C_2, \theta \xrightarrow{e}_{\mathsf{tid}} \top}$$

$$\frac{i \in \{1, 2\}}{C_1 + C_2, \theta \xrightarrow{\ell}_{\mathsf{tid}} C_i, \theta}$$

$$\frac{}{C^*, \theta \xrightarrow{\ell}_{\mathsf{tid}} C; C^*, \theta}$$

$$\frac{}{C^*, \theta \xrightarrow{e}_{\mathsf{tid}} \mathsf{done}, \theta}$$

$$\frac{C, \theta \xrightarrow{\ell}{}^{*}_{\mathsf{tid}} \mathsf{done}, \theta'}{\langle C \rangle_a, \theta \xrightarrow{a}_{\mathsf{tid}} \mathsf{done}, \theta'}$$

$$\frac{C, \theta \xrightarrow{\ell}{}^{*}_{\mathsf{tid}} \top}{\langle C \rangle_a, \theta \xrightarrow{a}_{\mathsf{tid}} \top}$$

$$\frac{C_{\mathsf{tid}}, \theta \xrightarrow{e}_{\mathsf{tid}} \top}{C_1 \parallel \ldots \parallel C_{\mathsf{tid}} \parallel \ldots \parallel C_n, \theta \xrightarrow{e} \top}$$

$$\frac{C_{\mathsf{tid}}, \theta \xrightarrow{e}_{\mathsf{tid}} C'_{\mathsf{tid}}, \theta'}{C_1 \parallel \ldots \parallel C_{\mathsf{tid}} \parallel \ldots \parallel C_n, \theta \xrightarrow{e} C_1 \parallel \ldots \parallel C'_{\mathsf{tid}} \parallel \ldots \parallel C_n, \theta'}$$

**Fig. 10.** Operational semantics of the programming language

$\mathcal{P}(\mathsf{RAM_e})^{\top}$ for the above primitive commands $\alpha$ are defined with the aid of the transition relation $\rightsquigarrow$ in Figure 11:

$$f^{\mathsf{tid}}_{\alpha}(\theta) = \bigcup\{\theta' \mid \alpha, \theta \rightsquigarrow \theta'\},$$

if $\alpha, \theta \not\rightsquigarrow \top$; otherwise, $f^{\mathsf{tid}}_{\alpha}(\theta) = \top$. These transformers are local [2]. Figure 12 shows instantiations of the LOCAL axiom of our logic for the above primitive commands.

## A.4 Additional Proof Rules

The proof rules of the logic omitted from Figure 4 are shown in Figure 13.

$$\begin{array}{lll}
[\mathtt{x}] = \mathsf{y}, & \theta[\mathtt{x} : (\_, \pi)] & \rightsquigarrow \theta[\mathtt{x} : (\mathsf{y}, \pi)] \\
[\mathsf{y}] = [\mathtt{x}], & \theta[\mathtt{x} : (v, \pi)][\mathsf{y} : (\_, \pi')] & \rightsquigarrow \theta[\mathtt{x} : (v, \pi)][\mathsf{y} : (v, \pi')] \\
[\mathsf{y}] = [\mathtt{x}], & \theta[\mathtt{x} : \mathsf{e}][\mathsf{y} : (\_, \pi)] & \rightsquigarrow \theta[\mathtt{x} : \mathsf{e}][\mathsf{y} : (\_, \pi)] \\
\mathsf{new}([\mathtt{x}]), & \theta[\mathtt{x} : (\_, \pi)] & \rightsquigarrow \theta[\mathtt{x} : (u, \pi)][u : (\_, 1)],\ \text{if } (\theta[\mathtt{x} : \_])(u)\!\uparrow \\
\mathsf{delete}([\mathtt{x}]), & \theta[\mathtt{x} : (u, \pi)][u : (\_, 1)] & \rightsquigarrow \theta[\mathtt{x} : (u, \pi)],\ \text{if } (\theta[\mathtt{x} : \_])(u)\!\uparrow \\
\mathsf{assume}([\mathtt{x}]), & \theta[\mathtt{x} : \mathsf{e}] & \rightsquigarrow \theta[\mathtt{x} : \mathsf{e}] \\
\mathsf{assume}([\mathtt{x}]), & \theta[\mathtt{x} : (u, \pi)] & \rightsquigarrow \theta[\mathtt{x} : (u, \pi)],\ \text{if } u \neq 0 \\
\mathsf{assume}([\mathtt{x}]), & \theta[\mathtt{x} : (u, \pi)] & \not\rightsquigarrow \quad \text{if } u = 0 \\
\alpha, & \theta & \rightsquigarrow \top,\ \text{otherwise}
\end{array}$$

**Fig. 11.** Transition relation for primitive commands over $\mathsf{RAM_e}$. We assume that $\mathtt{x} \neq \mathsf{y}$ in $[\mathsf{y}] = [\mathtt{x}]$. $\top$ indicates that the command faults. $\not\rightsquigarrow$ is used to denote that the command does not fault, but gets stuck. We use $\pi$ and $\pi'$ to denote a permission that is either $1$ or $\mathsf{m}$.

$$\frac{}{\{\mathtt{x} \mapsto_{\pi} \_\}\ [\mathtt{x}] = \mathsf{y}\ \{\mathtt{x} \mapsto_{\pi} \mathsf{y}\}}$$

$$\frac{}{\{\mathtt{x} \mapsto_{\pi} u * \mathsf{y} \mapsto_{\pi'} \_\}\ [\mathsf{y}] = [\mathtt{x}]\ \{\mathtt{x} \mapsto_{\pi} u * \mathsf{y} \mapsto_{\pi'} u\}}$$

$$\frac{}{\{\mathtt{x} \mapsto_{\pi} \_\}\ \mathsf{new}(\mathtt{x})\ \{\exists u.\, \mathtt{x} \mapsto_{\pi} u * u \mapsto_{1} \_\}}$$

$$\frac{}{\{\mathtt{x} \mapsto_{\pi} u * u \mapsto_{1} \_\}\ \mathsf{delete}(\mathtt{x})\ \{\mathtt{x} \mapsto_{\pi} u\}}$$

$$\frac{}{\{\mathtt{x} \mapsto_{\pi} u\}\ \mathsf{assume}([\mathtt{x}])\ \{\mathtt{x} \mapsto_{\pi} u \wedge u \neq 0\}}$$

$$\frac{}{\{\mathtt{x} \mapsto_{\mathsf{e}} \_\}\ \mathsf{assume}([\mathtt{x}])\ \{\mathtt{x} \mapsto_{\mathsf{e}} \_\}}$$

**Fig. 12.** Instantiations of the LOCAL axiom for the $\mathsf{RAM_e}$ algebra; `x` and `y` are constants. We use $\pi$ to denote a permission that is either $1$ or $\mathsf{m}$.

### A.5 Derivation of SHARED-I

The derivation of the SHARED-I is given in Figure 14.

## B Proofs of an Optimised Hazard Pointer Implementation

In Figures 15–18, we give a proof of memory safety of a non-blocking stack algorithm using hazard pointers. The implementation of hazard pointers in this algorithm is optimised or extended in several ways:

– Hazard pointers are dynamically allocated, which supports dynamic thread creation.
– The `reclaim` function scans the hazard pointer list only once. This optimisation is done in practice to ensure a bound on the number of non-reclaimed memory cells.
– For efficiency, the `detached` set is implemented as a list whose links are stored in the detached nodes themselves.

Note that the proof of `reclaim` and the actions involving hazard pointers do not mention the stack data structures. Thus this part of the proof is not specific to the stack algorithm. This is an improvement on the proof of §4 that we omitted there so as not to complicate presentation.

$$\frac{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_1\}\, C_1\, \{P_2\} \quad R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_2\}\, C_2\, \{P_3\}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_1\}\, C_1; C_2\, \{P_3\}}\ \textsc{Seq}$$

$$\frac{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C_1\, \{Q\} \quad R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C_2\, \{Q\}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C_1 + C_2\, \{Q\}}\ \textsc{Choice}$$

$$\frac{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C\, \{P\}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C^*\, \{P\}}\ \textsc{Loop}$$

$$\frac{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_1\}\, C\, \{Q_1\} \quad R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_2\}\, C\, \{Q_2\}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_1 \vee P_2\}\, C\, \{Q_1 \vee Q_2\}}\ \textsc{Disj}$$

$$\frac{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_1\}\, C\, \{Q_2\} \quad R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_2\}\, C\, \{Q_2\}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_1 \wedge P_2\}\, C\, \{Q_1 \wedge Q_2\}}\ \textsc{Conj}$$

$$\frac{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C\, \{Q\}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{\exists x.\, P\}\, C\, \{\exists x.\, Q\}}\ \textsc{Exists}_1$$

$$\frac{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C\, \{Q\}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{\forall x.\, P\}\, C\, \{\forall x.\, Q\}}\ \textsc{Forall}_1$$

$$\frac{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C\, \{Q\}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{\exists X.\, P\}\, C\, \{\exists X.\, Q\}}\ \textsc{Exists}_2$$

$$\frac{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C\, \{Q\}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{\forall X.\, P\}\, C\, \{\forall X.\, Q\}}\ \textsc{Forall}_2$$

**Fig. 13.** Additional proof rules of the logic

## C Proving Linearizability

In this section we show how the proof of the counter from §4 can be adjusted to establish its linearizability following the approach in [16]. We prove linearizability with respect to the following atomic specification:

```
int A = 0;

int abstract_inc() {
  int a;
  ⟨ a = A; A = a+1 ⟩;
  return a;
}
```

Following Vafeiadis [16], we embed the cell `A` storing the abstract state of the counter into the code of the algorithm. The above `abstract_inc` method is then executed at a single linearization point inside the implementation `inc`, which is a successful CAS.

We use the same actions as in the proof of memory safety, except Inc is redefined to change the abstract value of the counter:

$$\mathsf{C} \mapsto x * x \mapsto k * A \mapsto k * X \rightsquigarrow$$
$$\mathsf{C} \mapsto y * y \mapsto k+1 * A \mapsto k+1 * x \mapsto_{\mathsf{e}} \_ * X \quad (\mathsf{Inc})$$

$$\frac{\begin{array}{c} p \Rightarrow l * \mathsf{true} \qquad p_s \Rightarrow p'_s \qquad q_s \Rightarrow q'_s \qquad a = (l \mid p'_s \rightsquigarrow q'_s) \in G \\ (p \wedge \boxed{p_s} \wedge \neg(\boxed{g} \mathsf{\ since\ } \boxed{r})) \Rightarrow (p \wedge \neg(\boxed{g} \mathsf{\ since\ } \boxed{r}) \wedge \boxed{p_s \wedge \neg(g \wedge r)}) \\ \emptyset, \emptyset, \mathsf{true} \vdash_{\mathsf{tid}} \{p * (p_s \wedge \neg(g \wedge r))\}\, C\, \{q * (q_s \wedge (g \wedge r \Rightarrow c))\} \\ (q \wedge ((\neg(\boxed{g} \mathsf{\ since\ } \boxed{r}) \wedge \boxed{p_s \wedge \neg(g \wedge r)}) \lhd \boxed{q_s \wedge (g \wedge r \Rightarrow c)})) \Rightarrow (q \wedge \boxed{q_s} \wedge ((\boxed{g} \mathsf{\ since\ } \boxed{r}) \Rightarrow \boxed{c})) \end{array}}{\emptyset, G, \mathsf{true} \vdash_{\mathsf{tid}} \{p \wedge \boxed{p_s} \wedge \neg(\boxed{g} \mathsf{\ since\ } \boxed{r})\}\, \langle C \rangle_a\, \{q \wedge \boxed{q_s} \wedge ((\boxed{g} \mathsf{\ since\ } \boxed{r}) \Rightarrow \boxed{c})\}}$$

(SHARED, CONSEQ)

$$\frac{\begin{array}{c} p \Rightarrow l * \mathsf{true} \qquad p_s \Rightarrow p'_s \qquad q_s \Rightarrow q'_s \qquad a = (l \mid p'_s \rightsquigarrow q'_s) \in G \\ (p \wedge \boxed{p_s} \wedge (\boxed{g} \mathsf{\ since\ } \boxed{r}) \wedge \boxed{c}) \Rightarrow (p \wedge \boxed{p_s \wedge g \wedge c}) \\ \emptyset, \emptyset, \mathsf{true} \vdash_{\mathsf{tid}} \{p * (p_s \wedge g \wedge c)\}\, C\, \{q * (q_s \wedge (g \Rightarrow c))\} \\ (q \wedge (\boxed{p_s \wedge g \wedge c} \lhd \boxed{q_s \wedge (g \Rightarrow c)})) \Rightarrow (q \wedge \boxed{q_s} \wedge ((\boxed{g} \mathsf{\ since\ } \boxed{r}) \Rightarrow \boxed{c})) \end{array}}{\emptyset, G, \mathsf{true} \vdash_{\mathsf{tid}} \{p \wedge \boxed{p_s} \wedge (\boxed{g} \mathsf{\ since\ } \boxed{r}) \wedge \boxed{c}\}\, \langle C \rangle_a\, \{q \wedge \boxed{q_s} \wedge ((\boxed{g} \mathsf{\ since\ } \boxed{r}) \Rightarrow \boxed{c})\}}$$

(SHARED, CONSEQ)

$$\frac{\dfrac{\begin{array}{c} \emptyset, G, \mathsf{true} \vdash_{\mathsf{tid}} \{p \wedge \boxed{p_s} \wedge \neg(\boxed{g} \mathsf{\ since\ } \boxed{r})\}\, \langle C \rangle_a\, \{q \wedge \boxed{q_s} \wedge ((\boxed{g} \mathsf{\ since\ } \boxed{r}) \Rightarrow \boxed{c})\} \\ \emptyset, G, \mathsf{true} \vdash_{\mathsf{tid}} \{p \wedge \boxed{p_s} \wedge ((\boxed{g} \mathsf{\ since\ } \boxed{r}) \wedge \boxed{c})\}\, \langle C \rangle_a\, \{q \wedge \boxed{q_s} \wedge ((\boxed{g} \mathsf{\ since\ } \boxed{r}) \Rightarrow \boxed{c})\} \end{array}}{\emptyset, G, \mathsf{true} \vdash_{\mathsf{tid}} \{p \wedge \boxed{p_s} \wedge (\neg(\boxed{g} \mathsf{\ since\ } \boxed{r}) \vee ((\boxed{g} \mathsf{\ since\ } \boxed{r}) \wedge \boxed{c}))\}\, \langle C \rangle_a\, \{q \wedge \boxed{q_s} \wedge ((\boxed{g} \mathsf{\ since\ } \boxed{r}) \Rightarrow \boxed{c})\}}}{\emptyset, G, \mathsf{true} \vdash_{\mathsf{tid}} \{p \wedge \boxed{p_s} \wedge ((\boxed{g} \mathsf{\ since\ } \boxed{r}) \Rightarrow \boxed{c})\}\, \langle C \rangle_a\, \{q \wedge \boxed{q_s} \wedge ((\boxed{g} \mathsf{\ since\ } \boxed{r}) \Rightarrow \boxed{c})\}}$$

(DISJ, CONSEQ)

**Fig. 14.** Derivation of SHARED-I

The $I$ predicate, describing the layout of the shared state, now includes the abstract state as well:

$$I \Leftrightarrow \boxed{H * \exists y, k.\, \mathtt{C} \mapsto y * y \mapsto k * A \mapsto k * \mathsf{true_e}}$$

All other assertions are used without changes. The proof outline for `inc` is given in Figure 19; the proof of `retire` in Figure 7 does not change. We write `CAS'`$_{a,b}$`(addr,v1,v2)` for

```
if (nondet()) {
  ⟨assume(*addr == v1); *addr = v2;
    Res = abstract_inc() ⟩_a; return 1;
} else { ⟨assume(*addr != v1)⟩_b; return 0; }
```

The key assertion in the proof is the one in lines 24 and 28. As we already noted in §4, the conjunct $\boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}}$ ensures that the node pointed to by `s` cannot be reallocated until the CAS in line 29, and thus, the ABA problem does not occur. More formally, the assertion in lines 24 and 28 is stable, because the environment cannot execute an $\mathsf{Inc}$ action inserting $s$ while $s \mapsto_{\mathsf{e}} \_$ is in the shared state: $s \mapsto \_ * s \mapsto_{\mathsf{e}} \_$ is inconsistent. This ensures that, when the CAS in line 29 succeeds, the abstract value `A` is equal to `v`. This allows us to establish in the postcondition that the abstract return value, given by `Res`, is equal to the concrete one, given by `v`, which entails linearizability.

## D Reasoning about Epoch-based Reclamation

**Informal summary.** Figure 20 presents an implementation of the running example using epoch-based reclamation [5], which is in some respects more efficient than hazard

$$
\begin{aligned}
\mathsf{stack}(x) &\iff (\mathsf{emp} \wedge x = 0) \vee (\exists y, v.\, x \mapsto y, v * \mathsf{stack}(y))\\
\mathsf{lsh}_A(z) &\iff (A = \emptyset \wedge z = 0) \vee (z \in A \wedge \exists z'.\, z \mapsto \_, \_, z' * \mathsf{lsh}_{A \setminus \{z\}}(z'))\\
H &\iff \exists z, A.\, \mathtt{HPL} \mapsto z * \mathsf{lsh}_A(z)\\
I &\iff \boxed{H * \exists y.\, \mathtt{Top} \mapsto y * \mathsf{stack}(y) * \mathsf{true}_{\mathsf{e}}}\\
\mathsf{myhp}(h, t, x) &\iff \boxed{\exists z, A.\, h \in A \wedge \mathtt{HPL} \mapsto z * \mathsf{lsh}_A(z) * \mathsf{true}} \wedge \boxed{h \mapsto x, t, \_ * \mathsf{true}}\\
D(x, A) &\iff (A = \emptyset \wedge x = 0) \vee (x \in A \wedge \exists x'.\, x \mapsto x', \_ * D(x', A \setminus \{x\}))\\
\Upsilon_{\mathsf{DHP}} &\iff \forall h, x.\, \big(\mathsf{myhp}(h, \_, x) \text{ since } \boxed{\mathtt{Top} \mapsto x * x \mapsto \_ * \mathsf{true}}\big) \Rightarrow \boxed{x \mapsto_{\mathsf{e}} \_, \_ * \mathsf{true}}
\end{aligned}
$$

$$\mathtt{HPL} \mapsto z * X \;\rightsquigarrow\; \mathtt{HPL} \mapsto y * y \mapsto \_, t, z * X \qquad (\mathsf{DHP}_{\mathsf{tid}})$$

$$\mathtt{HPL} \mapsto y * (\mathsf{lsh}_A(y) \wedge (h \mapsto \_, t, z * Y)) * X \;\rightsquigarrow\; \mathtt{HPL} \mapsto y * (\mathsf{lsh}_A(y) \wedge (h \mapsto \_, t, z * Y)) * X \qquad (\mathsf{WHP}_{\mathsf{tid}})$$

$$X \rightsquigarrow X \qquad (\mathsf{Id})$$

$$\mathtt{Top} \mapsto x * X \;\rightsquigarrow\; \mathtt{Top} \mapsto z * z \mapsto x, v * X \qquad (\mathsf{Push})$$

$$\mathtt{Top} \mapsto x * x \mapsto y, v * X \rightsquigarrow \mathtt{Top} \mapsto y * x \mapsto_{\mathsf{e}} y, v * X \qquad (\mathsf{Pop})$$

$$x \mapsto_{\mathsf{m}} \_, \_ \mid x \mapsto_{\mathsf{e}} \_, \_ * X \;\rightsquigarrow\; X \qquad (\mathsf{Take})$$

$$x \mapsto_{\mathsf{e}} \_, \_ * X \;\rightsquigarrow\; x \mapsto_{\mathsf{e}} \_, \_ * X \qquad (\mathsf{Write})$$

$$G_{\mathsf{tid}} = \{\mathsf{DHP}_{\mathsf{tid}}, \mathsf{WHP}_{\mathsf{tid}}, \mathsf{Push}, \mathsf{Pop}, \mathsf{Take}, \mathsf{Id}, \mathsf{Write}\}; \quad R_{\mathsf{tid}} = \bigcup_{t \neq \mathsf{tid}} G_t$$

**Fig. 15.** Assertions, actions, and rely/guarantee conditions used in the proof of the stack with an optimised hazard pointer implementation

pointers. The algorithm uses a *global epoch* counter GE, incremented after every round of reclamation. In fact, it is sufficient to keep track only of the three most recent epochs, so GE $\in \{0, 1, 2\}$ and all arithmetic is done modulo 3. Every thread maintains its local snapshot of the counter, called its *local epoch*, in the corresponding entry of the LE array. At the beginning of every `inc` operation, the local epoch is updated to the global one (line 10), and it stays the same for the whole duration of `inc`. The following contract, included into the temporal invariant of the algorithm, is respected when reclaiming nodes:

"for all $t$ and $x$, if thread has stayed in the same epoch since it saw C pointing to $x$, then $x$ is allocated." (12)

The grace period for $t$ and $x$ thus starts when $t$ sees $x$ pointed to by C and lasts for as long as it does not update its local epoch. We now explain how `reclaim` ensures that (12) is preserved.

The algorithm batches deallocations using `detached` sets for every thread *and epoch*: `reclaim` stores the address of the detached node s into the `detached` set for the local epoch of the current thread (line 21). It can then decide to perform a round of reclamation, incrementing the global epoch GE at its end (line 36). To avoid conflicts when several threads decide to reclaim, the corresponding code is protected by a lock (lines 24 and 37).

The reclamation is only done *when all the threads have the same local epoch as the global one* (lines 25–29). Once the global epoch is incremented at line 36, every thread just needs to execute an `inc` operation for this condition to hold. In the counter implementations using hazard pointers and RCU, the thread that detached a node was in

```
struct Node { Node *next;
              int val; };
Node *Top = 0;
struct HazardNode { Node *p;
                    HazardNode *next;
                    TID tid; };
HazardNode *HPL = 0;
ThreadLocal HazardNode *h = 0;
ThreadLocal Node* detached = {∅};

int pop() {
  int v; Node *x, *p, *p2;

  $\{V \Vdash F_{\mathsf{tid}} \land (h \neq 0 \implies \mathsf{myhp}(h, \mathsf{tid}, \_)) \land I\}$
  if (h == 0) {
    h = new HazardNode();
    h->tid = tid();
    do {
      HazardNode *o = HPL;
      h->next = o;
    } while (!CAS$_{\mathsf{DHP_{tid}},\mathsf{Id}}$(&HPL,o,h);
  }
  $\{V \Vdash F_{\mathsf{tid}} \land \mathsf{myhp}(h, \mathsf{tid}, \_) \land I\}$

  do {
    $\{V \Vdash F_{\mathsf{tid}} \land \mathsf{myhp}(h, \mathsf{tid}, \_) \land I\}$
    do {
      $\{V \Vdash F_{\mathsf{tid}} \land \mathsf{myhp}(h, \mathsf{tid}, \_) \land I\}$
      $\langle$p = Top$\rangle_{\mathsf{Id}}$;
      $\{V \Vdash F_{\mathsf{tid}} \land \mathsf{myhp}(h, \mathsf{tid}, \_) \land I\}$
      if (p == NULL) return EMPTY;
      $\{V \Vdash p \neq 0 \land F_{\mathsf{tid}} \land \mathsf{myhp}(h, \mathsf{tid}, \_) \land I\}$
      $\langle$myhp->p = p$\rangle_{\mathsf{WHP_{tid}}}$;
      $\{V \Vdash p \neq 0 \land F_{\mathsf{tid}} \land \boxed{\mathsf{myhp}(h, \mathsf{tid}, p) * \mathsf{true}} \land I\}$
      $\langle$p2 = Top$\rangle_{\mathsf{Id}}$;
      $\{V \Vdash p \neq 0 \land F_{\mathsf{tid}} \land I \land (\boxed{\mathsf{myhp}(h, \mathsf{tid}, p) * \mathsf{true}}$ since
        $\boxed{\mathsf{Top} \mapsto p2 * \mathsf{stack}(p2) * \mathsf{true_e}})\}$
    } while (p != p2);
    $\{V \Vdash p \neq 0 \land F_{\mathsf{tid}} \land I \land (\boxed{\mathsf{myhp}(h, \mathsf{tid}, p) * \mathsf{true}}$ since
      $\boxed{\mathsf{Top} \mapsto p * \mathsf{stack}(p) * \mathsf{true_e}}) \land \boxed{p \mapsto_{\mathsf{e}} \_, \_ * \mathsf{true}}\}$
    $\langle$n = p->next;$\rangle_{\mathsf{Id}}$;
    $\{V \Vdash \exists x, y.\, p \neq 0 \land F_{\mathsf{tid}} \land \boxed{H * \mathsf{Top} \mapsto y * \mathsf{stack}(y) * \mathsf{true_e}} \land$
     $(\boxed{\mathsf{myhp}(h, \mathsf{tid}, p) * \mathsf{true}}$ since $\boxed{\mathsf{Top} \mapsto p * \mathsf{stack}(p) * \mathsf{true_e}}) \land$
     $\boxed{p \mapsto_{\mathsf{e}} x, \_ * \mathsf{true} \land (y = p \implies n = x)}\}$
  } while (!CAS$_{\mathsf{Pop},\mathsf{Id}}$(&Top, p, n));
  $\{V \Vdash p \mapsto_{\mathsf{m}} \_, \_ * F_{\mathsf{tid}} \land I \land \boxed{p \mapsto_{\mathsf{e}} \_, \_ * \mathsf{true}}\}$
  v = p->val;
  $\{V \Vdash p \mapsto_{\mathsf{m}} \_, \_ * F_{\mathsf{tid}} \land I \land \mathsf{myhp}(h, \mathsf{tid}, \_) \land$
   $\boxed{p \mapsto_{\mathsf{e}} \_, \_ * \mathsf{true}}\}$
  reclaim(p);
  $\{V \Vdash F_{\mathsf{tid}} \land \mathsf{myhp}(h, \mathsf{tid}, \_) \land I\}$
  return v;
}
```

**Fig. 16.** Proof outline for pop for a non-blocking stack with an optimised hazard pointer implementation. Here $V$ is $x$, $p$, $p2$, $n$, $v$, $o$, $h$, $c$, $S$, $used$.

```
void reclaim(int* p) {
  $\{V \Vdash p \mapsto_{\mathsf{m}} \_, \_ * D(detached, \_) \wedge \boxed{p \mapsto_{\mathsf{e}} \_, \_ * \mathsf{true}} \wedge I\}$
  insert(&detached,p);
  $\{V \Vdash D(detached, \_) \wedge I\}$
  if (nondet()) return;
  $\{V \Vdash \exists A.\, D(detached, A) \wedge I \wedge \forall x \in A.\, \neg \boxed{x \mapsto_1 \_, \_ * \mathsf{true}}\}$
  $\langle$HazardNode *c = HPL$\rangle_{\mathsf{Id}}$;
  $\{V \Vdash \exists A, P.\, D(detached, A) \wedge I \wedge \forall x \in A.\, (\neg \boxed{x \mapsto_1 \_, \_ * \mathsf{true}})$ since $\boxed{\exists c.\, \mathtt{HPL} \mapsto c * \mathsf{lsh}_P(c) * \mathsf{true}} \wedge$
    $\boxed{\exists z, N.\, P \subseteq N \wedge \mathtt{HPL} \mapsto z * \mathsf{lsh}_N(z)}\}$
  Set *S = $\emptyset$;
  $\{V \Vdash \exists A, P.\, D(detached, A) \wedge I \wedge \forall x \in A.\, (\neg \boxed{x \mapsto_1 \_, \_ * \mathsf{true}})$ since $\boxed{\exists c.\, \mathtt{HPL} \mapsto c * \mathsf{lsh}_P(c) * \mathsf{true}} \wedge$
    $\boxed{\exists z, N.\, P \subseteq N \wedge \mathtt{HPL} \mapsto z * \mathsf{lsh}_N(z)}\}$
  while (c != NULL) {
    $\langle$Node *p = c->p$\rangle_{\mathsf{Id}}$;
    insert(S,p);
    $\langle$c = c->next$\rangle_{\mathsf{Id}}$;
  }
  $\{V \Vdash \exists A, P.\, D(detached, A) \wedge I \wedge (\forall x \in A.\, (\neg \boxed{x \mapsto_1 \_, \_ * \mathsf{true}})$ since $\boxed{\exists c.\, \mathtt{HPL} \mapsto c * \mathsf{lsh}_P(c) * \mathsf{true}}) \wedge$
   $(\forall p \in P.\, \exists v \in S.\, \forall x \in A.\, (\neg \boxed{x \mapsto_1 \_, \_ * \mathsf{true}})$ since $\boxed{p \mapsto v * \mathsf{true}})\}$
  Node *used = 0;
  while (!isEmpty(&detached)) {
    $\{V \Vdash \exists A, P, n.\, D(detached, A) * A \neq \emptyset * D(\mathit{used}, \_) \wedge$
     $I \wedge (\forall x \in A.\, (\neg \boxed{x \mapsto_1 \_, \_ * \mathsf{true}})$ since $\boxed{\exists c.\, \mathtt{HPL} \mapsto c * \mathsf{lsh}_P(c) * \mathsf{true}}) \wedge$
    $(\forall p \in P.\, \exists v \in S.\, \forall x \in A.\, (\neg \boxed{x \mapsto_1 \_, \_ * \mathsf{true}})$ since $\boxed{p \mapsto v * \mathsf{true}})\}$
    Node *n = popd(&detached);
    if (!member(S,n)) {
      $\{V \Vdash \exists A, P.\, D(detached, A) * D(\mathit{used}, \_) * (n \mapsto_{\mathsf{m}} \_, \_ \wedge \boxed{n \mapsto_{\mathsf{e}} \_, \_ * \mathsf{true}}) \wedge$
       $I \wedge (\forall x \in A \cup \{n\}.\, (\neg \boxed{x \mapsto_1 \_, \_ * \mathsf{true}})$ since $\boxed{\exists c.\, \mathtt{HPL} \mapsto c * \mathsf{lsh}_P(c) * \mathsf{true}}) \wedge$
       $(\forall p \in P.\, \exists v \in S.\, \forall x \in A.\, (\neg \boxed{x \mapsto_1 \_, \_ * \mathsf{true}})$ since $\boxed{p \mapsto v * \mathsf{true}}) \wedge$
       $(\forall p \in P.\, (\neg \boxed{n \mapsto_1 \_, \_ * \mathsf{true}})$ since $\neg \boxed{p \mapsto n * \mathsf{true}})\}$
      $\langle$ ; $\rangle_{\mathsf{Take}}$
      $\{V \Vdash \exists A, P.\, * D(detached, A) * D(\mathit{used}, \_) * n \mapsto_1 \_, \_ \wedge$
       $I \wedge (\forall x \in A.\, (\neg \boxed{x \mapsto_1 \_, \_ * \mathsf{true}})$ since $\boxed{\exists c.\, \mathtt{HPL} \mapsto c * \mathsf{lsh}_P(c) * \mathsf{true}}) \wedge$
       $(\forall p \in P.\, \exists v \in S.\, \forall x \in A.\, (\neg \boxed{x \mapsto_1 \_, \_ * \mathsf{true}})$ since $\boxed{p \mapsto v * \mathsf{true}})\}$
      free(n);
    } else
      insert(&used, n);
  }
  $\{V \Vdash D(detached, \emptyset) * D(\mathit{used}, \_) \wedge I\}$
  moveAll(&detached, &used);
  $\{V \Vdash D(detached, \_) * D(\mathit{used}, \emptyset) \wedge I\}$
}
```

**Fig. 17.** Proof outline for `reclaim` for a non-blocking stack with an optimised hazard pointer implementation. Here $V$ is $x$, $p$, $p2$, $n$, $v$, $o$, $h$, $c$, $S$, $\mathit{used}$.

charge of deallocating it. Here, in contrast, a thread that performs a round of reclamation clears the `detached` sets of all threads *for the epoch behind the current one by two*, i.e., $\mathtt{GE} \ominus 2 = \mathtt{GE} \oplus 1$ ($\oplus$ and $\ominus$ denote the operations modulo 3). This clears the `detached` sets to be used by threads after `GE` is incremented at line 36. We note for the future that, since the global epoch is incremented only when all threads are in sync with it,

$$\text{“every thread is always either in the global epoch or one step behind it.”} \qquad (13)$$

```
$\{s \mapsto x * D(x, A) * p \mapsto_m \_, \_ \wedge \boxed{p \mapsto_e \_, \_ * \mathsf{true}}\}$
insert(s,p) {
  $\{\exists A.\, s \mapsto x * D(x, A) * p \mapsto_m \_, \_ \wedge \boxed{p \mapsto_e \_, \_ * \mathsf{true}}\}$
  $\langle$*p = x$\rangle_{\mathsf{Write}}$;
  $\{\exists x.\, s \mapsto x * D(x, A) * p \mapsto_m x, \_ \wedge \boxed{p \mapsto_e x, \_ * \mathsf{true}}\}$
  *s = p;
  $\{s \mapsto p * D(x, A) * p \mapsto_m x, \_ \wedge \boxed{p \mapsto_e x, \_ * \mathsf{true}}\}$
}
$\{s \mapsto p * D(p, A \uplus \{p\})\}$

$\{s \mapsto x * D(x, A)\}$
isEmpty(s) {
  $\{s \mapsto x * D(x, A)\}$
  return *s == NULL;
  $\{s \mapsto x * D(x, A) \wedge (\mathit{isEmpty} \iff (x = 0))\}$
}
$\{s \mapsto x * D(x, A) \wedge (\mathit{isEmpty} \iff A = \emptyset)\}$

$\{s \mapsto x * D(x, A) \wedge A \neq \emptyset\}$
popd(s) {
  $\{\exists y.\, s \mapsto x * x \mapsto_m y, \_ \wedge \boxed{x \mapsto_e y, \_ * \mathsf{true}} * D(y, A \setminus \{x\})\}$
  Node p = *s;
  *s = s->next;
  $\{\exists y.\, s \mapsto y * p \mapsto_m y, \_ \wedge \boxed{p \mapsto_e y, \_ * \mathsf{true}} * D(y, A \setminus \{x\})\}$
  return p;
}
$\{\exists y.\, s \mapsto y * \mathit{popd} \mapsto_m \_, \_ \wedge \boxed{\mathit{popd} \mapsto_e \_, \_ * \mathsf{true}} * D(y, A \setminus \{x\})\}$

$\{t \mapsto 0 * s \mapsto y * D(y, A)\}$
moveAll(t,s) {
  $\{t \mapsto 0 * s \mapsto y * D(y, A)\}$
  *t = *s;
  *s = 0;
  $\{s \mapsto 0 * t \mapsto y * D(y, A)\}$
}
$\{s \mapsto 0 * t \mapsto y * D(y, A)\}$
```

**Fig. 18.** The implementation of the linked list of detached nodes used in the stack with an optimised hazard pointer implementation

The check at lines 25–29 and the fact that nodes are reclaimed from the epoch two steps behind ensure the preservation of (12). Namely, assume that, at line 33, a thread is reclaiming a node $x$ from the `detached` set of a thread $t$ for epoch $e \oplus 1$; then the current epochs of all threads and the global one are equal to $e$. To ensure that this is safe, we need to establish the negation of the 'since' clause of (12) for the node $x$ and all threads $t'$, i.e.,

“for all $t'$, `C` has not pointed to $x$ <u>since</u> $t'$ was not in epoch $e$.” (14)

For algorithms using hazard pointers and RCU we could establish such a fact right in the `reclaim` function: after checking that the hazard pointers do not point to $x$, or after executing sync. However, here there is no obvious way to establish (14) in `reclaim`. Instead, it has to be established earlier, but when?

The earliest we could try to establish (14) is at the time when $x$ is detached, as from that moment and until its deallocation, `C` cannot point to it. We cannot establish (14)

```
int *C = new int(0), *HP[N] = {0}; Set detached[N] = {∅};
int A = 0;

int inc() {
  int v, *n, *s, *s2, Res;
  $\{V \Vdash F_{\mathsf{tid}} \land I\}$
  n = new int;
  do {
    $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \land I\}$
    do {
      $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \land I\}$
      $\langle$s = C$\rangle_{\mathsf{Id}}$;
      $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \land I\}$
      $\langle$HP[tid-1] = s$\rangle_{\mathsf{HP_{tid}}}$;
      $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \land I \land \boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}\}$
      $\langle$s2 = C$\rangle_{\mathsf{Id}}$;
      $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \land I \land (\boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}\ \mathsf{since}\ \boxed{\mathtt{C} \mapsto s2 * s2 \mapsto \_ * \mathsf{true}})\}$
    } while (s != s2);
    $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \land I \land \boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}} \land$
     $(\boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}\ \mathsf{since}\ \boxed{\mathtt{C} \mapsto s * s \mapsto \_ * \mathsf{true}})$
    $\langle$v = *s$\rangle_{\mathsf{Id}}$;
    $\{V \Vdash n \mapsto \_ * F_{\mathsf{tid}} \land \boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}} \land$
     $\boxed{H * \exists y, k.\, \mathtt{C} \mapsto y * y \mapsto k * A \mapsto k * \mathsf{true_e} \land (y = s \Rightarrow k = v)}$
     $\land (\boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}\ \mathsf{since}\ \boxed{\mathtt{C} \mapsto s * s \mapsto \_ * \mathsf{true}})$
    *n = v+1;
    $\{V \Vdash n \mapsto v+1 * F_{\mathsf{tid}} \land \boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}} \land$
     $\boxed{H * \exists y, k.\, \mathtt{C} \mapsto y * y \mapsto k * A \mapsto k * \mathsf{true_e} \land (y = s \Rightarrow k = v)}$
     $\land (\boxed{\mathtt{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}\ \mathsf{since}\ \boxed{\mathtt{C} \mapsto s * s \mapsto \_ * \mathsf{true}})$
  } while (!CAS'$_{\mathsf{Inc,Id}}$(&C, s, n));
  $\{V \Vdash v = Res \land s \mapsto_{\mathsf{m}} \_ * F_{\mathsf{tid}} \land I \land \boxed{s \mapsto_{\mathsf{e}} \_ * \mathsf{true}}\}$
  reclaim(s);
  $\{V \Vdash v = Res \land F_{\mathsf{tid}} \land I\}$
  return v; }
```

**Fig. 19.** Proof outline for the linearizability of `inc` with hazard pointers. Here $V$ is $v$, $n$, $s$, $s2$, $my$, $in\_use$, $i$, $Res$.

right when detaching $x$: then the thread $t$ is at epoch $e \oplus 1$, and by (13), we might well have other threads at epoch $e$ that have seen $x$. However, by (13), the global epoch at this point can be either $e \oplus 1$ or $e \oplus 2$, but not $e$. Hence, before $x$ is deallocated (at epoch $e$), the global epoch must be incremented from $e \oplus 2$ to $e$. It is at this point that we can establish (14), as all the threads $t'$ are at the epoch $e \oplus 2$, and $x$ is in a `detached` set and so cannot be pointed to by `C`. Thus, the following assertion becomes part of the temporal invariant:

$$\text{“If the global epoch is } e\text{, and } x \text{ is in } \mathtt{detached}[t][e \oplus 1] \text{ for some } t\text{, then for all } t'\text{, } \mathtt{C} \text{ has not pointed to } x \ \underline{\text{since}}\ t' \text{ was not in epoch } e\text{.”} \quad (15)$$

The temporal invariant of the algorithm is the conjunction of (12) and (15). Thus, the assertions describing the main protocol and the fact a reclaimer establishes before deallocating a node are of the same form as for the algorithms considered in §4 and §5. However, in this case, the latter assertion has to be carried around in the proof as part of the temporal invariant. Nevertheless, the resulting invariant is easy to establish and

```
1  int GE = 0;
2  int LE[N] = {0};
3  Set detached[N][3]
4    = {∅};
5  Lock Elock;
6
7  int inc() {
8    int v, *s, *n;
9    n = new int;
10   LE[tid-1] = GE;
11   do {
12     s = C;
13     v = *s;
14     *n = v+1;
15   } while
16     (!CAS(&C,s,n));
17   reclaim(s);
18   return v;
19 }
20 reclaim(int *s) {
21   insert(
22     detached[tid-1][LE[tid-1]],s);
23   if (nondet()) return;
24   lock(Elock);
25   for (int i = 0; i < N; i++)
26     if (GE != LE[i]) {
27       unlock(Elock);
28       return;
29     }
30   for (int i = 0; i < N; i++)
31     while (!isEmpty(
32       detached[i][(GE+1)%3])) {
33         free(pop(
34           detached[i][(GE+1)%3]));
35     }
36   GE = (GE+1)%3;
37   unlock(Elock);
38   return; }
```

**Fig. 20.** A shared counter with epoch-based reclamation

maintain (§D).

**Formal proof.** The proof outline for the algorithm in Figure 20 is given in Figures 22 and 23, and the auxiliary assertions, in Figure 21.

The key places in the proof are as follows. When a thread updates its local epoch to the global one at line 11, it transfers the detached set for the old epoch to the shared state. The second conjunct of $\Upsilon_{\mathsf{E}}$ is trivially satisfied for this detached set, as the global epoch is one step ahead of the old one, not one step behind. The safety of Pop at line 25 is ensured by the second conjunct in the temporal invariant. Finally, when the global epoch is incremented from $e \ominus 1$ to $e$ at line 31, all the threads are at epoch $e \ominus 1$, which implies the second conjunct of $\Upsilon_{\mathsf{E}}$.

## E Proof of Soundness

**Notation.** For each action $a = (l \mid p_s \rightsquigarrow q_s)$, we write $\theta_l, \theta_s, \theta'_s \in [\![a]\!]$ to mean that

$$\exists \mathbf{i}.\ (\theta_l, \mathbf{i} \vDash l) \wedge (\theta_s, \mathbf{i} \vDash p_s) \wedge (\theta'_s, \mathbf{i} \vDash q_s).$$

We write $\mathsf{stab}_{R,\Upsilon}(Q)$ as a shorthand for assertion $Q$ is stable under $R$ and temporal invariant $\Upsilon$. We leave write transitions with arrows which are not annotated to mean either an $\ell$-annotated arrow or an $a$-annotated one, for some action $a$.

### E.1 Soundness for Commands

**Proof strategy.** Our proof strategy follows Vafeiadis[2]: The $\mathsf{TSafe}^{\mathsf{tid}}$ predicate (Definition 4) formalises the notion of a command $C$ executed by a thread tid starting from an abstract local state $\theta_l$, a (non-empty) abstract history $\xi\theta_s$ (whose current shared state is $\theta_s$), and a logical environment $\mathbf{i}$; establishing a postcondition $Q$, while preserving the temporal invariant $\Upsilon$ and respecting the guarantee $G$–provided the environment modifies the shared state according to rely $R$ while preserving the same temporal invariant $\Upsilon$.

[2] V. Vafeiadis. Concurrent Separation Logic and Operational Semantics. In *MFPS*, 2011.

$$\mathtt{C} \mapsto x * x \mapsto \_ * X \rightsquigarrow \mathtt{C} \mapsto y * y \mapsto \_ * x \mapsto_e \_ * X \qquad (\mathsf{Inc})$$

$$\mathtt{GE} \mapsto e * \mathtt{LE}[\mathsf{tid}-1] \mapsto l * C(\mathsf{tid}, e) * X \rightsquigarrow$$
$$\mathtt{GE} \mapsto e * \mathtt{LE}[\mathsf{tid}-1] \mapsto e * C(\mathsf{tid}, l) * X \qquad (\mathsf{UpdateLE}_{\mathsf{tid}})$$

$$\mathtt{Elock} \mapsto 0 \rightsquigarrow \mathtt{Elock} \mapsto \mathsf{tid} * X \qquad (\mathsf{Lock}_{\mathsf{tid}})$$

$$\mathtt{Elock} \mapsto \mathsf{tid} \rightsquigarrow \mathtt{Elock} \mapsto 0 * X \qquad (\mathsf{Unlock}_{\mathsf{tid}})$$

$$\mathtt{Elock} \mapsto \mathsf{tid} * \mathtt{GE} \mapsto e *$$
$$\mathtt{detached}[t-1][e \oplus 1] \mapsto A \uplus \{x\} * x \mapsto \_ \rightsquigarrow$$
$$\mathtt{GE} \mapsto e * \mathtt{detached}[t-1][e \oplus 1] \mapsto A \qquad (\mathsf{Pop}_{\mathsf{tid}})$$

$$\mathtt{Elock} \mapsto \mathsf{tid} * \mathtt{GE} \mapsto e * (\circledast_t \mathtt{LE}[t-1] \mapsto e) \rightsquigarrow$$
$$\mathtt{Elock} \mapsto \mathsf{tid} * \mathtt{GE} \mapsto e \oplus 1 * (\circledast_t \mathtt{LE}[t] \mapsto e) \qquad (\mathsf{NextEpoch}_{\mathsf{tid}})$$

$$X \rightsquigarrow X \qquad (\mathsf{Id})$$

$$G_{\mathsf{tid}} = \{\mathsf{Inc}, \mathsf{UpdateLE}_{\mathsf{tid}}, \mathsf{Lock}_{\mathsf{tid}}, \mathsf{Unlock}_{\mathsf{tid}}, \mathsf{Pop}_{\mathsf{tid}}, \mathsf{NextEpoch}_{\mathsf{tid}}, \mathsf{Id}\};$$
$$R_{\mathsf{tid}} = \bigcup\{G_k \mid 1 \le k \le N \land k \ne \mathsf{tid}\}$$

$$\Upsilon_{\mathsf{E}} \iff (\forall t, l, x.\, (\boxed{\mathtt{LE}[t-1] \mapsto l * \mathsf{true}} \text{ since } \boxed{\mathtt{C} \mapsto x * x \mapsto \_ * \mathsf{true}})$$
$$\implies \boxed{x \mapsto_e \_ * \mathsf{true}}) \land$$
$$(\forall t, t', e, A.\, \boxed{\exists y.\, \mathtt{GE} \mapsto e * \mathtt{detached}[t-1][e \oplus 1] \mapsto A * \mathsf{true}} \implies$$
$$\forall x \in A.\, (\neg\boxed{\mathtt{C} \mapsto x * x \mapsto \_ * \mathsf{true}} \text{ since } \neg\boxed{\mathtt{LE}[t'-1] \mapsto e * \mathsf{true}})$$

$$C(t, l) \iff \exists A.\, \mathtt{detached}[t-1][l] \mapsto A * (\circledast_{x \in A}\, x \mapsto_m \_)$$

$$L(t, l) \iff \exists A.\, \mathtt{detached}[t-1][l] \mapsto A * (\circledast_{x \in A}\, x \mapsto_m \_ * \boxed{x \mapsto_e \_ * \mathsf{true}})$$

$$I(l, t) \iff \exists e, y.\, \mathtt{LE}[t-1] \mapsto l * \mathtt{GE} \mapsto e * \mathtt{C} \mapsto y * y \mapsto \_ *$$
$$\circledast_{t' \ne t}(\exists l.\, \mathtt{LE}[t'-1] \mapsto l * C(t', l \oplus 1) * C(t', l \oplus 2) \land$$
$$e \in \{l, l \oplus 1\}) * \exists l.\, C(t, l \oplus 1) * C(t, l \oplus 2) \land e \in \{l, l \oplus 1\}$$

**Fig. 21.** Rely/guarantee conditions, the temporal invariant and auxiliary assertions used in the proof of the counter algorithm with epoch-based reclamation

**Definition 2 (Extended Worlds)** *An* extended world *is a 7-tuple* $\omega = (\theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ *such that* $\theta_l$ *is an abstract state,* $\xi\theta_s$ *is a non-empty history (sequence of abstract states) whose last element–the current shared state–is* $\theta_s$*,* $\mathbf{i}$ *is a logical interpretation,* $R$ *is a set of (rely) actions,* $G$ *is a set of (guarantee) actions,* $\Upsilon$ *is a temporal invariant (a close assertion on histories), and* $Q$ *is an assertion on worlds.*

**Definition 3 (Thread Configuration)** *A* thread configuration *is an 8-tuple* $\kappa = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ *such that* $C$ *is a command,* $(\theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ *is an extended world, and the local state is consistent with the current shared state* $(\theta_l * \theta_s)\downarrow$.

**Definition 4 (Safety of Thread Configurations)** *A thread configuration* $\kappa = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ *is always* safe for 0 steps for *any* thread tid*, denoted* $\mathsf{TSafe}_0^{\mathsf{tid}}(\kappa)$. $\kappa$ *is* safe for $n+1$ steps for thread tid*, denoted* $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa)$*, if all of the following conditions hold:*

*(a) If* $C = \mathsf{done}$ *then* $\theta_l, \xi\theta_s, \mathbf{i} \vDash Q$ *and* $\xi\theta_s, \mathbf{i} \vDash \Upsilon$.

```
int GE = 0;
int LE[N] = {0};
Set detached[N][3] = {∅};
Lock Elock;

int inc() {
  int v, *s, *n;
  $\{V \Vdash \exists l.\, L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}}\}$
  n = new int;
  $\{V \Vdash \exists l.\, n \mapsto \_ * L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}}\}$
  $\langle$LE[tid-1] = GE;$\rangle_{\mathsf{UpdateLE_{tid}}}$
  $\{V \Vdash \exists l.\, n \mapsto \_ * L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}}\}$
  do {
    $\{V \Vdash \exists l.\, n \mapsto \_ * L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}}\}$
    s = C;
    $\{V \Vdash \exists l.\, n \mapsto \_ * L(\mathsf{tid}, l) \wedge \boxed{I(l, \mathsf{tid}) * \mathsf{true_e}} *$
      $(\boxed{\mathsf{LE}[\mathsf{tid}-1] \mapsto l * \mathsf{true}}\ \mathsf{since}\ \boxed{\mathsf{C} \mapsto s * s \mapsto \_ * \mathsf{true}})\}$
    $\{V \Vdash \exists l.\, n \mapsto \_ * L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}} *$
      $(\boxed{\mathsf{LE}[\mathsf{tid}-1] \mapsto l * \mathsf{true}}\ \mathsf{since}\ \boxed{\mathsf{C} \mapsto s * s \mapsto \_ * \mathsf{true}}) \wedge$
      $\boxed{s \mapsto_e \_ * \mathsf{true}}\}$
    $\langle$v = *s;$\rangle_{\mathsf{Id}}$
    *n = v+1;
    $\{V \Vdash \exists l.\, n \mapsto \_ * L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}} \wedge$
      $(\boxed{\mathsf{LE}[\mathsf{tid}-1] \mapsto l * \mathsf{true}}\ \mathsf{since}\ \boxed{\mathsf{C} \mapsto s * s \mapsto \_ * \mathsf{true}}) \wedge$
      $\boxed{s \mapsto_e \_ * \mathsf{true}}\}$
  } while (!$\langle$CAS(&C, s, n)$\rangle_{\mathsf{Inc}}$);
  $\{V \Vdash \exists l.\, s \mapsto_m \_ * L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}}\}$
  reclaim(s);
  $\{V \Vdash L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}}\}$
  return v;
}
```

**Fig. 22.** Proof outline for `inc` with epoch-based reclamation. Here $V$ is $v$, $n$, $s$, $s2$, $my$, $in\_use$, $i$.

*(b) For all $\theta_f$ such that $(\theta_l * \theta_s * \theta_f)\downarrow$,*

$$(C, \theta_l * \theta_s * \theta_f) \not\rightarrow_{\mathsf{tid}} \top.$$

*(c) For all $\theta_f, C', \theta'$ such that*

$$(\theta_l * \theta_s * \theta_f)\downarrow \wedge ((C, \theta_l * \theta_s * \theta_f) \xrightarrow{e}_{\mathsf{tid}} (C', \theta')),$$

*there exist $\theta'_l$ and $\theta'_s$ such that*
*(1) $\theta' = \theta'_l * \theta'_s * \theta_f$, and*
*(2) either $e = a$ for some $a \in G$ and*
  *(i) there exist $\theta_{l_0}$ and $\theta_{l_1}$ such that $\theta_l = \theta_{l_0} * \theta_{l_1}$ and $(\theta_{l_1}, \theta_s, \theta'_s, \mathbf{i}) \in [\![a]\!]$,*
  *(ii) $(\xi\theta_s\theta'_s, \mathbf{i}) \vDash \Upsilon$, and*
  *(iii) $\mathsf{TSafe}^{\mathsf{tid}}_n(C', \theta'_l, \xi\theta_s\theta'_s, \mathbf{i}, R, G, \Upsilon, Q)$;*
*(3) or $e = \ell$ and*
  *(i) $\theta'_s = \theta_s$ and*
  *(ii) $(\xi\theta_s\theta'_s, \mathbf{i}) \vDash \Upsilon$ and*

```
void reclaim(int *s) {
  $\{V \Vdash \exists l.\, s \mapsto_m \_ * L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}}\}$
  insert(detached[tid-1][LE[tid-1]], p);
  $\{V \Vdash \exists l.\, L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}}\}$
  if (nondet()) return;
  $\{V \Vdash \exists l.\, L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}}\}$
  $\langle$lock(Elock);$\rangle_{\mathsf{Lock_{tid}}}$
  $\{V \Vdash \exists l.\, L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \mathsf{tid} * I(l, \mathsf{tid}) * \mathsf{true_e}}$
  $\{V \Vdash \exists l.\, L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \mathsf{tid} * I(l, \mathsf{tid}) * \mathsf{true_e}}$
  for (int i = 0; i < N; i++) {
    if (GE != LE[i]) {
      $\langle$unlock(Elock);$\rangle_{\mathsf{Unlock_{tid}}}$
      $\{V \Vdash \exists l.\, L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e}}$
      return;
    }
  }
  $\{V \Vdash \exists l, e.\, L(\mathsf{tid}, e) \wedge$
   $\boxed{\exists y.\, \mathtt{GE} \mapsto e * \mathtt{C} \mapsto y * y \mapsto \_ * \mathtt{Elock} \mapsto \mathsf{tid} * \ldots}$
   $\boxed{\ldots \circledast_t (\mathsf{LE}[t] \mapsto e * C(t, e \oplus 1) * C(t, e \oplus 2)) * \mathsf{true_e}}\}$
  for (int i = 0; i < N; i++) {
    while (!$\langle$isEmpty(detached[i][(GE+1)%3])$\rangle_{\mathsf{Id}}$) {
      $\{V \Vdash \exists l, e.\, L(\mathsf{tid}, e) \wedge$
       $\boxed{\exists y.\, \mathtt{GE} \mapsto e * \mathtt{C} \mapsto y * y \mapsto \_ * \mathtt{Elock} \mapsto \mathsf{tid} * \ldots}$
       $\boxed{\ldots \circledast_t (\mathsf{LE}[t] \mapsto e * C(t, e \oplus 1) * C(t, e \oplus 2)) * \mathsf{true_e}}\}$
      free($\langle$pop(detached[i][(GE+1)%3])$\rangle_{\mathsf{Pop}}$);
    }
  }
  $\{V \Vdash \exists l, e.\, L(\mathsf{tid}, e) \wedge$
   $\boxed{\exists y.\, \mathtt{GE} \mapsto e * \mathtt{C} \mapsto y * y \mapsto \_ * \mathtt{Elock} \mapsto \mathsf{tid} * \ldots}$
   $\boxed{\ldots \circledast_t (\mathsf{LE}[t] \mapsto e * C(t, e \oplus 1) * C(t, e \oplus 2)) * \mathsf{true_e}}\}$
  $\langle$GE = (GE+1)%3;$\rangle_{\mathsf{NextEpoch_{tid}}}$
  $\langle$unlock(Elock);$\rangle_{\mathsf{Unlock_{tid}}}$
  $\{V \Vdash \exists l.\, L(\mathsf{tid}, l) \wedge \boxed{\mathtt{Elock} \mapsto \_ * I(l, \mathsf{tid}) * \mathsf{true_e})}\}$
  return;
}
```

**Fig. 23.** Proof outline for `inc` with epoch-based reclamation. Here $V$ is $v$, $n$, $s$, $s2$, $my$, $in\_use$, $i$.

*(iii)* $\mathsf{TSafe}_n^{\mathsf{tid}}(C', \theta'_l, \xi\theta_s\theta'_s, \mathbf{i}, R, G, \Upsilon, Q)$.

*(d) For every* $\theta^l$ *and* $\theta^s$ *if*

*(1)* $(\theta^l, \theta_s, \theta^s) \in [\![R]\!]$,

*(2)* $(\theta^l * \theta_l * \theta_s)\downarrow$ *and* $(\theta_l * \theta^s)\downarrow$ *, and*

*(3)* $\xi\theta_s\theta^s, \mathbf{i} \vDash \Upsilon$

*then* $\mathsf{TSafe}_n^{\mathsf{tid}}(C, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, Q)$.

*Thread configuration* $\kappa$ *is* safe*, denoted* $\mathsf{TSafe}^{\mathsf{tid}}(\kappa)$*, if it is safe for any* $0 \leq n$ *steps.*

**Proposition 1.** *For all sets of actions* $G$ *if* $a \in G$ *then* $[\![a]\!] \subseteq [\![G]\!]$.

**Proposition 2.** *For all sets of actions* $R$*,* $(\theta_l, \theta_s, \theta'_s) \in [\![R]\!]$ *only if there is* $a \in R$ *such that* $(\theta_l, \theta_s, \theta'_s) \in [\![a]\!]$.

We take the following definition and lemma are adaptations of ones taken from [16].

**Definition 5 (Strongest stable weaker assertion)** $\mathsf{sswa}_{R,\Upsilon}(Q)$ *is the strongest stable weaker assertion of an assertion* $Q$ *under rely* $R$ *and temporal invariant* $\Upsilon$*, and defined below*

– $Q \implies \mathsf{sswa}_{R,\Upsilon}(Q)$
– $\mathsf{sswa}_{R,\Upsilon}(Q)$ *is stable under* $R$ *with respect to* $\Upsilon$*.*
– *for all* $P$ *if* $P$ *is stable under* $R$ *and* $\Upsilon$ *and* $Q \implies P$ *then* $\mathsf{sswa}_{R,\Upsilon}(Q) \implies P$

**Lemma 1 (Properties of strongest stable weaker assertion).** *Let* $P$ *and* $Q$ *be assertions on worlds,* $R$ *a set of actions, and* $\Upsilon$ *a temporal invariant. The following holds*

*(i)* $\mathsf{sswa}_{\emptyset,\mathsf{true}}(P) \iff P$
*(ii)* $(P \wedge \Upsilon \implies Q) \implies (\mathsf{sswa}_{R,\Upsilon}(P) \implies \mathsf{sswa}_{R,\Upsilon}(Q))$
*(iii)* $\mathsf{sswa}_{R,\Upsilon}(P) \iff (\mathsf{sswa}_{R,\Upsilon}(P) \wedge \Upsilon)$
*(iv)* $(R \implies R') \implies (\mathsf{sswa}_{R',\Upsilon}(P) \implies \mathsf{sswa}_{R,\Upsilon}(P))$
*(v)* $\mathsf{sswa}_{R,\Upsilon}(P * Q) \iff \mathsf{sswa}_{R,\Upsilon}(P) * \mathsf{sswa}_{R,\Upsilon}(Q)$
*(vi)* *If* $Q$ *is closed under stuttering so is* $\mathsf{sswa}_{R,\Upsilon}(Q)$*.*

**Definition 6 (Meaning of Thread-local Judgements)** *We express the validity of a triple for the thread* $\mathsf{tid}$ *and the step* $n$ *by*

$$R, G, \Upsilon \vDash^{n}_{\mathsf{tid}} \{P\}\, C\, \{Q\}.$$

*This validity holds if and only if for every* $R' \subseteq R$ *and for every thread configuration* $\kappa = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R',\Upsilon}(Q))$ *if* $\theta_l, \xi\theta_s, \mathbf{i} \vDash \mathsf{sswa}_{R',\Upsilon}(P)$ *and* $\xi\theta_s, \mathbf{i} \vDash \Upsilon$ *then* $\mathsf{TSafe}^{\mathsf{tid}}_{n}(\kappa)$*. We also write*

$$R, G, \Upsilon \vDash_{\mathsf{tid}} \{P\}\, C\, \{Q\}$$

*when* $R, G, \Upsilon \vDash^{n}_{\mathsf{tid}} \{P\}\, C\, \{Q\}$ *for every* $0 \leq n$*.*

Intuitively, a Hoare triple $R, G, \Upsilon \vDash^{n}_{\mathsf{tid}} \{P\}\, C\, \{Q\}$ is valid, if a command $C$ which executes starting from a configuration in which the world satisfies the precondition (after the later is stabilized), and terminates then the world in the terminal configuration satisfies $Q$.

**Definition 7 (Closure under Stuttering)** *An assertion* $\Upsilon$ *is* closed under stuttering *if for every abstract state* $\theta_l$*, non-empty history* $\xi\theta_s\xi'$ *and interpretation* $\mathbf{i}$ *if* $\theta_l, \xi\theta_s\xi', \mathbf{i} \vDash \Upsilon$ *holds then* $\theta_l, \xi\theta_s\theta_s\xi', \mathbf{i} \vDash \Upsilon$ *also holds.*

**Theorem 8 (Soundness of Grace Logic for Commands).** *If*

$$R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C\, \{Q\}$$

*is derivable in our logic, where* $\Upsilon$*,* $P$*,* $Q$*, and every assertion in the proof is closed under stuttering, then*

$$R, G, \Upsilon \vDash_{\mathsf{tid}} \{P\}\, C\, \{Q\}$$

*holds.*

*Proof.* The proof is done by induction on the derivation tree: Intuitively, we prove that each rule is a sound implication if we replace all the $\vdash_{\mathsf{tid}}$ by $\vDash_{\mathsf{tid}}$. We handle each case separately by different lemmas in §E.3. □

## E.2 Soundness for Programs

**Definition 9 (Configuration)** *A* configuration *is an* 5*-tuple* $\rho = (\mathcal{P}, \theta_l, \xi\theta_s, \mathbf{i}, Q)$ *such that* $\mathcal{P}$ *is a program,* $\theta_l$ *is the combination of local states of threads,* $\xi\theta_s$ *is a history with* $(\theta_l * \theta_s){\downarrow}$*,* $\mathbf{i}$ *is an interpretation of logical variables and* $Q$ *is an assertion about worlds.*

**Definition 10 (Safety of Configurations)** *A configuration* $\rho = (\mathcal{P}, \theta_l, \xi\theta_s, \mathbf{i}, Q)$ *is always* safe for 0 steps*, denoted* $\mathsf{Safe}_0(\rho)$ . $\rho$ *is* safe for $n+1$ steps*, denoted* $\mathsf{Safe}_{n+1}(\rho)$ , *if all of the following conditions hold:*

*(I) If* $\mathcal{P} = \mathsf{done} \parallel \ldots \parallel \mathsf{done}$ *then* $(\theta_l, \xi\theta_s, \mathbf{i} \vDash Q)$.
*(II)* $(\mathcal{P}, \theta_l * \theta_s) \not\to \top$.
*(III) For all* $\mathcal{P}', \theta'$*, if*

$$(\mathcal{P}, \theta_l * \theta_s) \to (\mathcal{P}', \theta'),$$

*there exist* $\theta'_l$ *and* $\theta'_s$ *such that*

$$\mathsf{Safe}_n(\mathcal{P}', \theta'_l, \xi\theta_s\theta'_s, \mathbf{i}, Q)\,.$$

*Configuration* $\rho$ *is* safe*, denoted* $\mathsf{Safe}(\rho)$*, if it is safe for any* $0 \leq n$ *steps.*

**Definition 11 (Meaning of Global Judgements)** *We express the validity of a triple for the program* $\mathcal{P}$ *and the step* $n$ *by*

$$\vDash^n \{P\}\,\mathcal{P}\,\{Q\}.$$

*This holds if and only if for every configuration* $\rho = (\mathcal{P}, \theta_l, \xi\theta_s, \mathbf{i}, Q)$ *if* $\theta_l, \xi\theta_s, \mathbf{i} \vDash P$*, then* $\mathsf{Safe}_n(\rho)$*. We also write*

$$\vDash \{P\}\,\mathcal{P}\,\{Q\}$$

*when* $\vDash^n \{P\}\,\mathcal{P}\,\{Q\}$ *for every* $0 \leq n$.

**Lemma 2.** *Let* $\kappa_1, \ldots, \kappa_n$ *be thread configurations that have the following form:*

$$\kappa_i = (C_i, (\theta_l)_i, \xi\theta_s, \mathbf{i}, R_i, G_i, \Upsilon, Q_i)$$

*such that*

$$((\theta_l)_1 * \ldots * (\theta_l)_n * \theta_s){\downarrow} \wedge (R_i = \bigcup_{j \neq i} G_i).$$

*Also, let* $\rho$ *be the configuration that combines all of* $\kappa_i$*'s:*

$$\rho = (C_1 \parallel \ldots \parallel C_n, (\theta_l)_1 * \ldots * (\theta_l)_n, \xi\theta_s, \mathbf{i}, Q_1 * \ldots * Q_n)$$

*If* $\mathsf{Safe}(\kappa_i)$ *holds for every* $i$ *and every* $Q_i$ *is closed under stuttering then* $\mathsf{Safe}(\rho)$ *holds.*

*Proof.* The proof is done by induction on the number $k$ of steps for all $n$-tuples of possible configurations. For $k = 0$, $\mathsf{Safe}_0(\rho)$ holds by definition. For the induction step, assume the induction hypothesis (IH)

$$\begin{aligned}\forall 1 \leq i \leq n.\, \kappa_i = (C'_i, (\theta'_l)_i, \xi'\theta'_s, \mathbf{i}, R_i, G_i, \Upsilon, Q_i) \wedge{}\\ \mathsf{TSafe}^{\mathsf{tid}}_k(\kappa'_i) \wedge ((\theta'_l)_1 * \ldots * (\theta'_l)_n * \theta'_s){\downarrow} \implies\\ \mathsf{Safe}(C'_1 \parallel \ldots \parallel C'_n, (\theta'_l)_1 * \ldots * (\theta'_l)_n, \xi'\theta'_s, \mathbf{i}, Q'_1 * \ldots * Q'_n)\end{aligned}$$

and that

$$(\dagger_{\downarrow})\ ((\theta_l)_1 * \ldots * (\theta_l)_n * \theta_s){\downarrow}$$

holds and that

$$(\dagger_i)\ \mathsf{TSafe}^{\mathsf{tid}}_{k+1}(\kappa_i)$$

holds for $i = 1, \ldots, n$. We now show that $\mathsf{Safe}_{k+1}(\rho)$ holds.

**Condition (I)** if $C_i = \mathsf{done}$ for every $i = 1, \ldots, n$ then from $(\dagger_i)$ and Condition (a) of Definition 4, we get that for every $i = 1, \ldots, n$ it holds that $(\theta_l)_i, \xi\theta_s, \mathbf{i} \vDash_i Q_i$ and $\xi\theta_s, \mathbf{i} \vDash_i \Upsilon$. As $Q_i$ and $\Upsilon$ do not have `tid`, we get that for every $i = 1, \ldots, n$ it holds that

$$(\%_i)\ (\theta_l)_i, \xi\theta_s, \mathbf{i} \vDash Q_i \text{ and } \xi\theta_s, \mathbf{i} \vDash \Upsilon\,.$$

From $(\dagger_\downarrow)$, $(\%_1, \ldots \%_n)$, and the meaning of $*$ for assertions on world we get that $((\theta_l)_1 * \ldots * (\theta_l)_n, \xi\theta_s, \mathbf{i} \vDash Q_1 * \ldots Q_n)$.

**Condition (II)** Inspecting the operational semantics, we see that $(\mathcal{P}, \theta) \to \top$ only if there is a thread $t$ such that $(C_t, \theta) \to \top$. However, by $(\dagger_\downarrow)$, we get that for

$$\theta_f = (\theta_l)_1 * \ldots * (\theta_l)_{t-1} * (\theta_l)_{t+1} * \ldots * (\theta_l)_n$$

it holds that $((\theta_l)_t * \theta_s * \theta_f)\downarrow$. By assumption $(\dagger_t)$ and Condition (b) of Definition 4, we get that $(C_t, \theta) \not\to \top$.

**Condition (III)** Inspecting the operational semantics, we see that $(\mathcal{P}, \theta) \to (\mathcal{P}', \theta')$ only if there is a thread $t$ such that $(C_t, \theta) \to (C'_t, \theta')$ for some $1 \leq t \leq n$ and

$$\mathcal{P}' = C_1 \parallel \ldots \parallel C'_t \parallel \ldots \parallel C_n\,.$$

For $\theta_f$ as before, we get from assumption $(\dagger_t)$ and Condition (c) of Definition 4, we get that there exist $\theta'_l$ and $\theta'_s$ such that

$$(\$)\ (C_t, \theta) \to_{\mathsf{tid}} (C'_t, \theta'_l * \theta'_s * \theta_f)\ ,$$
$$(\&)\ ((\theta_l)_1 * \ldots * (\theta_l)_{t-1} * (\theta'_l) * (\theta_l)_{t+1} * \ldots * (\theta_l)_n * \theta'_s)\downarrow\ ,$$

and

1. either
$$(\xi\theta_s\theta'_s \vDash \Upsilon) \wedge \mathsf{TSafe}^{\mathsf{tid}}_n(C'_t, \theta'_l, \xi\theta_s, \mathbf{i}, R_t, G_t, \Upsilon, Q_t)$$
2. or
$$(\theta'_s = \theta_s) \wedge (\xi\theta_s \vDash \Upsilon) \wedge \mathsf{TSafe}^{\mathsf{tid}}_n(C'_t, \theta'_l, \xi\theta_s\theta'_s, \mathbf{i}, R_t, G_t, \Upsilon, Q_t).$$

We continue be case analysis.

1. Assume the first case hold. From assumptions $(\dagger_t)$, (\$), and Condition (c), we get that there exists $a \in G_t$ and states $\theta_{l_0}$ and $\theta_{l_1}$ such that (%) $(\theta_l)_t = \theta_{l_0} * \theta_{l_1}$ and (‡) $(\theta_{l_0}, \theta_s, \theta'_s) \in [\![a]\!]$.
Recall that or every $i \in \{j \neq t \mid 1 \leq j \leq n\}$, it holds by assumptions of the lemma that $G_t \subseteq R_i$. Hence, for every such $i$, we get from (%) and (‡), assumptions $(\dagger_t)$ and (\$) and Condition (d) that $\mathsf{TSafe}^{\mathsf{tid}}_n(C_i, (\theta_l)_i, \xi\theta_s\theta'_s, \mathbf{i}, R_i, G_i, Q_i)$ holds.
2. Assume the second case hold. To use the induction assumption, it suffuices to show that for every $i \neq t$ it holds that $\mathsf{TSafe}^{\mathsf{tid}}_i(\kappa_i)$. This is because the shared history was extended by the same state and we assume that $Q$ is closed under stuttering. We get the desired result from assumption $(\dagger_i)$ and Proposition 3.

Hence, we can now apply the induction hypothesis which, recall, was for arbitrary configurations up to step $n$. □

$$\frac{\begin{array}{c}R_1, G_1, \Upsilon \vdash_1 \{P_1\}\, C_1\, \{Q_1\} \;\ldots\; R_n, G_n, \Upsilon \vdash_n \{P_n\}\, C_n\, \{Q_n\} \\ R_{\mathsf{tid}} = \bigcup\{G_k \mid 1 \leq k \leq n \wedge k \neq \mathsf{tid}\} \qquad P_1 * \ldots * P_n \Rightarrow \Upsilon\end{array}}{\vdash \{P_1 * \ldots * P_n\}\, C_1 \parallel \ldots \parallel C_n\, \{Q_1 * \ldots * Q_n\}} \;\textsc{Par}$$

**Theorem 12 (Soundness of Grace Logic).** *If*

$$\vdash \{P\}\, \mathcal{P}\, \{Q\}$$

*is derivable in our logic, then*

$$\vDash \{P\}\, \mathcal{P}\, \{Q\}$$

*holds.*

*Proof.* Assume $\vdash \{P\}\, \mathcal{P}\, \{Q\}$ is derivable in our logic. Thus, there must be a proof tree whose root is derived by rule Par because this is the only rule that can be used to prove programs. Hence, it must be that there exist
($n$) $n > 0$,
($\dagger_C$) $\mathcal{P} = C_1 \parallel \ldots \parallel C_n$,
($\dagger_P$) $P = P_1 * \ldots * P_n$ and is closed under stuttering,
($\dagger_\Upsilon$) $P \implies \Upsilon$, for some assertion $\Upsilon$ on histories which is closed under stuttering.
($\dagger_Q$) $Q = Q_1 * \ldots Q_n$ and is closed under stuttering,
($\dagger_G^i$) there exist $G_1, \ldots G_n$, for $i = 1, \ldots, n$, such that ($\dagger_R^k$) $R_k = \bigcup_{\{j \neq k \mid 1 \leq j \leq n\}} G_k$, for $k = 1, \ldots, n$, and
($\dagger_\vdash^i$) the judgement $R_i, G_i, \Upsilon \vdash_i \{P_i\}\, C_i\, \{Q_i\}$, for $i = 1, \ldots, n$, is derivable in the logic.

Assume $(\theta_l, \xi\theta_s)$ is a world and $\mathbf{i}$ is an interpretation such that $\theta_l, \xi\theta_s, \mathbf{i} \vDash P$.

By ($\dagger_\Upsilon$), it must be that ($\ddagger_\Upsilon$) $\xi\theta_s, \mathbf{i} \vDash \Upsilon$. By ($\dagger_P$) and the meaning of $*$, there must be ($\ddagger_l$) $(\theta_l)_1, \ldots, (\theta_l)_n$ such that $\theta_l = (\theta_l)_1 * \ldots * (\theta_l)_n$ and ($\ddagger_i$) $(\theta_l)_i, \xi\theta_s, \mathbf{i} \vDash P_i$ for $i = 1, \ldots, n$. (Note that by the definition of worlds, it must be that

($\ddagger_\downarrow$) $(\theta_l * \theta_s)\downarrow$,

and from this and the cancelativity of $*$, it must be that

($\ddagger_l^i$) $((\theta_l)_i * \theta_s)\downarrow$ for $i = 1, \ldots, n$.)

By ($\dagger_\vdash^i$) and Theorem 8, it must be that ($\ddagger_\vDash^i$) $R_i, G_i, \Upsilon \vDash_i \{P_i\}\, C_i\, \{Q_i\}$, for $i = 1, \ldots, n$.

By ($\ddagger_i$), ($\ddagger_\Upsilon$), ($\ddagger_\vDash^i$), and Definition 6, for $i = 1, \ldots, n$, it must be that ($\$_i$) $\mathsf{TSafe}^{\mathsf{tid}}(\kappa_i)$ holds for $\kappa_i = (C_i, (\theta_l)_i, \xi\theta_s, \mathbf{i}, R_i, G_i, Q_i)$.

By ($\ddagger_l$) and ($\ddagger_\downarrow$), we get that (%) $((\theta_l)_1 * \ldots * (\theta_l)_n * \theta_s)\downarrow$.

By Lemma 2, (%), ($\$_1$),. . .,($\$_n$), ($\dagger_R^1$), . . ., ($\dagger_R^n$), it holds that (&) $\mathsf{Safe}(C_1 \parallel \ldots \parallel C_n, (\theta_l)_1 * \ldots * (\theta_l)_n, \xi\theta_s, \mathbf{i}, Q_1 * \ldots * Q_n)$.

By (&), ($\dagger_C$), ($\ddagger_l$), and ($\dagger_Q$), we get that $\mathsf{Safe}(\mathcal{P}, \theta_l, \xi\theta_s, \mathbf{i}, Q)$ holds. □

## E.3 Auxiliary Lemmas

**Proposition 3.** *Let* $\kappa = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ *be a configuration. If* $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa)$ *hold then* $\mathsf{TSafe}_m^{\mathsf{tid}}(\kappa)$ *hold for every* $0 \leq m \leq n$.

*Proof.* The proof is done by induction on $n$ for arbitrary configurations. The induction hypothesis (IH) is

$$\forall m, n, \kappa' = (C', \theta'_l, \xi'\theta'_s, \mathbf{i}', R', G', \Upsilon', Q'). \\ \mathsf{TSafe}^{\mathsf{tid}}_n(\kappa') \wedge 0 \leq m \leq n \implies \mathsf{TSafe}^{\mathsf{tid}}_m(\kappa') \, .$$

In the base case, $n = 0$ and hence $m = 0$, and the result holds from the definition of $\mathsf{TSafe}^{\mathsf{tid}}_0$ for any configuration. For the inductive step, assume that the lemma holds for $n$. We prove that it also holds for $n + 1$. It suffices to show that if $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa)$ holds then $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa)$ also hold: once we do so, the result will follow from the induction hypothesis for any $m < n$.

Assume

($\dagger$) $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ holds.

We show that $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa)$.

By assumption ($\dagger$), Condition (a) and Condition (b) hold as they are irrespective of $n$.

To show that Condition (c) holds, pick an arbitrary $\theta_f$ such that $(\theta_l * \theta_s * \theta_f)\downarrow$ holds and an arbitrary $C'$ such that $C, \theta_l * \theta_s * \theta_f \to C', \theta'$. Applying Condition (c1) to assumption ($\dagger$), we get that there exist $\theta'_l$ and $\theta'_s$ such that $\theta' = \theta'_l * \theta'_s * \theta_f$. Choose the same states to show that Condition (c) holds for $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa)$. Condition (c1) holds by selection. If Condition (c3) holds for $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa)$ then we have ($\sharp$) $\theta_s = \theta'_s$, ($\ddagger'$) $(\theta'_l, \xi\theta_s\theta'_s, \mathbf{i}) \vDash \Upsilon$, and ($\ddagger$) $\mathsf{TSafe}^{\mathsf{tid}}_n(C', \theta'_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ holds. Note, that ($\ddagger'$) establishes Condition (c(3)ii) holds. Recall that the induction hypothesis is for arbitrary configurations. Thus, we can use (IH) for ($\ddagger$) and $n$ to get that $\mathsf{TSafe}^{\mathsf{tid}}_{n-1}(C', \theta'_l, \xi\theta_s\theta'_s, \mathbf{i}, R, G, \Upsilon, Q)$ holds, which is what required to establish that Condition (c(3)iii) for $\kappa$ and $n$.

Establishing Condition (c2) and Condition (d) for $\kappa$ and $n$ is done in a similar way. □

## Soundness of Proof Rules for Primitive Commands

**Lemma 3 (Done).** *For all $n$ and for all configurations $\kappa = (\mathsf{done}, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, \mathsf{stab}_{R,\Upsilon}(Q))$ If*
*(i)* $\theta_l, \xi\theta_s, \mathbf{i} \vDash \mathsf{stab}_{R,\Upsilon}(Q)$ *and*
*(ii)* $\xi\theta_s, \mathbf{i} \vDash \Upsilon$
*then* $\mathsf{TSafe}^{\mathsf{tid}}(\kappa)$.

*Proof.* The proof is done by induction on $n$ and for arbitrary configurations for done. The induction hypothesis is (IH)

$$\forall n, \kappa' = (\mathsf{done}, \theta'_l, \xi'\theta'_s, \mathbf{i}', R', G', \Upsilon', \mathsf{stab}_{R',\Upsilon'}(Q')). \\ \theta'_l, \xi'\theta'_s, \mathbf{i}' \vDash Q' \wedge \xi'\theta'_s \vDash \Upsilon' \implies \mathsf{TSafe}^{\mathsf{tid}}_n(\kappa') \, .$$

For $n = 0$, the induction holds by definition. We assume now that the induction hypothesis holds for $n$ and show that it holds for $n + 1$: Condition (a) holds by assumptions (i) and (ii). Condition (b) and Condition (c) hold vacuously because no transition leaves done. We show that Condition (d) holds using the induction assumption for

$\mathsf{TSafe}^{\mathsf{tid}}_n(\mathsf{done}, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, \mathsf{stab}_{R,\Upsilon}(Q))$. This is possible because the induction hypothesis was for arbitrary configurations for done and all three requirements of the lemma hold: The first requirement (i) follows from the three assumptions made in Condition (d) on $\theta^s$ together with $\mathsf{stab}_{R,\Upsilon}(Q)$ being, by definition, stable under $R$ and $\Upsilon$. The second requirement (ii) holds because of the assumption made in Condition (d3) on $\theta^s$. □

**Lemma 4 (Local).** *For all $n$ and for all configurations $\kappa = (\alpha, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, q)$ if*
*(i) $\Upsilon$ is closed for stuttering,*
*(ii) $\xi\theta_s \vDash \Upsilon$,*
*(iii) $c \in \mathsf{PComm} \setminus \{\mathsf{done}\}$,*
*(iv) $f^{\mathsf{tid}}_\alpha(\theta_l) \subseteq [\![q]\!]_{\mathbf{i}}$, and*
*(v) $f^{\mathsf{tid}}_\alpha$ is local,[3] i.e., $\forall\theta.\, (\theta * \theta_l)\downarrow \wedge (f^{\mathsf{tid}}_c(\theta_l))\downarrow \Rightarrow f^{\mathsf{tid}}_c(\theta * \theta_l) = f^{\mathsf{tid}}_c(\theta_l) * \{\theta\}$*
*then* $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa)$.

*Proof.* Formally, the proof is done by induction on $n$. The induction hypothesis is (IH)

$$\begin{aligned}\forall\alpha, \theta'_l, \xi'\theta'_s, \mathbf{i}', R', G', \Upsilon', q'.\, f^{\mathsf{tid}}_\alpha(\theta'_l) \subseteq [\![q]\!]_{\mathbf{i}'} \implies \\ (\forall\theta.\, (\theta * \theta'_l)\downarrow \wedge (f^{\mathsf{tid}}_c(\theta'_l))\downarrow \Rightarrow f^{\mathsf{tid}}_\alpha(\theta * \theta'_l) = f^{\mathsf{tid}}_\alpha(\theta'_l) * \{\theta\}) \implies \\ \mathsf{TSafe}^{\mathsf{tid}}_n(\alpha, \theta'_l, \xi'\theta'_s, \mathbf{i}', R', G', \Upsilon', q')\,.\end{aligned}$$

In the base case, $\mathsf{TSafe}^{\mathsf{tid}}_0(\kappa)$ holds by definition. Assume the induction assumption holds for $n$. We show that $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa)$ also holds.
**Condition (a)**: Holds trivially as $\alpha \neq \mathsf{done}$.
**Condition (b)**: Pick an arbitrary $\theta_f$ such that (&) $(\theta_l * \theta_s * \theta_f)\downarrow$. By assumption $f^{\mathsf{tid}}_\alpha(\theta_l) \subseteq [\![q]\!]$. This implies that (†) $f^{\mathsf{tid}}_\alpha(\theta_l) \neq \top$. The result follows from (†) and the locality of $f^{\mathsf{tid}}_\alpha$.
**Condition (c)**: Consulting the operational semantics, we see that the only possible transitions for $\alpha$ and $\theta_f$ as above are $\alpha, \theta_l * \theta_s * \theta_f \xrightarrow{\ell}_{\mathtt{tid}} \mathsf{done}, \theta'$ where $\theta' \in f^{\mathsf{tid}}_\alpha(\theta_l * \theta_s * \theta_f)$. From the locality of $f^{\mathsf{tid}}_\alpha$ and (†), we get that there exists $\theta'_l$ such that (‡) $\theta'_l \in f^{\mathsf{tid}}_\alpha(\theta_l)$ and (♯) $\theta' = \theta'_l * \theta_s * \theta_f$. We show that Condition (c) holds using $\theta'_l$ and $\theta'_s = \theta_s$. This choice satisfies Condition (c1) by (♯) and Condition (c(3)i). It satisfies Condition (c(3)ii) by the assumption that $\Upsilon$ is closed under stuttering. To show that Condition (c(3)iii) is satisfied, we need to show that $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa')$ holds where $\kappa' = \mathsf{TSafe}^{\mathsf{tid}}_n(\mathsf{done}, \theta'_l, \xi\theta_s\theta_s, \mathbf{i}, R, G, \Upsilon, q)$.

We show that $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa')$ holds using Lemma 3. We can use the lemma because by the interpretation of assertions, we have that $q = \mathsf{sswa}_{R,\Upsilon}(q)$. Note that the two requirements of the lemma are satisfied:
– Requirement (i) is satisfied because by assumption (iv) and (‡) $\theta'_l, \mathbf{i} \vDash q$. As an assertion on worlds, $q$ places no restrictions on the history. Thus, it also holds that $\theta'_l, \xi\theta_s\theta_s\mathbf{i} \vDash q$.
– Requirement (ii) is satisfied by assumption (i) and (ii) of this lemma.
**Condition (d)**: We establish the result using the induction assumptions. This is possible because $q$ is independent of the history, $\theta_l$ is local and thus does not change, and the environment preserves $\Upsilon$. □

[3] For a discussion about local commands see §A.2.

## Soundness of Proof Rules for Composite Commands

**Lemma 5 (Seq).** *For all $n$ and for all configurations $\kappa_1 = (C_1, \theta_l^1, \xi^1\theta_s^1, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q_1))$ if*

*(i) $\Upsilon$ is closed for stuttering,*

*(ii) $Q_1$ is closed for stuttering,*

*(iii) $Q_2$ is closed for stuttering,*

*(iv) $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa_1)$ holds,*

*(v) $R, G, \Upsilon \vDash_{\mathsf{tid}}^n \{Q_1\}\, C\, \{Q_2\}$ holds, and*

*then $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa)$ where*

$$\kappa = (C_1; C_2, \theta_l^1, \xi^1\theta_s^1, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q_2))\,.$$

*Proof.* The proof is done by induction on $n$ and for arbitrary configurations. The induction hypothesis (IH) is

$$\begin{aligned}&\forall \kappa_1' = (C', \theta_l', \xi'\theta_s', \mathbf{i}', R', G', \Upsilon', \mathsf{sswa}_{R',\Upsilon'}(Q')), C'', Q''.\\ &\qquad (\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa_1) \wedge (R', G', \Upsilon \vDash_{\mathsf{tid}}^n \{Q'\}C''\{Q''\})) \implies\\ &\qquad\qquad \mathsf{TSafe}_n^{\mathsf{tid}}(C'; C'', \theta_l', \xi'\theta_s', \mathbf{i}', R', G', \Upsilon', \mathsf{sswa}_{R',\Upsilon'}(Q''))\,.\end{aligned}$$

In the base case, $\mathsf{TSafe}_0^{\mathsf{tid}}(\kappa)$ holds by definition. For the inductive step, assume ($\dagger_1$) $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa_1)$ and that assumption (v) holds for $n+1$. (Note that the latter together with Proposition 3 means that ($\dagger_2$) assumption (v) holds for $n$.) We now show that $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa)$ also holds.

**Condition (a)**: Holds trivially because $C_1; C_2$ is not done.

**Condition (b)**: Consider a state $\theta_f$ such that ($\dagger$) $(\theta_l^1 * \theta_s^1 * \theta_f)\downarrow$. Consulting the operational semantics, we see that $C_1; C_2, \theta_l^1 * \theta_s^1 * \theta_f$ can reduce in one step to $\top$ only if $C_1, \theta_l^1 * \theta_s^1 * \theta_f$ does so, which contradicts assumption ($\dagger_1$).

**Condition (c)**: For $\theta_f$ as above, we consult the operational semantics and see that there can be two cases. Either $C_1 = \mathsf{done}$, and $C_1; C_2, \theta_l^1 * \theta_s^1 * \theta_f \to C_2, \theta_l^1 * \theta_s^1 * \theta_f$ or $C_1, \theta_l * \theta_s * \theta_f \to C_1', \theta'$ and $C_1; C_2, \theta_l * \theta_s * \theta_f \to C_1'; C_2, \theta'$.

In the first case, we choose $\theta_l' = \theta_l^1$ and $\theta_s' = \theta_s^1$. Condition (c1), Condition (c(3)i), and Condition (c(3)ii) are satisfied by this choice because of ($\dagger$) and our assumption that $\Upsilon$ is closed under stuttering. To show that Condition (c(3)iii) is satisfied, we need to show that $\mathsf{TSafe}_n^{\mathsf{tid}}(C_2, \theta_l^1, \xi^1\theta_s^1\theta_s^1, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q_2))$ holds. To do so, we first use ($\dagger_1$) and Condition (a) of Definition 4, which, as $C_1 = \mathsf{done}$, give us that $(\xi^1\theta_s^1, \mathbf{i} \vDash \Upsilon)$ and $(\theta_l^1, \xi^1\theta_s^1, \mathbf{i} \vDash Q_1)$. By assumption, $Q_1$ is closed under stuttering, hence by Lemma 1(vi), so is $\mathsf{sswa}_{R,\Upsilon}(Q_1)$, which gives us $(\theta_l^1, \xi^1\theta_s^1\theta_s^1, \mathbf{i} \vDash \mathsf{sswa}_{R,\Upsilon}(Q_1))$. Using our assumption that $\Upsilon$ is also closed under stuttering gives us $(\xi^1\theta_s^1\theta_s^1, \mathbf{i} \vDash \Upsilon)$. Assumption (v) and the induction assumption give the desired result.

In the second case, we may assume that $C_1$ is not done. From ($\dagger_1$), there must be $\theta_l'$ and $\theta_s'$ such that $\theta' = \theta_l' * \theta_s' * \theta_f$ which justify the transition $C_1, \theta_l * \theta_s * \theta_f \to C_1', \theta'$. Pick the same states. By choice, these states satisfy Condition (c1) and either Condition (c2) or Condition (c3). Let $\xi' = \xi^1\theta_s^1\theta_s^{1'}$ in the first case and $\xi' = \xi^1\theta_s^1\theta_s^1$ in the second case. Note that the above choice also ensures that ($\ddagger$) $\mathsf{TSafe}_n^{\mathsf{tid}}(C_1', \theta_l', \xi', \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q_1))$ holds. Using ($\dagger_2$) and the induction hypothesis, which recall was for arbitrary configurations, we get that $\mathsf{TSafe}_n^{\mathsf{tid}}(C_1'; C_2, \theta_l', \xi', \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q_2))$ holds, as required.

**Condition (d)**: Pick an arbitrary $\theta^l$ and $\theta^s$ such that the premise of Condition (d) of $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa)$ holds. As the same $R$ and $\Upsilon$ are used in the premise of Condition (d) for $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa_1)$–which is assumed to hold ($\dagger_1$)–we get that ($\sharp$) $\mathsf{TSafe}^{\mathsf{tid}}_{n}(C_1, \theta^1_l, \xi^1\theta^1_s\theta^s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q_1))$ holds. Using ($\sharp$) and the induction hypothesis, we establish that $\mathsf{TSafe}^{\mathsf{tid}}_{n}(C_1; C_2, \theta^1_l, \xi^1\theta^1_s\theta^s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q_2))$ holds, as required. □

**Lemma 6 (Choice).** *For all $n$ and for all configurations $\kappa = (C_1 + C_2, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))$ if*
*(i) $\Upsilon$ is closed for stuttering,*
*(ii) $Q$ is closed for stuttering,*
*(iii) $\mathsf{TSafe}^{\mathsf{tid}}_{n}(C_1, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))$*
*(iv) $\mathsf{TSafe}^{\mathsf{tid}}_{n}(C_2, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))$*
*then $\mathsf{TSafe}^{\mathsf{tid}}_{n}(\kappa)$.*

*Proof.* The proof is done by induction on $n$ and for arbitrary configurations. The base case ($n = 0$) holds by definition. For the inductive step, assume ($\dagger_i$) $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(C_i, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ hold for $i = 1, 2$. We now show that $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(C_1 + C_2, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ also holds.
**Condition (a)** holds trivially because $C_1 + C_2 \neq \mathsf{done}$.
**Condition (b)** $C_1 + C_2$ never transitions to $\top$.
**Condition (c)** From the operational semantics, $C_1 + C_2, \theta \to C_i, \theta$ for either $i = 1$ or $i = 2$. The result follows from the induction hypothesis and assumptions ($\dagger_1$), ($\dagger_2$), and the closure under stuttering of $Q$ and $\Upsilon$.
**Condition (d)** To establish Condition (d), we need to show that $\mathsf{TSafe}^{\mathsf{tid}}_{n}(C_1 + C_2, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, Q)$ holds for $\theta^l$ and $\theta^s$ that satisfy the assumed conditions for $\theta_l$, $R$, and $\Upsilon$. As these conditions are insensitive to the command of the configuration, assumptions ($\dagger_1$) and ($\dagger_2$) gives us that $\mathsf{TSafe}^{\mathsf{tid}}_{n}(C_i, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, Q)$ for $i = 1$ and $i = 2$, respectively. Using the induction hypothesis (recall that the induction hypothesis is for arbitrary configurations) we get that $\mathsf{TSafe}^{\mathsf{tid}}_{n}(C_1 + C_2, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, Q)$ holds. □

**Lemma 7 (Iter).** *For all $n$ and for all configurations $\kappa_1 = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))$ if*
*(i) $\Upsilon$ is closed for stuttering,*
*(ii) $Q$ is closed for stuttering,*
*(iii) $\mathsf{TSafe}^{\mathsf{tid}}_{n}(\kappa_1)$,*
*(iv) $R, G, \Upsilon \vDash^{n}_{\mathsf{tid}} \{Q\}C\{Q\}$ holds, and*
*(v) $\theta_l, \xi\theta_s, \mathbf{i} \vDash \mathsf{sswa}_{R,\Upsilon}(Q)$ and $\xi\theta_s, \mathbf{i} \vDash \Upsilon$*
*then $\mathsf{TSafe}^{\mathsf{tid}}_{n}(\kappa)$ where*

$$\kappa = (C^*, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))\,.$$

*Proof.* The proof is done by induction on $n$. In the base case, $\mathsf{TSafe}^{\mathsf{tid}}_{0}(\kappa)$ holds by definition. For the inductive step, assume ($\dagger_1$) $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa_1)$ holds, ($\dagger_2$) assumption (v) holds, and the induction hypothesis (IH)

$$\forall \kappa' = (C', \theta_l', \xi'\theta_s', \mathbf{i}', R', G', \Upsilon', Q').$$
$$\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa') \wedge (R', G', \Upsilon \vDash \{Q'\}C'\{Q'\})$$
$$\implies (\mathsf{TSafe}_n^{\mathsf{tid}}(C'^*, \theta_l', \xi'\theta_s', \mathbf{i}', R', G', \Upsilon', Q') .$$

We now show that $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa)$ also holds.
**Condition (a)** holds trivially as $C^* \neq \mathsf{done}$.
**Condition (b)** $C^*$ never transitions to $\top$.
**Condition (c)**: Pick an arbitrary $\theta_f$ such that (†) $(\theta_l * \theta_s * \theta_f)\downarrow$. We show that Condition (c) holds using $\theta_l' = \theta_l$ and $\theta_s' = \theta_s$. This choice satisfies Condition (c1) and Condition (c(3)i) and together with our assumption that $\Upsilon$ is closed under stuttering, it also satisfies Condition (c(3)ii). To establish Condition (c(3)iii), we consult the operational semantics, and see that there can be two cases. Either $(C^*, \theta_l * \theta_s * \theta_f \xrightarrow{\ell}_{\mathsf{tid}} \mathsf{done}, \theta_l * \theta_s * \theta_f)$ or $(C^*, \theta_l * \theta_s * \theta_f \xrightarrow{\ell}_{\mathsf{tid}} C; C^*, \theta_l * \theta_s * \theta_f)$.

In the first case, we need to show that $\mathsf{TSafe}_n^{\mathsf{tid}}(\mathsf{done}, \theta_l, \xi\theta_s\theta_s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))$ holds. To do so, we use Lemma 3. We can use the lemma because $\theta_l, \xi\theta_s, \mathbf{i} \vDash \mathsf{sswa}_{R,\Upsilon}(Q)$ and $\xi\theta_s \vDash \Upsilon$ hold by assumption (v); by assumption, $Q$ and $\Upsilon$ are closed under stuttering. Hence, $\xi\theta_s\theta_s \vDash \Upsilon$ and, using Lemma 1(vi), we also get $\theta_l, \xi\theta_s\theta_s, \mathbf{i} \vDash \mathsf{sswa}_{R,\Upsilon}(Q)$.

In the second case, we need to show that $\mathsf{TSafe}_n^{\mathsf{tid}}(C; C^*, \theta_l, \xi\theta_s\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ holds. We do so using Lemma 5. We can use the lemma because assumption $(\dagger_1)$ and Proposition 3 ensure that $\mathsf{TSafe}_n^{\mathsf{tid}}(C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ holds and (IH) ensures, under this assumption, $R, G, \Upsilon \vDash_{\mathsf{tid}}^n \{Q\}C\{Q\}$ also holds.
**Condition (d)**: Pick an arbitrary $\theta^l$ and $\theta^s$ such that the premise of Condition (d) of $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa)$ holds. As the same $G$ and $\Upsilon$ are used in the premise of Condition (d) for $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa_1)$–which is assumed to hold $(\dagger_1)$–we get that (♯) $\mathsf{TSafe}_n^{\mathsf{tid}}(C, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, Q)$ holds. Using (♯) and the induction hypothesis, we establish that $\mathsf{TSafe}_n^{\mathsf{tid}}(C^*, \theta_l^1, \xi^1\theta_s^1\theta^s, \mathbf{i}, R, G, \Upsilon, Q_2)$ holds, as required. We can use the induction hypothesis because $\xi\theta_s\theta^s \vDash \Upsilon$ holds from Condition (d3) for (♯) and $\theta_l, \xi\theta_s\theta^s, \mathbf{i} \vDash Q$ holds from the stability of $Q$ under $R$ and $\Upsilon$.

□

**Soundness of Structural Proof Rules**

**Lemma 8 (Disj).** *For all $n$ and for all configurations $\kappa_1 = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q_1)$ and $\kappa_2 = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q_2)$ if*
*(i)* $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa_1)$ *hold or*
*(ii)* $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa_2)$ *hold*
*then* $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa)$ *holds where*

$$\kappa = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q_1 \vee Q_2) .$$

*Proof.* The proof is done by induction on $n$. In the base case, $\mathsf{TSafe}_0^{\mathsf{tid}}(\kappa)$ holds by definition. For the inductive step, assume the induction hypothesis (IH)

$$\begin{aligned}\forall C', \theta'_l, \xi'\theta'_s, \mathbf{i}', R', G', \Upsilon', Q'_1, Q'_2, .\ (\xi'\theta'_s \vDash \Upsilon' \wedge \\ (\mathsf{TSafe}^{\mathsf{tid}}_n(C', \theta'_l, \xi'\theta'_s, \mathbf{i}', R', G', \Upsilon', Q'_1) \vee \\ \mathsf{TSafe}^{\mathsf{tid}}_n(C', \theta'_l, \xi'\theta'_s, \mathbf{i}', R', G', \Upsilon', Q'_2))) \\ \implies \mathsf{TSafe}^{\mathsf{tid}}_n(C', \theta'_l, \xi'\theta'_s, \mathbf{i}', R', G', \Upsilon', Q'_1 \vee Q'_2)\end{aligned}$$

and that assumption (i) holds for $n+1$, i.e., that $(\dagger_1)$ $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa_1)$ holds. (The case where and assumption (i) hold is symmetric.) We now show that $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa)$ also holds.

**Condition (a)** If $C = \mathsf{done}$ then by assumption $(\dagger_1)$ and Condition (a) we get that $\theta_l, \xi\theta_s, \mathbf{i} \vDash Q_1$, and thus, $\theta_l, \xi\theta_s, \mathbf{i} \vDash Q_1 \vee Q_2$, and that $\xi\theta_s, \mathbf{i} \vDash \Upsilon$ holds.

**Condition (b)** If $C = \mathsf{done}$, we are done. Otherwise, pick an arbitrary $\theta_f$ such that (†) $(\theta_l * \theta_s * \theta_f)\downarrow$. If $C, \theta_l * \theta_s * \theta_f \rightarrow \top$ we get a contradiction to $(\dagger_1)$, which is assumed to hold.

**Condition (c)**: If $C = \mathsf{done}$, we are done. Otherwise, for the same $\theta_f$ as before, assume $C, \theta_l * \theta_s * \theta_f \rightarrow C, \theta'$. As $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa_1)$ is assumed to hold $(\dagger_1)$ there must be $\theta'_l$ and $\theta'_s$ such that (†) $\theta' = \theta'_l * \theta'_s * \theta_f$ which justify the transition. Pick the same states. By choice, these states satisfy Condition (c1) and either Condition (c2) or Condition (c3). Set $\xi' = \xi^1\theta^1_s\theta^{1'}_s$ in the first case and $\xi' = \xi^1\theta^1_s$ in the second case. Note that the above choice also ensures that (‡) $\mathsf{TSafe}^{\mathsf{tid}}_n(C'_1, \theta'_l, \xi', \mathbf{i}, R, G, \Upsilon, Q_1)$ holds. Using (‡) and the induction hypothesis, which recall was for arbitrary configurations, we get that $\mathsf{TSafe}^{\mathsf{tid}}_n(C, \theta'_l, \xi', \mathbf{i}, R, G, \Upsilon, Q_1 \vee Q_2)$ holds, as required.

**Condition (d)**: Pick an arbitrary $\theta^l$ and $\theta^s$ such that the premise of Condition (d) of $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa)$ holds. As the same $R$ and $\Upsilon$ are used in the premise of Condition (d) for $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa_1)$–which is assumed to hold $(\dagger_1)$–we get that (♯) $\mathsf{TSafe}^{\mathsf{tid}}_n(C, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, Q_1)$ holds. Using the induction hypothesis and that $\xi\theta_s\theta^s \vDash \Upsilon$ holds (by Condition (d3) for (♯)) we establish that $\mathsf{TSafe}^{\mathsf{tid}}_n(C, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, Q_1 \vee Q_2)$ as required. □

$$\frac{\begin{array}{c}P \wedge \Upsilon \Rightarrow P' \quad R \Rightarrow R' \quad G' \Rightarrow G \quad Q' \wedge \Upsilon \Rightarrow Q \\ R', G', \Upsilon \vdash_{\mathsf{tid}} \{P'\}\, C\, \{Q'\}\end{array}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C\, \{Q\}}\ \textsc{Conseq}$$

**Lemma 9 (Conseq).** *For all configurations*

$$\kappa = (C^0, \theta^0_l, \xi^0\theta^0_s, \mathbf{i}^0, R, G, \Upsilon, Q)$$

*and for all assertions $P'$ and $Q'$ and sets of actions $R'$ and $G'$. If*
1. *$Q'$ and $\Upsilon$ are closed under stuttering,*
2. *$\theta^0_l, \xi^0\theta^0_s, \mathbf{i} \vDash \mathsf{sswa}_{R,\Upsilon}(P)$,*
3. *$\xi^0\theta^0_s, \mathbf{i}^0 \vDash \Upsilon$,*
4. *$P \wedge \Upsilon \implies P'$,*
5. *$Q' \wedge \Upsilon \implies Q$,*
6. *$R \implies R'$,*
7. *$G' \implies G$, and*
8. *$R', G', \Upsilon \vDash^n_{\mathsf{tid}} \{P'\}C^0\{Q'\}$*

*then* $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa)$.

*Proof.* By assumptions (2) and (3), we get that

$$\theta_l^0, \xi^0\theta_s^0, \mathbf{i} \vDash \mathsf{sswa}_{R,\Upsilon}(P) \wedge \Upsilon\,.$$

(Recall that $\Upsilon$ does not restrict the local state.) By this and Lemma 1(iii), we get that

$$(*)\ \theta_l^0, \xi^0\theta_s^0, \mathbf{i} \vDash \mathsf{sswa}_{R,\Upsilon}(P \wedge \Upsilon)\,.$$

Applying Lemma 1(ii) and assumption (4), we get that

$$\theta_l^0, \xi^0\theta_s^0, \mathbf{i}^0 \vDash \mathsf{sswa}_{R,\Upsilon}(P')\,.$$

By Definition 6 and assumptions (6) and (8), we get that

$$R, G', \Upsilon \vDash_{\mathsf{tid}}^{n} \{P'\}C^0\{Q'\}\,.$$

From this and $(*)$, we have that

$$(\$\$)\ \mathsf{TSafe}^{\mathsf{tid}}(\theta_l^0, \xi^0\theta_s^0, \mathbf{i}^0, R, G', \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q'))\,.$$

We now show that $(\$\$)$ implies the desired result.

The proof continues by induction on the number of steps $n$. In the base case, $\mathsf{TSafe}_0^{\mathsf{tid}}(\kappa)$ holds by definition. For the inductive step, assume the induction hypothesis

$$(\$)\ \mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa') \quad \text{for} \quad \kappa' = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G', \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q'))\,.$$

We now show that $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa)$ also holds.
**Condition (a)** If $C = \mathsf{done}$ then by ($\$$), we get that $\theta_l, \xi\theta_s, \mathbf{i} \vDash \mathsf{sswa}_{R,\Upsilon}(Q')$ and that $\xi\theta_s \vDash \Upsilon$. Hence,

$$\theta_l, \xi\theta_s, \mathbf{i} \vDash \mathsf{sswa}_{R,\Upsilon}(Q') \wedge \Upsilon\,.$$

By this and assumption (5), and using Lemma 1(ii) and Lemma 1(iii) as before, we get that we get that $(\theta_l, \xi\theta_s, \mathbf{i} \vDash \mathsf{sswa}_{R,\Upsilon}(Q))$ and $(\xi\theta_s, \mathbf{i} \vDash \Upsilon)$.
**Condition (b)** If $C = \mathsf{done}$, we are done. Otherwise, pick an arbitrary $\theta_f$ such that $(\theta_l * \theta_s * \theta_f)\!\downarrow$. If $(C, \theta_l * \theta_s * \theta_f \to \top)$ we get a contradiction to ($\$$).
**Condition (c)**: If $C = \mathsf{done}$, we are done. Otherwise, for the same $\theta_f$ as before assume $(C, \theta_l * \theta_s * \theta_f \to C', \theta')$. By ($\$$), there must be $\theta_l'$ and $\theta_s'$ such that (†) $\theta' = \theta_l' * \theta_s' * \theta_f$ which justify the transition for $\kappa'$ to be safe up to $n+1$ steps.

Pick the same states $\theta_l'$ and $\theta_s'$ as above. We now show that Condition (c) also holds for $\kappa$ and $n+1$ steps. By choice, these states satisfy Condition (c1) and either Condition (c2) or Condition (c3) for $\kappa'$ to be safe up to $n+1$ steps. In the first case, the arrow of the transition is labeled with $a$. Choose the history of the shared state to be $\xi\theta_s\theta_s^{'}$. By Condition (c(2)i), there exist $a \in G'$ and states $\theta_{l_0}'$ and $\theta_{l_1}'$ such that $\theta_l' = \theta_{l_0}' * \theta_{l_1}'$ and $(\theta_l', \theta_s, \theta_s', \mathbf{i} \vDash a)$. Recall that $G' \implies G$, hence $[\![a]\!] \subseteq [\![G]\!]$. This satisfies Condition (c(2)i) for $\kappa$. By Condition (c(2)ii), $(\xi\theta_s\theta_s', \mathbf{i} \vDash \Upsilon)$. This satisfies Condition (c(2)ii) for $\kappa$. By Condition (c(2)iii), it holds that $\mathsf{TSafe}_n^{\mathsf{tid}}(C', \theta_l', \xi'\theta_s\theta_s', \mathbf{i}, R', G', \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q'))$. Using this and the induction hypothesis, we get $\mathsf{TSafe}_n^{\mathsf{tid}}(C', \theta_l', \xi'\theta_s\theta_s', \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))$, which satisfies Condition (c(2)iii) for $\kappa$.

In the second case, the arrow of the transition is labeled with $\ell$. Pick $\xi\theta_s\theta_s$ as the history of the shared state. As we extend the history with the same last state,

this satisfies Condition (c(3)i). As $\Upsilon$ is assumed to be closed under stuttering, we get that Condition (c(3)ii) holds. As Condition (c(3)iii) holds for $\kappa'$, we get that $\mathsf{TSafe}_n^{\mathsf{tid}}(C', \theta_l', \xi'\theta_s, \mathbf{i}, R', G', \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q'))$ holds. Using this and the induction hypothesis, we get $\mathsf{TSafe}_n^{\mathsf{tid}}(C', \theta_l', \xi'\theta_s\theta_s', \mathbf{i},', G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))$, which satisfies Condition (c(3)iii) for $\kappa$.
**Condition (d)**: Pick an arbitrary $\theta^l$ and $\theta^s$ such that the premise of Condition (d) of $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa)$ holds. The same rely $R$ is used in both configurations Hence, from Condition (d) for $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa_1)$–which is assumed to hold (\$)–we get that ($\sharp$) $\mathsf{TSafe}_n^{\mathsf{tid}}(C, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G', \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q'))$. Using the induction hypothesis we establish that $\mathsf{TSafe}_n^{\mathsf{tid}}(C, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))$ as required. □

**Lemma 10 (Exists$_2$).** *For all $n$ and for all logical variable $X$, state $\theta$, and configuration $\kappa_1 = (C, \theta_l, \xi\theta_s, \mathbf{i}[X : \theta], R, G, \Upsilon, Q)$ if $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa_1)$ then $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa)$ holds, where $\kappa = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, \exists X.\, Q)$.*

*Proof.* Before beginning the proof we remind ourselves that $C$, as all commands in our language does not modify the interpretation of (logical) variables of any type.

The proof is done by induction on $n$. In the base case, $\mathsf{TSafe}_0^{\mathsf{tid}}(\kappa)$ holds by definition. For the inductive step, assume the induction hypothesis (IH)

$$\begin{aligned}\forall C', \theta_l', \xi'\theta_s', \mathbf{i}', R'G', \Upsilon', Q', \theta'.\\ &\mathsf{TSafe}_n^{\mathsf{tid}}(C', \theta_l', \xi'\theta_s', \mathbf{i}'[X : \theta], R', G', \Upsilon', Q')\\ &\qquad \implies \mathsf{TSafe}_n^{\mathsf{tid}}(C', \theta_l', \xi'\theta_s', \mathbf{i}', R', G', \Upsilon', \exists X.\, Q')\end{aligned}$$

and that ($\dagger_1$) $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa_1)$ holds. We now show that $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa)$ also holds.
**Condition (a)** If $C = \mathsf{done}$ then by assumption ($\dagger_1$) we get that $\theta_l, \xi\theta_s, \mathbf{i}[X : \theta] \vDash Q$ and that $\xi\theta_s \vDash \Upsilon$. By the (standard) semantic definition of $\exists$ we get that $\theta_l, \xi\theta_s, \mathbf{i} \vDash \exists X.\, Q$.
**Condition (b)** If $C = \mathsf{done}$, we are done. Otherwise, pick an arbitrary $\theta_f$ such that ($\dagger$) $(\theta_l * \theta_s * \theta_f)\!\downarrow$. If $C, \theta_l * \theta_s * \theta_f \to \top$ we get a contradiction to ($\dagger_1$), which is assumed to hold.
**Condition (c)**: If $C = \mathsf{done}$, we are done. Otherwise, for the same $\theta_f$ as before assume $C, \theta_l * \theta_s * \theta_f \to C', \theta'$. As $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa_1)$ is assumed to hold ($\dagger_1$), there must be $\theta_l'$ and $\theta_s'$ such that ($\dagger$) $\theta' = \theta_l' * \theta_s' * \theta_f$ which justify the transition. Pick the same states. By choice, these states satisfy Condition (c1) and either Condition (c2) or Condition (c3). Choose the history of the shared state to be $\xi' = \xi^1\theta_s^1\theta_s^{1'}$ in the first case and $\xi' = \xi^1\theta_s^1\theta_s^1$ in the second case. Note that the above choice also ensures that ($\ddagger$) $\mathsf{TSafe}_n^{\mathsf{tid}}(C_1', \theta_l', \xi', \mathbf{i}[X : \theta], R, G, \Upsilon, Q)$ holds. Using ($\ddagger$) and the induction hypothesis, which recall was for arbitrary configurations, we get that $\mathsf{TSafe}_n^{\mathsf{tid}}(C, \theta_l', \xi', \mathbf{i}, R, G, \Upsilon, \exists X.\, Q)$ holds, as required.
**Condition (d)**: Pick an arbitrary $\theta^l$ and $\theta^s$ such that the premise of Condition (d) of $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa)$ holds. As the same $G$ and $\Upsilon$ are used in the premise of Condition (d) for $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa_1)$–which is assumed to hold ($\dagger_1$)–we get that ($\sharp$) $\mathsf{TSafe}_n^{\mathsf{tid}}(C, \theta_l, \xi\theta_s\theta^s, \mathbf{i}[X : \theta], R, G, \Upsilon, Q)$. Using the induction hypothesis and that $\xi\theta_s\theta^s \vDash \Upsilon$ holds (by Condition (d3) for ($\sharp$)) we establish that $\mathsf{TSafe}_n^{\mathsf{tid}}(C, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, \exists X.\, Q)$ as required. □

The proof for the other quantification rules is similar, and thus omitted.

$$\frac{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_1\}\, C\, \{Q_2\} \quad R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_2\}\, C\, \{Q_2\}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P_1 \wedge P_2\}\, C\, \{Q_1 \wedge Q_2\}}\ \textsc{Conj}$$

**Lemma 11 (Conj).** *For all $n$ and for all configurations $\kappa_1 = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q_1)$, $\kappa_2 = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q_2)$, and $\kappa = (C, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q_1 \wedge Q_2)$, if (i) $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa_1)$ and (ii) $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa_2)$ then $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa)$.*

*Proof.* The proof is done by induction on $n$. In the base case, $\mathsf{TSafe}_0^{\mathsf{tid}}(\kappa)$ holds by definition. For the inductive step, assume the induction hypothesis (IH)

$$\begin{aligned}\forall C', \theta_l', \xi'\theta_s', \mathbf{i}', R', G', \Upsilon', Q_1', Q_2', .\\ \big(\mathsf{TSafe}_n^{\mathsf{tid}}(C', \theta_l', \xi'\theta_s', \mathbf{i}', R', G', \Upsilon', Q_1') \wedge\\ \mathsf{TSafe}_n^{\mathsf{tid}}(C', \theta_l', \xi'\theta_s', \mathbf{i}', R', G', \Upsilon', Q_2')\big)\\ \implies \mathsf{TSafe}_n^{\mathsf{tid}}(C', \theta_l', \xi'\theta_s', \mathbf{i}', R', G', \Upsilon', Q_1' \wedge Q_2')\end{aligned}$$

and that ($\dagger_1$) $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa_1)$ and ($\dagger_2$) $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa_2)$ hold. We now show that $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa)$ also holds.
**Condition (a)** If $C = \mathsf{done}$ then by assumption ($\dagger_1$) we get that $\theta_l, \xi\theta_s, \mathbf{i} \vDash Q_1$ and by ($\dagger_2$) we get that $\theta_l, \xi\theta_s, \mathbf{i} \vDash Q_2$. Thus, $\theta_l, \xi\theta_s, \mathbf{i} \vDash Q_1 \wedge Q_2$. Both ($\dagger_1$) and ($\dagger_2$) give us that $\xi\theta_s \vDash \Upsilon$.
**Condition (b)** If $C = \mathsf{done}$, we are done. Otherwise, pick an arbitrary $\theta_f$ such that ($\dagger$) $(\theta_l * \theta_s * \theta_f)\downarrow$. If $C, \theta_l * \theta_s * \theta_f \to \top$ we get a contradiction to both ($\dagger_1$) and ($\dagger_2$), which are assumed to hold.
**Condition (c)**: If $C = \mathsf{done}$, we are done. Otherwise, for the same $\theta_f$ as before, assume $C, \theta_l * \theta_s * \theta_f \to_{\mathsf{tid}} C^c, \theta^c * \theta_f$. As $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa_i)$ is assumed to hold ($\dagger_i$) for $i = 1, 2$, there must be ($\sharp$) $\theta^c = \theta_l^{1'} * \theta_s^{1'} = \theta_l^{2'} * \theta_s^{2'}$ such that the requirements of Condition (c) are satisfied for $Q_i$ by $\theta_l^{i'}$ and $\theta_s^{i'}$ where $i = 1, 2$. To establish, Condition (c), it suffices to show that $\theta_l^{1'} = \theta_l^{2'}$ and $\theta_s^{1'} = \theta_s^{2'}$. Once we do that, we can use the induction hypothesis to establish the desired result.

There can be two cases. (I) If the arrow of the transition is labeled with an action $a$ from ($\dagger_1$) and ($\dagger_2$) we get that there exist $\theta_{l_0^i}$ and $\theta_{l_1^i}$, precise assertions $p_s$ and $q_s$, a variable over states $X$, and abstract states $\theta_X^i$ such that $a = l \mid p_s * X \rightsquigarrow q_s * X$ and $\theta_{l_1^i}, \theta_s, \theta_s^{i'}, \mathbf{i}[X \mapsto \theta_X^i] \vDash a$ for $i = 1, 2$. Now, as $\theta_s, \mathbf{i}[X \mapsto \theta_X^i] \vDash p_s * X$ for both $i = 1$ and $i = 2$ and $p_s$ is precise, we get from $*$ being cancelative that $\theta_X^1 = \theta_X^2$. Recall that $\theta_s^{i'}, \mathbf{i}[X \mapsto \theta_X^1] \vDash q_s * X$. Hence, there exists $\theta_{q_s}^{i'}$ for $i = 1, 2$ such that (&) $\theta_s^{i'} = \theta_{q_s}^{i'} * \theta_X^1$ and (%) $\theta_{q_s}^{i'}, \mathbf{i} \vDash q_s$. From ($\sharp$), we have that $\theta_l^{1'} * \theta_s^{1'} = \theta_l^{2'} * \theta_s^{2'}$. Hence, $\theta_l^{1'} * \theta_{q_s}^{1'} * \theta_X^1 = \theta_l^{2'} * \theta_{q_s}^{2'} * \theta_X^1$. As $*$ is cancelative, we get that $\theta_l^{1'} * \theta_{q_s}^{1'} = \theta_l^{2'} * \theta_{q_s}^{2'}$. From this equality, (%), and the precision of $q_s$ we get that $\theta_{q_s}^{1'} = \theta_{q_s}^{2'}$, and hence $\theta_l^{1'} = \theta_l^{2'}$. The latter, ($\sharp$), and $*$ being cancelative, implies that $\theta_s^{1'} = \theta_s^{2'}$.

(II) If the arrow of the transition is labeled with an$\ell$ by Condition (c2) and ($\dagger_1$) and ($\dagger_2$) we get that $\theta_s^{1'} = \theta_s^{2'} = \theta_s$. Because $*$ is cancelative, we get from ($\sharp$) that $\theta_l^{1'} = \theta_l^{2'}$.

**Condition (d)**: Pick an arbitrary $\theta^l$ and $\theta^s$ such that the premise of Condition (d) of $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa)$ holds. As the same $R$ and $\Upsilon$ are used in the premise of Condition (d) for $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa_i)$–which are assumed to hold $(\dagger_i)$ (for $i = 1, 2$)– we get that $(\sharp)$ $\mathsf{TSafe}^{\mathsf{tid}}_{n}(C, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, Q_i)$. Using the induction hypothesis and that $\xi\theta_s\theta^s \vDash \Upsilon$ holds (by Condition (d3) for $(\sharp)$) we establish that $\mathsf{TSafe}^{\mathsf{tid}}_{n}(C, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, Q_1 \wedge Q_2)$ as required. □

$$\frac{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\, C\, \{Q\} \quad F \text{ is stable under } R \cup G \text{ and } \Upsilon}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P * F\}\, C\, \{Q * F\}}\ \textsc{Frame}$$

**Lemma 12 (Frame).** *For all configurations*

$$\kappa^0 = (C^0, \theta^0_l, \xi^0\theta^0_s, \mathbf{i}^0, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q^0 * F))$$

*and for all assertions $F$. If*
*1. $\Upsilon$, $Q^0$ and $F$ are closed to stuttering.*
*2. $\theta^0_l, \xi^0\theta^0_s, \mathbf{i} \vDash P * F$,*
*3. $\xi\theta^0_s, \mathbf{i}^0 \vDash \Upsilon$,*
*4. $\mathsf{stab}_{R\cup G,\Upsilon}(F)$, and*
*5. $R, G, \Upsilon \vDash_{\mathsf{tid}} \{P\}C^0\{Q\}$*
*then $\mathsf{TSafe}^{\mathsf{tid}}_{n}(\kappa^0)$.*

*Proof.* By assumptions (2) and the interpretation of $*$, we get that there exist $\theta^{P0}_l$ and $\theta^{F0}_l$ such that

$$(\dagger_1)\ \theta^0_l = \theta^{P0}_l * \theta^{F0}_l,$$
$$(\dagger_2)\ \theta^{P0}_l, \xi^0\theta^0_s, \mathbf{i}^0 \vDash P, \text{ and}$$
$$(\dagger_3)\ \theta^{F0}_l, \xi^0\theta^0_s, \mathbf{i}^0 \vDash F\,.$$

By $(\dagger_2)$ and assumptions (3) and (5), we get that

$$(\$\$)\ \mathsf{TSafe}^{\mathsf{tid}}(\kappa^0) \text{ where}$$
$$\kappa^0 = (C^0, \theta^{P0}_l, \xi^0\theta^0_s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))\,.$$

We now show that (\$\$) implies the desired result.

The proof continues by induction on $n$. Pick

$$\kappa = (C, \theta^P_l * \theta^F_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q * F))$$

such that $\theta^P_l, \xi\theta_s, \mathbf{i} \vDash P$ and $(*_0)$ $\theta^F_l, \xi\theta_s, \mathbf{i} \vDash F$.

In the base case, $\mathsf{TSafe}^{\mathsf{tid}}_{0}(\kappa)$ holds by definition. For the inductive step, assume the induction hypothesis (IH)

$$\begin{aligned}
&\forall C^1, \theta^1_l, \xi^1\theta^1_s, \mathbf{i}^1, R^1, G^1, \Upsilon, Q^1, \theta^2_l, F^1.\\
&\qquad \Big(\theta^2_l, \xi^1\theta^1_s, \mathbf{i}^1 \vDash F^1 \wedge \mathsf{stab}_{R\cup G,\Upsilon}(F) \wedge (\theta^1_l * \theta^2_l * \theta^1_s)\!\downarrow \wedge\\
&\qquad \mathsf{TSafe}^{\mathsf{tid}}_{n}(C^1, \theta^1_l, \xi^1\theta^1_s, \mathbf{i}^1, R^1, G^1, \Upsilon, \mathsf{sswa}_{R^1,\Upsilon}(Q^1))\Big) \implies\\
&\qquad\qquad \mathsf{TSafe}^{\mathsf{tid}}_{n}(C^1, \theta^1_l * \theta^2_l, \xi^1\theta^1_s, \mathbf{i}^1, R^1, G^1, \Upsilon, \mathsf{sswa}_{R^1,\Upsilon}(Q^1 * F^1))
\end{aligned}$$

and that the following also holds:

$$(\$)\ \ \mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa') \text{ for } \kappa' = (C, \theta^P_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))$$
$$(\%)\ (\theta^F_l, \xi\theta_s, \mathbf{i}, F)$$
$$(\ddagger)\ \ (\theta^P_l * \theta^F_l * \theta^1_s)\!\downarrow$$

We now show that $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa)$ also holds.
**Condition (a)** If $C = \mathsf{done}$ then by (\$), we get that

$$(\&)\ \theta_l^P, \xi\theta_s, \mathbf{i} \vDash Q \wedge \xi\theta_s, \mathbf{i} \vDash \Upsilon\,.$$

By (%), (‡), and (&) we get that $\theta_l, \xi\theta_s, \mathbf{i} \vDash Q * F$. By (&), we also get that $\xi\theta_s, \mathbf{i} \vDash \Upsilon$.
**Condition (b)** If $C = \mathsf{done}$, we are done. Otherwise, pick an arbitrary $\theta_f$ such that $(\theta_l^P * \theta_l^F * \theta_s * \theta_f)\!\downarrow$. If $(C, \theta_l^P * \theta_l^F * \theta_s * \theta_f \rightarrow_{\mathtt{tid}} \top)$, we get a contradiction to (\$).
**Condition (c)**: If $C = \mathsf{done}$, we are done. Otherwise, for the same $\theta_f$ as before assume $(C, \theta_l^P * \theta_l^F * \theta_s * \theta_f \rightarrow_{\mathtt{tid}} C', \theta')$. By (\$), there must be $\theta'_l$ and $\theta'_s$ such that (†) $\theta' = \theta'_l * \theta'_s * \theta_l^F * \theta_f$ which justify the transition for $\kappa'$ to be safe up to $n+1$ steps. Pick the same states $\theta'_l$ and $\theta'_s$ as above. Our choice satisfies Condition (c1) to $\kappa$. Recall that the same $G$ and $\Upsilon$ appear in $\kappa$ and $\kappa'$. Thus, either Condition (c(2)i) and Condition (c(2)ii) hold for $\kappa$ or Condition (c(3)i) and Condition (c(3)ii) do.

The challenge, is to show that after the step, the claim about the frame still holds. There can be two case.

1. In case the transition was labeled with action $a$ and the new history of the shared state is $\xi'$, then $(\theta_l^F, \xi', \mathbf{i} \vDash F)$ still holds because by Condition (c2) $\xi'$ is of the form $\xi' = \xi\theta_s\theta_s^{'}$. By Condition (c(2)i), $a \in G$ and there exists states $\theta_{l_0}$ and $\theta_{l_1}$ such that $\theta_l^P = \theta_{l_0} * \theta_{l_1}$ and $(\theta_{l_0}, \theta_s, \theta'_s, \mathbf{i} \vDash a)$. Recall that $F$ is assumed to be stable for $G$ under $\Upsilon$. As Condition (c(2)ii) for $\kappa'$ ensures that $\xi', \mathbf{i} \vDash \Upsilon$. Thus, we get that $(*_1)$ $(\theta_l^F, \xi', \mathbf{i} \vDash F)$ holds.
2. Otherwise, the transition was labeled with $\ell$ and the history of the shared state was extended to $\xi' = \xi\theta_s\theta_s$, then $(*_2)$ $(\theta_l^F, \xi\theta_s\theta_s, \mathbf{i} \vDash F)$ holds because, by (†), $\theta_l^F$ was not modified and we assume that $F$ is closed under stuttering.

By Condition (c(2)iii), in the first case, and Condition (c(3)iii), in the second case, we get that for $\kappa'$ it holds that

$$\mathsf{TSafe}^{\mathsf{tid}}_n(C', \theta'_l, \xi', \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))\,.$$

From the above, (†), $(*_1)$, $(*_2)$, and the induction hypothesis we get that

$$\mathsf{TSafe}^{\mathsf{tid}}_n(C', \theta'_l * \theta_l^P, \xi', \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q) * F)\,.$$

By assumption, $F$ is stable under $R \cup G$ and $\upsilon$. From Lemma iv, we get that $\mathsf{stab}_{R,\Upsilon}(F)$ holds. Hence, $F = \mathsf{sswa}_{R,\Upsilon}(F)$. This, using Lemma v allows us to infer that $\mathsf{sswa}_{R,\Upsilon}(Q) * F \iff \mathsf{sswa}_{R,\Upsilon}(Q * F)$, and hence that

$$\mathsf{TSafe}^{\mathsf{tid}}_n(C', \theta'_l * \theta_l^P, \xi', \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q * F))\,,$$

which is what required to show (Condition (c(2)iii)) resp. Condition (c(3)iii) holds.
**Condition (d)**: Pick an arbitrary $\theta^l$ and $\theta^s$ such that the premise of Condition (d) of $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa)$ holds. We need to show that

$$\mathsf{TSafe}^{\mathsf{tid}}_n(\theta_l * \theta_l^F, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q * F))\,.$$

By the definition of stability, we have that $(*)$ $(\theta_l * \theta_l^F * \theta^s)\!\downarrow$. By (4), we have that $F$ is stable for $R$ under $\Upsilon$. From this and (%), we get that (/) $(\theta_l^F, \xi\theta_s\theta^s, \mathbf{i} \vDash F)$. From Condition (d) for $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa_1)$–which is assumed to hold (\$)–we get that

($\sharp$) $\mathsf{TSafe}_n^{\mathsf{tid}}(C, \theta_l^P, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q))$. Using the induction hypothesis, ($/$) ($*$), we establish that

$$\mathsf{TSafe}_n^{\mathsf{tid}}(C, \theta_l^P * \theta_l^F, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, \mathsf{sswa}_{R,\Upsilon}(Q * F))$$

holds. □

## Soundness of Shared Proof Rules

**Definition 13 (Local Configurations)** *A* local configuration *is a configuration of the form* $\kappa = (C, \theta, \xi\epsilon, \mathbf{i}, \emptyset, \emptyset, \mathsf{true}, q)$ *where* $C$ *is a command which does not contain atomic blocks.*

$$\frac{Q \Rightarrow \Upsilon \qquad P, Q \text{ are stable under } R \text{ and } \Upsilon \qquad \emptyset, G, \mathsf{true} \vdash_{\mathsf{tid}} \{P\}\,\langle C\rangle_a\,\{Q\}}{R, G, \Upsilon \vdash_{\mathsf{tid}} \{P\}\,\langle C\rangle_a\,\{Q\}}\ \textsc{Shared-R}$$

**Lemma 13 (Shared-R).** *Let* $\kappa = (\langle C\rangle_a, \theta_l, \xi\theta_s, \mathbf{i}, R, G, \Upsilon, Q)$ *be a configuration. If*
*1.* $\theta_l, \xi\theta_s, \mathbf{i} \vDash P$ *and* $\xi\theta_s, \mathbf{i} \vDash \Upsilon$*,*
*2.* $\emptyset, G, \mathsf{true} \vDash \{P\}\,\langle C\rangle_a\,\{Q\}$*,*
*3.* $Q \implies \Upsilon$*, and*
*4.* $P$ *and* $Q$ *are stable under* $R$ *and* $\Upsilon$
*then* $\mathsf{TSafe}^{\mathsf{tid}}(\kappa)$ *holds.*

*Proof.* We show that $\mathsf{TSafe}_n^{\mathsf{tid}}(\kappa)$ holds for all $0 \leq n$. The proof is done by induction on $n$. In the base case, $\mathsf{TSafe}_0^{\mathsf{tid}}(\kappa)$ holds by definition. For the inductive step, assume, in addition to (1–4), the induction hypothesis (IH)

$$\begin{aligned}&\forall a', C', \theta_l', \xi'\theta_s', \mathbf{i}', R', G', \Upsilon', Q'.\\ &\quad\Big(\theta_l', \xi'\theta_s', \mathbf{i}' \vDash P' \wedge \xi'\theta_s', \mathbf{i}' \vDash \Upsilon' \wedge \emptyset, G', \mathsf{true} \vDash \{P'\}\,\langle C'\rangle_{a'}\,\{Q'\} \wedge\\ &\qquad\qquad (Q' \implies \Upsilon') \wedge \mathsf{stab}_{R',\Upsilon'}(P') \wedge \mathsf{stab}_{R',\Upsilon'}(Q')\Big)\\ &\qquad\qquad\qquad \implies \mathsf{TSafe}_n^{\mathsf{tid}}(\langle C'\rangle_{a'}, \theta_l', \xi'\theta_s', \mathbf{i}', R', G', \Upsilon', Q')\,.\end{aligned}$$

We now show that $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa)$ also holds.
**Condition (a)** Holds vacuously because $\langle C\rangle_a \neq \mathsf{done}$.
**Condition (b)** Pick an arbitrary $\theta_f$ such that (†) $(\theta_l * \theta_s * \theta_f)\!\downarrow$. From assumptions (1) and (2) we get that

($) $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa_1)$ holds, where
$\kappa_1 = (\langle C\rangle_a, \theta_l, \xi\theta_s, \mathbf{i}, \emptyset, G, \mathsf{true}, Q)$.

Note that $\langle C'\rangle_a, \theta_l * \theta_s * \theta_f \to \top$ contradicts ($).
**Condition (c)**: For the same $\theta_f$ as above, assume

$$\langle C\rangle_a, \theta_l * \theta_s * \theta_f \xrightarrow{a}_{\mathsf{tid}} C', \theta'\,.$$

By ($), there are $\theta_l'$ and $\theta_s'$ such that (‡) $\theta' = \theta_l' * \theta_s' * \theta_f$ which justify this transition for $\mathsf{TSafe}_{n+1}^{\mathsf{tid}}(\kappa_1)$. Pick the same states. Condition (c1) holds by (‡). Condition (c(2)i) holds by ($) because the guarantee set in $\kappa$ is the same one as in $\kappa_1$ and the annotation of arrows does not change.

To see that Condition (c(2)ii) holds, we recall from the operational semantics that the only possible derivation for $\langle C\rangle_a, \theta_l * \theta_s * \theta_f \xrightarrow{a}_{\mathsf{tid}} C', \theta'$ is one such that $C' = \mathsf{done}$. Below, we establish that $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa')$ holds, where

$$\kappa' = (\mathsf{done}, \theta'_l, \xi\theta_s\theta'_s, \mathbf{i}, R, G, \Upsilon, Q).$$

If $n = 0$ then $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa')$ holds by definition. If $0 < n$ then by applying the definition of safety to (\$), we get that ($\sharp$) $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa'_1)$ also holds where $\kappa'_1 = (\mathsf{done}, \theta'_l, \xi\theta_s\theta'_s, \mathbf{i}, \emptyset, G, \mathsf{true}, Q)$. As $0 < n$, we get from the definition of safety for configurations that $\theta'_l, \xi\theta_s\theta'_s, \mathbf{i} \vDash Q$. By assumption (3), we get that $\xi\theta_s\theta'_s, \mathbf{i} \vDash \Upsilon$ because $\Upsilon$ does not restricts the local state. We now can apply Lemma 3 and get that $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa')$.

**Condition (d)**: Pick an arbitrary $\theta^l$ and $\theta^s$ such that the premise of Condition (d) of $\mathsf{TSafe}^{\mathsf{tid}}_{n+1}(\kappa)$ holds. We need to show that $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa^e)$ holds where $\kappa^e = (\langle C\rangle_a, \theta_l, \xi\theta_s\theta^s, \mathbf{i}, R, G, \Upsilon, Q)$. Recall that $P$ is stable under $R$ and $\Upsilon$. Hence, $\theta_l, \xi\theta_s\theta^s, \mathbf{i} \vDash P$ and $\xi\theta_s\theta^s, \mathbf{i} \vDash \Upsilon$. Now, we can apply the induction hypothesis for $\kappa^e$ to get the desired result. □

$$\frac{\begin{array}{c} p \Rightarrow l * \mathsf{true} \qquad \{l \mid p_s \rightsquigarrow q_s\} \Rightarrow \{a\} \qquad a \in G \\ \emptyset, \emptyset, \mathsf{true} \vdash_{\mathsf{tid}} \{p * p_s\}\, C\, \{q * q_s\} \end{array}}{\emptyset, G, \mathsf{true} \vdash_{\mathsf{tid}} \{p \land \tau \land \boxed{p_s}\}\, \langle C\rangle_a\, \{q \land ((\tau \land \boxed{p_s}) \lhd \boxed{q_s})\}}\ \textsc{Shared}$$

**Lemma 14 (Shared).** *Let*

$$\kappa = (\langle C\rangle_a, \theta, \xi\theta_s, \mathbf{i}, \emptyset, G, \mathsf{true}, \mathsf{sswa}_{\emptyset,\mathsf{true}}(Q))$$

*be a configuration where* $Q = q \land ((\tau \land \boxed{p_s}) \lhd \boxed{q_s})$ *. If*
1. $\theta, \mathbf{i} \vDash p$,
2. *there exist* $\theta_{l_0}$ *and* $\theta_{l_1}$ *such that* $\theta = \theta_{l_0} * \theta_{l_1}$ *and* $\theta_{l_1}, \mathbf{i} \vDash l$,
3. $\theta_s, \mathbf{i} \vDash p_s$,
4. $\xi\theta_s, \mathbf{i} \vDash \tau$,
5. $\emptyset, \emptyset, \mathsf{true} \vDash \{p * p_s\}\, C\, \{q * q_s\}$*, and*
6. $[\![l \mid p_s \rightsquigarrow q_s]\!] \subseteq [\![a]\!]$ *and* $a \in G$*,*

*then* $\mathsf{TSafe}^{\mathsf{tid}}(\kappa)$ *holds.*

*Proof.* We first note that by Lemma 1(i),

$$Q \iff \mathsf{sswa}_{\emptyset,\mathsf{true}}(Q)\,.$$

This allows us to simplify $\kappa$ to be

$$\kappa = (\langle C\rangle_a, \theta, \xi\theta_s, \mathbf{i}, \emptyset, G, \mathsf{true}, Q)\,.$$

We show that $\mathsf{TSafe}^{\mathsf{tid}}_n(\kappa)$ for all $0 \le n$. For $n = 0$, $\mathsf{TSafe}^{\mathsf{tid}}_0(\kappa)$ holds by definition. Thus, assume $n = m + 1$.

**Condition (a)** holds trivially as $\langle C\rangle_a$ is not done.

**Condition (b)**: Pick an arbitrary $\theta_f$ such that (\$) $(\theta * \theta_s * \theta_f)\downarrow$. (Note that, in particular, $(\theta * \theta_s)\downarrow$.) Assume, by contradiction, that (†) $\langle C \rangle, \theta * \theta_s * \theta_f \rightarrow \top$. Consulting the operational semantics, we see that (†) holds only if (‡) $C, \theta * \theta_s * \theta_f \rightarrow^* \top$. By assumptions (1) and (3) it holds that $\theta * \theta_s * \epsilon, \mathbf{i} \vDash p * p_s$. Hence, $\kappa_L = (C, \theta * \theta_s, \xi\epsilon, \mathbf{i}, \emptyset, \emptyset, \mathsf{true}, q * q_s)$ is a (local) configuration. By assumption (5) we get that $\mathsf{TSafe}^{\mathsf{tid}}(\kappa_L)$ holds. From Lemma 15 and (\$), we get that (‡) cannot hold, and hence neither can (†).

**Condition (c)**: Pick $\theta_f$ as above. Consulting the operational semantics, the only way for the one step reduction

$$\langle C \rangle_a, \theta * \theta_s * \theta_f \xrightarrow{a}_{\mathsf{tid}} C', \theta'$$

to occur is if $C' = \mathsf{done}$ and $C, \theta * \theta_s * \theta_f \rightarrow^* \mathsf{done}, \theta'$. Consider, again, the local configuration $\kappa_L$ and recall that $\mathsf{TSafe}^{\mathsf{tid}}(\kappa_L)$ holds. From Lemma 15, we get that there exist $\theta''$, $\theta'_l$, and $\theta'_s$ such that $\theta' = \theta'' * \theta_f$, $\theta'' = \theta'_l * \theta'_s$, (♯) $\theta'_l, \mathbf{i} \vDash q$, and (&) $\theta'_s, \mathbf{i} \vDash q_s$.

Using the aforementioned states, we get that $(\theta'_l * \theta'_s * \theta_f)\downarrow$, and hence Condition (c1) holds. We now show that Condition (c2) holds: Condition (c(2)i) holds by (&) and assumptions (2), (3) and (6). Condition (c(2)ii) holds trivially as the temporal invariant is true. To show that Condition (c(2)iii) holds, we need to establish that $\mathsf{TSafe}^{\mathsf{tid}}_m(\mathsf{done}, \theta'_l, \xi\theta_s\theta'_s, \mathbf{i}, \emptyset, G, \varUpsilon, Q)$. We do so using Lemma 3. To apply the latter we need to show that:

(i) $\theta'_l, \xi\theta_s\theta'_s, \mathbf{i} \vDash Q$. By definition of the satisfaction relation over worlds, the latter holds if and only if (a) $\theta'_l, \mathbf{i} \vDash q$, which holds by (♯), (b) $\theta_s, \mathbf{i} \vDash p_s$ and $\xi\theta_s, \mathbf{i} \vDash \tau$, which hold by assumptions (3) and (4), respectively, and (c) $\theta'_s, \mathbf{i} \vDash q_s$, which holds by (&).

(ii) $\xi\theta_s\theta'_s, \mathbf{i} \vDash \mathsf{true}$, holds trivially.

(iii) $Q$ is stable under $\emptyset$ and $\mathsf{true}$, which holds trivially.

**Condition (d)**: holds trivially since the rely is an emptyset. □

**Lemma 15 (Properties of Runs of Local Configurations).** *Let $\kappa = (C, \theta, \xi\epsilon, \mathbf{i}, \emptyset, \emptyset, \mathsf{true}, q)$ be a safe local configuration. For any $\theta_f$ such that $(\theta * \theta_f)\downarrow$ it holds that*

*1. $\neg(C, \theta * \theta_f, \mathbf{i} \rightarrow^* \top)$ and*

*2. if $C, \theta * \theta_f, \mathbf{i} \rightarrow^* \mathsf{done}, \mathbf{i}, \theta', \mathbf{i}$ then there exists $\theta''$ such that $\theta' = \theta'' * \theta_f$ and $\theta'', \mathbf{i} \vDash q$.*

**Sketch** The lemma is proved by induction on the number of derivation steps used in $\rightarrow^*$. We note that for local configurations, the notion of safety in Definition 4 simplifies to Definition 14.

**Definition 14 (Bounded Safety for Local Configurations)** *A configuration $\kappa = (C, \theta, \xi\epsilon, \mathbf{i}, \emptyset, \emptyset, \mathsf{true}, q)$ is always* locally safe for 0 steps*, denoted* $\mathsf{LSafe}^t_0(\kappa)$ . *$\kappa$ is* locally safe for $n+1$ steps*, denoted* $\mathsf{LSafe}^t_{n+1}(\kappa)$ , *if the following holds*

*(a) if $C = \mathsf{done}$ then $\theta, \mathbf{i} \vDash q$,*

*(b) for all $\theta_f$ if $(\theta * \theta_f)\downarrow$ then $C, \theta * \theta_f \not\rightarrow \top$,*

*(c) whenever $C, \theta * \theta_f \xrightarrow{\ell}_{tid} C', \theta'$*

*(1) there exists $\theta''$ such that $\theta' = \theta'' * \theta_f$ and*

*(2) it holds that $\mathsf{TSafe}^{\mathsf{tid}}_n(C', \theta'', \xi\epsilon, \mathbf{i}, \emptyset, \emptyset, \mathsf{true}, q)$.*

*A local configuration $\kappa$ is safe, denoted $\mathsf{LSafe}^t(\kappa)$, if it is safe for any $0 \leq n$ steps.*

**Proposition 4.** *For a local configuration $\kappa = (C, \theta, \xi\epsilon, \mathbf{i}, \emptyset, \emptyset, \mathsf{true}, q)$ it holds that $\mathsf{TSafe}^{\mathsf{tid}}(\kappa)$ if and only if $\mathsf{LSafe}^t(\kappa)$.*