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.

M. Felleisen and P. Gardner (Eds.): ESOP 2013, LNCS 7792, pp. 249–269, 2013.

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*** [12] 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)*** [11] 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 [14,6,3,16], 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

```
1  int *C = new int(0);
2  int inc() {
3    int v, *s, *n;
4    n = new int;
5    do {
6      s = C;
7      v = *s;
8      *n = v+1;
9    } while
10     (!CAS(&C,s,n));
11   // free(s);
12   return v;
13 }
```

(a)

```
14 int *C = new int(0);
15 int *HP[N] = {0};
16 Set detached[N] = {∅};
17 int inc() {
18   int v, *n, *s, *s2;
19   n = new int;
20   do {
21     do {
22       s = C;
23       HP[tid-1] = s;
24       s2 = C;
25     } while (s != s2);
26     v = *s;
27     *n = v+1;
28   } while(!CAS(&C,s,n));
29   reclaim(s);
30   return v; }
```

(b)

```
31 void reclaim(int *s) {
32   insert(detached[tid-1],s);
33   if (nondet()) return;
34   Set in_use = ∅;
35   while (!isEmpty(
36       detached[tid-1])) {
37     bool my = true;
38     int *n =
39       pop(detached[tid-1]);
40     for (int i = 0;
41         i < N && my; i++)
42       if (HP[i] == n)
43         my = false;
44     if (my) free(n);
45     else insert(in_use, n);
46   }
47   moveAll(detached[tid-1],
48     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.

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 "η since μ", 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 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 "η since μ", 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 [17,4] 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 [7, §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 [7, §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 [8]. 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 [9]. 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* [12]. 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 since C pointed to x", (1)

where 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 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 since 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 μ_x, and "x is allocated" by λ_x. Then (1) is $(\eta_{t,x} \mathsf{\ since\ } \mu_x)$, (2) is $(\forall t, x.\, (\eta_{t,x} \mathsf{\ since\ } \mu_x) \implies \lambda_x)$, and (3) is $(\forall t.\, \neg\mu_x \mathsf{\ since\ } \neg\eta_{t,x})$. The combination of (1) and (3) is inconsistent due to the following tautology:

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

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 [7, §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) [11] 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.

```
1  bool rcu[N] = {0};
2  rcu_enter() {⟨rcu[tid-1]=1⟩RCU_tid;}
3  rcu_exit() {⟨rcu[tid-1]=0⟩RCU_tid;}
4  sync() {
5    bool r[N] = {0};
6    for(int i = 0; i < N; i++)
7      ⟨r[i] = rcu[i]⟩ld;
8    for(int i = 0; i < N; i++)
9      if(r[i]) {while(⟨rcu[i]⟩ld);}
10 }
```

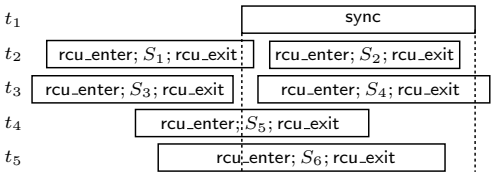

Fig. 2. An abstract RCU implementation and an illustration of the semantics of sync. Blocks represent the time spans of RCU critical sections or an execution of sync.

```
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 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.

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 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 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 [10,15], 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 [17]—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 [17] 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 [7, §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 Σ, together with a partial commutative, associative and cancellative operation $*$ on Σ 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 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 $f_\alpha^t : \mathsf{State} \to \mathcal{P}(\mathsf{State})^\top$. Here $\mathcal{P}(\mathsf{State})^\top$ is the set of subsets of 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_α^t, deferred to [7, §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 $\mathsf{LIVar} = \{x, y, \ldots\}$ range over integers, and those from $\mathsf{LSVar} = \{X, Y, \ldots\}$, 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 $\mathsf{State} \times \mathsf{LInt}$:

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

The interpretation of interesting connectives is as follows:

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

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 ξ by $|\xi|$, its i-th element by ξ_i, and its i-th prefix, by $\xi|_i$ (so that $|\xi|_i| = i$.) We refer to the last state $\xi_{|\xi|}$ in a history ξ as the *current* state. We define assertions denoting subsets of $\mathsf{History} \times \mathsf{LInt}$:

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

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

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 $(\tau_1 \;\mathsf{since}\; \tau_2)$ describes those histories where both τ_1 and τ_2 held at some point in the past, and since then, τ_1 has held continuously. The assertion $\tau \triangleleft \boxed{p}$ (τ *extended with* $\boxed{p}$) describes histories obtained by appending a state satisfying p to the end of a history satisfying τ. 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 $\mathsf{World} \times \mathsf{LInt}$:

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

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

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 τ 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 [17], 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 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

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

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

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

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

Fig. 4. Proof rules of the logic

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* [13], if for every state θ and interpretation $\mathbf{i}$, there exists at most one substate θ_1 satisfying r, i.e., such that $\theta_1, \mathbf{i} \models r$ and $\theta = \theta_1 * \theta_2$ for some θ_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 Υ 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 Υ; in return, a thread can rely on the environment not violating the invariant. We require that Υ 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 Υ, 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 Υ, if it is insensitive to changes to the shared state permitted by the action that preserve the invariant:

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

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 θ_l is inconsistent with the current thread-local state θ and the current shared state θ_s. An assertion is stable under R and Υ, when it is stable under every action in R together with Υ. 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 Υ, 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 Υ.

The proof rules of our logic are given in Figure 4. We have omitted the more standard rules [7, §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^t_α 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 Υ.

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 Υ. Note that to establish the latter in practice, we can always add Υ 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 τ 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 Υ.

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 Prog the set of programs $\mathcal{P}$ with an additional command 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 [7, §A]. The following theorem is proved in [7, §E].

Theorem 1 (Soundness). *Assume* $\vdash \{P\}\,\mathcal{P}\,\{Q\}$ *and take* θ_l, θ_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* θ'_l, θ'_s *and* ξ *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 State to $\mathsf{RAM_e} = \mathbb{N} \rightharpoonup_{fin} ((\mathbb{Z} \times \{1, \mathsf{m}\}) \cup \{\mathsf{e}\})$. 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 m is a *master* permission, which allows reading and writing the cell, but not deallocating it; and e is an *existential* permission, which only allows reading the cell and does not give any guarantees regarding its contents. The transformers f^t_α over $\mathsf{RAM_e}$ are given in [7, §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, which is enough for our purposes. For $\theta_1, \theta_2 \in \mathsf{RAM_e}$, $\theta_1 * \theta_2$ 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 $\mathsf{RAM_e}$, 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., $[[\![E]\!]_{\mathbf{i}} : ([\![F]\!]_{\mathbf{i}}, 1)], \mathbf{i} \models E \mapsto F$ 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 $\mathsf{true_e}$ 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 $\mathtt{var}$ 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 $\mathtt{var}$ 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_{tid} and the guarantee G_{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 $\mathtt{C}$ 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

$$X \rightsquigarrow X \quad (\mathsf{Id}) \qquad x \mapsto_{\mathsf{m}} \_ \mid x \mapsto_{\mathsf{e}} \_ * X \rightsquigarrow X \quad (\mathsf{Take})$$
$$\mathtt{HP}[\mathsf{tid}-1] \mapsto \_ * X \rightsquigarrow \mathtt{HP}[\mathsf{tid}-1] \mapsto \_ * X \quad (\mathsf{HP}_{\mathsf{tid}})$$
$$\mathtt{C} \mapsto x * x \mapsto \_ * X \rightsquigarrow \mathtt{C} \mapsto y * y \mapsto \_ * x \mapsto_{\mathsf{e}} \_ * X \quad (\mathsf{Inc})$$
$$G_{\mathsf{tid}} = \{\mathsf{HP}_{\mathsf{tid}}, \mathsf{Inc}, \mathsf{Take}, \mathsf{Id}\}; \qquad R_{\mathsf{tid}} = \bigcup \{G_k \mid 1 \le k \le N \wedge k \neq \mathsf{tid}\}$$
$$\Upsilon_{\mathsf{HP}} \iff \forall x, t.\ (\boxed{\mathtt{HP}[t-1] \mapsto x * \mathsf{true}}\ \mathsf{since}\ \boxed{\mathtt{C} \mapsto x * x \mapsto \_ * \mathsf{true}}) \Rightarrow \boxed{x \mapsto_{\mathsf{e}} \_ * \mathsf{true}})$$

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

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 Υ_{HP} in Figure 5. Temporal invariants used for proofs of algorithms based on grace periods are of the form "$\forall x, t.\ (\boxed{g}\ \mathsf{since}\ \boxed{r}) \Rightarrow \boxed{c}$", where "$\boxed{g}\ \mathsf{since}\ \boxed{r}$" 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 $\mathtt{CAS}_{a,b}$(addr,v1,v2) as a shorthand for the following, where the assume command "assumes" its parameter to be non-zero [7, §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 [17]. Temporal reasoning is needed, e.g., to check that every command changing the shared state preserves the temporal invariant Υ_{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_{tid}, defined later, which describes the thread-local detached set.

```
1  int *C = new int(0), *HP[N] = {0};
2  Set detached[N] = {∅};
3  int inc() {
4    int v, *n, *s, *s2;
5    {V ⊩ F_tid ∧ I}
6    n = new int;
7    do {
8      {V ⊩ n ↦ _ ∗ F_tid ∧ I}
9      do {
10       {V ⊩ n ↦ _ ∗ F_tid ∧ I}
11       ⟨s = C⟩_ld;
12       {V ⊩ n ↦ _ ∗ F_tid ∧ I}
13       ⟨HP[tid-1] = s⟩_HP_tid;
14       {V ⊩ n ↦ _ ∗ F_tid ∧ I ∧
15        HP[tid − 1] ↦ s ∗ true}
16       ⟨s2 = C⟩_ld;
17       {V ⊩ n ↦ _ ∗ F_tid ∧ I ∧
18        (HP[tid − 1] ↦ s ∗ true
19         since C ↦ s2 ∗ s2 ↦ _ ∗ true)}
20     } while (s != s2);
21     {V ⊩ n ↦ _ ∗ F_tid ∧ I ∧ s ↦_e _ ∗ true ∧
22      (HP[tid − 1] ↦ s ∗ true
23       since C ↦ s ∗ s ↦ _ ∗ true)}
24     ⟨v = *s⟩_ld;
25     *n = v+1;
26     {V ⊩ n ↦ _ ∗ F_tid ∧ I ∧ s ↦_e _ ∗ true ∧
27      (HP[tid − 1] ↦ s ∗ true
28       since C ↦ s ∗ s ↦ _ ∗ true)}
29   } while (!CAS_Inc,ld(&C, s, n));
30   {V ⊩ s ↦_m _ ∗ F_tid ∧ I ∧ s ↦_e _ ∗ true}
31   reclaim(s);
32   {V ⊩ F_tid ∧ I}
33   return v; }
```

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

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 $\boxed{\mathsf{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}$, which is stable, as R_{tid} and G_{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 $\boxed{\mathsf{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}} \wedge \boxed{\mathtt{C} \mapsto s2 * s2 \mapsto \_ * \mathsf{true}}$. 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 η is. Since $\boxed{\mathsf{HP}[\mathsf{tid}-1] \mapsto s * \mathsf{true}}$ 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 Υ_{HP}. This allows us to exploit Υ_{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 Υ_{HP}, we use the following rule Shared-I, derived from Shared [7, §A]:

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

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

$$(\boxed{g}\ \mathsf{since}\ \boxed{r}) \Rightarrow \boxed{c} \tag{9}$$

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 $(\boxed{g}\ \mathsf{since}\ \boxed{r}) \Rightarrow \boxed{c}$ may hold before the execution of C: $\neg(\boxed{g}\ \mathsf{since}\ \boxed{r})$ or $(\boxed{g}\ \mathsf{since}\ \boxed{r}) \wedge \boxed{c}$.

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 $\neg(\boxed{g}\ \mathsf{since}\ \boxed{r})$. Since $\boxed{g}\ \mathsf{since}\ \boxed{r}$ 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: $((\neg(\boxed{g}\ \mathsf{since}\ \boxed{r})) \triangleleft \boxed{g \wedge r \Rightarrow c}) \Rightarrow ((\boxed{g}\ \mathsf{since}\ \boxed{r}) \Rightarrow \boxed{c})$.

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

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 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 Υ, 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 Υ 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 $\boxed{g}\ \mathsf{since}\ \boxed{r}$ being false before the atomic block. The triple instantiates to

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

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

```
1  void reclaim(int *s) { {V ⊩ s ↦m _ ∗ F_tid ∧ [s ↦e _ ∗ true] ∧ I}
2    insert(detached[tid-1], s);
3    if (nondet()) return;
4    Set in_use = ∅;
5    while (!isEmpty(detached[tid-1])) {
6      {V ⊩ ∃A. detached[tid − 1] ↦ A ∗ D(A) ∗ D(in_use) ∧ A ≠ ∅ ∧ I}
7      bool my = true;
8      Node *n = pop(detached[tid-1]);
9      {V ⊩ my ∧ ∃A. detached[tid − 1] ↦ A ∗ D(A) ∗ D(in_use) ∗ n ↦m _ ∧ [n ↦e _ ∗ true] ∧ I}
10     for (int i = 0; i < N && my; i++) {
11       {V ⊩ my ∧ ∃A. detached[tid − 1] ↦ A ∗ D(A) ∗ D(in_use) ∗ n ↦m _ ∗ [n ↦e _ ∗ true] ∧
12        0 ≤ i < N ∧ I ∧ [H ∗ ∃y. y ≠ n ∧ C ↦ y ∗ y ↦ _ ∗ true_e] ∧
13        ∀0 ≤ j < i. ([∃y. y ≠ n ∧ C ↦ y ∗ y ↦ _ ∗ true_e] since [∃x. x ≠ n ∧ HP[j] ↦ x ∗ true])}
14       if (⟨HP[i] == n⟩_Id) my = false;
15     }
16     if (my) {
17       {V ⊩ ∃A. detached[tid − 1] ↦ A ∗ D(A) ∗ D(in_use) ∗ n ↦m _ ∧ [n ↦e _ ∗ true] ∧ I ∧
18        ∀t. ¬[C ↦ n ∗ true] since ¬[HP[t − 1] ↦ n ∗ true]}
19       ⟨ ; ⟩_Take
20       {V ⊩ ∃A. detached[tid − 1] ↦ A ∗ D(A) ∗ D(in_use) ∗ n ↦ _ ∧ I}
21       free(n);
22     } else { insert(in_use, n); }
23   } {V ⊩ detached[tid − 1] ↦ ∅ ∗ D(in_use) ∧ I}
24   moveAll(in_use, detached[tid-1]); {V ⊩ F_tid ∧ I}
25 }
```

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

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_{tid} describes the `detached` set of thread tid:

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

F_{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_{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:

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

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 [7, §B], we demonstrate this on the example of a non-blocking stack with several optimisations of hazard pointers used in practice [12]: 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 [7, §C], we also present an adaptation of the above proof to establish the linearizability of the algorithm following the approach in [17] (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 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

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

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

Fig. 8. Specification of RCU commands

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 RCU_k for $k \neq \mathsf{tid}$ and actions that do not access the `rcu` array. Second, we require that Υ be preserved under the actions that RCU functions execute (condition 2).

The specification for sync is the most interesting one. The precondition $p \wedge \tau$ 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 $\forall t.\, \tau \ \mathsf{since}\ S(t,0)$ 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 $r, i \Vdash p \wedge \forall t.\, (t < i+1 \vee r[t-1] = 0) \Rightarrow (\tau \ \mathsf{since}\ S(t,0))$. As usual, here we obtain the since clause by weakening: $(\tau \wedge S(\mathsf{tid},0)) \Rightarrow (\tau \ \mathsf{since}\ S(\mathsf{tid},0))$.

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 $R_{\mathsf{tid}} = \bigcup\{G_k \mid 1 \leq k \leq N \wedge k \neq \mathsf{tid}\}$. The following invariant formalises (5):

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

The proof outline for the RCU-based counter is given in Figure 9. The assertion F_{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:

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

The key points are as follows. After reading `C` at line 12, we obtain an unstable assertion $S(\mathsf{tid},1) \wedge \boxed{\mathtt{C} \mapsto s * s \mapsto \_ * \mathsf{true}}$, which we weaken to a stable one $(S(\mathsf{tid},1) \ \mathsf{since}\ \boxed{\mathtt{C} \mapsto s * s \mapsto \_ * \mathsf{true}})$. Then Υ_{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 $x \mapsto_{\mathsf{m}} \_ \Rightarrow \neg\boxed{\mathtt{C} \mapsto x * x \mapsto \_ * \mathsf{true}}$. At line 36, we apply the specification of `sync` with $\tau = \boxed{(\circledast_{x \in A}\, x \mapsto_{\mathsf{e}} \_) * \mathsf{true}} \wedge (\circledast_{x \in A} \neg\boxed{\mathtt{C} \mapsto x * x \mapsto \_ * \mathsf{true}})$ 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 Υ_{RCU}.

```
1  int *C=new int(0); bool rcu[N]={0};
2  Set detached[N]={∅};
3  int inc() {
4    int v, *n, *s;
5    {V ⊩ F_tid ∧ I ∧ S(tid,0)}}
6    n = new int;
7    {V ⊩ n ↦ _ ∗ F_tid ∧ I ∧ S(tid,0)}}
8    rcu_enter();
9    do { {V ⊩ n ↦ _ ∗ F_tid ∧ I ∧ S(tid,1)}
10     rcu_exit();
11     rcu_enter();
12     ⟨s = C⟩_ld;
13     {V ⊩ n ↦ _ ∗ F_tid ∧ I ∧ [s ↦_e _ ∗ true] ∧
14      (S(tid,1) since [C ↦ s ∗ s ↦ _ ∗ true])}
15     ⟨v = *s⟩_ld;
16     *n = v+1;
17     {V ⊩ n ↦ _ ∗ F_tid ∧ I ∧ [s ↦_e _ ∗ true]
18      (S(tid,1) since [C ↦ s ∗ s ↦ _ ∗ true])}
19   } while (!CAS_Inc,ld(&C, s, n));
20   rcu_exit();
21   {V ⊩ s ↦_m _ ∗ F_tid ∧ I ∧ S(tid,0) ∧
22    [s ↦_e _ ∗ true]}
23   reclaim(s);
24   {V ⊩ F_tid ∧ I ∧ S(tid,0)}
25   return v; }

26 void reclaim(int* s) {
27   {V ⊩ s ↦_m _ ∗ F_tid ∧ I ∧ S(tid,0) ∧
28    [s ↦_e _ ∗ true]}
29   insert(detached[tid-1], s);
30   if (nondet()) return;
31   {V ⊩ I ∧ S(tid,0) ∧
32    ∃A. detached[tid − 1] ↦ A ∗
33    (⊛_{x∈A} x ↦_m _) ∧
34    [(⊛_{x∈A} x ↦_e _) ∗ true] ∧
35    (⊛_{x∈A} ¬[C ↦ x ∗ x ↦ _ ∗ true])}
36   sync();
37   {V ⊩ I ∧ S(tid,0) ∧
38    ∃A. detached[tid − 1] ↦ A ∗
39    ⊛_{x∈A}((x ↦_m _ ∧ [x ↦_e _ ∗ true]) ∧ ∀t.
40    ¬[C ↦ x ∗ x ↦ _ ∗ true] since S(t,0))}
41   ⟨ ; ⟩_Take
42   {V ⊩ I ∧ S(tid,0) ∧
43    ∃A. detached[tid − 1] ↦ A ∗
44    (⊛_{x∈A} x ↦ _)}
45   while (!isEmpty(detached[tid]))
46     free(pop(detached[tid]));
47   {V ⊩ F_tid ∧ I ∧ S(tid,0)}
48 }
```

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

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 [7, §B] have been verified: in concurrent separation logic [14], a combination of separation logic and temporal logic [6], a reduction-based tool [3] and interval temporal logic [16]. 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 [7, §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. Boyland, J.: Checking Interference with Fractional Permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
2. Calcagno, C., O'Hearn, P., Yang, H.: Local action and abstract separation logic. In: LICS (2007)
3. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL (2009)
4. Feng, X., Ferreira, R., Shao, Z.: On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 173–188. Springer, Heidelberg (2007)
5. Fraser, K.: Practical lock-freedom. PhD Thesis. University of Cambridge (2004)
6. Fu, M., Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about Optimistic Concurrency Using a Program Logic for History. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 388–402. Springer, Heidelberg (2010)
7. Gotsman, A., Rinetzky, N., Yang, H.: Verifying concurrent memory reclamation algorithms with grace. Technical Report 7/13, School of Computer Science, Tel-Aviv University (2013), `http://www.cs.tau.ac.il/~maon`
8. Herlihy, M., Luchangco, V., Moir, M.: The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 339–353. Springer, Heidelberg (2002)
9. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. TOPLAS (1990)
10. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress (1983)
11. McKenney, P.: Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels. PhD Thesis. OGI (2004)
12. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. (2004)
13. O'Hearn, P.: Resources, concurrency and local reasoning. TCS (2007)
14. Parkinson, M., Bornat, R., O'Hearn, P.: Modular verification of a non-blocking stack. In: POPL (2007)
15. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Logics and Models of Concurrent Systems (1985)
16. Tofan, B., Schellhorn, G., Reif, W.: Formal Verification of a Lock-Free Stack with Hazard Pointers. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 239–255. Springer, Heidelberg (2011)
17. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD Thesis. University of Cambridge (2008)