Concurrent Data Structures Linked in Time

G. A. Delbianco[1], I. Sergey[2], A. Nanevski[1], and A. Banerjee[1]

1 **IMDEA Software Institute, Madrid, Spain**
{german.delbianco,aleks.nanevski,anindya.banerjee}@imdea.org
2 **University College London, UK**
i.sergey@ucl.ac.uk

Abstract

Arguments about linearizability of a concurrent data structure are typically carried out by specifying the linearization points of the data structure's procedures. Proofs that use such specifications are often cumbersome as the linearization points' position in time can be *dynamic*, *non-local* and *non-regional*: it can depend on the interference, run-time values and events from the past, or even future, appear in procedures other than the one considered, and might be only determined after the considered procedure has terminated.

In this paper we propose a new method, based on a Hoare-style logic, for reasoning about concurrent objects with such linearization points. We embrace the dynamic nature of linearization points, and encode it as part of the data structure's *auxiliary state*, so that it can be dynamically modified in place by auxiliary code, as needed when some appropriate run-time event occurs.

We name the idea *linking-in-time*, because it reduces temporal reasoning to spatial reasoning. For example, modifying a temporal position of a linearization point can be modeled similarly to a pointer update in a heap. We illustrate the method by verifying an intricate optimal snapshot algorithm due to Jayanti.

1 Introduction

Formal verification of concurrent objects commonly requires reasoning about linearizability [11]. This is a standard correctness criterion whereby a concurrent execution of an object's procedures is proved equivalent, via a simulation argument, to some sequential execution. The clients of the object can be verified under the sequentiality assumption, rather than by inlining the procedures and considering their interleavings. Linearizability is often established by describing the *linearization points* (LP) of the object, which are points in time where procedures take place, *logically*. In other words, even if the procedure physically executes across a time interval, exhibiting its linearization point enables one to pretend, for reasoning purposes, that it occurred instantaneously; hence, an interleaved execution of a number of procedures can be reduced to a sequence of instantaneous events.

However, reasoning with linearizability, and about linearization points, can be tricky. Many times, a linearization point of a procedure is not *local*, but may appear in another procedure or thread. Equally bad, a linearization points' place in time may not be determined statically, but may vary based on the past, and even future, *run-time* information. This complicates the simulation arguments, leading to unwieldy formal logical proofs.

This paper presents a novel specification and verification method for concurrent objects, based on Hoare logic. It achieves the same goal as linearizability; that is, client proofs can reason out of Hoare triples of the object's procedures, rather than inline the procedures and consider all interleavings. The method improves on linearizability by offering natural abstractions for specifying dynamic and non-local linearization points, based on familiar concepts from Hoare logics for shared-memory concurrency. It also permits better *information hiding*, allowing us to expose to clients less information about the internals of the object compared to linearizability.


Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

More specifically, the method consists of two components. First, we use shared *auxiliary state* [17] to record, as a list of timed events (e.g., writes occurring at a given time), the logical order in which the object's procedures are perceived to execute, each instantaneously. Tracking this time-related information through state enables us to specify the linearization points *dynamically*. In particular, we can use auxiliary code to mutate the logical order in place, thereby permuting the sequencing of the procedures, as may be needed when some run-time event occurs. This mutation is similar to updating pointers to reorder a linked list, except that it is executed over auxiliary state storing time-related data, rather than over real state. This is why we refer to the idea as *linking-in-time*.

Second, unlike in linearizability, where a concurrent program is specified by an equivalent sequential one, our Hoare triples specify programs *in relation* to the behavior of the interfering threads. Specifically, our Hoare triples scope over *two local* variables H_s (aka. *self*-variable) and H_o (aka. *other*-variable) that store the histories of the events attributed to the specified program, and to its interfering environment, respectively. The specifications can relate the events in H_s and H_o to each other, and to the shared auxiliary list of logical times described above. For example, an event with a timestamp t appearing in H_s of a procedure A, models that a call to A was linearized at time t. But this timestamp may also be seen as a pointer into the list of logical times. "A's linearization point appearing in procedure B" will be manifested by the auxiliary code of B rearranging this list, to permute the node pointed by t. However, the rearrangement does not change A's ownership of the event occurring at t, as t still appears in H_s of A. The setup will enable us to specify A *locally*, in terms of auxiliary state that A manipulates, rather than in terms of line numbers in the code of B.

Encoding temporal information by way of mutable state, will allow us to use, off-the-shelf, a variant of fine-grained concurrent separation logic (FCSL) [19] to verify example programs. FCSL has been implemented in the proof assistant Coq, and we are now finishing the Coq mechanization of the example from this paper. While several recent Hoare logics have targeted concurrent programs with non-thread-local and future-dependent linearization points [14, 21], they only allowed to establish a procedure's LP position based on the observations made *during* the procedure's symbolic execution, i.e., scoped within its *region*. However, a number of modern concurrent data structures exhibit executions whose linearization order can only be established *after* the involved overlapping procedure calls have terminated [4, 12]. We call the linearization points of such executions *non-regional*, and our method is the first that provides a logic for reasoning about non-regional linearization points.

We illustrate the method by applying it to a snapshot algorithm of Jayanti [12] (Section 2), whose linearization points depend on dynamic information in an intricate non-regional way. We show how the auxiliary state and code should be designed to provide local specifications for this algorithm (Sections 3,4), and sketch some aspects of our version of the proof (Section 4). We discuss the implications of the design to client-side reasoning (Section 5) and conclude with the related work (Section 6).

2 Verification challenge and main ideas

Jayanti's snapshot algorithm [12] provides the functionality of a shared array of size m, operated on by two procedures: `write`, which stores a given value into an element, and `scan`, which returns the array's contents. We use the *single-writer/single-scanner* version of the algorithm, which assumes that at most one thread writes into an element, and at most one thread invokes the scanner, at any given time. This is the simplest of Jayanti's algorithms, but it already exhibits linearization points of dynamic nature. We also restrict the array size

```
1 write (p : ptr, v : A) {
2  p := v;
3  b ← read(S);
4  if b
5  then (f_of p) := v // forward p to fp
6  else return }

   f_of (p : ptr) {
    return p = x ? fx : fy }
```

```
7 scan (): A × A  {
8  S := true;
9  fx := ⊥; fy := ⊥;
10 vx ← read(x); vy ← read(y);
11 S := false;
12 ox ← read(fx); oy ← read(fy);
13 rx ← if (ox ≠ ⊥) then ox else vx;
14 ry ← if (oy ≠ ⊥) then oy else vy;
15 return (rx, ry) }
```

Figure 1 Jayanti's single-scanner/single-writer snapshot algorithm.

```
l:  write (x,2);  ||  c:  scan ()  ||  r:  write (x,3)
    write (y,1)   ||                  ||
```

(a) Parallel composition of three threads `l, c, r`.

```
1  c: S := true              8  l: fx := 2              15 c: S := false
2  c: fx := ⊥                9  l: ret ()               16 r: read(S)  // b <- false
3  c: fy := ⊥                10 r: x := 3               18 r: ret ()
4  c: read(x)  // vx <- 5    11 l: y := 1               18 c: read(fx) // ox <- 2
5  c: read(y)  // vy <- 0    12 l: read(S)  // b <- true 19 c: read(fy) // oy <- 1
6  l: x := 2                 13 l: fy := 1              20 c: ret (2,1)
7  l: read(S)  // b <- true  14 l: ret ()
```

(b) A possible interleaving of the threads in (a).

Figure 2 An example leading to a scanner miss.

to $m = 2$ (i.e., we consider two pointers x and y, instead of an array). This removes some tedium from verification, but exhibits the same conceptual challenges.

The difficulty in this snapshot algorithm is ensuring that the scanner returns sound values of x and y. A naive scanner, which simply reads x and y in succession, is unsound. To see why, consider the following scenario, starting with $x = 5$, $y = 0$. The scanner reads x, but before it reads y, another thread preempts it, and changes x to 2 and y to 1. The scanner continues to read y, and returns $x = 5, y = 1$, but this was never the contents of the memory.

To ensure a sound snapshot, Jayanti's algorithm internally keeps additional *forwarding pointers* fx and fy, and a boolean *scanner bit* S. The implementation is given in Figure 1, and *assumes* the single-writer/single-scanner setup (the assumption can be enforced with wrapper locking code, which we omit here for simplicity, but consider in our Coq proofs). The intuition is as follows. A writer storing v into p (line 2), will additionally store v into the forwarding pointer for p (line 5). If the scanner missed the write and instead read the old value of p (line 10), it will have a chance to catch v via the forwarding pointer (line 12). The scanner bit S is used by writers (line 3) to detect a scan in progress, and forward v.

As Jayanti proves, this implementation is linearizable. Informally, every overlapping calls to `write` and `scan` can be rearranged to appear as if they occurred sequentially. To illustrate, consider the program in Figure 2a, and one possible interleaving of its primitive memory operations in Figure 2b. The threads `l`, `c`, and `r`, start with $x = 5, y = 0$. The thread `c` is scheduled first, and through lines 2-6 sets the scanner bit, clears the forwarding pointers, and reads $x = 5, y = 0$. Then `l` intervenes, and in lines 7-11, overwrites x with 2, and seeing S being set, forwards 2 to fx. Next, `r` and `l` overlap, writing 3 into x and 1 into y. However, while 1 gets forwarded to fy (line 13), 3 is not forwarded to fx, because S was turned off in line 15. Thus, when `c` reads the forwarded values (lines 18, 19), it returns $x = 2, y = 1$.

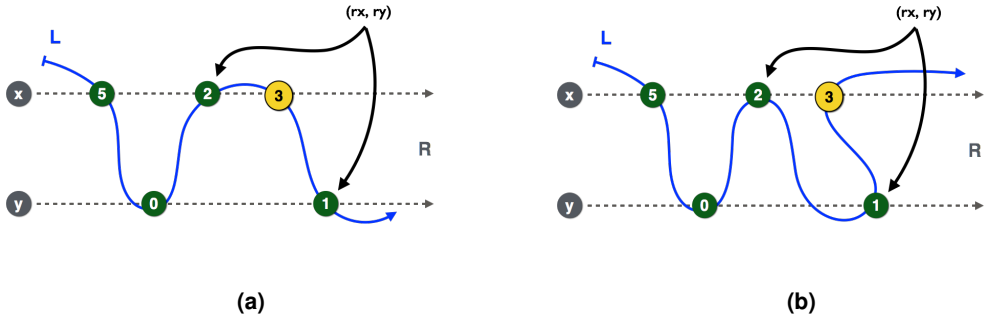

Figure 3 Changing the logical ordering (solid line L) of write events from (5, 0, 2, 3, 1) in (a) to (5, 0, 2, 1, 3) in (b), to reconcile with `scan` returning the snapshot $x = 2, y = 1$, upon missing the write of 3. Dashed lines R represent real-time ordering.

While $x = 2, y = 1$ was never the contents of the memory, returning this snapshot is nevertheless justified because we can *pretend* that the scanner *missed* `r`'s write of 3. Specifically, the events in Figure 2b can be *reordered* to represent the following sequential execution.

$$\texttt{write}(\texttt{x}, 2);\ \texttt{write}(\texttt{y}, 1);\ \texttt{scan}();\ \texttt{write}(\texttt{x}, 3) \tag{1}$$

The client programs cannot discover that a different scheduling actually took place in real time, because they can access the internal state of the algorithm only via `write` and `scan`.

This kind of temporal reordering is the most characteristic aspect of linearizability proofs, which typically describe the reordering by listing the linearization points of each procedure. At a linearization point, the procedure's operations can be spliced into the execution history as an uninterrupted chunk. For example, in Jayanti's proof, the linearization point of `scan` is at line 11 (Figure 1), where the scanner bit is unset. The linearization point of `write`, however, may vary. If `write` starts before an overlapping `scan`'s line 11, and moreover, the `scan` misses the `write` (note the dynamic nature of this property), then `write` should appear after `scan`; that is, the `write`'s linearization point is right after `scan`'s linearization point at line 11. Otherwise, `write`'s linearization point is at line 2. In the former case, `write` has a *future-dependent*, *non-local*, *non-regional* linearization point, as we can only identify it in the execution of `scan` in a different thread, and only *after* the execution of `write` has terminated. For instance, in Figure 2b the execution of the second `write` in `l` terminates at step 14, yet the decision of its position in the linearization order is taken only when the scanner is at step 15.

Obviously, the high-level pattern of the proof requires tracking a logical ordering of the `write` and `scan` events, which differs from the real time. As the logical ordering is inherently dynamic, depending on properties such as `scan` missing a `write`, we formalize it in Hoare logic, by keeping it as a list of events in auxiliary state that can be dynamically reordered as needed. For example, Figure 3a shows the situation in the execution of `scan` that we reviewed above, where the logical ordering coincides with real-time ordering, but is unsound for the snapshot $x = 2, y = 1$ that `scan` wants to return. In that case, the auxiliary code with which we annotate `scan`, will change the list in-place, as shown in Figure 3b.

Our challenge then lies in reconciling the following two conflicting requirements. First, we need to implement the reordering discipline so that the subsequent calls to `write` and `scan` preserve the established logical order of the past events. This will be accomplished by introducing yet further structures into the auxiliary state and code. Second, we have to

$$\texttt{write}\ (p : \texttt{ptr}, n : \texttt{int}) : \{H_\mathsf{s} = \mathsf{empty} \wedge h \subseteq H_\mathsf{o} \wedge \omega \subseteq \sqsubseteq_L\}$$
$$\{\exists t.\ H_\mathsf{s} = t \mapsto (p, n) \wedge h \subseteq H_\mathsf{o} \wedge \omega \subseteq \sqsubseteq_L \wedge h \subseteq H^{\not\sqsubseteq_L t}\}$$
$$\texttt{scan} : \{H_\mathsf{s} = \mathsf{empty} \wedge\ h \subseteq H_\mathsf{o} \wedge\ \omega \subseteq \sqsubseteq_L\}$$
$$\{H_\mathsf{s} = \mathsf{empty} \wedge\ h \subseteq H_\mathsf{o} \wedge\ \omega \subseteq \sqsubseteq_L \wedge\ \exists t.\ h \subseteq H^{\sqsubseteq_L t} \wedge\ \mathsf{chain}\ H^{\sqsubseteq_L t} \wedge\ r = \mathsf{eval}\ H^{\sqsubseteq_L t}\}$$

Figure 4 Specification of `write` and `scan` methods.

engineer Hoare triples for `write` and `scan` to be *intuitive* and *usable* by clients, but also to *not expose* the specifics of the reordering discipline, which is internal to the snapshot object.

3 Formal proof structures

Specification. For the purposes of specification and proof, we record a history of the snapshot object as a set of entries of the form $t \mapsto (p, v)$. The entry says that at time t (a natural number), the value v was written into the pointer p. Notice that we identify a write event with a *single* moment in time t, in contrast to linearizability which keeps a time interval. As we shall see, internally, our proofs record intervals too, but we hide the interval end-points from the clients. Also, unlike linearizability, we do not need to (though we could) keep track of scan events. Scans do not modify the object state in ways observable by clients, and may thus be omitted from specifications. These two points illustrate the improvement in information hiding over linearizability that we mentioned in Section 1.

The auxiliary variables H_s and H_o, which are local to the specification, record the *finished* write events of each procedure, and of the interfering environment, respectively. We also have a local variable H_j, which records the set of write events *in progress*. When a call to `write` is initiated, an auxiliary code places a write entry into H_j; when a call finishes, the entry is moved from H_j to H_s. We name by H the union $H_\mathsf{s} \cup H_\mathsf{o} \cup H_\mathsf{j}$, which is the history of the overall snapshot data structure. This is a *disjoint union*, enforcing that H_s, H_o and H_j do not contain common timestamps.

The timestamps in H determine the *real-time* ordering of write events. We record the *logical* ordering in another auxiliary variable L, whose type is list (i.e., mathematical sequence). This list stores the permutation of timestamps from H, and we write $t_1 \leqslant_L t_2$ if t_1 appears before t_2 in L. The list L is the "linking-in-time" for the snapshot algorithm, and can be dynamically modified by the threads. The order $\leqslant_L$ is total (i.e., a chain), but as $\leqslant_L$ is dynamic, we introduce further notation, and name by $\sqsubseteq_L$ a *partial suborder* of $\leqslant_L$ that is *stable*, i.e., it can only grow over time to add new relations, but cannot change the old ones. For example, in Figure 3, the initial value of $\leqslant_L$ is (in lattice notation) $5{-}0{-}2{-}3{-}1$, which changes to $5{-}0{-}2{-}1{-}3$. Stable order $\sqsubseteq_L$ is the lattice $5{-}0{-}2{<}\begin{matrix}1\\3\end{matrix}$: 1 and 3 are unrelated.

We will formally define $\sqsubseteq_L$ later, but we can already use it to give Hoare triples for `write` and `scan`. In fact, it will be important for client reasoning (Section 5) to keep the definition of $\sqsubseteq_L$ *hidden*, so that different snapshot algorithms can provide different definitions, without influencing the clients. Figure 4 presents our specifications for `write` and `scan`. These are partial correctness Hoare triples, describing how the program changes the state from the precondition (first braces) to the postcondition (second braces), possibly influencing the return result r. The expression $H^{\sqsubseteq_L t}$ selects the entries in the history H *up to and including* the timestamp t, according to the ordering $\sqsubseteq_L$. Likewise, $H^{\not\sqsubseteq_L t}$ is defined to exclude t. We

describe the specifications next, before commenting on their relationship to linearizability.

The specification for `write` says the following. We start with the empty self history indicating that the procedure did not yet make any writes. We use the variable h to name an arbitrary subset of the initial value of H_o (hence, of *completed* write events of *all other threads*), and the variable ω for a subset of the initial ordering $\sqsubseteq_L$. The postcondition says that when `write` terminates, one write event $t \mapsto (p, v)$ has finished, and is hence placed into H_s. H_o and $\sqsubseteq_L$ may have changed from the previous values, due to interference, but they still include h and ω as subsets. That is, H_o could only grow, because other threads could create new write events, and similarly for $\sqsubseteq_L$. Lastly, the conjunct $h \subseteq H^{\sqsubseteq_L t}$ says that the write events that have been completed before the call to `write` (and are hence stored in h) will be *ordered before* the event t in $\sqsubseteq_L$, no matter how L is permuted by interfering threads. In other words, we cannot logically reorder past events, which is a basic soundness requirement. But notice how stating it requires that H_o is in scope, thus directly relating `write` to the interfering threads.

The specification for `scan` starts with the same precondition. In the postcondition, it says that H_s is empty, because `scan` itself does not create write events. However, by the time `scan` returns the pair r as a snapshot, we know that there exists a timestamp t in the collective history H at which the snapshot *appears* to be taken. First, the conjunct $h \subseteq H^{\sqsubseteq_L t}$ indicates that the snapshot is taken *after* the call to `scan`, because the finished events stored in h are ordered before (or at best, at) timestamp t. Second, the timestamps from $H^{\sqsubseteq_L t}$ form a sub-chain in $\sqsubseteq_L$, encoding how the writes progressed logically in time up to the snapshot moment t. Moreover, since $\sqsubseteq_L$ is stable, this view of the time will not change in the future to invalidate the result r as a snapshot. Finally, if the chain of writes is evaluated in the order given by $\sqsubseteq_L$, it produces r.

Notice how these specifications directly capture what linearizability would have given us: in both procedures we get a time moment t at which the procedure executed *logically instantaneously*, as part of a larger *sequence* H. Additionally, `write` performed a write event at t, and `scan` returned a snapshot consistent with scanning at t. In the case of linearizability, the last property requires showing a simulation between the concurrent implementations of `write` and `scan`, and some sequential equivalents. If we then want to verify Hoare triples of clients, we need to establish Hoare triples for the sequential equivalents. In our method, the process is streamlined by immediately giving Hoare triples for the concurrent implementations, avoiding intermediary sequential equivalents and simulation proofs.

Internal auxiliary state. The proofs of `write` and `scan` require further auxiliary state, which does not feature in the specifications, and is hence hidden from clients.

First, we track the point of execution in which `write` and `scan` are, but instead of line numbers in the code, we use datatypes, to encode extra information in the constructors. For example, the scanner's state is a triple (S_s, S_x, S_y). S_s is drawn from $\{\mathsf{S_{on}}, \mathsf{S_{off}}\ t\}$. If $\mathsf{S_{on}}$, then the scanner is in lines 8–10 in Figure 1. If $\mathsf{S_{off}}\ t$, the the scanner reached line 11 at time t, and is now in 11–15. S_x is a boolean, set when the scanner clears fx in line 9, and reset upon scanner's termination (dually for S_y and fy). Writers' state for x is tracked by the auxiliary W_x (dually, W_y). These are drawn from $\{\mathsf{W_{off}}, \mathsf{New}\ t, \mathsf{NeedsFwd}\ t, \mathsf{Done}\ t\}$, where t marks the beginning of the write. If $\mathsf{W_{off}}$, then no write is in progress. If $\mathsf{New}\ t$, then the writer is in line 2. If $\mathsf{NeedsFwd}\ t$, then b has been set in line 3, triggering forwarding. If $\mathsf{Done}\ t$, the writer is free to exit.

Second, like in linearizability, we record the ending times of events. We use the auxiliary variable E, which is a function mapping a timestamp t of a *finished* write event from H, to a

timestamp identifying the event's *ending time*. Events t_1 and t_2 which are non-overlapping (i.e., $E(t_1) < t_2$ or $E(t_2) < t_1$), will never be reordered, thus `write` and `scan` cannot modify the past history.

▸ **Proposition 1.** $\forall t_1 \in \mathsf{dom}(E), t_2 \in \mathsf{dom}(H)$, if $E(t_1) < t_2$ then $t_1 <_L t_2$.

Third, we track the rearrangement status of write events wrt. an ongoing *active* scan, by *colors*. A scan is *active* if it has cleared the forwarding pointers in line 9, and is ready to read x and y. The auxiliary variable C is a function mapping each timestamp in H to a color, as follows.

- Green timestamps identify events whose position in the logical order is fixed in the following sense: if $C(t_1) = \mathsf{green}$ and $t_1 <_L t_2$, then $t_1 <_{L'} t_2$ for every L' to which L may step by auxiliary code execution (Section 4). For example, since we only reorder overlapping events, and only the scanner reorders events, every event that finished before the active scan started will be green. Also, a green timestamp never changes the color.
- Yellow timestamps identify events whose order is not fixed yet (i.e., they are not linearized), but which *may* be manipulated by the ongoing active scan, as follows. The scan can *push* a yellow timestamp in logical time, *past* another green or yellow timestamp, but not past a red one. *This is the only way the logical ordering can be modified.*
- Red timestamps identify events whose order is not fixed yet, but which will *not* be manipulated by the active scan, and are left for the next one.

There is a number of invariants that relate colors and timestamps. For brevity, we only list the most important ones, and defer to the Coq code for the others [1]. In the sequel, we use H_p to denote the set of writes into the pointer p that appear in the history H.

▸ **Proposition 2** (Color Invariant)**.** The colors of H_p are described by the regular expression $\mathbf{g}^+\mathbf{y}^?\mathbf{r}^*$: there is a non-empty prefix of green timestamps, followed by *at most* one yellow, and arbitrary number of reds.

This proposition identifies, by the yellow color, the write event for p that is the unique candidate for reordering by the ongoing active scan. Moreover, all the writes into p prior to the yellow one, will have already been painted green (i.e., fixed in time, linearized), whether they overlapped with the scanner or not.

▸ **Proposition 3** (Green/Yellow Forwarded Values)**.**
If $S_s = \mathsf{S_{off}}\ t_{\mathsf{off}}$ and $S_p = \mathsf{True}$ (i.e., scanner is in lines 12–14), and a value $v \neq \bot$ has been forwarded to p (i.e., $fp = v$), then the event of writing v into p is in the history, i.e., exists t such that $t \mapsto (p, v) \in H_p$. Moreover, t is the last green, or the yellow timestamp in H_p.

This proposition restricts the set of events that could have forwarded a value to the scanner, to only two: the event with the (unique) yellow timestamp, or the one with the last green timestamp. By Proposition 2, the two timestamps are consecutive in H_p.

▸ **Proposition 4** (Red-Zone Invariant)**.** If $S_s = \mathsf{S_{off}}\ t_{\mathsf{off}}, S_x = \mathsf{True}, S_y = \mathsf{True}$, then H satisfies the $(\mathbf{g}|\mathbf{y})^+\mathbf{r}^*$ pattern. Moreover, for every $t' \in \mathsf{dom}(H)$:

- $C(t') = \mathsf{green} \implies t' \leqslant t_{\mathsf{off}}$
- $C(t') = \mathsf{yellow} \implies t' \leqslant t_{\mathsf{off}} \leqslant E(t')$
- $C(t') = \mathsf{red} \implies t_{\mathsf{off}} < t'$

This proposition restricts the global history H (not the projections H_p). First, the red events in H do not mix with the green and yellow ones. Thus, when a scanner pushes a

```
1  write (p, v): () {
2   ⟨p := v; register(p,v)⟩;
3   ⟨b ← read(S); check(p,b)⟩;
4   if b
5   then ⟨(f_of p) := v; forward(p,b))⟩
6   else ⟨finalize(); return ()⟩}
```

```
7  scan (): A × A  {
8  ⟨ S := true; setS(true)⟩;
9  ⟨fx := ⊥; clear(x)⟩; ⟨fy := ⊥; clear(y)⟩;
10 vx ← read(x); vy ← read(y);
11 ⟨ S := false; setS(false)⟩;
12 ox ← read(fx); oy ← read(fy);
13 rx ← if (ox ≠ ⊥) then ox else vx;
14 ry ← if (oy ≠ ⊥) then oy else vy;
15 ⟨relink(rx, ry); return (rx, ry)⟩}
```

Figure 5 Snapshot procedures annotated with auxiliary code.

yellow event past a green or yellow event, it will not "jump over" any reds, thus keeping the invariant that ordering on reds is not manipulated by the scan. Second, the proposition relates the colors to the time t_{off} at which the scanner was turned off (in line 11, Figure 1). This moment is important for the algorithm; e.g., it is the linearization point for `scan` in Jayanti's proof [12]. In our proof, the inequalities wrt. t_{off} are used to prove that the events reordered by the scanner *do* overlap, as per Proposition 1.

We can now define the stable logical order $\sqsubseteq_L$, which relies on the internal auxiliary state.

▸ **Definition 5** (Logical Order). $t_1 \sqsubseteq_L t_2 \;\widehat{=}\; t_1 = t_2 \vee E(t_1) < t_2 \vee (t_1 <_L t_2 \wedge C(t_1) = \mathsf{green})$.

The proposition $t_1 \sqsubseteq_L t_2$ is stable (i.e., invariant under interference), since threads do not change the ending times E, the color of green events, or the order of green events in $<_L$.

4 Auxiliary code and sketch of the proof

Figure 5 annotates Jayanti's procedures with auxiliary code (typed in *italic*), with ⟨angle braces⟩ denoting that the enclosed real and auxiliary code execute *simultaneously*. The auxiliary code builds the object histories and evolves the logical ordering in-place, while respecting the invariants from Section 3. Thus, it is the *constructive* component of our proofs.

The auxiliary code is divided into several procedures, all of which are sequences of state reads followed by updates to auxiliary variables. We present them as Hoare triples in Figure 6, listing reads in the precondition, and writes in the postcondition, with the unmentioned state considered unchanged. Following are the characteristic cases.

For example, *register*(p, v) creates the write event for the assignment of v to p in line 2. It allocates a fresh timestamp t, inserts the entry $t \mapsto (p, v)$ into H_{j}, and adds t to the end of L, thus making t be the last write event, until and unless L is permuted later. The variable W_p records that the writer finished line 2 with a fresh t. The color of t is set to yellow (i.e., the rearrangement status of the event is left undetermined), if $(S_s = \mathsf{S_{on}})\&S_p$, i.e., an active scanner is in line 10. Otherwise, t is colored red, indicating that the logical order of t will be determined by a future active scan.

In line 5, *forward* paints the allocated timestamp t green, if an active scanner is in line 10, because such a scanner will definitely see the write, either by reading the original value in line 10, or by reading the forwarded value in line 12. Thus, the logical order of t becomes fixed. In fact, it is possible to derive from the invariants in Section 3, that this order is the same one t was assigned at registration, i.e., the linearization point of this write is line 2.

The key auxiliary procedure of our method is *relink*. It is executed just before the scanner returns the pair (r_x, r_y). Its task is to modify the logical order of the writes, to make (r_x, r_y)

(a)

$$
\begin{array}{ll}
\mathit{register}(p,v): & \{L = l,\ H_\mathsf{j} = j,\ W_p = \mathsf{W_{off}},\ C = c\} \\
& \{L = \mathsf{snoc}\ l\ t, H_\mathsf{j} = j \cup t \mapsto (p,v), W_p = \mathsf{New}\ t, \\
& \ C = \text{if } (S_s = \mathsf{S_{on}}) \& S_p \text{ then } c[t \mapsto \mathsf{yellow}] \text{ else } c[t \mapsto \mathsf{red}]\} \text{ where } t = \mathsf{fresh}\ H \\
\mathit{check}(p,b): & \{W_p = \mathsf{New}\ t\} \quad \{W_p = \text{if } b \text{ then } \mathsf{NeedsFwd}\ t \text{ else } \mathsf{Done}\ t\} \\
\mathit{forward}(p): & \{W_p = \mathsf{NeedsFwd}\ t,\ C = c\} \\
& \{W_p = \mathsf{Done}\ t,\ C = \text{if } (S_s = \mathsf{S_{on}}) \& S_p \text{ then } c[t \mapsto \mathsf{green}] \text{ else } c\} \\
\mathit{finalize}(p): & \{W_p = \mathsf{Done}\ t,\ H_\mathsf{s} = h,\ H_\mathsf{j} = j \cup t \mapsto (p,v), E = e\} \\
& \{W_p = \mathsf{W_{off}},\ H_\mathsf{s} = h \cup t \mapsto (p,v),\ H_\mathsf{j} = j,\ E = e \cup t \mapsto \mathsf{last}\ H\}
\end{array}
$$

(b)

$$
\begin{array}{ll}
\mathit{setS}(b): & \{S_s = \text{if } b \text{ then } \mathsf{S_{off}}\ \_ \text{ else } \mathsf{S_{on}},\ S_x = \neg b, S_y = \neg b\} \\
& \{S_s = \text{if } b \text{ then } \mathsf{S_{on}} \text{ else } \mathsf{S_{off}}\ (\mathit{max}\,(\mathsf{dom}(H))),\ S_x = \neg b, S_y = \neg b\} \\
\mathit{clear}(p): & \{S_s = \mathsf{S_{on}},\ S_p = \mathsf{False},\ C = c\} \quad \{S_s = \mathsf{S_{on}},\ S_p = \mathsf{True},\ C = c[H_p \mapsto \mathsf{green}]\} \\
\mathit{relink}(r_x, r_y): & \{S_s = \mathsf{S_{off}}\ t_\mathsf{off}, C = c, L = l,\ H = h \cup t_x \mapsto (x, r_x) \cup t_y \mapsto (y, r_y), \\
& \ p \in \{x,y\}: \ S_p = \mathsf{True}, (t_p = \mathit{last\_green}\ H_p \vee C(t_p) = \mathsf{yellow})\} \\
& \{S_s = \mathsf{S_{off}}\ t_\mathsf{off}, S_x = \mathsf{False}, S_y = \mathsf{False}, C = c[t_x, t_y \mapsto \mathsf{green}], \\
& \ L = \text{if } (d = \mathsf{Yes}\ x\ t') \text{ then } \mathit{push}\ t'\ t_y\ l \\
& \qquad \text{else if } (d = \mathsf{Yes}\ y\ t') \text{ then } \mathit{push}\ t'\ t_x\ l \text{ else } l\} \quad \text{where } d = \mathit{inspect}\ t_x\ t_y\ l\ C
\end{array}
$$

Figure 6 Auxiliary procedures for (a) `write` and (b) `scan`. Small caps variables (e.g., l) record the pre-values of auxiliaries (e.g., L).

appear as a valid snapshot. This will always be possible under the precondition of *relink* that the timestamps t_x, t_y of the events that wrote r_x, r_y respectively, are either the last green or the yellow ones in the respective histories H_x and H_y, and *relink* will consider all four cases. This precondition holds after line 14 in Figure 5, as one can prove from the algorithm invariants (e.g., Propositions 2 and 3).

Relink uses two helper procedures *inspect* and *push*, to change the logical order. *inspect* decides if the selected t_x and t_y determine a valid snapshot, and *push* performs the actual reordering. The snapshot determined by t_x and t_y is valid if there is no event t' such that $t_x <_L t' <_L t_y$ and t' is a write to x (or, symmetrically $t_y <_L t' <_L t_x$, and t' is a write to y). If such t' exists, *inspect* returns $\mathsf{Yes}\ x\ t'$ (or $\mathsf{Yes}\ y\ t'$ in the symmetric case). The reordering is completed by *push*, which moves t' right after t_y (after t_x in the symmetric case) in $\leqslant_L$. Finally, *relink* paints t_x and t_y green, to fix them in $\sqsubseteq_L$. We can then prove that (r_x, r_y) is a valid snapshot wrt. $\sqsubseteq_L$, and remains so under interference. Notice that the timestamp t' returned by *inspect* is always uniquely determined, and yellow. Indeed, since t_x and t_y are not red, no timestamp between them can be red either (Proposition 4). If $t_x <_L t' <_L t_y$ and t' is a write to x (and the other case is symmetric), then t_x must be the last green in H_x, forcing t' to be the unique yellow timestamp in H_x, by Proposition 2.

To illustrate, in Figure 3a we have $r_x = 2$, $r_y = 1$, t_x and t_y are both the last green timestamp of H_x and H_y, respectively, and $t_x <_L t_y$. However, there is a yellow timestamp t' in H_x coming after t_x, encoding a write of 3. Because $t_x <_L t' <_L t_y$, the pair (r_x, r_y) is not a valid snapshot, thus *inspect* returns $\mathsf{Yes}\ x\ t'$, after which *push* moves 3 after 1.

We devote the rest of the section to illustrating the structure of our proofs of `write` and `scan`, including aspects of mechanization in FCSL/Coq. For lack of space, we cannot present the full theory, but just focus on the structure and a few characteristic properties.

The first stage of proof (and the mechanization) involves defining the state space of the algorithm; that is, the definitions of the auxiliaries, and the invariants that relate them. We presented selected invariants in Section 3. There is a number of properties that need to be established under the state space, e.g., that it is closed under exchanging history entries

between H_s and H_o. The latter facilitates dynamic thread creation, because it allows the reasoning to focus on individual threads in a large parallel composition. The second stage involves defining the auxiliary code, as in Figure 6, and proving a number of its properties. In addition to correctness, the properties involve *erasure* (i.e., auxiliary code only mutates auxiliary state, but not the real one), *framing* (i.e., if auxiliary code is executable in some input heap and input history, then executing it in an extended heap and history leads to the same results, and the extensions remain invariant), preservation of invariants (from Section 3), and *termination*. The correctness aspect of auxiliary code involves proving that the code preserves the state-space invariants, and that the precondition and postcondition hold. For example, the correctness proof of *relink*, relies on the following two helper lemmas.

▸ **Lemma 6** (Correctness of *inspect*). *If t_x, t_y are timestamps for write events of r_x, r_y, then inspect t_x t_y L C correctly determines that (r_x, r_y) is a valid snapshot under ordering $\leqslant_L$ and coloring C, or otherwise, returns an "offending" timestamp. More formally, if $S_s = \mathsf{S}_{\mathsf{off}}\ t_{\mathsf{off}}, S_x = \mathsf{True}, S_y = \mathsf{True}$, and for each $p \in \{x, y\}$, $t_p \mapsto (p, r_p) \in H$ and $(t_p = last\_green\ H_p \vee C(t_p) = \mathsf{yellow})$, the following are exhaustive possibilities.*

1. *If $t_x <_L t_y$ and $C(t_x) = \mathsf{yellow}$, then inspect t_x t_y L $C = \mathsf{No}$. Symmetrically for $t_y <_L t_x$.*
2. *If $t_x <_L t_y$, $t_x = last\_green\ H_x$, and $\forall t' \in H_x.\ t_x <_L t' \implies t_y <_L t'$, then inspect t_x t_y L $C = \mathsf{No}$. Symmetrically for $t_y <_L t_x$.*
3. *If $t_x <_L t_y$, $t_x = last\_green\ H_x$, $t' \in H_x$, and $t_x <_L t' <_L t_y$, then inspect t_x t_y L $C = \mathsf{Yes}\ x\ t'$ and $C(t') = \mathsf{yellow}$. Symmetrically for $t_y <_L t_x$.*

▸ **Lemma 7** (Correctness of *push*). *If $S_s = \mathsf{S}_{\mathsf{off}}\ t_{\mathsf{off}}, S_x = \mathsf{True}, S_y = \mathsf{True}$, and for $p \in \{x, y\}$, we have $t_p \mapsto (p, r_p) \in H$ and $(t_p = last\_green\ H_p \vee C(t_p) = \mathsf{yellow})$, and inspect t_x t_y L $C = \mathsf{Yes}\ p\ t'$, then:*

1. *If $p = x$, then (r_x, r_y) is a valid snapshot under $\leqslant_{push\ t'\ t_y\ L}$. Symmetrically for $p = y$.*
2. *If $p = x$, then $L' = push\ t'\ t_y\ L$ and $C' = C[t_x, t_y \mapsto \mathsf{green}]$ satisfy the state space invariants (Propositions 1-4). Symmetrically for $p = y$.*

We now have the following main correctness result.

▸ **Theorem 8** (Correctness of `write` and `scan`). *The implementations for* `scan` *and* `write` *in Figure 5 can be ascribed the specification given in Figure 4.*

The theorem is proved following a customary style of Floyd-Hoare symbolic evaluation using the inference rules of FCSL [15], a variant of Hoare logic. We have mechanized this proof in FCSL/Coq, including all the properties from the first two stages described above. The proof is subject to the following helper stability properties that we have not yet mechanized, but have proved manually, and their proofs are in Appendix B.

▸ **Lemma 9** (Stability of Assertions). *If by environment stepping L evolves to L', then:*

1. *$\sqsubseteq_L \subseteq \sqsubseteq_{L'}$ i.e. the logical ordering is monotone.*
2. *$H^{\sqsubseteq_L t} \subseteq H^{\sqsubseteq_{L'} t}$. Similarly for $\sqsubsetneq$.*
3. *If $\mathsf{chain}\ H^{\sqsubseteq_L t}$ then $\mathsf{chain}\ H^{\sqsubseteq_{L'} t}$.*
4. *If $\mathsf{eval}\ H^{\sqsubseteq_L t}$ then $\mathsf{eval}\ H^{\sqsubseteq_{L'} t}$.*

5 Discussion

We already argued in Section 3 that the Hoare triples from Figure 4 capture precisely what linearizability would be used for, namely that the operations of the snapshot object can be sequenced. The triples do so without the intermediary sequential specifications for `write`

and `scan`, and the attending simulation proofs. In this section, we argue that our approach via Hoare logic can still do a bit more, specifically in the cases of client side reasoning, and reasoning about parallel composition.

By relying on FCSL, we immediately inherit the separation logic mechanism for reasoning about programs with nested parallel composition (i.e., dynamic, well-bracketed, forking and joining). We refer the interested reader to the Appendix A for a brief background on FCSL, and the separation-logic style inference rule for parallel composition that requires that program state, real and auxiliary, satisfies the properties of a Partial Commutative Monoid, or PCM (and indeed, our histories form a PCM under the operation of disjoint union, with the empty history as a unit). This gave us a thread-local way to reason about programs, via local variables such as H_s and H_o, rather than relying on global abstractions, such as *thread-id*s, to specify the behavior of each thread. This is in contrast to linearizability, whose very definition requires identifying threads by thread-ids, thus focusing on programs that are a top-level parallel composition of a fixed number of sequential threads, but not providing convenient abstractions for reasoning about nested parallel compositions.

As a brief example of reasoning about parallel composition, it is possible to prove that the program $P_1 =$ `write` $(x, 1) \parallel$ `write` $(y, 2)$ satisfies the spec which says that two events are executed, in an unspecified order. Then we can reason about a larger program, say $P_2 = P_1;$ `write` $(x, 3)$ out of that spec, to prove that P_2 executes three events, with the write of 3 appearing last. Moreover, the *substitution principle* holds; that is, the proof remains valid, even if we replace P_1 by another program, say $P_1' =$ `write` $(x, 1);$ `write` $(y, 2)$ which satisfies a stronger spec (explicitly ordering the two events), but which *can be weakened* to the spec of P_1 by forgetting the ordering by means of strengthening the precondition and weakening the postcondition, as customary in Hoare logic. Thus, our client proofs *can ignore* the internal thread-structure of component programs.

The substitution principle can be pushed further. In particular, we can use a different snapshot algorithm, without modifying the proofs of P_1, P_2, P_1' or any other client, as long as `write` and `scan` satisfy the expected specs. We have confirmed this property on the toy example given in Figure 7 (we present only `scan`, as `write` is trivial) [19]. In this example, the snapshot structure consists of pointers x and y storing tuples (c_x, v_x) and (c_y, v_y), respectively. c_x and c_y are the payload of x and y, whereas v_x and v_y are version numbers, internal to the structure. Writes to x and y increment the version number. `Scan` reads x, y and x again, in succession, but avoids snapshot inconsistency by restarting if the two version numbers of x differ. The definition of $\sqsubseteq_L$ used to satisfy the specs equals the real time one, as no dynamic reordering is needed.

```
1  scan(): A × A  {
2    (cx, vx) ← read(x);
3    (cy, _)  ← read(y);
4    (_, tx)  ← read(x);
5    if vx == tx
6    then return (cx, cy)
7    else scan (); }
```

Figure 7 `Scan` using version numbers.

6 Related work

The proof method for establishing linearizability of concurrent objects based on the notion of *linearization points* has been presented in the original paper by Herlihy and Wing [11]. The first Hoare-style logic, employing this method for compositional proofs of linearizability was introduced in Vafeiadis' PhD thesis [22]. However, that logic, while being inspired by the combination of Rely-Guarantee reasoning and Concurrent Separation logic [24] with syntactic treatment of linearization points [23], did not have a soundness proof with respect

to any program semantics. Furthermore, the work [22] did not connect reasoning about linearizability to the verification of client programs that make use of linearizable objects in a concurrent environment (*cf.* Section 5).

Both these shortcomings were addressed in more recent works on program logics for linearizability [14], or, equivalently [5], *observational refinement* [21], which provided semantically sound methodologies for verifying linearizability/refinement of concurrent objects *as well as* for giving the objects Hoare-style specifications. However, in [14,21], these two properties are established separately, thus doubling the proving effort. What is more important, none of these approaches supports reasoning about non-regional LPs, as they require the linearization order of a procedure being verified to be determined (with respect to other LPs) by its end.

A series of more recent Hoare logics focused on specifying concurrent behavior *without* resorting to linearizability [3, 13, 19, 20]. This paper continues the same line of thinking, building on [19], which explored patterns of assigning Hoare-style specifications with self/other auxiliary histories to concurrent objects, including *higher-order* ones (e.g., flat combiner [10]) in FCSL [15], but has not considered future-dependent linearization points, as required by Jayanti's algorithm. The alternative logics [3, 13, 20] would have to make use of *prophecy variables*, in order to tackle future-dependent LP. Supporting prophecies would require revising these logics' metatheory, and, to the best of our knowledge, it has not been done yet.

The key novelty of the current work with respect to previous results on Hoare logics with histories [2, 6, 7, 9, 14, 19] is the idea of dynamically changing (i.e., *re-linking*) the auxiliary logical histories to enable effective specification and constructive reasoning about dynamic linearization points. Since re-linking is just a manipulation of otherwise standard auxiliary state, we were able to use FCSL *off the shelf*, with no extensions to its metatheory. Furthermore, we expect to be able to use FCSL's higher-order features to reason about higher-order (*i.e.*, parametrized by another data structure) snapshot-based concurrent constructions [18], which is our immediate future work.

Related to our result, O'Hearn *et al.* have shown how to employ history-based reasoning and Hoare-style logic to *non-constructively* prove the existence of linearization points for concurrent objects out of the data structure invariants [16]—the result is known as *the Hindsight Lemma*. The reasoning principle presented in this paper generalizes that idea, since the Hindsight Lemma is only applicable to "pure" concurrent methods (*e.g.*, concurrent set's `contains` [8]) that do not influence the position of other threads' linearization points. In contrast, our history re-linking handles such cases, as showcased by Jayanti's construction, where the linearization point of `write` depends on the (future) outcome of `scan`.

Finally, we note that our proof of Jayanti's algorithm seems very different from his original proof. Jayanti relies on so-called *forwarding principles*, as a key property of the proof. For example, Jayanti's First Forwarding Principle says (in paraphrase) that if `scan` misses the value of a concurrent write in line 10 of Figure 1, but the write finishes before the scanner goes through line 11 (the scan's linearization point), then the scanner will catch the value in the forwarding pointer in line 12. Instead of forwarding principles, we rely on colors to algorithmically construct the status of each write event as it progresses through time, and express our assertions using formal logic. For example, though we did not use the First Forwarding Principle, we nevertheless can express and prove a property similar to it: If $S_s = \mathsf{S}_{\mathsf{off}}\ t_{\mathsf{off}}$ and $S_p = \mathsf{True}$, then $\forall t \in H_p.\ t \leqslant E(t) < t_{\mathsf{off}} \implies C(t) = \mathsf{green}$.

References

1 Concurrent data structures linked in time: Coq mechanization files. Available from `http://software.imdea.org/~germand/relink`.

2 Christian J. Bell, Andrew W. Appel, and David Walker. Concurrent separation logic for pipelined parallelization. In *SAS*, volume 6337 of *LNCS*, pages 151–166. Springer, 2010.

3 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, volume 8586 of *LNCS*, pages 207–231, 2014.

4 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *POPL*, pages 233–246. ACM, 2015.

5 Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52), 2010.

6 Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, volume 6269 of *LNCS*, pages 388–402. Springer, 2010.

7 Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP*, volume 7792, pages 249–269. Springer, 2013.

8 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, volume 3974, pages 3–16. Springer, 2006.

9 Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. Modular verification of concurrency-aware linearizability. In *DISC*, volume 9363 of *LNCS*. Springer, 2015.

10 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364. ACM, 2010.

11 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

12 Prasad Jayanti. An optimal multi-writer snapshot algorithm. In *STOC*. ACM, 2005.

13 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650. ACM, 2015.

14 Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470. ACM, 2013.

15 Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, volume 8410 of *LNCS*, pages 290–310. Springer, 2014.

16 Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *PODC*, pages 85–94. ACM, 2010.

17 Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

18 Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *DISC*, volume 8205 of *LNCS*, pages 224–238. Springer, 2013.

19 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, volume 9032, pages 333–358. Springer, 2015.

20 Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, volume 8410 of *LNCS*. Springer, 2014.

21 Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. ACM, 2013.

22 Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

23 Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP*, pages 129–136. ACM, 2006.

24 Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, volume 4703, pages 256–271. Springer, 2007.

Optional appendices

A Some background on FCSL

A state of a resource in FCSL, such as that of snapshot data structure discussed in this paper, always consists of three distinct auxiliary variables that we name a_s, a_o and a_j. These stand for the abstract self state, other state, and shared (joint) state.

However, the user can pick the types of these variables based on the application. In this paper, we have chosen a_s and a_o to be histories, and have correspondingly named them H_s and H_o. On the other hand, a_j consists of all the other auxiliary components that we discussed, such as the variables H_j, E, C, S_x, S_y, W_x and W_y. These variables become merely projections out of a_j.

It is essential that a_s and a_o have a common type, which moreover, exhibits the algebraic structure of a *partial commutative monoid* (PCM). A PCM requires a partial binary operation $\bullet$ which is commutative and associative, and has a unit. PCMs are important, as they give a generic way to define the inference rule for parallel composition.

$$\frac{e_1 : \{P_1\}\ \{Q_1\} \quad e_2 : \{P_2\}\ \{Q_2\}}{e_1 \parallel e_2 : \{P_1 \circledast P_2\}\ \{[r.1/r]Q_1 \circledast [r.2/r]Q_2\}}$$

Here, $\circledast$ is defined over state predicates P_1 and P_2 as follows.

$$(P_1 \circledast P_2)(a_\mathsf{s}, a_\mathsf{j}, a_\mathsf{o}) \iff \exists x_1\ x_2.\, a_\mathsf{s} = x_1 \bullet x_2, P_1(x_1, a_\mathsf{j}, x_2 \bullet a_\mathsf{o}), P_2(x_2, a_\mathsf{j}, x_1 \bullet a_\mathsf{o})$$

The inference rule, and the definition of $\circledast$, formalize the intuition that when a parent thread forks e_1 and e_2, then e_1 is part of the environment for e_2 and vice-versa. This is so because the *self* component a_s of the parent thread is split into x_1 and x_2; x_1 and x_2 become the *self* parts of e_1, and e_2 respectively, but x_2 is also added to the *other* component a_o of e_1, and dually, x_1 is added to the *other* component of e_2.

In this paper, the PCM we chose is that of histories, which are a PCM under the operation of disjoint union $\cup$, with the empty history as the unit. More common in separation logic is to use heaps, which, similarly to histories, form PCM under disjoint union and empty. In FCSL, these can be combined into a Cartesian product PCM, to enable reasoning about both space and time in the same system.

The frame rule is a special case of the parallel composition rule, obtained when e_2 is taken to be the idle thread.

$$\frac{e : \{P\}\ \{Q\}}{e : \{P \circledast R\}\ \{Q \circledast R\}}\ R \text{ is stable}$$

For the purpose of this paper, the rule is important because it allows us to generalize the specifications of `write` and `scan` from Figure 4. In that figure, both procedures start with the precondition that $H_\mathsf{s} = \mathsf{empty}$. But what do we do if the procedures are invoked by another one which has already completed a number of writes, and thus its H_s is non-empty. By $\circledast$-ing with the frame predicate $R \mathrel{\widehat{=}} (H_\mathsf{s} = k)$, the frame rule allows us to generalize these specs into ones where the input history equals an arbitrary k:

$$\begin{aligned}
&\texttt{write}\ (p : \texttt{ptr}, n : \texttt{int}) : \{H_\mathsf{s} = k \wedge h \subseteq H_\mathsf{o} \wedge \omega \subseteq \sqsubseteq_L\} \\
&\qquad\qquad \{\exists t.\ H_\mathsf{s} = k \cup t \mapsto (p, n) \wedge h \subseteq H_\mathsf{o} \wedge \omega \subseteq \sqsubseteq_L \wedge k \cup h \subseteq H^{\sqsubseteq_L t}\} \\
&\texttt{scan} : \{H_\mathsf{s} = k \wedge\ h \subseteq H_\mathsf{o} \wedge\ \omega \subseteq \sqsubseteq_L\} \\
&\qquad \{H_\mathsf{s} = k \wedge\ h \subseteq H_\mathsf{o} \wedge\ \omega \subseteq \sqsubseteq_L \wedge\ \exists t.\ k \cup h \subseteq H^{\sqsubseteq_L t} \wedge\ \mathsf{chain}\ H^{\sqsubseteq_L t} \wedge\ r = \mathsf{eval}\ H^{\sqsubseteq_L t}\}
\end{aligned}$$

B Proving the Stability of Assertions

In this appendix we discuss the stability of the assertions we used in the specifications of `write` and `scan`. A proposition P over state is stable if it is invariant under environment interference. That is, if the state s evolves to s' by environment steps, then $P\ s$ implies $P\ s'$. By the nature of our setting [15], such environment steps are restricted to the auxiliary code used by the concurrent object; in this case, the auxiliary procedures from Figure 6. In particular, this appendix develops the stability proofs for the properties from Lemma 9.

We begin with a property of the logical order $<_L$. This order is not stable, as it can be changed, for example, by the helper procedure *push* $i\ j\ L$, used in *relink* (Figure 6). However, we consider the special cases of elements for which the ordering is preserved, as this will become important for the subsequent lemmas.

▸ **Lemma 10** (Push Mono). *Given elements a, b, i, j, all in L:*

1. *If $a <_L i$ then $a <_L b \implies a <_{push\ i\ j\ L} b$.*
2. *If $j <_L b$ then $a <_L b \implies a <_{push\ i\ j\ L} b$.*
3. *If $a \neq i$ then $a <_L b \implies a <_{push\ i\ j\ L} b$*

Proof. First, as L is used to order timestamps, it contains only distinct elements. Second, *push* is a surgery operation on L defined as follows. Let $L = L_{<_i} +\!\!+\ i +\!\!+\ L_{i..j} +\!\!+\ j +\!\!+\ L_{>_j}$, then

$$push\ i\ j\ L = L_{<_i} +\!\!+\ L_{i..j} +\!\!+\ j +\!\!+\ i +\!\!+ L_{>_j}$$

The changes in the order $<_L$ can therefore occur only in the segment $i +\!\!+\ L_{i..j} +\!\!+\ j$ of L. Thus, (1) and (2) are trivial, as changing the position of i does not affect $a <_L b$. For (3), observe that the only relations invalidated by *push* are those $i <_L b$, where $b \in L_{i..j} +\!\!+\ j$. Since $a \neq i$, it follows that $a <_{push\ i\ j\ L} b$. ◂

We now consider the statements of Lemma 9, proving each one in order, as a separate lemma. First is the lemma on monotonicity of the full history H_x wrt. environment steps.

▸ **Lemma 11** (Monotonic Histories). *If L steps to L' by environment steps, then $H \subseteq H'$ (i.e., histories grow monotonically).*

Proof. The proof follows straightforwardly from the definitions of the auxiliary code (Figure 6). In particular, the evolution of the internal state does not remove entries from histories, but either (i) adds new entries via *register*, or (ii) shuffles entries from H_j to H_s via *finalize*. In either case, the total history grows. We have mechanized this proof in Coq [1]. ◂

▸ **Lemma 12** ($\sqsubseteq$ is stable). *If L steps to L' by environment steps, then $\sqsubseteq_L \subseteq \sqsubseteq_{L'}$.*

Proof. This proof has also been mechanized, so we present here only the interesting case of *relink*, which is the only auxiliary procedure that can invalidate existing relations in $<_L$. We assume that $a \sqsubseteq_L b$, and prove that $a \sqsubseteq_{L'} b$. From the definition of $a \sqsubseteq_L b$, the cases of this proof are (i) $a = b$, (ii) $E(a) < b$, and (iii) $a <_L b \wedge C(a) = \mathsf{green}$. The first two follow immediately, by definition of $a \leqslant_{L'} b$.

In the proof of $a \sqsubseteq_{L'} b$ in the case of (iii), the interesting situation is when in the call to relink, the environment obtains *inspect* $t_x\ t_y\ L\ C = \mathsf{Yes}\ x\ t'$. By Lemma 6, $C(t') = \mathsf{yellow}$, and thus $a \neq t'$, since $C(a) = \mathsf{green}$. But then by Lemma 10.3, we immediately get $a <_{L push\ t'\ t_y\ L} b$. Since L' is precisely equal to *push* $t'\ t_y\ L$, we obtain the result. ◂

This gives us the first statement of Lemma 9. The second statement of Lemma 9 also follows easily from the previous two lemmas, by mere unfolding of the definition of $H^{\sqsubseteq_L t} = \{r \mapsto (p, v) \in H \mid r \sqsubseteq_L t\}$.

▸ **Lemma 13** ($H^{\sqsubseteq t}$ is stable)**.** *If L steps to L' by environment interference then forall t, $H^{\sqsubseteq_L t} \subseteq H^{\sqsubseteq_{L'} t}$.*

Now, to consider the stability of chain and eval, which are the third and fourth statements of Lemma 9, we first introduce some intermediate concepts and results.

▸ **Definition 14** ($H^{\leqslant_L t}$)**.** The downward restriction on $<_L$ is defined as $H^{\leqslant_L t} = \{r \mapsto (p, v) \in H | r \leqslant_L t\}$, analogously to $H^{\sqsubseteq_L t}$.

▸ **Lemma 15.** $\mathsf{dom}(\_ \leqslant_L t) = \mathsf{dom}(H^{\leqslant_L t})$.

Proof. The statement of this lemma is a direct consequence of one of the invariants we omitted in Section 3, which states that $\mathsf{dom}(L) = \mathsf{dom}(H)$. ◂

We now prove that, if $H^{\leqslant_L t}$ is a all green, which we denote as $C(H^{\leqslant_L t}) = \mathsf{green}$, then $H^{\leqslant_L t}$ is invariant under environment interference.

▸ **Lemma 16** (all green stable)**.** *If L steps to L' by environment interference, then $C(H^{\leqslant_L t}) = \mathsf{green} \implies H^{\leqslant_L t} = H^{\leqslant_{L'} t}$.*

Proof. The proof is by induction on environment steps. The inductive case is trivial, and of the base cases, the only interesting one is stepping by *register*. Other auxiliary procedures do not extend L, and do not affect the position in L of green timestamps.

By definition, *register* adds a fresh timestamp $t' = \mathsf{fresh}\ L$ to the tail of L, to obtain L'. Hence $t <_{L'} t'$, and t' does not belong to $\mathsf{dom}(H^{\leqslant_{L'} t})$. By Lemma 11, we know that $\mathsf{dom}(H^{\leqslant_L t}) \subseteq \mathsf{dom}(H^{\leqslant_{L'} t})$. The difference between the two sets can be at most t', but since t' does not belong to $\mathsf{dom}(H^{\leqslant_{L'} t})$, we obtain the desired equality. ◂

▸ **Definition 17** (Complete)**.** Downward restriction $H^{\sqsubseteq_L t}$ is complete iff $H^{\sqsubseteq_L t} = H^{\leqslant_L t}$.

▸ **Lemma 18** (Complete Greens)**.** *If $C(H^{\leqslant_L t}) = \mathsf{green}$ then $H^{\sqsubseteq_L t}$ is* complete*,* i.e. *$H^{\sqsubseteq_L t} = H^{\leqslant_L t}$.*

Proof. From the definition of $H^{\leqslant_L t}$, and of the stable order $\sqsubseteq_L$ (Definition 5), we infer that $\mathsf{dom}(H^{\sqsubseteq_L t}) \subseteq \mathsf{dom}(H^{\leqslant_L t})$. Then, from $C(H^{\leqslant_L t}) = \mathsf{green}$ we know that for all $a \in H^{\leqslant_L t}$, $C(a) = \mathsf{green} \wedge a \leqslant_L t$, and therefore $a \sqsubseteq_L t$. It follows that $\mathsf{dom}(H^{\leqslant_L t}) \subseteq \mathsf{dom}(H^{\sqsubseteq_L t})$, and thus we conclude, since $\mathsf{dom}(H^{\leqslant_L t}) = \mathsf{dom}(H^{\sqsubseteq_L t})$. ◂

▸ **Lemma 19** (chain $H^{\sqsubseteq_L t}$ is stable)**.** *If L steps to L' by environment interference then if* chain *$H^{\sqsubseteq_L t}$ and $C(H^{\leqslant_L t}) = \mathsf{green}$ then* chain *$H^{\sqsubseteq_{L'} t}$.*

Proof. The proof follows up from Lemmas 18 and 16 above: by Lemma 18 we infer that $H^{\sqsubseteq_L t} = H^{\leqslant_L t}$. Then by Lemma 16, $H^{\leqslant_{L'} t} = H^{\leqslant_L t}$ and, moreover, $C(H^{\leqslant_{L'} t}) = \mathsf{green}$. From the latter fact we apply Lemma 18 again to get $H^{\sqsubseteq_{L'} t} = H^{\leqslant_{L'} t}$. Then, by transitivity $H^{\sqsubseteq_L t} = H^{\sqsubseteq_{L'} t}$ and we conclude. ◂

▸ **Lemma 20** (eval $H^{\sqsubseteq_L t}$ is stable)**.** *If L steps to L' by environment interference then if* eval *$H^{\sqsubseteq_L t}$ and $C(H^{\leqslant_L t}) = \mathsf{green}$ then* chain *$H^{\sqsubseteq_{L'} t}$.*

Proof. The proof is analogous to that of Lemma 19 above. ◂