Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee

IMDEA Software Institute, Spain
{ilya.sergey, aleks.nanevski, anindya.banerjee}@imdea.org

Abstract. We present a lightweight approach to Hoare-style specifications for fine-grained concurrency, based on a notion of *time-stamped histories* that abstractly capture atomic changes in the program state. Our key observation is that histories form a *partial commutative monoid*, a structure fundamental for representation of concurrent resources. This insight provides us with a unifying mechanism that allows us to treat histories just like heaps in separation logic. For example, both are subject to the same assertion logic and inference rules (*e.g.*, the frame rule). Moreover, the notion of ownership transfer, which usually applies to heaps, has an equivalent in histories. It can be used to formally represent helping—an important design pattern for concurrent algorithms whereby one thread can execute code on behalf of another. Specifications in terms of histories naturally abstract away the internal interference, so that sophisticated fine-grained algorithms can be given the same specifications as their simplified coarse-grained counterparts, making them equally convenient for client-side reasoning. We illustrate our approach on a number of examples and validate all of them in Coq.

1 Introduction

For sequential programs and data structures, Hoare-style specifications (or specs) in the form of pre- and postconditions are a declarative way to express a program's behavior. For example, an abstract specification of stack operations can be given as follows:

$$\begin{array}{l} \{\ s \mapsto xs\ \}\ \texttt{push}(x)\ \{\ s \mapsto x :: xs\ \} \\ \{\ s \mapsto xs\ \}\ \ \texttt{pop}()\ \ \left\{ \begin{array}{l} \mathsf{res} = \mathsf{None} \wedge xs = \mathsf{nil} \wedge s \mapsto \mathsf{nil} \vee \\ \mathsf{res} = \mathsf{Some}\ x \wedge \exists xs',\ xs = x :: xs' \wedge s \mapsto xs' \end{array} \right\} \end{array} \tag{1}$$

where s is an "abstract pointer" to the data structure's logical contents, and the logical variable xs is universally quantified over the spec. The result `res` of `pop` is either `Some` x, if x was on the top of the stack, or `None` if the stack was empty. The spec (1) is usually accepted as canonical for stacks: it hides the details of method implementation, but exposes what is important about the method behavior, so that a verification of a stack *client* does not need to explore the implementations of `push` and `pop`.

The situation is much more complicated in the case of concurrent data structures. In the concurrent setting, the spec (1) is of little use for implementations with server-side locking, as the interference of the threads executing concurrently may invalidate the assertions about the stack. For example, a call to `pop` may encounter an empty stack, and decide to return `None`, but by the time it returns, the stack may be filled by the other threads, thus invalidating the postcondition of `pop` in (1). To soundly reason about concurrent data structures, one has to devise specs that are *stable* (*i.e.*, invariant

under interference), but this may require trade-offs with respect to the specifications' expressivity and precision for the client's needs.

Reasoning about concurrent data structures is further complicated by the fact that their implementations are often *fine-grained*. Striving for better performance, they avoid explicit locking, and implement sophisticated synchronization patterns that deliberately rely on interference. For reasoning purposes, however, it is desirable that the clients can perceive such fine-grained implementations as if they were *coarse-grained*; that is, as if the effects of their methods take place *atomically*, at singular points in time. The standard correctness criteria of *linearizability* [16] establishes that a fine-grained data structure implementation *contextually refines* a coarse-grained one [10]. One can make use of a refined, fine-grained, implementation for efficiency in programming, but then soundly replace it with a more abstract coarse-grained implementation to simplify the reasoning about clients.

Semantically, one program linearizes to another if the *histories* of the first program (*i.e.*, the sequence of actions it executed) can be transformed, in a suitable sense, into the histories of the second. Thus, histories are an essential ingredient in specifying fine-grained concurrent data structures. However, while a number of logical methods exist for establishing the linearizability relation between two programs, for a class of data structures [7, 20, 24, 33], in general, it is a non-trivial property to prove and use. First, in a setting that employs Hoare-style reasoning, showing that a fine-grained structure refines a coarse-grained one is not an end in itself. One still needs to ascribe a stable spec to the coarse-grained version [20, 31]. Second, the standard notion of linearizability does not directly account for modern programming features, such as ownership transfer of state between threads, pointer aliasing, and higher-order procedures. Theoretical extensions required to support these features are a subject of active ongoing research [4, 13]. Finally, being a relation on *two* programs, deriving linearizability by means of logical inference inherently requires a *relational program logic* [20, 31], even though the spec one is ultimately interested in may be expressed using a Hoare triple that operates over a *single* program.

In this paper, we propose a novel method to specify and verify fine-grained programs by directly reasoning about histories in the specs of an elementary Hoare logic. We propose using *timestamped* histories, which carry information about the atomic changes in the abstract state of the program, indexed by discrete timestamps, and tracking the history of a program as a form of auxiliary state.

While using histories in Hoare-style specs is a simple and natural idea, and has been used before [1, 11, 12], in our paper it comes with two additional novel observations.

First, timestamped histories are technically very similar to heaps, as both satisfy the algebraic properties of a *partial commutative monoid* (PCM). A PCM is a set $\mathbb{U}$ with an associative and commutative *join* operation $\bullet$ and unit element $\mathbb{1}$. Both heaps and histories (considered as sets of actions with distinct timestamps, correspondingly) form a PCM with disjoint union and empty heap/history as the unit. Also, a singleton history $t \mapsto a$ is similar to the singleton heap $x \mapsto v$ containing only the pointer x with value v. We emphasize the connection by using the same notation for both. The common PCM structure makes it possible to reuse for histories the ideas and results developed for heaps in the work on separation logic [3]. In particular, in this paper, we make both

heaps and histories subject to the same assertion logic, the same rules of inference (*e.g.*, the frame rule), and thus the same style of *local reasoning*. Moreover, concepts such as ownership transfer, well-studied for heaps, apply to histories as well. For example, in Section 5, we use ownership transfer on histories to formalize the important design pattern of *helping* [14], whereby a concurrent thread may execute a task on behalf of other threads. That helping corresponds to a kind of ownership transfer (though not on histories, but on auxiliary commands) has been noticed before [20, 32]. However, commands do not form a PCM, while histories do—a fact that makes our development simple and uniform.

Second, we argue that precise history-based specs have to differentiate between the actions that have been performed by the specified thread, from the actions that have been performed by the thread's concurrent environment. Thus, our specs will range over *two* different history-typed variables, capturing the timestamped actions of the specified thread (*self*) and its environment (*other*), respectively. This split between self and other will provide us with a novel and very direct way of relating the functional behavior of a program to the interference of its environment, leading to specs that have a similar canonical "feel" in the concurrent setting, as the specs (1) have in the sequential one.

The self/other dichotomy required of histories is a special case of the more general specification pattern of *subjectivity*, observed in the recent related work on Subjective and Fine-grained Concurrent Separation Logic (FCSL) [19, 22]. That work generalized Concurrent Separation Logic (CSL) [23] to apply not only to heaps, but to any abstract notion of state (real or auxiliary) satisfying the PCM properties. We thus reuse FCSL [22] *off-the-shelf*, and instantiate it with histories, *without any additions to the logic or its meta-theory*. Surprisingly, the FCSL style of auxiliary state is sufficient to enable expressive history-based proofs of realistic fine-grained algorithms, including those with helping.

Specifications with histories also allow the clients of a fine-grained data structure to pretend, for the sake of simplifying their own reasoning, that they are using a coarse-grained version of the data structure. In this sense, we consider a program *logically* atomic (irrespective of the physical granularity of its implementation), if its specification is a singleton history $t \mapsto a$, containing only an abstract action a time-stamped with t. This spec provides an abstraction that the effect a of the program takes place at a singular point in time t, as if the program were coarse-grained, thus providing a uniform way to reason about coarse- and fine-grained programs.[1]

We show how a number of well-known algorithms can be proved logically atomic, and illustrate how the specs with histories facilitate client-side reasoning. We consider an atomic pair snapshot data structure [20, 26] (Section 2), a Treiber stack [30] along with its clients (Section 4), and Hendler *et al.*'s flat combining algorithm [14], a non-trivial example employing first-class functions and helping (Section 5). All our proofs, including the theory of histories, have been checked mechanically in Coq, and the sources are available online [27].

[1] An orthogonal aspect of granularity abstraction is the ability of a logical framework to express synchronized changes to auxiliary state that is spread across several shared data structures. We do not consider such abstraction in this paper, but elaborate in Section 6 on how to extend FCSL to support it, as well as on related approaches [5, 17, 28, 29].

2 Overview: specifying snapshots with histories

In this section, we illustrate history-based specifications by applying them to the fine-grained *atomic pair snapshot* data structure [20, 26]. This data structure contains a pair of pointers, x and y, pointing to tuples (c_x, v_x) and (c_y, v_y), respectively. The components c_x and c_y of type A represent the accessible contents of x and y, that may be read and updated by the client. The components v_x and v_y are nats, encoding "version numbers" for x and y. They are internal to the structure and not directly accessible by the client.

The data structure interface exports three methods: `readPair`, `writeX`, and `writeY`. `readPair` is the main method, and the focus of the section. It returns the *snapshot* of the data structure, *i.e.*, the accessible contents of x and y as they appear together at the moment of the call. However, while x and y are being read by `readPair`, other threads may change them, by invoking `writeX` or `writeY`. Thus, a naïve implementation of `readPair` which first reads x, then y, and returns the pair (c_x, c_y) does not guarantee that c_x and c_y ever appeared together in the structure. One may have `readPair` first lock x and y to ensure exclusive access, but here we consider a fine-grained implementation which relies on the version numbers to ensure that `readPair` returns a valid snapshot.

Fig. 1 Atomic pair snapshot

```
1 readPair(): A × A  {
2   (cx, vx) <- readX();
3   (cy, _)  <- readY();
4   (_, tx)  <- readX();
5   if vx == tx
6   then return (cx, cy);
7   else readPair();}
```

The idea is that `writeX(cx)` (and symmetrically, `writeY(cy)`), changes the logical contents of x to `cx`, while incrementing the internal version number, *simultaneously*. Since the operation involves changes to the contents of a single pointer, in this paper we assume that it can be performed atomically (*e.g.*, by some kind of read-modify-write operation [15, §5.6]). We also assume atomic operations `readX` and `readY` for reading from x and y respectively. Then the implementation of `readPair` (Figure 1) reads from x and y in succession, but makes a check (line 5) to compare the version numbers for x obtained before and after the read of y. If x's version has changed, the procedure restarts.

We want to specify and prove that such an implementation of `readPair` is correct; that is, if it returns a pair (c_x, c_y), then c_x and c_y occurred simultaneously in the structure. To do so, we use histories as auxiliary state of every method of the structure. Histories, ranged over by τ, are finite maps from the natural numbers to pairs of elements of some type S; *i.e.*, $\mathsf{hist}\ S \mathrel{\widehat{=}} \mathsf{nat} \rightharpoonup S \times S$.[2] The natural numbers represent the moments in time, and the pairs represent the change of state. Thus, a singleton history $t \mapsto (s_1, s_2)$ encodes an atomic change from abstract state s_1 to abstract state s_2 at the time moment t. We will only consider *continuous* histories, for which $t \mapsto (s_1, s_2)$ and $t+1 \mapsto (s_3, s_4)$ implies $s_2 = s_3$. We use the following abbreviations to work with histories:

$$\begin{array}{lcl} \tau[t] & \mathrel{\widehat{=}} & s, \text{ such that } \exists s',\ \tau(t) = (s', s) \\ \tau \le t & \mathrel{\widehat{=}} & \forall t' \in \mathsf{dom}(\tau), t' \le t \\ \tau \sqsubseteq \tau' & \mathrel{\widehat{=}} & \tau \text{ is a subset of } \tau' \end{array} \tag{2}$$

Similarly to heaps, histories form a PCM under the operation $\cup$ of disjoint union, with the `empty` history as the unit. The type S can be chosen arbitrarily, depending on the

[2] Other sets for time-stamping are possible besides nat, as will be mentioned in Section 6.

application, to capture whichever logical aspects of the actual physical state are of interest. For the snapshot structure, we take $S = A \times A \times \mathsf{nat}$. That is, the entries in the histories for pair snapshot will be of the form

$$t \mapsto (\langle c_x, c_y, v_x\rangle, \langle c'_x, c'_y, v'_x\rangle). \tag{3}$$

The entry encodes that at time moment t, the contents of x, y, and the version of x have changed from (c_x, c_y, v_x) to (c'_x, c'_y, v'_x). We ignore v_y, as it does not factor in the implementation of `readPair` (even though it is present for the sake of symmetry).

All the threads working over the pair snapshot structure respect a protocol on histories consisting of the following three properties. We explain in Section 3 how these are formally specified and enforced, but for now simply assume them. They will be important in the proof outline for `readPair`.

(*i*) Whenever a thread modifies x or y (*e.g.*, by calling `writeX` or `writeY`), its history gets augmented by an entry such as (3), where the timestamp t is chosen afresh. Thus, histories only grow, and only by adding valid snapshots (*i.e.*, snapshots corresponding to values of x and y, *simultaneously* present in the data structure).

(*ii*) Whenever the contents of x is changed in a history, its version number changes too. In contrapositive form, if $\tau[t_1] = \langle c_1, -, v\rangle$ and $\tau[t_2] = \langle c_2, -, v\rangle$, then $c_1 = c_2$.

(*iii*) Version numbers in a history grow monotonically. That is, if $\tau[t_1] = \langle -, -, v_1\rangle$ and $\tau[t_2] = \langle -, -, v_2\rangle$ and $t_1 \le t_2$, then $v_1 \le v_2$.

Specification. We now describe an FCSL spec for `readPair` and explain how it captures that its result is a valid snapshot of x and y.

$$\begin{array}{c} \{\exists \tau_O.\, \ell \overset{s}{\mapsto} \mathsf{empty} \wedge \ell \overset{o}{\mapsto} \tau_O \wedge \tau \sqsubseteq \tau_O\} \\ \texttt{readPair()} \\ \left\{\exists \tau_O\ t.\, \ell \overset{s}{\mapsto} \mathsf{empty} \wedge \ell \overset{o}{\mapsto} \tau_O \wedge \tau \sqsubseteq \tau_O \wedge \tau \le t \wedge \tau_O[t] = \langle \mathsf{res.1}, \mathsf{res.2}, -\rangle\right\} \end{array} \tag{4}$$

First, note the label ℓ, which serves as an "abstract pointer" that differentiates the instance of the pair snapshot structure from any other structure that may exist in the program. In particular, ℓ identifies the histories of concern to `readPair`. Each thread keeps track of two such histories: the self-history, describing the operations that the thread itself has executed, and the other-history for the operations executed by all the other threads combined. They are captured by the assertions $\ell \overset{s}{\mapsto} \tau$ and $\ell \overset{o}{\mapsto} \tau$, respectively.

Thus, the precondition in (4) requires that `readPair` starts with the empty self-history, *i.e.*, the calling thread has not performed any updates to x or y. We show in Section 3 that the frame rule can be used to relax the requirement, so that `readPair` can be invoked by threads with an arbitrary self history. The precondition allows an arbitrary initial other-history τ_O. As τ_O is bound locally in the precondition, we use the logical variable τ and a conjunct $\tau \sqsubseteq \tau_O$ to propagate the information about τ_O into the postcondition. Because τ and τ_O are related by inclusion, the precondition is stable under growth of τ_O due to interfering threads, according to (*i*).

The postcondition states that `readPair` does not perform any changes to x and y; it is a *pure* method, thus its self-history remains empty. The main novelty of the specification is that the postcondition directly relates the result of `readPair` to the interference of the environment, *i.e.*, to the value of τ_O. This is in contrast to the extant logics, which do not keep track of the *other* component, and hence cannot specify `readPair` as directly.

In particular, the postcondition says that $\tau_O[t] = \langle \mathsf{res}.1, \mathsf{res}.2, - \rangle$, *i.e.*, that the components of the returned pair res appear in the environment history. Since according to the property (*i*) above, the histories only store valid snapshots, the resulting pair must be a valid snapshot too. In other words, `readPair` behaves as if it read x and y atomically, at time t. Moreover, $\tau \le t$, *i.e.*, the read occurred after `readPair` was invoked.

The specification pattern whereby a logical variable τ names the initial history of the environment is very common, so we streamline it by introducing the following notation.

$$\ell \hookrightarrow (\tau_S, \tau_O, \tau) \;\widehat{=}\; \ell \overset{S}{\mapsto} \tau_S \wedge \ell \overset{O}{\mapsto} \tau_O \wedge \tau \sqsubseteq \tau_S \cup \tau_O \tag{5}$$

Proof outline. Figure 2 contains the proof outline for `readPair`, which we discuss next. The relation $\tau \sqsubseteq \tau_O$ is folded into the definition of $\ell \hookrightarrow (\mathsf{empty}, \tau_O, \tau)$. Lines 1 and 3 abbreviate the precondition in (4). The `readX` method has the following spec:

$$\left\{\ell \hookrightarrow (\mathsf{empty}, -, \tau)\right\}\ \texttt{readX()}\ \left\{\exists \tau_O\, t.\, \ell \hookrightarrow (\mathsf{empty}, \tau_O, \tau) \wedge \tau \le t \wedge \tau_O[t] = \langle \mathsf{res}.1, -, \mathsf{res}.2\rangle\right\}$$

Since the "initial" other-history is bounded by τ in the precondition, and the "final" τ_O may only grow, we require $\tau \le t$ in the postcondition to ensure that we will not get a value from the history, which has "expired" *before* the call to `readX`. Thus in line 5 of the proof, we infer the existence of the history τ_1 and time stamp $t_1 \ge \tau$, such that the cx and vx appear in τ_1 at the time t_1. Similarly, `readY` has the spec:

$$\left\{\ell \hookrightarrow (\mathsf{empty}, -, \tau)\right\}\ \texttt{readY()}\ \left\{\exists \tau_O\, t.\, \ell \hookrightarrow (\mathsf{empty}, \tau_O, \tau) \wedge \tau \le t \wedge \tau_O[t] = \langle -, \mathsf{res}.1, -\rangle\right\}$$

To obtain line 7, instantiate τ with τ_1 in the spec of `readY`. This derives the existence of τ_2, t_2, c and v, such that $\ell \hookrightarrow (\mathsf{empty}, \tau_2, \tau_1)$, $\tau_1 \le t_2$, and $\tau_2[t_2] = \langle c, \mathsf{cy}, v\rangle$. Because $t_1 \in \mathsf{dom}(\tau_1)$, it must be that $t_1 \le t_2$. Moreover, because $\tau \sqsubseteq \tau_1 \sqsubseteq \tau_2$, we further obtain $\ell \hookrightarrow (\mathsf{empty}, \tau_2, \tau)$, and $\tau \le t_2$, and lifting from line 5, $\tau_2[t_1] = \langle \mathsf{cx}, -, \mathsf{vx}\rangle$. Because t_1, t_2 appear in the same history τ_2, with versions vx and v, respectively, by property (*iii*), $\mathsf{vx} \le v$. Similarly, instantiating τ in the spec of `readX` with τ_2, and invoking (*iii*), derives line 9 of the proof outline, and in particular $\mathsf{vx} \le v \le \mathsf{tx}$.

Fig. 2 Proof outline for `readPair`.

```
1  { ℓ ↪ (empty, −, τ) }
2  readPair(): A × A {
3  { ℓ ↪ (empty, −, τ) }
4    (cx, vx) <- readX();
5  { ℓ ↪ (empty, τ1, τ) ∧ τ ≤ t1 ∧ τ1[t1] = ⟨cx, −, vx⟩ }
6    (cy, _) <- readY();
7  { ℓ ↪ (empty, τ2, τ) ∧ τ ≤ t1 ≤ t2 ∧ vx ≤ v ∧
   { τ2[t1] = ⟨cx, −, vx⟩ ∧ τ2[t2] = ⟨c, cy, v⟩
8    (_, tx) <- readX();
9  { ℓ ↪ (empty, τ3, τ) ∧ τ ≤ t1 ≤ t2 ≤ t3 ∧ vx ≤ v ≤ tx ∧
   { τ3[t1] = ⟨cx, −, vx⟩ ∧ τ3[t2] = ⟨c, cy, v⟩ ∧ τ3[t3] = ⟨−, −, tx⟩ }
10   if vx == tx
11   { ℓ ↪ (empty, τ3, τ) ∧ τ ≤ t2 ∧ cx = c ∧ τ3[t2] = ⟨cx, cy, v⟩ }
12     then return (cx, cy);
13   { ∃τO t. ℓ ↪ (empty, τO, τ) ∧ τ ≤ t ∧ τO[t] = ⟨res.1, res.2, −⟩ }
14   else readPair();}
15 { ∃τO t. ℓ ↪ (empty, τO, τ) ∧ τ ≤ t ∧ τO[t] = ⟨res.1, res.2, −⟩ }
```

From this property, if $\mathsf{vx} = \mathsf{tx}$ in the conditional on line 10, it must be that $\mathsf{vx} = v$, and thus by (*ii*), $\mathsf{cx} = c$. Substituting c by cx in line 9 gives us $\tau_3[t_2] = \langle \mathsf{cx}, \mathsf{cy}, v\rangle$, which, after (cx, cy) are returned in res, obtains the postcondition of `readPair`. Otherwise, if $\mathsf{vx} \neq \mathsf{tx}$ in the conditional 10, we perform the recursive call to `readPair`. The precondition for the call is $\ell \hookrightarrow (\mathsf{empty}, -, \tau)$, which is clearly met in line 9, so the postcondition immediately follows.

Monolithic histories. We compare the spec (4) with an alternative spec where the history is not split into self/other portions, but is kept monolithically as a *joint* (or shared) state. We use the predicate $\ell \overset{j}{\mapsto} \tau$ to specify such state:

$$\{\exists \tau_O.\, \ell \overset{j}{\mapsto} \tau_O \wedge \tau \sqsubseteq \tau_O\}\ \texttt{readPair()}\ \left\{ \begin{array}{c} \exists \tau_O\, t.\, \ell \overset{j}{\mapsto} \tau_O \wedge \tau \sqsubseteq \tau_O\ \wedge \\ \tau \leq t \wedge \tau_O[t] = \langle \mathsf{res.1}, \mathsf{res.2}, - \rangle \end{array} \right\} \tag{6}$$

Note that the spec (6) imposes no restrictions on the growth of τ_O (unlike (4) which keeps the self history `empty`). Thus, (6) is weaker than (4), as it allows more behaviors. In particular, it can be ascribed to any program which, in addition to calling `readPair`, also modifies x and y. This substantiates our claim from Section 1 that the self/other dichotomy is required to prevent history-based specs from losing precision. We provide further evidence for this claim in Section 4, where we show that subjective specs for *stacks* generalize the sequential canonical ones (1). The latter can be derived from the former by restricting τ_O to be the empty history. Such a restriction is not possible if the history is kept monolithic.

3 Background: a review of FCSL

In this section we review the relevant aspects of the previous work on Fine-grained Concurrent Separation Logic (FCSL) [22]. We explain FCSL by showing how it can be specialized to our novel contribution of specifying concurrent objects by means of histories. FCSL has been previously implemented as a shallow embedding in Coq; thus our assertions will freely use Coq's higher-order logic and datatype definition mechanism.

FCSL is a Hoare logic, generalizing CSL, hence its assertions are predicates on state. But unlike in CSL where state is a heap, in FCSL state may consist of a number of labeled components (sometimes dubbed as "regions" or "islands" in the literature [6, 28, 31]), each of which may represent state by a different type. If the type used by some label is non-heap, then that label encodes auxiliary state, used for logical specification, but erased at run time. For example, histories are an auxiliary state identified by the label ℓ in the atomic snapshot example. If we had a program which used two different atomic snapshot structures, we may label these by ℓ_1 and ℓ_2, *etc*.

3.1 Subjectivity

The state recorded in labels is further divided across another orthogonal axis – ownership. Each label identifies three different chunks of state: self, joint and other portion. The self portion is private to the specified thread, and cannot be accessed by the other threads. Dually, other is private to the environment threads, and cannot be accessed by the one being specified. Finally, the joint section is shared and can be accessed by everyone. The self and other portions of any given label have to belong to a common PCM (the joint portion, though, is not required to be a PCM element, as it is not a subject of a split between threads, as we will see below), and are often combined together by means of the $\bullet$ operation of that PCM. Of course, different labels can use different PCMs, and, therefore, the points-to assertions are implicitly parametrized with a PCM type.

The FCSL assertions reflect the division across these axes. We have already illustrated the assertions $\ell \overset{s}{\mapsto} v$, $\ell \overset{j}{\mapsto} v$ and $\ell \overset{o}{\mapsto} v$, which identify the self/joint/other component stored in the label ℓ of the state. These three basic assertions, constraining only one state component correspondingly (and leaving the two other unconstrained), can

be, therefore, combined by the usual propositional connectives, such as $\wedge$ and $\vee$, as we have already shown in Section 2. FCSL further provides two connectives that generalize the *separating conjunction* $*$ from separation logic, along the two axes of state splitting. We next illustrate the *subjective separating conjunction* $\circledast$, and defer the discussion of the *resource separating conjunction* $*$ until additional technical material has been introduced. The formal definitions of all the connectives can be found in [27, Appendix A]. The subjective conjunction $\circledast$ models the division of state between concurrent threads upon forking and joining. In particular, the parallel composition rule of FCSL is:

$$\frac{\{p_1\}\, c_1\, \{q_1\}@\mathcal{U} \qquad \{p_2\}\, c_2\, \{q_2\}@\mathcal{U}}{\{p_1 \circledast p_2\}\, c_1 \parallel c_2\, \{q_1 \circledast q_2\}@\mathcal{U}} \tag{7}$$

Ignoring $\mathcal{U}$ and the result types of c_1 and c_2 for now, we describe how $\circledast$ works. In this rule, it splits the pre-state of $c_1 \parallel c_2$ into two parts, satisfying p_1 and p_2 respectively. The parts contain the same labels, and equal joint portions, but the self and other portions are recombined to match the thread-relative views of c_1 and c_2. Concretely, in the case of one label ℓ, with a PCM $\mathbb{U}$ and values $a, b, c \in \mathbb{U}$, we have the following implication.

$$\ell \overset{s}{\mapsto} a \bullet b \wedge \ell \overset{o}{\mapsto} c \implies (\ell \overset{s}{\mapsto} a \wedge \ell \overset{o}{\mapsto} b \bullet c) \circledast (\ell \overset{s}{\mapsto} b \wedge \ell \overset{o}{\mapsto} a \bullet c) \tag{8}$$

Thus, if before the fork, the self-state of the parent thread contained $a \bullet b$, and the other-state contained c, then after the fork, the children will have self-states a and b, and the other-states $b \bullet c$ and $a \bullet c$, respectively. In the opposite direction:

$$\begin{array}{l}(\ell \overset{s}{\mapsto} a \wedge \ell \overset{o}{\mapsto} c_1) \circledast (\ell \overset{s}{\mapsto} b \wedge \ell \overset{o}{\mapsto} c_2) \implies \\ \quad \exists c.\, c_1 = b \bullet c \wedge c_2 = a \bullet c \wedge \ell \overset{s}{\mapsto} a \bullet b \wedge \ell \overset{o}{\mapsto} c\end{array} \tag{9}$$

That is, if the state can be subjectively split between two child threads so that their other-views are c_1, c_2 (with self-views a, b), then there exists a common c—the other-view of the parent thread—such that $c_1 = b \bullet c$ and $c_2 = a \bullet c$. In this sense, the rule for parallel composition models the important effect that upon a split, c_1 becomes an environment thread for c_2, and vice-versa.

There are a few further equations that illustrate the interaction between the different assertions. First, every label contains all three of the self/joint/other components. Thus:

$$\ell \overset{s}{\mapsto} a \iff \ell \overset{s}{\mapsto} a \wedge \ell \overset{j}{\mapsto} - \wedge \ell \overset{o}{\mapsto} - \tag{10}$$

and similarly for $\ell \overset{j}{\mapsto} a$ and $\ell \overset{o}{\mapsto} a$. Also:

$$\ell \overset{s}{\mapsto} a \bullet b \iff \ell \overset{s}{\mapsto} a \circledast \ell \overset{s}{\mapsto} b \tag{11}$$

which is provable from (8), (9) and (10).

FCSL also provides a *frame rule*, obtained as a special case of parallel composition when c_2 is the idle thread, and $p_2 = q_2 = r$ is a stable predicate, as usual in fine-grained logics [6, 8, 33].

$$\frac{\{p\}\, c\, \{q\}@\mathcal{U}}{\{p \circledast r\}\, c\, \{q \circledast r\}@\mathcal{U}} \quad r \text{ stable under } \mathcal{U} \tag{12}$$

We illustrate the frame rule by deriving from the `readPair` spec (4) a relaxed spec which allows `readPair` to apply when the calling thread has non-trivial self history τ_S:

$$\{\ \ell \hookrightarrow (\tau_S, -, \tau)\ \}\ \texttt{readPair()}\ \left\{ \begin{array}{l} \exists \tau_O\ t.\ \ell \hookrightarrow (\tau_S, \tau_O, \tau) \wedge \tau \leq t \wedge \\ \quad (\tau_S \cup \tau_O)[t] = \langle \mathsf{res.1}, \mathsf{res.2}, - \rangle \end{array} \right\} \tag{13}$$

Note that (13), when compared to (4), changes the self component from `empty` to τ_S, but also $\tau_O[t]$ changes into $(\tau_S \cup \tau_O)[t]$. The latter accounts for the possibility that the returned snapshot may have been recorded in τ_S as a consequence of the thread itself changing x or y, immediately before invoking `readPair`.

The spec (13) derives from (4) by framing with the predicate $r = \ell \overset{S}{\mapsto} \tau_S$. r is trivially stable, as it describes self-state, which is inaccessible to the interfering threads. We only show how to weaken the framed postcondition of (4) to the postcondition in (13); the preconditions can be strengthened similarly. Abbreviating $\tau \sqsubseteq \tau_O \wedge \tau \leq t \wedge \tau_O[t] = \langle \mathsf{res.1}, \mathsf{res.2}, - \rangle$ by $P(\tau_O)$, which is a label-free (*i.e.*, pure) assertion, and thus commutes with $\circledast$, we get:

$$\begin{array}{l} (\ell \overset{S}{\mapsto} \mathsf{empty} \wedge \ell \overset{O}{\mapsto} \tau_O \wedge P(\tau_O)) \circledast (\ell \overset{S}{\mapsto} \tau_S) \implies \text{by (10) and } P\text{-pure} \\ (\ell \overset{S}{\mapsto} \mathsf{empty} \wedge \ell \overset{O}{\mapsto} \tau_O) \circledast (\ell \overset{S}{\mapsto} \tau_S \wedge \ell \overset{O}{\mapsto} -) \wedge P(\tau_O) \implies \text{by (9)} \\ \exists \tau'_O.\ \tau_O = \tau_S \cup \tau'_O \wedge \ell \overset{S}{\mapsto} \tau_S \wedge \ell \overset{O}{\mapsto} \tau'_O \wedge P(\tau_O) \implies \text{by substituting } \tau_O \\ \exists \tau'_O.\ \ell \hookrightarrow (\tau_S, \tau'_O, \tau) \wedge \tau \leq t \wedge (\tau_S \cup \tau'_O)[t] = \langle \mathsf{res.1}, \mathsf{res.2}, - \rangle. \end{array}$$

Intuitively, in (13) the frame history τ_S is "subtracted" from the other-history τ_O of (4), and moved to the self-history, illustrating one important difference between the frame rule of FCSL and that of CSL. In FCSL, the frame is always subtracted from the other component, whereas in CSL it simply materializes out of nowhere. On the flip side, CSL does not consider the other component, and cannot easily express a spec such as (4).

3.2 Concurroids

We now turn to the component $\mathcal{U}$ of the FCSL specs, which is called *concurroid*. Concurroids are responsible for enforcing the invariants on the evolution of the state. For example, the properties (*i*)–(*iii*) in Section 2 will be enforced by defining an appropriate concurroid to govern the pair-snapshot structure. Thus, concurroids formally represent concurrent data structures, over which the programs operate.

A concurroid is (a form of) a state transition system (STS). It is a quadruple $\mathcal{U} = (L, W, I, E)$ where: (1) L is a set of labels, identifying different data structures; (2) W is a set of admissible states (alternatively, an FCSL assertion); (3) I is the set of *internal transitions* on W; (4) E is a set of pairs (α, ρ), where α is a *heap-acquiring* and ρ is a *heap-releasing* transition, collectively called *external* transitions. The internal transitions are relations on states, describing how a state of the STS evolves in one atomic step. The external transitions serve for transfer of state ownership. The concurroids thus bound the moves of the concurrent programs that operate on a data structure, and therefore represent a structured form of rely/guarantee transitions from Rely/Guarantee logics [8, 9, 18, 33, 34]. We next illustrate concurroids by example.

Pair-snapshot concurroid. Given a label ℓ, pointers x, y, and the type A of the accessible contents of x and y, the concurroid for the pair-snapshot structure is $\mathcal{S} = (\{\ell\}, W_{\mathcal{S}}, \{wr_x, wr_y, \mathsf{id}\}, \emptyset)$. The set of states $W_{\mathcal{S}}$ is described below. We assume that

τ_S, τ_O are histories, $c_x, c_y : A$ and $v_x, v_y : \mathsf{nat}$, and are implicitly existentially quantified.

$$W_{\mathcal{S}} \mathrel{\widehat{=}} \ell \overset{s}{\mapsto} \tau_S \wedge \ell \overset{j}{\mapsto} (x \mapsto (c_x, v_x) \cup y \mapsto (c_y, v_y)) \wedge \ell \overset{o}{\mapsto} \tau_O \wedge \\ \tau_S, \tau_O \text{ satisfy } (ii) - (iii), \tau_S \cup \tau_O \text{ is continuous, and} \\ \text{if } t = \mathsf{last}(\tau_S \cup \tau_O), \text{ then } (\tau_S \cup \tau_O)[t] = (c_x, c_y, v_x)$$

A state in $W_{\mathcal{S}}$ consists of the auxiliary part, which are histories in the self and other components, and concrete part, which is a joint heap, storing pointers x and y, with accessible contents c_x, c_y, and version numbers v_x, v_y, respectively.[3] It requires several additional properties of the auxiliary histories. First, the combined history $\tau_S \cup \tau_O$ is continuous; that is, adjacent timestamps have matching states. Second, the last timestamp in $\tau_S \cup \tau_O$ correctly reflects what is stored in x and y. Finally, $W_{\mathcal{S}}$ also bakes in the properties $(ii) - (iii)$ required in the proof outline of `readPair`, so the specification (4) and its proof were, in fact, carried out in the concurroid context $@\mathcal{S}$, which was omitted.

The internal transitions wr_x and wr_y synchronize the changes to x and y with histories. The transitions operate only on self and joint portions of the state, and the other-portion, τ_O, is fixed (*cf.* notation (10)). That is, the transitions essentially define the concurroid's Guarantee. In both transitions, $t_{\text{fresh}}^{\tau_S \cup \tau_O}$ is the smallest timestamp unused by τ_S and τ_O.

$$wr_x \mathrel{\widehat{=}} \ell \overset{j}{\mapsto} (x \mapsto (c_x, v_x) \cup y \mapsto (c_y, v_y)) \wedge \ell \overset{s}{\mapsto} \tau_S \quad \rightsquigarrow \\ \ell \overset{j}{\mapsto} (x \mapsto (c'_x, v_x + 1) \cup y \mapsto (c_y, v_y) \wedge \ell \overset{s}{\mapsto} \tau_S \cup t_{\text{fresh}}^{\tau_S \cup \tau_O} \mapsto (\langle c_x, c_y, v_x \rangle, \langle c'_x, c_y, v_x + 1 \rangle)$$

$$wr_y \mathrel{\widehat{=}} \ell \overset{j}{\mapsto} (x \mapsto (c_x, v_x) \cup y \mapsto (c_y, v_y)) \wedge \ell \overset{s}{\mapsto} \tau_S \quad \rightsquigarrow \\ \ell \overset{j}{\mapsto} (x \mapsto (c_x, v_x) \cup y \mapsto (c'_y, v_y + 1) \wedge \ell \overset{s}{\mapsto} \tau_S \cup t_{\text{fresh}}^{\tau_S \cup \tau_O} \mapsto (\langle c_x, c_y, v_x \rangle, \langle c_x, c'_y, v_x \rangle)$$

The first conjunct after $\rightsquigarrow$ in wr_x (and wr_y is similar) allows that the version number of x can only increase by 1 in an atomic step. The second conjunct shows that simultaneously with the change of x, the snapshot of the changed state is committed to the self-history of the invoking thread. Together, wr_x and wr_y ensure that histories only grow, and only by adding valid snapshots; *i.e.*, precisely the property (i) from Section 2.

$\mathcal{U}$ also contains the identity transition `id`, whose presence enables programs that do not modify the state at all. In the pair-snapshot example, these are the `readX` and `readY` actions, and the `readPair` method. The pair-snapshot example does not involve ownership transfer, so $\mathcal{S}$ has no external transitions, but these will be important in the forthcoming examples.

Entanglement and private heaps. Larger concurroids may be constructed out of smaller ones. A particularly common construction is *entanglement* [22]. Given concurroids $\mathcal{U}$ and $\mathcal{V}$, the entanglement $\mathcal{U} \bowtie \mathcal{V}$ is a concurroid whose state space is the Cartesian product $W_{\mathcal{U}} \times W_{\mathcal{V}}$, and the transitions allow the $\mathcal{U}$ portion to perform a $\mathcal{U}$ transition, while the $\mathcal{V}$ portion remains idle, and vice-versa. Additionally, $\mathcal{U}$ and $\mathcal{V}$ portions can communicate to *transfer a heap* between themselves, by having one take a heap-acquiring, and the other *simultaneously* taking a heap-releasing transition.

The most common is the entanglement with the concurroid $\mathcal{P}$ of *private heaps* [27, Appendix B]. Entangling with $\mathcal{P}$ lets the concurroids temporarily move heaps to a private section, via the communication discussed above, where threads may then perform the customary operations of reading, writing, allocating, and deallocating pointers,

[3] Notice the overloading of the $\mapsto$ notation for singleton heaps and histories.

without interference.[4] $\mathcal{P}$ comes with a dedicated label pv. As an illustration, the following assertion may describe one possible state in the state space of the entanglement $\mathcal{P} \bowtie \mathcal{S}$ with the snapshot concurroid.

$$\mathsf{pv} \overset{s}{\mapsto} (z \mapsto 0) * \ell \overset{j}{\mapsto} (x \mapsto (c_x, v_x) \cup y \mapsto (c_y, v_y))$$

The $\ell \overset{j}{\mapsto} -$ portion describes the part of the state coming from $\mathcal{S}$, which is joint, containing pointers x and y, as explained before. The $\mathsf{pv} \overset{s}{\mapsto} (z \mapsto 0)$ describes the part of the state coming from $\mathcal{P}$. In this case, it contains a heap with a single pointer z. The heap is private, *i.e.*, owned by the self thread, so z cannot be modified by other threads. Notice that the assertions about pv and ℓ are separated by the resource separating conjunction $*$, which splits the state into portions with disjoint labels and heaps. In this particular case, it signifies that the labels pv and ℓ are distinct, as are the pointers z, x and y.

3.3 Extending and hiding concurroids

Concurroids represent concurrent data structures; thus it is important to be able to introduce and eliminate them. FCSL provides two programming constructors (both no-ops operationally), and corresponding inference rules for that purpose. For completeness, we introduce them here, but postpone the illustration until Section 4.

The injection rule shows that if a program is proved correct with respect to a smaller concurroid $\mathcal{U}$, then it can be extended to $\mathcal{U} \bowtie \mathcal{V}$, without invalidating the proof.

$$\frac{\{p\}\, c\, \{q\}@\mathcal{U}}{\{p * r\}\, [c]\, \{q * r\}@\mathcal{U} \bowtie \mathcal{V}} \quad r \subseteq W_{\mathcal{V}} \text{ stable under } \mathcal{V} \tag{14}$$

This is a form of framing rule, along the axis of adding new resources. The operator $*$ splits the state into portions with disjoint labels, and the side-condition that $r \subseteq W_{\mathcal{V}}$ forces r to remove the labels of the concurroid $\mathcal{V}$, so that c is verified *wrt.* the labels of $\mathcal{U}$. The program constructor $[-]$ is a coercion from $\mathcal{U}$ to $\mathcal{U} \bowtie \mathcal{V}$.

Hiding is the ability to introduce a concurroid $\mathcal{V}$, *i.e.*, install it in a private heap, for the scope of a thread c. The children forked by c can interfere on $\mathcal{V}$'s state, respecting $\mathcal{V}$'s transitions, but $\mathcal{V}$ is hidden from the environment of c, To the environment, $\mathcal{V}$'s state changes look like changes of the private heap of c. Upon termination of c, $\mathcal{V}$ is deinstalled.

$$\frac{\{\mathsf{pv} \overset{s}{\mapsto} h * p\}\, c\, \{\mathsf{pv} \overset{s}{\mapsto} h' * q\}@(\mathcal{P} \bowtie \mathcal{U}) \bowtie \mathcal{V}}{\{\Psi\ g\ h * (\Phi(g) \mathrel{-\!\!*} p)\}\, \mathsf{hide}_{\Phi,g}\, c\, \{\exists g'.\Psi\ g'\ h' * (\Phi(g') \mathrel{-\!\!*} q)\}@\mathcal{P} \bowtie \mathcal{U}}$$
$$\text{where } \Psi\ g\ h = \exists k{:}\mathsf{heap}.\ \mathsf{pv} \overset{s}{\mapsto} h \cup k \wedge \Phi(g) \text{ erases to } k \tag{15}$$

Since installing $\mathcal{V}$ consumes a chunk of private heap, the rule requires the overall concurroid to support private heaps, *i.e.*, to be an entanglement of $\mathcal{P}$ with an arbitrary $\mathcal{U}$. In programs, we use the coercion hide c to indicate the change from $(\mathcal{P} \bowtie \mathcal{U}) \bowtie \mathcal{V}$ to $\mathcal{P} \bowtie \mathcal{U}$. If $\mathcal{U}$ is of no interest, one can take it to be the empty concurroid $\mathcal{E}$, which is a right unit for $\bowtie$ [27, Appendix B.4].

[4] Our Coq proofs actually use two different concurroids, one for reading/writing, another for allocation/deallocation, which we entangle to provide all four operations. For simplicity, here we assume a monolithic implementation.

The annotation Φ is a predicate; it describes an invariant that holds within the scope of hide, parametrized by an argument. It is subject to a number of conditions [27, Appendix D.3]. g is the initial argument, so $\Phi(g)$ holds in the initial state into which $\mathcal{V}$ is placed upon installation. The rule guarantees that the ending state of c satisfies $\exists g'.\, \Phi(g')$. The surrounding connectives $*$ and $-\!\!*$ merely mediate between $\mathcal{U}$, $\mathcal{V}$, and the erasure of $\mathcal{V}$ to heaps. We explain the precondition, and the postcondition is similar.

In the precondition, $*$ separates private heaps from $\mathcal{U}$, and Ψ requires that every state in $\Phi(g)$ obtains the same private heap when the auxiliary fields are erased. $-\!\!*$ is inherited from separation logic. $\Phi(g) -\!\!* p$ says that if the initial state (which is in $W_\mathcal{U}$) is extended with a state from $\Phi(g)$ (which is in $W_\mathcal{V}$), then the result is a state satisfying p. In other words, if a state satisfying $\Phi(g)$ is installed in the initial state of c, while its heap footprint is removed from the private heaps, then c's precondition is satisfied.

4 Treiber stack and its client

In this section we illustrate how histories can be used to specify and verify the fine-grained data structure of Treiber stack [30]. We also show how the specs can be used by clients, where they provide an abstraction that facilitates client reasoning as if the structure were coarse-grained.

The Treiber stack works as follows. Physically, the stack is kept as a singly-linked list in the heap, with a sentinel pointer *snt* pointing to the stack top p1. The call to push(e) allocates a node p that is supposed to go to the top of stack, and attempts to link the node into the stack, by changing the sentinel to p. Clearly, this operation should not succeed if some interfering thread has in the meantime changed the top by pushing or popping elements. Thus push applies a CAS read-modify-write operation [15], which atomically reads *snt*, compares its contents with p1, and if the two are equal (*i.e.*, if the stack's top has not changed), writes p into *snt*, thus en-linking the new top. Otherwise, push is restarted. pop() behaves similarly. It reads the first node p, pointed to by *snt*, and obtains its value e and pointer p1 to the next node. Then it tries to de-link p, by changing the sentinel to p1 using a CAS to identify interference. Note that pop does not deallocate the de-linked node p (this is enforced by the design of the appropriate concurroid as we will soon see), which thus remains in the data structure as garbage. This is by design, to prevent the ABA problem [15, §10]: if p is deallocated, then some other push may allocate it again, and place it back on top of the stack. A procedure that observed p on top of the stack, but has not performed its CAS yet may thus be fooled as follows. Its CAS may encounter p on top of the stack, and proceed as if the stack had not changed, producing invalid results.

Fig. 3 Treiber stack methods.

```
1 push(e : A): Unit {
2  p <- alloc();
3  fix loop() {
4   p1 <- readSentinel();
5   write(p, (e, p1));
6   ok <- tryPush(p1, p);
7   if ok then return ();
8   else loop();}();
9 }
```

```
1 pop(): option A {
2  p <- readSentinel();
3  if p == null
4  then return None;
5  else {
6   (e,p1) <- readNode(p);
7   ok     <- tryPop(p,p1);
8   if ok then return Some e;
9   else pop();}}
```

The described code of the Treiber stack operations is given in Figure 3, where we used descriptive names for the atomic operations. Instead of CAS, we used tryPush and tryPop, and instead of pointer read, we used readSentinel and readNode. The reason for the descriptive names is that the atomic operations in FCSL operate not only on concrete

heap pointers, but on auxiliary state as well. In the particular case of Treiber, the auxiliary state will be histories, which tryPush and tryPop change in different ways, even though they both operationally perform a CAS. Similarly, readSentinel and readNode deduce different facts about the histories, even though they both simply read from a pointer. We elide here any further discussion on how the atomic operations are specified and verified in FCSL (it can be found in [22] and [27, Appendix C]). Instead, whenever needed, we simply state the Hoare specs for the atomics and proceed to use them in proof outlines, as if the atomics were ordinary procedures. Of course, our Coq files contain proofs that all such Hoare triples are valid.

Treiber concurroid. Given a label tb, the sentinel pointer *snt*, and the type A of the stack elements, the state space of the Treiber concurroid $\mathcal{T}$ is described as follows. Its auxiliary self/other components are histories τ_S and τ_O that store mathematical sequences l corresponding to the logical contents of the stack at various timestamps. The joint component contains a heap h_s storing a sentinel *snt* pointing to a linked list, a heap h implementing the list, and a garbage section *grb* of de-linked nodes.

$$\begin{aligned} W_{\mathcal{T}} &\mathrel{\widehat{=}} \exists \tau_S\ \tau_O\ h_s.\ \mathsf{tb} \overset{s}{\mapsto} \tau_S \wedge \mathsf{tb} \overset{o}{\mapsto} \tau_O \wedge \mathsf{tb} \overset{j}{\mapsto} h_s \wedge I\ (\tau_S \cup \tau_O)\ h_s \\ I\ \tau\ h_s &\mathrel{\widehat{=}} \exists p\ h\ grb\ l.\ h_s = (snt \mapsto p) \cup h \cup grb \wedge \mathsf{list}(p, l, h) \wedge \\ &\qquad \mathsf{complete}(\tau) \wedge \mathsf{continuous}(\tau) \wedge \mathsf{stacklike}(\tau) \wedge \tau[\mathsf{last}(\tau)] = l \end{aligned} \tag{16}$$

The auxiliary predicates are:

$$\begin{aligned} \mathsf{list}(p, l, h) &\mathrel{\widehat{=}} p = \mathtt{null} \wedge l = \mathsf{nil} \wedge h = \mathsf{empty} \vee \\ &\qquad \exists e\ p'\ l'\ h'.\ l = e :: l' \wedge h = p \mapsto (e, p') \cup h' \wedge \mathsf{list}(p', l', h') \\ \mathsf{complete}(\tau) &\mathrel{\widehat{=}} \exists l_0.\ \tau(0) = (l_0, l_0) \wedge \forall t.\ t < |\mathsf{dom}(\tau)| \Rightarrow t \in \mathsf{dom}(\tau) \\ \mathsf{stacklike}(\tau) &\mathrel{\widehat{=}} \forall t \in \mathsf{dom}(\tau).\ t > 0 \ \Rightarrow \exists l\ e.\ \tau(t) = (l, e :: l) \vee \tau(t) = (e :: l, e) \end{aligned}$$

In particular: (1) the overall history $\tau_S \cup \tau_O$ is complete, *i.e.* no gaps exist between timestamps (this property was irrelevant for the pair snapshot structure, but essential for stacks to ensure the absence of the ABA-problem); (2) aside from the initialization in timestamp 0, the history only stores events corresponding to pushing or popping, and (3) the last recorded state in the history captures the current contents of the stack. For simplicity, we disable reasoning about the structure's inherent memory leak by not relating histories to *grb* in (16).

The transitions of $\mathcal{T}$ allow for popping and pushing only.

$$\begin{aligned} pop &\mathrel{\widehat{=}} \mathsf{tb} \overset{j}{\mapsto} snt \mapsto p \cup h \cup grb \wedge \mathsf{tb} \overset{s}{\mapsto} \tau_S \wedge h = (p \mapsto (e, p') \cup h') \wedge \mathsf{list}(p, (e :: l), h) \rightsquigarrow \\ &\qquad \mathsf{tb} \overset{j}{\mapsto} snt \mapsto p' \cup h' \cup (p \mapsto (e, p') \cup grb) \wedge \mathsf{tb} \overset{s}{\mapsto} \tau_S \cup t_{\mathrm{fresh}}^{\tau_S \cup \tau_O} \mapsto (e :: l, l) \\ push_{p', e, p} &\mathrel{\widehat{=}} \mathsf{tb} \overset{j}{\mapsto} snt \mapsto p \cup h \cup grb \wedge \mathsf{tb} \overset{s}{\mapsto} \tau_S \wedge \mathsf{list}(p, l, h) \rightsquigarrow \\ &\qquad \mathsf{tb} \overset{j}{\mapsto} snt \mapsto p' \cup (p' \mapsto (e, p) \cup h) \cup grb \wedge \mathsf{tb} \overset{s}{\mapsto} \tau_S \cup t_{\mathrm{fresh}}^{\tau_S \cup \tau_O} \mapsto (l, e :: l) \end{aligned}$$

In *pop*, the sentinel pointer is swapped from used-to-be head p to its next one, p', whereas $(p \mapsto -)$ logically joins the garbage. The transition *push* describes how a heap of the shape $p' \mapsto (e, p)$, describing the node to be pushed, is acquired and placed at the top of the stack. It is an external transition, which means it only fires when entangled with a concurroid from which the heap $p' \mapsto (e, p)$ can be taken away. In our case, that will be the concurroid $\mathcal{P}$ for private state. Indeed, both transitions preserve the state invariant I (16). Importantly, $\mathcal{T}$ does not have a release transition; once a memory chunk is in the joint state, it never leaves, capturing that $\mathcal{T}$ does not allow deallocation.

Method specs. We give the following history-based specs.

$$\left\{\begin{array}{c}\mathsf{pv} \overset{s}{\mapsto} \mathsf{empty} * \\ \mathsf{tb} \hookrightarrow (\mathsf{empty}, -, \tau)\end{array}\right\} \mathsf{push}(e) \left\{\begin{array}{c}\exists t\, l.\, \mathsf{pv} \overset{s}{\mapsto} \mathsf{empty} * \\ \mathsf{tb} \hookrightarrow (t \mapsto (l, e :: l), -, \tau) \wedge \tau < t\end{array}\right\} @ \mathcal{P} \bowtie \mathcal{T}$$

$$\left\{\mathsf{tb} \hookrightarrow (\mathsf{empty}, -, \tau)\right\} \quad \mathsf{pop}() \quad \left\{\begin{array}{c}\exists e\, t\, l.\, \mathsf{res} = \mathsf{Some}\ e \wedge \mathsf{tb} \hookrightarrow (t \mapsto (e :: l, l), -, \tau) \wedge \tau < t \vee \\ \exists \tau_o\, t.\, \mathsf{res} = \mathsf{None} \wedge \mathsf{tb} \hookrightarrow (\mathsf{empty}, \tau_o, \tau) \wedge \tau_o[t] = \mathsf{nil}\end{array}\right\} @ \mathcal{T} \tag{17}$$

A call to push runs with empty private heap and history, thus by framing, it can run with any private heap and history. After termination, the self history is incremented by a singleton exposing that a push event has been executed at a time stamp t; $\tau < t$ indicates that the push event appeared strictly after the events preceding the call. The spec for pop is slightly more complicated as pop checks for stack emptiness, but ultimately proceeds in the similar manner. push works over the entangled concurroid $\mathcal{P} \bowtie \mathcal{T}$, as it needs to allocate memory; pop works over $\mathcal{T}$ only, as it does not deallocate.

Fig. 4 A proof outline of Treiber's push method.

```
1  { pv ↦ˢ empty * tb ↪ (empty, −, τ) }
2  p <- [alloc()];
3  { pv ↦ˢ p ↦ − * tb ↪ (empty, −, τ) }
4  fix loop() {
5  { pv ↦ˢ p ↦ − * tb ↪ (empty, −, τ) }
6  p1 <- [readSentinel()];
7  { pv ↦ˢ p ↦ − * tb ↪ (empty, −, τ) }
8  [write(p, (e, p1))];
9  { pv ↦ˢ p ↦ (e, p1) * tb ↪ (empty, −, τ) }
10 ok <- tryPush(p1, p);
11 { ok ∧ ∃t l. pv ↦ˢ empty * tb ↪ (t ↦ (l, e :: l), −, τ) ∧ τ < t ∨
   { ¬ok ∧ pv ↦ˢ p ↦ (e, p1) * tb ↪ (empty, −, τ)
12 if ok then return ();
13 { ∃t l. pv ↦ˢ empty * tb ↪ (t ↦ (l, e :: l), −, τ) ∧ τ < t }
14 else
15 { pv ↦ˢ p ↦ − * tb ↪ (empty, −, τ) }
16 loop();}();
17 { ∃t l. pv ↦ˢ empty * tb ↪ (t ↦ (l, e :: l), −, τ) ∧ τ < t }
```

Verification of push and pop implementations relies on the specifications of the atomic actions alloc and write, which are specific to the $\mathcal{P}$ concurroid.

$$\begin{array}{l}\{\ \mathsf{pv} \overset{s}{\mapsto} \mathsf{empty}\ \}\quad \mathsf{alloc}()\quad \{\ \mathsf{pv} \overset{s}{\mapsto} \mathsf{res} \mapsto -\ \} @ \mathcal{P} \\ \{\ \mathsf{pv} \overset{s}{\mapsto} x \mapsto -\ \}\ \mathsf{write}(x, e)\ \{\ \mathsf{pv} \overset{s}{\mapsto} x \mapsto e\ \} @ \mathcal{P}\end{array} \tag{18}$$

In Figure 4, we present the proof outline for push (the proof for pop can be found in the Coq files). It is mostly self-explanatory, so we only point out a few technicalities. The actions alloc and write have to be explicitly injected into $\mathcal{P} \bowtie \mathcal{T}$, by means of the coercion $[-]$, introduced in Section 3. Similarly for readSentinel, whose concurroid is $\mathcal{T}$. Somewhat surprisingly, the call to readSentinel in line 6 is irrelevant for the (partial) correctness of tryPush; thus, line 7 does not say anything about p1.[5] The proof rule for fix allows assuming the spec of a procedure in the proof of the body, and is presented in [27, Appendix D]. The tryPush action appears in the proof outline with its precise specification; that is, line 9 contains its precondition, and 11 contains the postcondition, describing that a successful outcome of tryPush removed a heap from $\mathcal{P}$, moved it to the joint heap of $\mathcal{T}$, and updated the history, following the *push* transition.

[5] Though, taking a random p1 here will affect liveness, as push will keep looping until it finds the chosen p1 at the top of the stack.

Recovering sequential specifications. We next show that the subjective spec (17) is a generalization of the canonical sequential spec (1). In particular, if there is no interference from other threads, (17) can be reduced to (1). The mechanism for achieving the reduction relies on the self/other dichotomy, thus substantiating our point that the dichotomy is important for precise reasoning with histories.

To this end, we use the hide constructor from Section 3. It introduces a concurroid in a delimited scope, and prohibits the environment threads from interfering on it. The heap for the introduced concurroid is appropriated from the private heap. In the case of push, we will appropriate a heap storing the sentinel and the linked list of the stack, install the $\mathcal{T}$ concurroid over this heap, perform push with interference disabled, then return the heap back to private heaps. We will derive the following specification, which is essentially an elaborated version of (1), modulo the memory leak inherent to Treiber stack (hence *grb* in the postcondition).

Fig. 5 Proof of sequential spec for push.

1 $\{\exists p\, h.\, \mathsf{pv} \overset{s}{\mapsto} (snt \mapsto p \cup h) \wedge \mathsf{list}(p, l, h)\}$
2 $\{\Psi\ \mathsf{empty}\ \mathsf{empty} * (\Phi(\mathsf{empty}) \mathrel{-\!\!*} \mathsf{tb} \hookrightarrow (0 \mapsto (l, l), -, -))\}$
3 $\mathsf{hide}_{\Phi,\mathsf{empty}}$ {
4 $\{\ \mathsf{pv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{tb} \hookrightarrow (0 \mapsto (l, l), -, -)\ \}$
5 push(e);
6 $\left\{\begin{array}{l}\exists t\, l'.\, \mathsf{pv} \overset{s}{\mapsto} \mathsf{empty} * \\ \mathsf{tb} \hookrightarrow (0 \mapsto (l, l) \cup t \mapsto (l', e :: l'), -, -)\end{array}\right\}$ }
7 $\left\{\begin{array}{l}\exists \tau.\, \Psi\ \tau\ \mathsf{empty} * \\ (\Phi(\tau) \mathrel{-\!\!*} \exists t\, l'.\, \mathsf{tb} \hookrightarrow (0 \mapsto (l, l) \cup t \mapsto (l', e :: l'), -, -))\end{array}\right\}$
8 $\left\{\begin{array}{l}\exists t\, l'\ \tau.\, \tau = 0 \mapsto (l, l) \cup t \mapsto (l', e :: l') \wedge \\ \mathsf{complete}(\tau) \wedge \mathsf{continuous}(\tau) \wedge \Psi\ \tau\ \mathsf{empty}\end{array}\right\}$
9 $\{\exists \tau.\, \tau = 0 \mapsto (l, l) \cup 1 \mapsto (l, e :: l) \wedge \Psi\ \tau\ \mathsf{empty}\}$
10 $\{\ \exists p'\, h.\, \mathsf{pv} \overset{s}{\mapsto} (snt \mapsto p' \cup h \cup -) \wedge \mathsf{list}(p', e :: l, h)\ \}$

$$\begin{array}{c}\{\ \exists p\, h.\, \mathsf{pv} \overset{s}{\mapsto} (snt \mapsto p \cup h) \wedge \mathsf{list}(p, l, h)\ \} \\ \mathsf{hide}_{\Phi,\mathsf{empty}}\ \{\ \mathsf{push}(e);\ \} \\ \{\ \exists p\, h\, grb.\, \mathsf{pv} \overset{s}{\mapsto} (snt \mapsto p \cup h \cup grb) \wedge \mathsf{list}(p, e :: l, h)\ \}@\mathcal{P}\end{array} \tag{19}$$

The self/other dichotomy affords explicit access to other-owned histories, so that we can define the following predicate Φ stating that other-histories remain empty within the scope of hide.

$$\Phi(\tau) \mathrel{\widehat{=}} \exists l.\, \mathsf{tb} \overset{s}{\mapsto} ((0 \mapsto (l, l)) \cup \tau) \wedge \mathsf{tb} \overset{o}{\mapsto} \mathsf{empty} \wedge W_{\mathcal{T}} \tag{20}$$

Inside hide, the stack is initialized (the history contains the singleton $0 \mapsto (l, l)$), there is no interference ($\mathsf{tb} \overset{o}{\mapsto} \mathsf{empty}$), and the state is a valid one for $\mathcal{T}$ (*i.e.*, it is captured by the definition (16)).

One can prove that if the histories are erased from any state in $\Phi(\tau)$, the remaining concrete heap consists of *snt* and the stack. Moreover, the contents of the stack is the last entry of τ (or l if τ is empty). In other words, using Ψ (15), defined in Section 3:

$$\Psi\ \tau\ \mathsf{empty} \iff \exists p\, h.\, \mathsf{pv} \overset{s}{\mapsto} (snt \mapsto p \cup h \cup -) \wedge \mathsf{list}(p, l', h) \tag{21}$$

where $l' = \tau[\mathsf{last}(\tau)]$ (or $l' = l$ if τ is empty).

The derivation is in Figure 5, and we comment on the main points. In line 2, the right conjunct uses the property inherent in Ψ, that $\Phi(\mathsf{empty})$ erases to the heap storing l. Thus, this is the l that appears in the consequent of $-\!\!*$. In line 7, the second conjunct implies that the history τ, whose existence obtains from the rule for hiding (15), must be the self-history returned by push. Hence, it is equal to $0 \mapsto (l, l) \cup t \mapsto (l', e :: l')$ for some t and l'. But, we also know that τ must be complete (no gaps between timestamps) and continuous. Hence $t = 1$ and $l' = l$ in line 9, which derives the postcondition by (21).

Fig. 6 A parallel stack-based producer/consumer program.

```
1 produce(n: nat, i: nat) {
2  if i == n
3  then return ();
4  else {
5   e <- ap[i];
6   push_tb(e);
7   produce(i + 1);
8  }
9 }
```

```
1 consume(n: nat, i: nat) {
2  if i == n
3  then return ();
4  else {
5   r <- pop_tb();
6   if r == Some e
7   then {
8    ac[i] := e;
9    consume(i + 1);}
10  else consume(i);}}
```

```
1 exchange(n: nat): Unit {
2  hide_{Φ,empty} {
3    produce(n, 0); || consume(n, 0);
4  }
5 }
```

A stack client. We next illustrate how the specs (17) are exploited by the *concurrent* clients of Treiber stack to abstract from the fine-grained nature of Treiber's implementation. The example code in Figure 6 presents two procedures, `produce` and `consume`, that communicate via a common Treiber stack `tb`. `produce` pushes onto the stack the elements of its array `ap` in order, whereas `consume` pops from the stack, to fill its array `ac`. Both arrays are of equal size n. The procedure `exchange` runs `produce` and `consume` concurrently. We will prove that after `exchange` terminates, `ap` has been copied to `ac`, modulo element permutation. The inference will only use the specs (17) but not the code of stack methods, thus obtaining a coarse-grained view of effects provided by histories.

We use several auxiliary predicates. First, $\mathsf{Arr}_n(a, l, h)$ defines an array of size n as a sequence of consecutive pointers in the heap h, starting from pointer a, and storing elements of the list l:

$$\mathsf{Arr}_n(a, l, h) \;\widehat{=}\; |\, l \,| = n \wedge h = \bigcup\nolimits_{i<n} (a + i) \mapsto l(i) \tag{22}$$

Next, the predicates `Pushed` and `Popped` extract the lists of pushed and popped elements from a stack history τ.

$$\begin{aligned} \mathsf{Pushed}(\tau, l) &\;\widehat{=}\; l =_{/\mathrm{mset}} \{\!\{e \mid \exists t\, l.\, t \mapsto (l, e :: l) \in \tau \vee 0 \mapsto (l, l) \in \tau \wedge e \in l\}\!\} \\ \mathsf{Popped}(\tau, l) &\;\widehat{=}\; l =_{/\mathrm{mset}} \{\!\{e \mid \exists t\, l.\, t \mapsto (e :: l, l) \in \tau\}\!\} \end{aligned} \tag{23}$$

The notation $\{\!\{-\}\!\}$ stands for multisets, and $=_{/\mathrm{mset}}$ is multiset equality, which we conflate with list equality modulo permutation. We can now ascribe the following specs to `produce` and `consume`:

$$\begin{aligned} &\left\{\mathsf{Pr}(h_p, l_{<i}) \wedge \mathsf{Arr}_n(\mathsf{ap}, l, h_p)\right\}\ \mathsf{produce}(n, i)\ \left\{\mathsf{Pr}(h_p, l) \wedge \mathsf{Arr}_n(\mathsf{ap}, l, h_p)\right\} \\ &\left\{\exists h_c\ l.\, \mathsf{Cn}(h_c, l_{<i}) \wedge \mathsf{Arr}_n(\mathsf{ac}, l, h_c)\right\}\ \mathsf{consume}(n, i)\ \left\{\exists h_c\ l.\, \mathsf{Cn}(h_c, l) \wedge \mathsf{Arr}_n(\mathsf{ac}, l, h_c)\right\} \end{aligned} \tag{24}$$

both over the $\mathcal{P} \bowtie \mathcal{T}$ concurroid. `Pr` and `Cn` are defined as follows:

$$\begin{aligned} \mathsf{Pr}(h_p, l) &\;\widehat{=}\; \mathsf{pv} \overset{s}{\mapsto} h_p * \mathsf{tb} \overset{s}{\mapsto} \tau_\mathsf{S} \wedge \mathsf{Pushed}(\tau_\mathsf{S}, l) \wedge \mathsf{Popped}(\tau_\mathsf{S}, \mathsf{nil}) \\ \mathsf{Cn}(h_c, l) &\;\widehat{=}\; \mathsf{pv} \overset{s}{\mapsto} h_c * \mathsf{tb} \overset{s}{\mapsto} \tau_\mathsf{S} \wedge \mathsf{Pushed}(\tau_\mathsf{S}, \mathsf{nil}) \wedge \mathsf{Popped}(\tau_\mathsf{S}, l), \end{aligned}$$

so they essentially describe the producer/consumer loop invariants; $l_{<i}$ is a prefix of l for elements with indices less than i. The specs (24) show that `produce` pushes all the elements from `ap`, and `consume` fills `ac` with elements of some sequence of the length n. The proofs of both specs (available in our Coq development) derive easily from (17) after these are framed to allow running in arbitrary initial self heap and history.

The interesting part of the example is proving `exchange`, where we compose `produce` and `consume` in parallel, and then use hiding to infer that the `ap` and `ac` arrays in the end contain the same elements, modulo permutation. The proof outline is in Figure 7, and it relies on the following important lemmas about histories.

Lemma 1. $\mathsf{Pushed}(\tau_1, l_1) \land \mathsf{Popped}(\tau_1, \mathsf{nil}) \land \mathsf{Popped}(\tau_2, l_2) \land \mathsf{Pushed}(\tau_2, \mathsf{nil}) \implies \mathsf{Pushed}(\tau_1 \cup \tau_2, l_1) \land \mathsf{Popped}(\tau_1 \cup \tau_2, l_2)$.

Lemma 2. *If* $\mathsf{complete}(\tau)$ *and* $\mathsf{stacklike}(\tau)$ *then* $\mathsf{Pushed}(\tau, l_1) \land \mathsf{Popped}(\tau, l_2) \land |l_1| = |l_2| \implies l_1 =_{/\mathrm{mset}} l_2$.

Fig. 7 Proof outline for producer/consumer.

$$
\begin{array}{c}
\left\{ \mathsf{pv} \overset{s}{\mapsto} h_p \cup h_c \cup \mathit{snt} \mapsto \mathtt{null} \land \mathsf{Arr}_n(\mathsf{ap}, l, h_p) \land \mathsf{Arr}_n(\mathsf{ac}, -, h_c) \right\} \\
\mathsf{hide}_{\Phi,\mathsf{empty}}\ \{ \\
\left\{ \begin{array}{c} \mathsf{pv} \overset{s}{\mapsto} h_p \cup h_c \land \mathsf{Arr}_n(\mathsf{ap}, l, h_p) \land \mathsf{Arr}_n(\mathsf{ac}, -, h_c) * \\ \mathsf{tb} \overset{s}{\mapsto} 0 \mapsto (\mathsf{nil}, \mathsf{nil}) \land \mathsf{tb} \overset{o}{\mapsto} \mathsf{empty} \end{array} \right\} \\
\left\{ \left(\begin{array}{c} \mathsf{pv} \overset{s}{\mapsto} h_p \land \mathsf{Arr}_n(\mathsf{ap}, l, h_p) \\ * \mathsf{tb} \overset{s}{\mapsto} 0 \mapsto (\mathsf{nil}, \mathsf{nil}) \end{array} \right) \circledast \left(\begin{array}{c} \mathsf{pv} \overset{s}{\mapsto} h_c \land \mathsf{Arr}_n(\mathsf{ac}, -, h_c) \\ * \mathsf{tb} \overset{s}{\mapsto} \mathsf{empty} \end{array} \right) \right\} \\
\left\{ \mathsf{Pr}(h_p, l_{<0}) \land \mathsf{Arr}_n(\mathsf{ap}, l, h_p) \right\} \\
\mathtt{produce}(n, 0); \\
\left\{ \mathsf{Pr}(h_p, l) \land \mathsf{Arr}_n(\mathsf{ap}, l, h_p) \right\}
\end{array}
\Bigg\| \begin{array}{c}
\left\{ \exists l'.\ \mathsf{Cn}(h_c, l'_{<0}) \land \mathsf{Arr}_n(\mathsf{ac}, l', h_c) \right\} \\
\mathtt{consume}(n, 0); \\
\left\{ \exists h'_c\ l'.\ \mathsf{Cn}(h_c, l') \land \mathsf{Arr}_n(\mathsf{ac}, l', h'_c) \right\}
\end{array}
$$

$$
\left\{ \left(\mathsf{Pr}(h_p, l) \land \mathsf{Arr}_n(\mathsf{ap}, l, h_p) \right) \circledast \left(\exists h'_c\ l'.\ \mathsf{Cn}(h_c, l') \land \mathsf{Arr}_n(\mathsf{ac}, l', h'_c) \right) \right\}
$$

$$
\left\{ \begin{array}{c} \exists h'_c\ l'.\ \mathsf{pv} \overset{s}{\mapsto} h_p \cup h_c \land \mathsf{Arr}_n(\mathsf{ap}, l, h_p) \land \mathsf{Arr}_n(\mathsf{ac}, l', h'_c) \\ * \exists \tau_\mathsf{S}, \mathsf{tb} \overset{s}{\mapsto} \tau_\mathsf{S} \land \mathsf{Pushed}(\tau_\mathsf{S}, l) \land \mathsf{Popped}(\tau_\mathsf{S}, l') \land \mathsf{tb} \overset{o}{\mapsto} \mathsf{empty} \end{array} \right\} \}
$$

$$
\left\{ \begin{array}{c} \exists h'_c\ l'.\ \mathsf{pv} \overset{s}{\mapsto} h_p \cup h'_c \cup (\mathit{snt} \mapsto -) \cup - \land \\ \mathsf{Arr}_n(\mathsf{ap}, l, h_p) \land \mathsf{Arr}_n(\mathsf{ac}, l', h'_c) \land l =_{/\mathrm{mset}} l' \end{array} \right\}
$$

The proof outline in Figure 7 starts in the concurroid $\mathcal{P}$, which extends to $\mathcal{P} \bowtie \mathcal{T}$ in the scope of `hide`. The invariant Φ of `hide` is the one we already used, defined in (20). It introduces a Treiber stack structure with an initial history $0 \mapsto (\mathsf{nil}, \mathsf{nil})$. Also, the heaplet $\mathit{snt} \mapsto \mathtt{null}$ with the sentinel pointer has been donated to the state space of the Treiber stack, so it is removed from the private heap. Next, the self-heap and history are split via $\circledast$; the parts are given to `produce` and `consume`, respectively, according to the parallel composition rule (7). Next, we reason out of specifications (24) for producer/consumer and combine the subjective views back via $\circledast$ upon joining of the parallel threads: we thus derive that the contents of `ap` and `ac`, are l and l' respectively. By unfolding the definitions of `Pr` and `Cn`, and using Lemma 1, we derive $\mathsf{Pushed}(\tau_\mathsf{S}, l) \land \mathsf{Popped}(\tau_\mathsf{S}, l')$, where τ_S is the combined history of `produce` and `consume`. Finally, τ_S is complete and stack-like (since other-history is provably `empty` thanks to hiding). Moreover, both l and l' have size n, as ensured by the assertion Arr_n constraining both of them. Thus, in the last assertion, we can use Lemma 2 to obtain the desired equality of l and l' modulo permutation. Note also that the sentinel pointer is returned back to the private heap, along with the garbage heap (existentially abstracted by the $-$ placeholder).

5 Flat combining

This section shows how PCMs in general, and histories in particular, can formalize the concurrent algorithm design pattern of helping, whereby one concurrent thread may execute code on behalf of another. We use Hendler *et al.*'s flat combining algorithm as an example [14]. Unlike other proofs of this algorithm [4, 31], we do not require any additional logical infrastructure aside from ordinary auxiliary state, represented by a PCM [19, 22]. We verify the algorithm *wrt.* a generic PCM, and then instantiate with the PCM of histories. Thus, our proof is usable even in examples where the specs do not rely on histories.

The flat combiner structure (FC) generalizes a coarse-grained lock [22,23,25]. In the case of a lock, threads acquire exclusive access to the shared resource protected by the lock, *in succession*. With the flat combiner, threads register the work that they want to perform over the shared resource. The lock-acquiring thread (aka. the *combiner*) then executes all the registered work, so the other threads do not need to compete for the lock anymore. This reduces the contention on the lock, and improves performance.

Fig. 8 Flat combining algorithm.

```
1  flatCombine(f: A → B, x: A): B {
2   reqHelp(tid, f, x);
3   fix loop() {
4    locked <- tryLock();
5    if locked then {
6     for i∈{0, ..., n − 1} {
7      req <- readReq(i);
8      if req == Req fi xi then {
9       w <- fi(xi);
10      doHelp(i, w);
11     }}
12    unlock();}
13  rc <- tryCollect(tid);
14  if rc == Some w
15  then return w;
16  else loop();}();}
```

The higher-order `flatCombine` procedure (Figure 8) works as follows.[6] It takes as input a *sequential* function f and argument x, and registers the invoking thread for help with executing $f\ x$ over the shared resource. It does so by storing `Req f x` into the shared *publication* array, at index `tid` (line 2), where `tid` is the id of the invoking thread. It next enters the main loop (line 3) and tries to acquire the lock to the shared heap (line 4). The acquiring thread becomes a combiner (line 5); it traverses the publication array, where the global variable n bounds the number of threads, checking for help requests (lines 6–11). For each request found (which can arrive even while the combiner holds the lock), the combiner executes the appropriate function with the provided arguments (line 9) over the shared heap. It informs the requesting thread `i` of the result `w`, by writing `Resp w` into the slot `i` of the publication array (line 10). After the traversal, the combiner releases the lock (line 12). Finally, the thread (combiner or otherwise), checks the publication array to see if it has been helped (line 13). If so, it extracts the result `w` from its slot in the publication array, and fills the slot with `Init` (all line 13). The result of the help, if one exists, is returned in line 15. Otherwise, the thread loops for help again.

To supply the intuition behind the spec for FC, we first review how ordinary locks work with auxiliary state, in the subjective setting of FCSL [22]. In CSL [23], and the Owicki-Gries method [25], a lock comes with a resource invariant I that restricts the heap of the shared resource. Such restriction implicitly assumes a presence of "hard-coded" auxiliary state, describing the contents of the corresponding shared heap (the explicit parametrization over the auxiliary state, which we make use of here, is explained in the introduction of [19]). When the lock is not taken, the shared heap satisfies I. When the lock is taken, the heap is in the exclusive possession of the acquiring thread, which can invalidate I, but has to restore it before releasing the lock. The subjective setting is similar, except the values of the auxiliary state are drawn from a PCM $\mathbb{U}$, and specs keep track of two values $\mathsf{g_S}$ and $\mathsf{g_O}$, describing how much the thread (*self*) and its environment (*other*) have contributed to the resource, respectively. When the lock is free, the heap of the shared resource satisfies $I(\mathsf{g_S} \bullet \mathsf{g_O})$. When the lock is released by a thread, the thread may update its $\mathsf{g_S}$ by some value g_Δ, reflecting that its contribution

[6] For simplicity, we consider a modified version of the original algorithm. In particular, (a) we use an array rather than a priority queue for registration of help requests, and (b) we do not expunge help requests that have not been served for sufficiently long time.

to the resource changed. Thus, if before locking, the resource satisfied $I(\mathsf{g_s} \bullet \mathsf{g_o})$, after unlocking it will satisfy $I(\mathsf{g_s} \bullet \mathsf{g}_\Delta \bullet \mathsf{g_o})$, as shown by examples in Section 3 of [22].

The setup of the flat combiner is similar, but in addition to $\mathsf{g_s}$ and $\mathsf{g_o}$, FC also keeps an array g_p storing a $\mathbb{U}$-value for each thread. The entry $\mathsf{g}_p[i]$ signifies how much the thread i has been helped by the combiner. If $\mathsf{g}_p[i] = \mathsf{g}_\Delta$ is non-unit, i can collect the help by joining g_Δ to its own $\mathsf{g_s}$, and setting $\mathsf{g}_p[i]$ to the unit $\mathbb{1}$ of $\mathbb{U}$, after which it can ask for help again. Thus, the overall relation between the auxiliary state and the resource heap, when the lock is free, is captured by the invariant $I\,(\bigodot_{i=1}^{n} \mathsf{g}_p[i] \bullet \mathsf{g_s} \bullet \mathsf{g_o})$.

5.1 Flat combiner state and transitions

The states of the FC concurroid $\mathcal{F}$ are described by the assertion:

$$W_{\mathcal{F}} \mathrel{\widehat{=}} \mathsf{fc} \overset{s}{\mapsto} (\mathsf{t_s}, \mathsf{m_s}, \mathsf{g_s}) \wedge \mathsf{fc} \overset{o}{\mapsto} (\mathsf{t_o}, \mathsf{m_o}, \mathsf{g_o}) \wedge \mathsf{fc} \overset{j}{\mapsto} \langle lk \mapsto b \cup h_p \cup h_r, \mathsf{g}_p \rangle \wedge \exists l_p.\, \mathsf{Arr}_n(a_p, l_p, h_p)$$

The auxiliary state in the self/other components consists of the following. $\mathsf{t_s}$ and $\mathsf{t_o}$ are sets of thread ids, which form a PCM under disjoint union.[7] $\mathsf{m_s}$ and $\mathsf{m_o}$ are elements of the *mutual exclusion* set $O = \{\cancel{\mathsf{Own}}, \mathsf{Own}\}$ [19, 22] and record whether the lock lk is owned by the thread, or the environment. O is a PCM under the operation defined as $x \bullet \cancel{\mathsf{Own}} = \cancel{\mathsf{Own}} \bullet x = x$, with $\mathsf{Own} \bullet \mathsf{Own}$ undefined. The unit element is $\cancel{\mathsf{Own}}$, and the undefinedness of $\mathsf{Own} \bullet \mathsf{Own}$ means that two threads cannot simultaneously own the lock. $\mathsf{g_s}$ and $\mathsf{g_o}$ are elements of a generic PCM $\mathbb{U}$, as described above. The self/other triples form a PCM with component-wise lifted joins and units.

The joint component of $\mathcal{F}$ contains a concrete heap, and the auxiliary array g_p. The concrete heap keeps the pointer $lk \mapsto b$, which stands for the lock, with the boolean b representing the lock status. It also stores the publication array with the origin pointer a_p into the heaplet h_p (see notation (22)). The array stores elements of type $\mathsf{Stat} \mathrel{\widehat{=}} \mathsf{Init} \mid \mathsf{Req}\ f\ x \mid \mathsf{Resp}\ w$, as already apparent from Figure 8. We abuse the notation and refer to the array represented by h_p as a_p. The heap h_r is the resource protected by the FC lock. Upon locking it moves to the exclusive ownership of the combiner.

We further assume the following properties of $W_{\mathcal{F}}$:

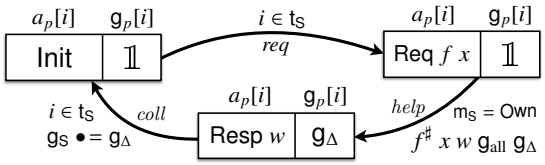
(*i*) for any *tid*, if $\mathsf{g}_p[tid] \neq \mathbb{1}$, then $a_p[tid] = \mathsf{Resp}\ w$ for some w;

(*ii*) if b is true then $h_r = \mathsf{empty}$ and $\mathsf{m_s} \bullet \mathsf{m_o} = \mathsf{Own}$; otherwise $\mathsf{m_s} \bullet \mathsf{m_o} = \cancel{\mathsf{Own}}$ and $I\,(\bigodot_{i=1}^{n} \mathsf{g}_p[i] \bullet \mathsf{g_s} \bullet \mathsf{g_o})\ h_r$.

Property (*i*) ensures that the auxiliary array g_p holds a pending contribution in a cell *tid* only if the corresponding entry in the publication array a_p points to the response with some (uncollected) result. Property (*ii*) formally relates the auxiliary state to the resource heap h_r, as already described.

Flat combiner concurroid's external transitions intuitively correspond to locking and unlocking the heap h_r, thus moving it from the joint to private state, and vice-versa. We do not present them formally, as they are similar to the transitions in CSL [22]. The internal transitions *req*, *help* and *coll* synchronously change the contents of a_p and g_p for a particular thread id i (one at a time) as the following diagram illustrates.

[7] One thread may hold many thread id's, which it distributes between its children upon forking.



The transition *req* can be taken only by a thread holding the thread id i; it changes the value of $a_p[i]$ from Init to Req f x for some f and x. The transition *help* can be performed by any thread that owns the lock (not necessarily the one with the id i); it replaces the contents of $a_p[i]$ and $g_p[i]$ with an appropriate result w and an auxiliary delta g_Δ, respectively. The two are valid *wrt.* the input x and the cumulative auxiliary g_{all}, as ensured by the constraint $f^\sharp$. Finally, *coll* is invoked by the thread with id i; it flushes the contents of $g_p[i]$, into the self-contribution g_S and puts Init into $a_p[i]$.

5.2 Flat combiner specification

We now provide a spec for `flatCombine` in terms of the concurroid $\mathcal{F}$. We assume $f : A \to B$, $x : A$, and f comes with the following spec.[8]

$$\{\ \exists h.\, pv \overset{s}{\mapsto} h \wedge I\ g\ h\ \}\ \ f(x)\ \ \{\ \exists h'\ g_\Delta.\, pv \overset{s}{\mapsto} h' \wedge I\ (g \bullet g_\Delta)\ h' \wedge f^\sharp\ x\ res\ g\ g_\Delta\ \}@\mathcal{P} \tag{25}$$

The spec allows the input heap h to change to h'. The resource invariant I has to be preserved, up to a change of the auxiliary state, from g to $g \bullet g_\Delta$. $f^\sharp$ is a client-supplied predicate which specifies f. We call it *validity predicate*; it is functional with respect to g_Δ, and relates the input value v, the result value res, the initial auxiliary state g and the "auxiliary delta" g_Δ resulting from the invocation of f. For instance, if f were a sequential push operation on stacks, with g and g_Δ being set to histories τ and τ_Δ, we might choose the following validity predicate:

$$\text{push}^\sharp\ x\ res\ \tau\ \tau_\Delta \ \widehat{=}\ res = () \wedge \tau_\Delta = t^{\tau}_{fresh} \mapsto (l, x :: l), \tag{26}$$

where $l = \tau[\text{last}(\tau)]$. That is, $\text{push}^\sharp$ fixes the result of push to be unit and its effect to be the singleton history describing the action of pushing.

For the `flatCombine` spec, we need two auxiliary predicates. NoReq indicates that the thread *tid* does not request help. $\cdot \hookrightarrow (\cdot)$, generalizes (5) from histories to PCM $\mathbb{U}$.

$$\begin{aligned} \text{NoReq}(tid) \ &\widehat{=}\ fc \overset{s}{\mapsto} (\{tid\}, \cancel{\text{Own}}, -) \wedge a_p[tid] = \text{Init} \\ fc \hookrightarrow (g_S, g_O, g) \ &\widehat{=}\ fc \overset{s}{\mapsto} (-, -, g_S) \wedge fc \overset{o}{\mapsto} (-, -, g_O) \wedge g \sqsubseteq \textstyle\bigodot_{i=1}^{n} g_p[i] \bullet g_S \bullet g_O \end{aligned} \tag{27}$$

Here, the partial order $\sqsubseteq$ on PCM elements is defined as $g_1 \sqsubseteq g_2 \ \widehat{=}\ \exists g, g_2 = g_1 \bullet g$. It generalizes the relation $\sqsubseteq$ from histories to the PCM $\mathbb{U}$, and in the specs captures that the value g_1 was "current" before g_2.

The spec for `flatCombine` is given *wrt.* a specific thread id *tid*.

$$\begin{gathered} \{pv \overset{s}{\mapsto} \text{empty} * fc \hookrightarrow (\mathbb{1}, -, g) \wedge \text{NoReq}(tid)\} \\ \texttt{flatCombine}(f, x) : B \\ \left\{\exists g'\ g_\Delta.\, pv \overset{s}{\mapsto} \text{empty} * fc \hookrightarrow (g_\Delta, -, g') \wedge \text{NoReq}(tid) \wedge g \sqsubseteq g' \wedge f^\sharp\ x\ res\ g'\ g_\Delta\right\}@\mathcal{P} \bowtie \mathcal{F} \end{gathered} \tag{28}$$

[8] Thus, we do not require f to be sequential (*i.e.*, in addition to just manipulating the privately-owned state, f can also allocate new concurroids via hiding, and fork children threads), but every sequential function can be given a spec in $\mathcal{P}$.

A call to `flatCombine` starts and ends in a state in which the thread *tid* does not request the help (`NoReq`), and in which $\mathtt{g}$ names the sum total of the contributions. It does not change the privately-owned heap, but increases self-contribution by amount of an auxiliary delta $\mathtt{g}_\Delta$. The mediating value $\mathtt{g}'$ is a sum-total of the contributions at the moment when the thread received help; thus, $f^\sharp\ x\ \mathtt{res}\ \mathtt{g}'\ \mathtt{g}_\Delta$. As $\mathtt{g}'$ is current sometime after the initial $\mathtt{g}$, the spec postulates $\mathtt{g} \sqsubseteq \mathtt{g}'$. Due to space limitations, we omit a detailed discussion on verification of the spec (28) of the flat combiner (it can be found in [27, Appendix E] or in the accompanying Coq files).

To strengthen the analogy with coarse-grained CSL-style locks, let us note that if one were to implement a procedure $\mathtt{coarseGrainedCombine}(f, x) = \{\mathtt{lock}();\ f(x);\ \mathtt{unlock}()\}$, its specification would be the same as (28), modulo the `NoReq` conjunct and the join with all $\mathtt{g}_p[i]$ components in (27), which would not be present in the coarse-grained case, as they are artefacts of the helping machinery.[9]

5.3 Instantiating the flat combiner for stacks

To illustrate that the abstract spec for the flat combiner follows the expected intuition, we consider an instance where $\mathtt{g_s}, \mathtt{g_o}, \mathtt{g}_p$ are histories, and f is the sequential `push` method for stacks, satisfying the generic sequential spec (25) with the validity predicate $\mathrm{push}^\sharp$ defined by (26) and the stack invariant (16). So by instantiating (28), after some simplification, we obtain:

$$\begin{array}{c} \left\{\mathsf{pv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{fc} \hookrightarrow (\mathsf{empty}, -, \tau) \wedge \mathtt{NoReq}(\mathit{tid})\right\} \\ \mathtt{flatCombine}(\mathrm{push}, e) : \mathsf{Unit} \\ \left\{\exists t\ l.\ \mathsf{pv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{fc} \hookrightarrow (t \mapsto (l, e :: l), -, \tau) \wedge \tau < t \wedge \mathtt{NoReq}(\mathit{tid})\right\} \end{array} \tag{29}$$

Note that (29) is very similar to the spec (17) for Treiber `push`; the only difference, again, is in the FC-specific components such as thread id's, the `NoReq` predicate, and the lock status views used in the definition of `NoReq`. Thus, the spec (28) is adequate. A similar derivation can be done for an FC-specification of pop.

6 Related and future work

Histories are a recurring idea in the semantics of shared-memory concurrency, in one form or another. For example, the classical Brookes' semantics [2] uses *traces* to give a model for CSL. Traces are similar to histories, but do not contain time stamps. The explicit time-stamping makes it straightforward to define a merge (*i.e.*, join) for histories, and endows them with PCM structure. While Brookes uses traces in the semantics, we use histories in the specs.

Temporal reasoning about shared-memory concurrent programs has also been employed before. For example, O'Hearn *et al.* [24] advocate *hindsight lemmas* to directly and elegantly capture the intuition about linearizability of a class of concurrent data structures. In this paper, we put histories to use in ordinary Hoare-style specs. This avoids the relational reasoning about permuting traces of *two* programs, as required by linearizability, but is strong enough to provide Hoare logic specs that are expressive, and capable of abstracting granularity. In our experience, deriving history-based specs very much resembles reasoning by hindsight (*e.g.*, verifying `locate` [24] and `readPair`).

[9] To provide truly *the same* specs, we need abstract predicates to hide these artefacts. As abstract predicates are easily available in Coq, we omit the further discussion.

HLRG by Fu *et al.* is a Hoare logic for concurrency that admits history-based assertions [11]. However, their histories are hard-coded into the logic. In contrast, our histories are just a specific PCM, that one can use to instantiate the general framework of FCSL. This affords greater flexibility: if history-based specifications are not needed (*e.g.*, the incrementation example [22]), they do not have to be used. HLRG defines separating conjunction $*$ over histories as follows: conjoined histories must have equal length, and their corresponding entry heaps are merged via disjoint union. In contrast, our histories are not required to have heaps in the codomain. One can choose an arbitrary datatype to capture what is important for an example at hand.

Bell *et al.* use a variant of concurrent separation logic augmented with a monoid of *sets of histories* to reason about programs with asynchronous communication via channels [1]. Their logic is tailored for producer/consumer pattern (similar to the example we have considered in Section 4), and it features dedicated produce/consume predicates PHist and CHist defined for a particular channel and a set of histories. However, without time-stamping, Bell *et al.*'s sets of histories do not enjoy the unifomity with heaps, hence, they are a subject of a series of dedicated inference rules.

Gotsman *et al.* use temporal reasoning to verify several concurrent memory reclamation algorithms using the notion of *grace period* [12]. Their logic extends RGSep [34] with a very specific notion of histories, which live in the shared state. In contrast, we use histories not as shared, but as private auxiliary state, following the self/other dichotomy. This enables us to directly reuse the frame rule and other logical infrastructure from the separation logic FCSL, without any extensions.

Several recent approaches, such as Turon *et al.*'s CaReSL [31] (which also verifies the flat combiner), and the logic of Liang and Feng (L&F) [20] support granularity abstraction by unifying Hoare-style reasoning with linearizability and contextual refinement. In contrast, in this paper, we argue that a form of granularity abstraction achieved by these works can already be obtained *without* relying on linearizability. Instead, by using histories, one obtains Hoare-style specs which hide the fine-grained nature of the underlying programs. This can be done in a simple Hoare logic (and we reuse FCSL off-the-shelf), whereas CaReSL and L&F require significant additional logical infrastructure [21, 32], as linearizability is a stronger property than our specs. One example of the additional infrastructure has to do with helping (*e.g.*, in the flat combiner), where these logics consider the refined effectful commands as resources, and make them subject to ownership transfer [31]. While on the surface there is a similarity between commands-as-resources and histories-as-resources, there are also significant differences. Commands-as-resources are about executing specification-level programs (and an effectful abstract program, once executed, cannot be "re-executed", since it has reached a value), while histories are about what has transpired. Unlike commands-as-resources, histories also contain information about the order in which something happened in the form of timestamps, thus enabling temporal reasoning by hindsight [24]. Histories have a PCM structure, whereas commands-as-resources do not. Hence, histories in FCSL are subject to the same set of inference rules as heaps, in contrast to commands-as-resources which requires a number of dedicated inference rules.

Many of our history-based proofs are very close in spirit to proofs of linearizability (*e.g.*, the proofs of Treiber stack in Section 4 compared to the proofs in L&F [20]), since

adding an entry to a self-history can be seen as linearizing an effectful operation. However, we obtain some simplification in the proofs of pure methods such as `readPair`. In particular, L&F and related logics require *prophecy variables* [26] (or, equivalently, *speculations* [20, 32]) in their proofs of `readPair`, but we do not. We do expect, however, that prophecy variables will be required in examples where the shape of the event to be inserted into the history cannot be fully determined at the moment when it logically takes place (*e.g.*, Harris *et al.*'s MCAS [33]). We plan to address such examples in the future work, by choosing another history-based PCM; that of branching-time histories, in contrast to the linear-time ones used here.

In this work, we argued for the abstraction of granularity via the singleton histories of the form $t \mapsto (s_1, s_2)$, which describe the atomic changes in the abstract state, although other ways are possible to express what it means for a program to behave "like an atomic one" in a setting of a Hoare-style logic.

In particular, a different approach to express atomicity abstraction is suggested by da Rocha Pinto *et al.*'s logic TaDA [5] (a successor of the Concurrent Abstract Predicates framework (CAP) [6]) using the notion of an "atomic Hoare triple" of the form $\langle p \rangle\, c\, \langle q \rangle$, where the precondition p is required to be stable, whereas q is not. TaDA proposes a *make_atomic* command and a number of related inference rules, which allow one to specify *synchronized changes* of auxiliary resources across *several* shared regions. The changes themselves do not have to be physically atomic; it is sufficient that they appear atomic from the point of view of specs. TaDA's assertions range over *atomic tracking* resources, similar to the operations-as-resources [20, 31]. Unlike histories, these resources do not have the PCM structure, and thus require special treatment in TaDA's metatheory. The atomic tracking resources are not subject of ownership transfer, which is why TaDA currently does not support reasoning about helping.

Yet another view of atomicity abstraction and canonical concurrent specifications, which also bypasses linearizability, is advocated by Svendsen *et al.* in a series of papers on Higher-Order and Impredicative Concurrent Abstract Predicates [28, 29]. Both HOCAP and iCAP leverage the idea, originated by Jacobs and Piessens [17], of parametrizing specs of concurrent data types by a user-provided auxiliary code. Such auxiliary code can be seen as a callback, which, when invoked at some point during the execution of a specified method, changes the values of auxiliary resources in several regions simultaneously. Thus, when proving a parametrized spec, one should locate a right moment to invoke the provided auxiliary code, so its precondition would be ensured and the postcondition handled properly, a reasoning similar to locating a linearization point. The use of the first-class auxiliary code can introduce circularity in the domain underlying the logic—the issue tackled in HOCAP by means of indirection via "region types" and resolved in iCAP by providing a (non-elementary) model in the topos of trees. One difference between iCAP and TaDA is that *make_atomic* in TaDA presents a more *localized* view of atomicity, whereas the specs in iCAP have to predict the uses of the data structure, and provide hooks for callbacks. The hooks lead to somewhat indirect specs, and propagate client-side information into the reasoning about the structure.

We have not considered either of these two ways of exploiting abstract atomicity in the current paper, but plan to add *make_atomic* to FCSL in the future work. The challenge will be to generalize *make_atomic* to work with different notions of histories

(*e.g.*, branching-time histories may be useful, as mentioned above). We believe that the PCM approach (together with subjectivity), neither of which is exploited by TaDA and iCAP, will be beneficial in that respect. In particular, we plan to use PCMs to generalize the notion of logical atomicity afforded by histories, that we explored in this paper. Given a PCM $\mathbb{U}$, the element $x \in \mathbb{U}$ is *prime* if it cannot be represented as $x = x_1 \bullet x_2$, for non-unit x_1, x_2. For example, in the PCM of heaps, the prime elements are the singleton heaps. In the PCM of natural numbers with multiplication, the prime elements are the prime numbers. In the PCM of histories, the prime elements are the singleton histories $t \mapsto a$. A program can be considered logically atomic if it augments the self-owned portion of its state by a prime element, or by a unit. According to this definition, all the examples presented in this paper are atomic. We expect it should be possible to soundly apply *make_atomic* to programs that are atomic in this logical sense.

7 Conclusion

In this work we proposed using specifications over auxiliary state in the form of histories as means of providing general and expressive specifications for fine-grained concurrent data structures in a separation style logic.

Histories satisfy the algebraic properties of PCMs, and thus can directly reuse the underlying infrastructure from an employed separation logic, such as its assertion logic and frame rule, enabling a separation logic style of local reasoning about histories that has usually been reserved for heaps. Moreover, as we illustrated with the formalization of the flat combiner Section 5, the concept of ownership transfer from separation logic, when specialized to the PCM of histories, captures the design pattern of helping.

In addition to the flat combiner, we have verified a number of benchmark fine-grained structures, such as the pair snapshot structure, and the Treiber stack. The novelty of the specs and the proofs is that they all rely in an essential way on the subjective dichotomy between self and other auxiliary state, in order to directly relate the result of a program execution with the interference of other threads. Such explicit dichotomy provides for what we consider very concise proofs, as demonstrated by our implementation in Coq.

Acknowledgements. We thank the anonymous ESOP 2015 reviewers for their feedback. This research was partially supported by Ramon y Cajal grant RYC-2010-0743.

References

1. C. J. Bell, A. W. Appel, and D. Walker. Concurrent separation logic for pipelined parallelization. In *SAS*, volume 6337 of *LNCS*. Springer, 2010.
2. S. Brookes. A semantics for concurrent separation logic. *Th. Comp. Sci.*, 375(1-3), 2007.
3. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.
4. A. Cerone, A. Gotsman, and H. Yang. Parameterised Linearisability. In *ICALP*, volume 8573 of *LNCS*, 2014.
5. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, volume 8586 of *LNCS*, 2014.
6. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, volume 6183 of *LNCS*, 2010.
7. T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS*, volume 6015 of *LNCS*, 2010.
8. X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

9. X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, volume 4421 of *LNCS*, 2007.
10. I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52), 2010.
11. M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, volume 6269 of *LNCS*, 2010.
12. A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP*, volume 7792 of *LNCS*, 2013.
13. A. Gotsman and H. Yang. Linearizability with Ownership Transfer. In *CONCUR*, volume 7454 of *LNCS*, 2012.
14. D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.
15. M. Herlihy and N. Shavit. *The art of multiprocessor programming*. M. Kaufmann, 2008.
16. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3), 1990.
17. B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, 2011.
18. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.
19. R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, 2013.
20. H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, 2013.
21. H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.
22. A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP*, volume 8410 of *LNCS*, 2014.
23. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Th. Comp. Sci.*, 375(1-3), 2007.
24. P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC*, 2010.
25. S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5), 1976.
26. S. Qadeer, A. Sezgin, and S. Tasiran. Back and forth: Prophecy variables for static verification of concurrent programs. Technical Report MSR-TR-2009-142, 2009.
27. I. Sergey, A. Nanevski, and A. Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. Extended Version and Supporting Material. Available from `http://ilyasergey.net/projects/histories`.
28. K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, volume 8410 of *LNCS*, 2014.
29. K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, volume 7792 of *LNCS*, 2013.
30. R. K. Treiber. Systems programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.
31. A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, 2013.
32. A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, 2013.
33. V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.
34. V. Vafeiadis and M. J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*, volume 4703 of *LNCS*, 2007.