A Separation Logic for Fictional Sequential Consistency

Filip Sieczkowski[1], Kasper Svendsen[1],
Lars Birkedal[1], and Jean Pichon-Pharabod[2]

[1] Aarhus University
{filips, ksvendsen, birkedal}@cs.au.dk
[2] University of Cambridge
Jean.Pichon@cl.cam.ac.uk

Abstract. To improve performance, modern multiprocessors and programming languages typically implement *relaxed* memory models that do not require all processors/threads to observe memory operations in the same order. To relieve programmers from having to reason directly about these relaxed behaviors, languages often provide efficient synchronization primitives and concurrent data structures with stronger high-level guarantees about memory reorderings. For instance, locks usually ensure that when a thread acquires a lock, it can observe all memory operations of the releasing thread, prior to the release. When used correctly, these synchronization primitives and data structures allow clients to recover a *fiction* of a *sequentially consistent* memory model.
In this paper we propose a new proof system, iCAP-TSO, that captures this fiction formally, for a language with a TSO memory model. The logic supports reasoning about libraries that directly exploit the relaxed memory model to achieve maximum efficiency. When these libraries provide sufficient guarantees, the logic hides the underlying complexity and admits standard separation logic rules for reasoning about their more high-level clients.

1 Introduction

Modern multiprocessors and programming languages typically implement *relaxed* memory models that allow the processor and compiler to reorder memory operations. While these reorderings cannot be observed in a sequential setting, they *can* be observed in the presence of concurrency. Relaxed memory models help improve performance by allowing more agressive compiler optimizations and avoiding unnecessary synchronization between processes. However, they also make it significantly more difficult to write correct and efficient concurrent code: programmers now have to explicitly enforce the orderings they rely on, but enforcing too much ordering negates the performance benefits of the relaxed memory model.

To help programmers, several languages [2, 1] provide standard libraries that contain efficient synchronization primitives and concurrent data structures. These constructs restrict the reordering of low-level memory operations in order to express more high-level concepts, such as acquiring or releasing a lock, or pushing and popping an element from a stack. For instance, the collections provided by `java.util.concurrent` enforce that memory operations in a first thread prior to adding an element to a collection cannot be reordered past the subsequent removal by a second thread. Provided the library is used correctly, these high-level guarantees suffice for clients to recover a fiction of a sequentially consistent memory model, without introducing unnecessary synchronization in client code.

The result is a two-level structure: At the low-level we have libraries that directly exploit the relaxed memory model to achieve maximum efficiency, but enforce enough ordering to provide a fiction of sequential consistency; at the high-level we have clients that use these libraries for synchronization. While we have to reason about relaxed behaviors when reasoning about low-level libraries, ideally we should be able to use standard reasoning for the high-level clients. In this paper we propose a new proof system, iCAP-TSO, specifically designed to support this two-level approach, for a language with a TSO memory model.

We focus on TSO for two reasons. Firstly, while the definition of TSO is simple, reasoning about TSO programs is difficult, especially modular reasoning. Reasoning therefore greatly benefits from a program logic, in particular with the fiction of sequential consistency we provide. Moreover, a logic specifically tailored for TSO allows us to reason about idioms that are valid under TSO but not necessarily under weaker memory models, such as double-checked initialization (see examples).

In the TSO memory model, each thread is connected to main memory via a FIFO store buffer, modeled as a sequence of (address, value) pairs, see, e.g., [21]. When a value is written to an address, the write is recorded in the writing thread's store buffer. Threads can commit these buffered writes to main memory at any point in time. When reading from a location, a thread first consults its own store buffer; if it contains buffered writes to that location, then the thread reads the value of its last buffered write to that location; otherwise, it consults main memory. Each thread thus has its own *subjective* view of the current state of memory, which might differ from other threads'.

In contrast, in a sequentially consistent memory model, threads read and write directly to main memory and thus share an *objective* view of the current state of the memory. In separation logics for languages with sequentially consistent memory models we thus use assertions such as $\mathsf{x} \mapsto 1$, which express an *objective* property of the value of location x. Since in the TSO setting each thread has a subjective view of the state, in order to preserve the standard proof rules for reading and writing, we need a *subjective* interpretation of pre- and postconditions. The first component of our proof system, the SC logic (for sequentially consistent), provides exactly this kind of subjective interpretation.

In the SC logic we use specifications of the form $\{\mathsf{P}\}\ \mathsf{e}\ \{\mathsf{r.Q}\}$, which express that if e is executed by some thread t from an initial state that satisfies P *from the point of view* of t and e terminates with some value v, then the terminal state satisfies $\mathsf{Q}[\mathsf{v}/\mathsf{r}]$ from the point of view of thread t. Informally, an assertion

P holds from the point of view of a thread t if the property holds in a heap updated with t's store buffer. Additionally, to ensure that other threads' store buffers cannot invalidate the property, no store buffer other than t's can contain buffered writes to the parts of the heap described by P. In particular, $\mathsf{x} \mapsto \mathsf{v}$ holds from the point of view of thread t, if the value of x that t can observe is v. We shall see that this interpretation justifies the standard separation logic read and write rules.

What about transfer of resources? In separation logics for sequentially consistent memory models, assertions about resources are objective and can thus be transferred freely between threads. However, since assertions in the SC logic are interpreted subjectively, they may not hold from the point of view of other threads. To transfer resources between threads, their views of the resources must match. Thus, the SC logic is not expressive enough to reason about implementations of low-level concurrency primitives. To verify such data structures, we use the TSO logic, which allows us to reason about the complete TSO machine state, including store buffers. Importantly, in cases where the data structure provides enough synchronization to transfer resources between two threads, we can verify the implementation against an SC specification. This gives us a *fiction of sequential consistency* and allows us to reason about the *clients* of such data structures using the SC logic.

Example. To illustrate, consider a simple spin-lock library with acquire and release methods. We can specify the lock in the SC logic as follows.

$$
\begin{aligned}
&\exists \mathrm{isLock}, \mathrm{locked} : \mathsf{Prop}_{\mathsf{sc}} \times \mathsf{Val} \to \mathsf{Prop}_{\mathsf{sc}}.\ \forall \mathsf{R} : \mathsf{Prop}_{\mathsf{sc}}.\ \mathsf{stable}(\mathsf{R}) \ \Rightarrow \\
&\qquad\quad \{\mathsf{R}\}\ \mathsf{Lock}()\ \{\mathsf{r}.\ \mathrm{isLock}(\mathsf{R}, \mathsf{r})\} \\
&\qquad \wedge\ \{\mathrm{isLock}(\mathsf{R}, \textbf{this})\}\ \mathsf{Lock.acquire}()\ \{\mathrm{locked}(\mathsf{R}, \textbf{this}) * \mathsf{R}\} \\
&\qquad \wedge\ \{\mathrm{locked}(\mathsf{R}, \textbf{this}) * \mathsf{R}\}\ \mathsf{Lock.release}()\ \{\top\} \\
&\qquad \wedge\ \mathsf{valid}(\forall \mathsf{x} : \mathsf{Val}.\ \mathrm{isLock}(\mathsf{R}, \mathsf{x}) \Leftrightarrow \mathrm{isLock}(\mathsf{R}, \mathsf{x}) * \mathrm{isLock}(\mathsf{R}, \mathsf{x})) \\
&\qquad \wedge\ \forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathrm{isLock}(\mathsf{R}, \mathsf{x})) \wedge \mathsf{stable}(\mathrm{locked}(\mathsf{R}, \mathsf{x}))
\end{aligned}
$$

Here $\mathsf{Prop}_{\mathsf{sc}}$ is the type of propositions of the SC logic, and isLock and locked are thus abstract representation predicates. The predicate $\mathrm{isLock}(\mathsf{R}, \mathsf{x})$ expresses that x is a lock protecting the resource invariant R, while $\mathrm{locked}(\mathsf{R}, \mathsf{x})$ expresses that the lock x is indeed locked. Acquiring the lock grants ownership of R, while releasing the lock requires the client to relinquish ownership of R. Since the resource invariant R is universally quantified, this is a very strong specification; in particular, the client is free to instantiate R with any SC proposition. This specification requires the resource invariant to be stable, $\mathsf{stable}(\mathsf{R})$. The reason is that R could in general refer to shared resources and to reason about shared resources we need to ensure we only use assertions that are closed under interference from the environment. This is what stability expresses.

Note that this specification is expressed in the SC logic and the specification of the acquire method thus grants ownership of the resource R from the caller's point of view. Likewise, the release method only requires that R holds from the caller's point of view. This specification thus provides a fiction of sequential consistency, by allowing transfer of SC resources. Crucially, since the lock specification is an SC specification, we can reason about the clients that use it to transfer resources *entirely* using the standard proof rules of the SC logic. We illustrate this by verifying a shared bag in Section 3.

Using the TSO logic we can verify that an *efficient* spin-lock implementation satisfies this specification. The spin-lock that we verify is inspired by the Linux spin-lock implementation [3], which allows the release to be buffered. To verify the implementation we must prove that between releasing and acquiring the lock, the releasing and acquiring threads' views of the resource R match. Intuitively, this is the case because if R holds from the point of view of the releasing thread, once the buffered release makes it to main memory, R holds objectively. This style of reasoning relies on the ordering of buffered writes. To capture this intuition, we introduce a new operator in the TSO logic for expressing such ordering dependencies. This operator supports modular reasoning about many of the ordering dependencies that arise naturally in TSO-optimized data structures. In Section 5 we illustrate how to use the TSO logic to verify the spin-lock and briefly discuss other case studies we have verified.

iCAP iCAP-TSO builds on iCAP [23], a recent extension of higher-order separation logic [5] for modular reasoning about concurrent higher-order programs with shared mutable state. While the meta-theory of iCAP is intricate, understanding it is not required to understand this paper. By building on iCAP, we can use higher-order quantification to express abstract specifications that abstract over both internal data representations and client resources (such as the resource invariant R in the lock specification). This is crucial for modular verification, as it allows libraries and clients to be verified *independently* against abstract specifications and thus to scale verification to large programs. In addition, by abstractly specifying possible interference from the environment, iCAP allows us to reason about shared mutable state without having to consider all possible interleavings.

Summary of Contributions. We provide a new proof system, iCAP-TSO, for a TSO memory model, which features:

– a novel logic for reasoning about low-level *racy* code, called the TSO logic; this logic features new connectives for expressing ordering dependencies introduced by store buffers;
– a *standard* separation logic, called the SC logic, that allows *simple* reasoning for clients that transfer resources through libraries that provide sufficient synchronization;
– a notion of *fiction of sequential consistency* which allows us to provide SC specifications for libraries that provide synchronized resource transfer, even if the implementations exhibit relaxed behaviors.

Moreover, we prove soundness of iCAP-TSO. We use the logic to verify *efficient* spin-lock and bounded ticket lock implementations, double-checked initialization that uses a spin-lock internally, a circular buffer, and Treiber's stack against SC specifications. Crucially, this means that we can reason about clients of these libraries *entirely* using standard separation logic proof rules!

$$\begin{aligned}\mathsf{Val} \ni \mathsf{v} ::= &\ \mathsf{x} \mid \textbf{null} \mid \textbf{this} \mid \mathsf{o} \mid \mathsf{n} \mid \mathsf{b} \mid \textbf{()}\\ \mathsf{Exp} \ni \mathsf{e} ::= &\ \mathsf{v} \mid \textbf{let}\ \mathsf{x} = \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2 \mid \textbf{if}\ \mathsf{v}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ \mathsf{e}_2 \mid \textbf{new}\ \mathsf{C}(\bar{\mathsf{v}})\\ &\mid \mathsf{v.f} \mid \mathsf{v}_1\mathsf{.f} := \mathsf{v}_2 \mid \mathsf{v.m}(\bar{\mathsf{v}}) \mid \textbf{CAS}(\mathsf{v}_1\mathsf{.f}, \mathsf{v}_2, \mathsf{v}_3) \mid \textbf{fence} \mid \textbf{fork}(\mathsf{v.m})\end{aligned}$$

Fig. 1. Syntax of the programming language. In the definition of values, n ranges over machine integers, b over booleans, and o over object references. In the definition of expressions, f ranges over the field names, and m over the method names.

Outline. In Section 2 we introduce the programming language that we reason about and its operational semantics. Section 3 illustrates how the fiction of sequential consistency allows us to reason about shared resources using standard separation logic. Section 4 introduces the TSO logic and connectives for reasoning about store buffers. In Section 5 we illustrate the use of the TSO logic to verify an efficient spin-lock and briefly discuss the other case-studies we have verified. In Section 6 we discuss the iCAP-TSO soundness theorem. Finally, in Sections 7 and 8 we discuss related work and future work and conclude. Details and proofs can be found in the accompanying technical report [22]. The technical report is available online at `http://cs.au.dk/~filips/icap-tso-tr.pdf`.

2 Language

We build our logic for a simple, class-based programming language. For simplicity of both semantics and the logic, the language uses let-bindings and expressions, but we keep it relatively low-level by ensuring that all the values are machine-word size. The values include variables, natural numbers, booleans, unit, object references (pointers), the null pointer and the special variable **this**. The expressions include values, let bindings, conditionals, constructor and method calls, field reads and writes, atomic compare-and-swap expressions, a fork call and an explicit fence instruction. The syntax of values and expressions is shown in Figure 1. The class and method definitions are standard and therefore omitted; they can be found in the accompanying technical report.

To simplify the construction of the logic, we follow the Views framework [10] and split the operational semantics into two components. The first is a thread-local small-step semantics labeled with actions that occur during the step, the second — an action semantics that defines the effect of each action on the machine state, which in our case consists of the heap and the store buffer pool. In the thread-local semantics, a thread, which consists of a thread identifier and an expression, takes a single step of evaluation to a finite set of threads that contains besides the original thread also the threads spawned by this step. It also emits the action that describes the interaction with the memory. For instance, the Write rule in Figure 2 applies when the expression associated with thread t is an assignment (possibly in some evaluation context). It reduces by replacing the assignment with a unit value, and emits a write action that states that thread t wrote the value v to the field f of object o.

$$\frac{}{(t, E[\mathsf{o.f} := \mathsf{v}]) \xrightarrow{write(t,\mathsf{o},\mathsf{f},\mathsf{v})} \{(t, E[\mathbf{()}])\}}\ \textsc{Write} \qquad \frac{}{(t, E[\mathsf{o.f}]) \xrightarrow{read(t,\mathsf{o},\mathsf{f},\mathsf{v})} \{(t, E[\mathsf{v}])\}}\ \textsc{Read}$$

$$\frac{}{(t, E[\mathbf{CAS}(\mathsf{o.f}, \mathsf{v_o}, \mathsf{v_n})]) \xrightarrow{cas(t,\mathsf{o},\mathsf{f},\mathsf{v_o},\mathsf{v_n},\mathsf{r})} \{(t, E[\mathsf{r}])\}}\ \text{CAS} \qquad \frac{}{(t, \mathsf{e}) \xrightarrow{flush(t)} \{(t, \mathsf{e})\}}\ \textsc{Flush}$$

$$\frac{\mathsf{body}(C, m) = (\mathsf{unit}\ m() = e) \qquad t \neq t'}{(t, E[\mathbf{fork}(\mathsf{o.m})]) \xrightarrow{fork(t,o,C,t')} \{(t, E[\mathbf{()}]), (t', e[o/\mathbf{this}])\}}\ \textsc{Fork}$$

Fig. 2. Selected cases of the thread-local operational semantics.

The non-fault memory state consists of a heap — a finite map from pairs of an object reference and a field to semantic values (i.e., all the values that are not variables) — and a store buffer pool, which contains a sequence of buffered updates for each of the finitely many thread identifiers. The memory can also be in a fault state (written $\lightning$), which means that an error in the execution of the program has occurred. The action semantics interprets the actions as functions from memory states to sets of memory states: if it is impossible for the action to occur in a given state, the result is an empty set; if, however, the action may occur in the given state but it would be erroneous, the result is the fault state. Consider the write action emitted by reducing an assignment. In Figure 3 we can see the interpretation of the action: there are three distinct cases. The action is successful if there is a store buffer associated with the thread that emitted the action and the object is allocated in the heap, and has the appropriate field. In this case, the write gets added to the end of the thread's buffer. However, the write action can have two additional outcomes: if there is no store buffer associated with the thread in the store buffer pool, the initial state had to be ill-formed, and so the interpretation of the action is an empty set; however, if the thread is defined, but the reference to the field $o.f$ is not found in the heap, the execution will fault.

The state of a complete program consists of the thread pool and a memory state, and is consistent if the memory state is a fault, or the domain of the store buffer pool equals the domain of the thread pool. The complete semantics proceeds by reducing one of the threads using the thread-local semantics, then interpreting the resulting action with the action semantics, and reducing to a memory state in the resultant set:

$$\frac{t \in \operatorname{dom} T \qquad (t, T(t)) \xrightarrow{a} T' \qquad \mu' \in [\![a]\!](\mu)}{(\mu, T) \to (\mu', (T - t) \uplus T')}$$

Note how in some cases, notably read, this might require "guessing" the return value, and checking that the guess was right using the action semantics. Some of the cases of the semantics are written out in Figures 2 and 3. In particular, we show the reduction and action semantics that correspond to the (nondeterministic) flushing of a store buffer: a flush action can be emitted by a thread at any time, and the action is interpreted by flushing the oldest buffered write to

$$\llbracket \mathit{read}(t,o,f,v)\rrbracket(h,U) =$$
$$\begin{cases} \{(h,U)\} & \text{if } (o,f) \in \operatorname{dom} h \text{ and } \operatorname{lookup}(o.f, U(t), h) = v \\ \emptyset & \text{if } t \notin \operatorname{dom} U \text{ or } (o,f) \in \operatorname{dom} h \text{ and } \operatorname{lookup}(o.f, U(t), h) \neq v \\ \{\lightning\} & \text{if } (o,f) \notin \operatorname{dom} h \end{cases}$$
$$\llbracket \mathit{write}(t,o,f,v)\rrbracket(h,U) =$$
$$\begin{cases} \{(h, U[t \mapsto U(t) \cdot (o,f,v)])\} & \text{if } (o,f) \in \operatorname{dom} h \text{ and } t \in \operatorname{dom} U \\ \{\lightning\} & \text{if } (o,f) \notin \operatorname{dom} h \\ \emptyset & \text{if } t \notin \operatorname{dom} U \end{cases}$$
$$\llbracket \mathit{cas}(t,o,f,v_o,v_n,r)\rrbracket(h,U) =$$
$$\begin{cases} \{(\operatorname{flush}(h, U(t) \cdot (o,f,v_n)), U[t \mapsto \varepsilon])\} & \text{if } (o,f) \in \operatorname{dom} h,\ r = \textbf{true} \\ & \text{and } \operatorname{lookup}(o.f, U(t), h) = v_o \\ \{(\operatorname{flush}(h, U(t)), U[t \mapsto \varepsilon])\} & \text{if } (o,f) \in \operatorname{dom} h,\ r = \textbf{false} \\ & \text{and } \operatorname{lookup}(o.f, U(t), h) \neq v_o \\ \{\lightning\} & \text{if } (o,f) \notin \operatorname{dom} h \\ \emptyset & \text{otherwise} \end{cases}$$
$$\llbracket \mathit{flush}(t)\rrbracket(h,U) =$$
$$\begin{cases} \{(h[(o,f) \mapsto v], U[t \mapsto \alpha])\} & \text{if } U(t) = (o,f,v) \cdot \alpha \text{ and } (o,f) \in \operatorname{dom} h \\ \emptyset & \text{if } t \notin \operatorname{dom}(U),\ U(t) = \varepsilon, \text{ or } (o,f) \notin \operatorname{dom} h \end{cases}$$

Fig. 3. Selected cases of the action semantics. The lookup function finds the newest value associated with the field, including the store buffer, while the flush function applies all the updates from the store buffer to the heap in order.

the memory. Note also the rules for the compare-and-swap expression: similarly to reading, the return value has to be guessed by the thread-local semantics. However, whether the guess matches the state of the memory or not, the *whole* content of the store buffer is written to main memory. Moreover, if the compare-and-swap succeeds, the update resulting from it is also written to main memory. Thus, this expression can serve as a synchronization primitive.

Note that our operational semantics is the usual TSO semantics of the x86 [21] adapted to a high-level language. The only difference is that we have a notion of allocation of objects, which does not exist in the processor-level semantics. Our semantics allocates the new object directly on the heap to avoid different threads trying to allocate the same object.

3 Reasoning in the SC logic

The SC logic of iCAP-TSO allows us to reason about code that always uses synchronization to transfer resources *using standard separation logic*, without having to reason about store buffers. Naturally, this also includes standard mutable data structures without any sharing. We can thus easily verify a list library in the SC logic against the standard separation logic specification as it enforces a *unique* owner. Crucially, within the SC logic we can also use libraries that provide synchronized resource transfer. For instance, we can use the specification of the spin-lock from the Introduction and the fiction of sequential consistency that it provides. We illustrate this point by verifying a shared bag library, implemented as a list protected by a lock.

The SC logic. The SC logic is an intuitionistic higher-order separation logic. Recall that the SC logic features Hoare triples of the form $\{\mathsf{P}\}\ \mathsf{e}\ \{\mathsf{r.\ Q}\}$, where P and Q are SC assertions. Formally, SC assertions are terms of type $\mathsf{Prop}_{\mathsf{SC}}$. SC assertions include the usual connectives and quantifiers of higher-order separation logic and language specific assertions such as points-to, $\mathsf{x.f} \mapsto \mathsf{v}$, for asserting the value of field f of object x.

Recall that SC triples employ a subjective interpretation of the pre- and postcondition: $\{\mathsf{P}\}\ \mathsf{e}\ \{\mathsf{r.\ Q}\}$ expresses that if thread t executes the expression e from an initial state where P holds from the point of view of thread t and e terminates with value v then $\mathsf{Q}[\mathsf{v}/\mathsf{r}]$ holds for the terminal state from the point of view of thread t. An assertion P holds from the point of view of a thread t if P's assertions about the heap hold from the point of view of t's store buffer and main memory *and* no other thread's store buffer contains a buffered write to these parts of the heap. The assertion $\mathsf{x.f} \mapsto \mathsf{v}$ thus holds from the point of view of thread t if

- the value of the most recently buffered write to $\mathsf{x.f}$ in t's store buffer is v
- or t's store buffer does not contain any buffered writes to $\mathsf{x.f}$ and the value of $\mathsf{x.f}$ in main memory is v

and no other threads store buffer contains a buffered write to $\mathsf{x.f}$. The condition that no other thread's store buffer can contain a buffered write to $\mathsf{x.f}$ ensures that flushing of store buffers cannot invalidate $\mathsf{x.f} \mapsto \mathsf{v}$ from the point of view of a given thread.

If $\mathsf{x.f} \mapsto \mathsf{v}$ holds from the point of view of thread t and thread t attempts to read $\mathsf{x.f}$ it will thus read the value v either from main memory or its own store buffer. Likewise, if $\mathsf{x.f} \mapsto \mathsf{v}_1$ holds from the point of view of thread t and thread t writes v_2 to $\mathsf{x.f}$, afterwards $\mathsf{x.f} \mapsto \mathsf{v}_2$ holds from the point of view of thread t. We thus get the standard rules for reading and writing to a field in our SC logic:

$$\frac{}{\{\mathsf{x.f} \mapsto \mathsf{v}\}\ \mathsf{x.f}\ \{\mathsf{r.\ x.f} \mapsto \mathsf{v} * \mathsf{r} = \mathsf{v}\}}\ \textsc{S-Read} \qquad \frac{}{\{\mathsf{x.f} \mapsto \mathsf{v}_1\}\ \mathsf{x.f} := \mathsf{v}_2\ \{\mathsf{r.\ x.f} \mapsto \mathsf{v}_2\}}\ \textsc{S-Write}$$

Using SC specifications: a shared bag To illustrate how we can use the lock specification from the Introduction, consider a shared bag implemented using a list. Each shared bag maintains a list of elements and a lock to ensure exclusive access to the list of elements. Each bag method acquires the lock and calls the corresponding method of the list library before releasing the lock.

We take the following specification, which allows unrestricted sharing of the bag, to be our specification of a shared bag. This is not the most general specification we can express — we discuss a more general specification of a stack in Appendix A — but it suffices to illustrate that verification of the shared bag against such specifications is standard. Since the specification allows unrestricted sharing (the bag predicate is duplicable), no client can know the contents of the bag; instead, the specification allows clients to associate ownership of additional resources (expressed using the predicate P) with each element in the bag.

$$
\begin{aligned}
&\exists \mathrm{bag} : \mathsf{Val} \times (\mathsf{Val} \to \mathsf{Prop_{sc}}) \to \mathsf{Prop_{sc}}.\ \forall \mathsf{P} : \mathsf{Val} \to \mathsf{Prop_{sc}}.\\
&\quad (\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathsf{P}(\mathsf{x})))\ \Rightarrow\\
&\qquad \{\top\}\ \mathsf{Bag}()\ \{\mathsf{r}.\ \mathrm{bag}(\mathsf{r}, \mathsf{P})\}\ \wedge\\
&\qquad \{\mathrm{bag}(\textbf{this}, \mathsf{P}) * \mathsf{P}(\mathsf{x})\}\ \mathsf{Bag.push}(\mathsf{x})\ \{\top\}\ \wedge\\
&\qquad \{\mathrm{bag}(\textbf{this}, \mathsf{P})\}\ \mathsf{Bag.pop}()\ \{\mathsf{r}.\ (\mathsf{P}(\mathsf{r}) \vee \mathsf{r} = \textbf{null})\}\ \wedge\\
&\qquad \forall \mathsf{x} : \mathsf{Val}.\ \mathsf{valid}(\mathrm{bag}(\mathsf{x}, \mathsf{P}) \Leftrightarrow \mathrm{bag}(\mathsf{x}, \mathsf{P}) * \mathrm{bag}(\mathsf{x}, \mathsf{P}))
\end{aligned}
$$

Pushing an element x thus requires the client to transfer ownership of $\mathsf{P}(\mathsf{x})$ to the bag. Likewise, either pop returns **null** or the client receives ownership of the resources associated with the returned element.

To verify the implementation against this specification, we first have to define the abstract bag representation predicate. To define bag we first need to choose the resource invariant of the lock. Intuitively, the lock owns the list of elements and the resources associated with the elements currently in the list. This is expressed by the following resource invariant $\mathsf{R_{bag}}(\mathsf{xs}, \mathsf{P})$, where xs refers to the list of elements.

$$\mathsf{R_{bag}}(\mathsf{xs}, \mathsf{P}) \overset{\mathrm{def}}{=} \exists \mathsf{l} : \mathrm{list}\ \ \mathsf{Val}.\ \mathrm{lst}(\mathsf{xs}, \mathsf{l}) * \circledast_{\mathsf{y} \in mem(\mathsf{l})} \mathsf{P}(\mathsf{y})$$

The bag predicate asserts *read-only* ownership of the lock and elms fields, and that the lock field refers to a lock with the above resource invariant.

$$\mathrm{bag}(\mathsf{x}, \mathsf{P}) \overset{\mathrm{def}}{=} \exists \mathsf{y}, \mathsf{xs} : \mathsf{Val}.\ \ \mathsf{x.lock} \overset{-}{\mapsto} \mathsf{y} * \mathsf{x.elms} \overset{-}{\mapsto} \mathsf{xs} * \mathrm{isLock}(\mathsf{R_{bag}}(\mathsf{xs}, \mathsf{P}), \mathsf{y})$$

Now, we are ready to verify the bag methods. The most interesting method is pop, as it actually returns the resources associated with the elements it returns. A proof outline of pop is presented below. The crucial thing to note is that since locks introduce sufficient synchronization, they can mediate ownership transfer. Thus, once a thread t acquires the lock, it receives the resource invariant $\mathsf{R_{bag}}(\mathsf{xs}, \mathsf{P})$ *from the point of view of* t. Since it now owns the list, t can call the List.pop method, and finally — again using the fiction of sequential consistency provided by the lock — release the lock.

```
class Bag {
 Lock lock; List elms;
 Object pop() =
     $\{\mathrm{bag}(\textbf{this}, \mathsf{P})\}$
     let x = this.lock in let xs = this.elms in x.acquire();
     $\{\textbf{this}.\mathsf{elms} \overset{-}{\mapsto} \mathsf{xs} * \mathrm{locked}(\mathsf{R_{bag}}(\mathsf{xs}, \mathsf{P}), \mathsf{x}) * \mathsf{R_{bag}}(\mathsf{xs}, \mathsf{P})\}$
     let z = xs.pop() in
     $\{\mathrm{locked}(\mathsf{R_{bag}}(\mathsf{xs}, \mathsf{P}), \mathsf{x}) * \mathsf{R_{bag}}(\mathsf{xs}, \mathsf{P}) * (\mathsf{z} = \textbf{null} \vee \mathsf{P}(\mathsf{z}))\}$
```

```
    x.release();
    {isLock(R_bag(xs, P), x) * (z = null ∨ P(z))}
    z
    {r. r = null ∨ P(r)}
  ...
}
```

This example illustrates the general pattern that we can use to verify clients of libraries that provide fiction of sequential consistency. As long as these clients only transfer resources using libraries that provide sufficient synchronization, the verification can proceed entirely within the SC logic.

4 TSO logic and connectives

In this section we describe the TSO logic and introduce our new TSO connectives that allow us to reason about the kinds of relaxed behaviors that occur in low-level concurrency libraries.

We can express the additional reorderings that the memory model allows by extending the space of states over which the assertions are built. In the case of our TSO model, we include the store buffer pool as an additional component of the memory state. However, reasoning about the buffers directly would be extremely unwieldy and contrary to the spirit of program logics. Hence, we introduce new logical connectives that allow us to specify this interference abstractly, and provide appropriate reasoning rules.

The triples and assertions of the TSO logic First, however, we need to consider how the TSO logic is built. As mentioned in the Introduction, its propositions extend the propositions of SC logic by adding the store buffer pool component. Just like SC assertions, this space forms a higher-order intuitionistic separation logic, with the usual rules for reasoning about assertion entailment. However, we are still reasoning about the code running in a particular thread and we often need to state properties that hold of its own store buffer. Thus, formally, the typing rule for the TSO logic triples is as follows:

$$\frac{\mathsf{P} : \mathsf{TId} \to \mathsf{Prop}_{\mathsf{TSO}} \qquad \mathsf{Q} : \mathsf{TId} \to \mathsf{Val} \to \mathsf{Prop}_{\mathsf{TSO}}}{[\mathsf{P}]\ \mathsf{e}\ [\mathsf{Q}] : \mathsf{Spec}}$$

where TId is the type of thread identifiers and Spec is the type of specifications. We usually keep this quantification over thread identifiers implicit, by introducing appropriate syntactic sugar for the TSO-level connectives. The logic also includes another family of Hoare triples, the atomic triples, with the following typing rule:

$$\frac{\mathsf{atomic(e)} \qquad \mathsf{P} : \mathsf{TId} \to \mathsf{Prop}_{\mathsf{TSO}} \qquad \mathsf{Q} : \mathsf{TId} \to \mathsf{Val} \to \mathsf{Prop}_{\mathsf{TSO}}}{\langle \mathsf{P} \rangle\ \mathsf{e}\ \langle \mathsf{Q} \rangle : \mathsf{Spec}}$$

As the rule states, these triples can only be used to reason about atomic expressions — read, write, fence, and compare-and-swap. This feature is inherited from iCAP, as a means of reasoning about the way the shared state changes at the atomic updates. We give an example of such reasoning in Section 5.

Note that the triples above use a different space of assertions than the SC triples introduced in Section 3. Hence, in order to provide the fiction of sequential consistency and prove SC specifications for implementations whose correctness involves reasoning about buffered updates, we need to use of both of these spaces in the TSO logic. To this end we define two embeddings of $\mathsf{Prop}_{\mathsf{SC}}$ into $\mathsf{Prop}_{\mathsf{TSO}}$.

The subjective embedding The subjective embedding is denoted $\ulcorner - \textbf{ in } - \urcorner : \mathsf{Prop}_{\mathsf{SC}} \times \mathsf{TId} \to \mathsf{Prop}_{\mathsf{TSO}}$. Intuitively, $\ulcorner \mathsf{P} \textbf{ in } t \urcorner$ means that P holds from the perspective of thread t — including the possible buffered updates in the store buffer of t, but forbidding buffered updates that "touch" P by other threads. Thus, it means that if the buffer of thread t is flushed to the memory, P will hold in the resulting state. Note that this corresponds to the interpretation of the assertions in the SC triples.

For a concrete example of what this embedding means, consider an assertion $\mathsf{x.f} \mapsto \mathsf{v} : \mathsf{Prop}_{\mathsf{SC}}$. Clearly, we can use our embedding to get $\ulcorner \mathsf{x.f} \mapsto \mathsf{v} \textbf{ in } t \urcorner$. This assertion requires that the reference $\mathsf{x.f}$ is defined, and there are no buffered updates in store buffers of threads other than t. As for t's store buffer, the last update of $\mathsf{x.f}$ has to set its value to v or, if there are no buffered updates of $\mathsf{x.f}$, the value associated with it in main memory is v. This means that from the point of view of thread t, $\mathsf{x.f} \mapsto \mathsf{v}$ holds, but from the point of view of the other threads, the only information is that $\mathsf{x.f}$ is defined.

The objective embedding The objective embedding is denoted $\ulcorner - \urcorner : \mathsf{Prop}_{\mathsf{SC}} \to \mathsf{Prop}_{\mathsf{TSO}}$. The idea is that $\ulcorner \mathsf{P} \urcorner$ holds in a state that does include store buffers if P holds in the state where we ignore the buffers *and* none of the buffers contain buffered updates to the locations mentioned by P. The intuition behind this embedding is that P should hold *in main memory*, and as such from the point of view of *all* threads. This makes it very useful to express resource transfer: an assertion that holds for all threads can be transferred to any of them.

Using the points-to example again, $\ulcorner \mathsf{x.f} \mapsto \mathsf{v} \urcorner$ means precisely that the reference $\mathsf{x.f}$ is defined, its associated value in the heap is v, and there are no buffered updates in any of the store buffers to the field $\mathsf{x.f}$.

Semantics of assertions and embeddings In the following, we provide a simplified presentation of parts of the model for the interested reader, to flesh out the intuitions given above. We concentrate on the interpretation of TSO-specific constructs and elide the parts inherited from iCAP, which are orthogonal.

Following the Views framework [10], TSO assertions (terms of type $\mathsf{Prop}_{\mathsf{TSO}}$) are modeled as predicates over *instrumented states*. In addition to the underlying machine state, instrumented states contain shared regions, protocols and phantom state. The instrumented states form a Kripke model in which worlds consist of allocated regions and their associated protocols. Since iCAP-TSO inherits iCAP's impredicative protocols [23], worlds need to be recursively defined. Hence we use a meta-theory that supports the definition of sets by guarded recursion, namely the so-called internal language of the topos of trees [6]. We refer readers to the accompanying technical report [22] for details and proofs.

Propositions are interpreted as subsets of instrumented states upwards-closed wrt. extension ordering. The states are instrumented with shared regions and protocols which we inherit from iCAP. In the following these are denoted with X, and we elide their definition.

$$[\![\mathsf{Prop}_{\mathsf{TSO}}]\!] \stackrel{\text{def}}{=} \mathcal{P}^{\uparrow}(\mathit{LState} \times \mathit{SPool} \times X) \qquad [\![\mathsf{Prop}_{\mathsf{SC}}]\!] \stackrel{\text{def}}{=} \mathcal{P}^{\uparrow}(\mathit{LState} \times X)$$

In these definitions *LState* denotes the local state, including the partial physical heap, while *SPool* is the store-buffer pool that directly corresponds to the operational semantics. Note that the interpretation of $\mathsf{Prop}_{\mathsf{SC}}$ does not consider store buffer pools, only the local state and the instrumentation. This allows us to interpret the connectives at this level in a standard way.

At the level of $\mathsf{Prop}_{\mathsf{TSO}}$, we have several important connectives, namely separating conjunction, and both embeddings we have introduced before. These are defined as follows:

$$\begin{aligned}
\mathrm{lfd}(l, U) &\stackrel{\text{def}}{=} \forall t, v.\ \forall (o, f) \in \mathrm{dom}(l).\ (o, f, v) \notin U(t) \\
[\![\ulcorner \mathsf{P} \urcorner]\!] &\stackrel{\text{def}}{=} \{(l, U, x) \mid \exists l' \leq l.\ (l', x) \in [\![\mathsf{P}]\!] \wedge \mathrm{lfd}(l', U)\} \\
[\![\ulcorner \mathsf{P}\ \mathbf{in}\ t \urcorner]\!] &\stackrel{\text{def}}{=} \{(l, U, x) \mid (\mathrm{flush}(l, U(t)), U[t \mapsto \varepsilon], x) \in [\![\ulcorner \mathsf{P} \urcorner]\!]\} \\
[\![\mathsf{P} * \mathsf{Q}]\!] &\stackrel{\text{def}}{=} \{(l, U, x) \mid \exists l_1, l_2.\ l = l_1 \bullet l_2 \wedge (l_1, U, x) \in [\![\mathsf{P}]\!] \wedge (l_2, U, x) \in [\![\mathsf{Q}]\!]\}.
\end{aligned}$$

The embeddings are defined using the auxiliary "locally flushed" $\mathrm{lfd}(l, U)$ predicate, which ensures that no updates to $\mathrm{dom}(l)$ are present in U. We only require this on a sub-state of the local state to ensure good behavior with respect to the extension ordering. The subjective embedding is then defined in terms of the objective one, with all the updates in the corresponding store-buffer flushed. Finally, the separating conjunction is defined as a composition of local states. Separating conjunction does not split the instrumentation or store-buffer pool and both conjuncts thus have to hold with the same pools of buffered updates.

Reasoning about buffered updates To effectively reason about the store buffers, we need an operator that describes how the state *changes* due to an update. To this end, we define $- \mathcal{U}_{-} - : \mathsf{Prop}_{\mathsf{TSO}} \times \mathsf{TId} \times \mathsf{Prop}_{\mathsf{TSO}} \to \mathsf{Prop}_{\mathsf{TSO}}$. Because of its role, this connective has a certain temporal feel: in fact, it behaves in a way that is somewhat similar to the classic "until" operator. Intuitively, $\mathsf{P}\ \mathcal{U}_t\ \mathsf{Q}$ means that there exists a buffered update in the store buffer of thread t, such that until this update is flushed the assertion P holds, while after the update gets written to memory, the assertion Q holds. Thus, it can be used to describe the ordering dependencies introduced by the presence of store buffers. This intuition should become clearer by observing the proof rules in Figure 4 (explained in the following).

Again, let us consider a simple example. In the state described by $\ulcorner \mathsf{x.f} \mapsto 1 \urcorner\ \mathcal{U}_t\ \ulcorner \mathsf{x.f} \mapsto 2 \urcorner$, we know that the value of $\mathsf{x.f}$ in the heap is 1, and that there exists a buffered update in thread t. Before that update there are no updates to $\mathsf{x.f}$, due to the use of $\ulcorner - \urcorner$, so it has to be the first update to $\mathsf{x.f}$ in the store buffer of t. Additionally, after it gets flushed $\ulcorner \mathsf{x.f} \mapsto 2 \urcorner$ holds — so the update must set $\mathsf{x.f}$

to 2. Since the right-hand side of $\mathcal{U}_t$ also uses $\ulcorner - \urcorner$, we also know that there are no further buffered updates to $\mathsf{x.f}$. This means that the thread t can observe the value of $\mathsf{x.f}$ to be 2, while all of the other threads can observe it to be 1. Note that, since $\mathcal{U}_t$ is a binary operator on $\mathsf{Prop}_{\mathsf{TSO}}$, it is possible to use it to express multiple buffered updates.

The semantics of the until operator follow very closely the intuition given above. Note that for some assertions and states, several choices of the update would validate the conditions. However, this rarely occurs in practice due to the use of the objective embedding, which requires no updates in its footprint.

$$\begin{aligned}\llbracket \mathsf{P}\ \mathcal{U}_t\ \mathsf{Q} \rrbracket \overset{\text{def}}{=} \{(l, U, x) \mid\ & \exists \alpha, \beta, o, f, v.\ U(t) = \alpha \cdot (o, f, v) \cdot \beta\ \wedge \\ & (l, U[t \mapsto \alpha], x) \in \llbracket \mathsf{P} \rrbracket \wedge (\mathrm{flush}(l, \alpha \cdot (o, f, v)), U[t \mapsto \beta], x) \in \llbracket \mathsf{Q} \rrbracket\}\end{aligned}$$

Relating the two embeddings The two embeddings we have defined are in fact quite related. Since an assertion under an objective embedding holds from the perspective of any thread, we get $\ulcorner \mathsf{P} \urcorner \Rightarrow \ulcorner \mathsf{P}\ \textbf{in}\ t \urcorner$. We also have $\mathsf{P}\ \mathcal{U}_t\ \ulcorner \mathsf{Q} \urcorner \Rightarrow \ulcorner \mathsf{Q}\ \textbf{in}\ t \urcorner$: since there is a buffered update at which $\ulcorner \mathsf{Q} \urcorner$ starts to hold, Q holds from t's perspective.

Since most of the time we are reasoning from the perspective of a particular thread, we also include some syntactic sugar: $\mathcal{U}$ is a shorthand for an update in the current thread, while $\mathcal{U}^{\mathrm{o}}$ is a shorthand for an update in some thread *other than the current one.* We also use $\overline{\mathsf{P}}$ as a shorthand for $\ulcorner \mathsf{P}\ \textbf{in}\ t \urcorner$, where t is the current thread. To make the syntax simpler, whenever we need to refer to the thread identifier explicitly, we use an assertion $\mathsf{iam}(t)$. This is just syntactic sugar for a function $\lambda t'.\ t = t' : \mathsf{TId} \to \mathsf{Prop}_{\mathsf{TSO}}$, which allows us to bind the thread identifier of the thread we are reasoning about to a logical variable.

Reading and writing state. The presence of additional connectives that mention the state makes reading fields of an object and writing to them more involved in the TSO logic than in standard separation logic. We deal with this by introducing additional judgments that specify when we can read a value and what the result of flushing a store buffer will be. Intuitively, $\mathsf{P} \vdash_{\mathrm{rd}(t)} \mathsf{x.f} \mapsto \mathsf{v}$ specifies that thread t can read the value v from the reference $\mathsf{x.f}$ — precisely what we need for reading the state. The other new judgment, $\mathsf{P} \vdash_{\mathrm{fl}(t)} \mathsf{Q}$, means that if we flush thread t's store buffer in a state specified by P, the resulting state will satisfy Q. This action judgment is clearly useful for specifying actions that flush the store buffer: compare-and-swap and fences. However, it is also used to specify the non-flushing writes. To see this, consider the rule A-Write in Figure 4. Since the semantics of assignment will introduce a buffered update, we know that *after* this new update reaches main memory, all the other updates will also have reached it. Thus, at that point in time, Q will also hold, since it is disjoint from the reference $\mathsf{x.f}$. The other interesting rules are related to the CAS expression. In A-CAS-True, we do not need to establish the read judgment, since the form of the flush judgment ensures that the value we can observe is $\mathsf{v_o}$. Aside from that, the rule behaves like a combination of writing and flushing. The rule A-CAS-False, on the other hand, requires a separate read judgment. This is because it does not perform an assignment, and so the current value does not

$$\frac{}{\triangleright\ulcorner \mathsf{x.f} \mapsto \mathsf{v}\ \textbf{in}\ t\urcorner \vdash_{\mathrm{rd}(t)} \mathsf{x.f} \mapsto \mathsf{v}}\ \textsc{Rd-Ax} \qquad \frac{\mathsf{P} \vdash_{\mathrm{rd}(t)} \mathsf{x.f}\mapsto \mathsf{v} \qquad t\neq t'}{\mathsf{P}\ \mathcal{U}_{t'}\ \mathsf{Q} \vdash_{\mathrm{rd}(t)} \mathsf{x.f}\mapsto \mathsf{v}}\ \textsc{Rd-}\mathcal{U}\textsc{-NEq}$$

$$\frac{}{\triangleright\ulcorner \mathsf{P}\ \textbf{in}\ t\urcorner \vdash_{\mathrm{fl}(t)} \ulcorner \mathsf{P}\urcorner}\ \textsc{Fl-Ax} \qquad \frac{\mathsf{P}(t)\vdash_{\mathrm{fl}(t)} \ulcorner \mathsf{x.f}\mapsto -\urcorner * \mathsf{Q}(t)}{\langle \mathsf{P} * \mathsf{iam}(t)\rangle\ \mathsf{x.f} := \mathsf{v}\ \langle \_.\ \mathsf{P}\ \mathcal{U}\ (\mathsf{Q}*\ulcorner \mathsf{x.f}\mapsto \mathsf{v}\urcorner)\rangle^{C}}\ \textsc{A-Write}$$

$$\frac{\mathsf{Q}\vdash_{\mathrm{fl}(t)} \mathsf{R}}{P\ \mathcal{U}_t\ \mathsf{Q} \vdash_{\mathrm{fl}(t)} \mathsf{R}}\ \textsc{Fl-}\mathcal{U}\textsc{-Eq} \qquad \frac{\mathsf{P}(t)\vdash_{\mathrm{rd}(t)} \mathsf{x.f}\mapsto \mathsf{v}}{\langle \mathsf{P}*\mathsf{iam}(t)\rangle\ \mathsf{x.f}\ \langle \mathsf{r.}\ \mathsf{P} * \mathsf{r} = \mathsf{v}\rangle^{C}}\ \textsc{A-Read}$$

$$\frac{\mathsf{P}(t)\vdash_{\mathrm{fl}(t)}\ulcorner \mathsf{x.f}\mapsto \mathsf{v_o}\urcorner * \mathsf{Q}(t)}{\langle \mathsf{P}*\mathsf{iam}(t)\rangle\ \mathbf{CAS}(\mathsf{x.f},\mathsf{v_n},\mathsf{v_o})\ \langle \mathsf{r.}\ \mathsf{r} = \mathbf{true} * \mathsf{Q} * \ulcorner \mathsf{x.f}\mapsto \mathsf{v_n}\urcorner\rangle^{C}}\ \textsc{A-CAS-True}$$

$$\frac{\mathsf{P}(t)\vdash_{\mathrm{rd}(t)} \mathsf{x.f}\mapsto \mathsf{v} \qquad \mathsf{P}(t)\vdash_{\mathrm{fl}(t)} \mathsf{Q}(t) \qquad \mathsf{v}\neq \mathsf{v_o}}{\langle \mathsf{P}*\mathsf{iam}(t)\rangle\ \mathbf{CAS}(\mathsf{x.f},\mathsf{v_n},\mathsf{v_o})\ \langle \mathsf{r.}\ \mathsf{r}=\mathbf{false}*\mathsf{Q}\rangle^{C}}\ \textsc{A-CAS-False}$$

$$\frac{\langle \mathsf{P}\rangle\ \mathsf{e}\ \langle \mathsf{Q}\rangle^{C} \qquad \mathsf{atomic}(\mathsf{e})}{[\mathsf{P}]\ \mathsf{e}\ [\mathsf{Q}]}\ \textsc{A-Start} \qquad \frac{[\mathsf{P}]\ \mathsf{e_1}\ [\mathsf{r.}\ \mathsf{Q}(\mathsf{r})] \qquad [\mathsf{Q}(\mathsf{x})]\ \mathsf{e_2}\ [\mathsf{r.}\ \mathsf{R}(\mathsf{r})]}{[\mathsf{P}]\ \textbf{let}\ \mathsf{x} = \mathsf{e_1}\ \textbf{in}\ \mathsf{e_2}\ [\mathsf{r.}\ \mathsf{R}(\mathsf{r})]}\ \textsc{Bind}$$

$$\frac{[\mathsf{P}]\ \mathsf{e}\ [\mathsf{Q}] \qquad \mathsf{stable}(\mathsf{R})}{[\mathsf{P}*\mathsf{R}]\ \mathsf{e}\ [\mathsf{Q}*\mathsf{R}]}\ \textsc{Frame} \qquad \frac{[\overline{\mathsf{P}}]\ \mathsf{e}\ [\overline{\mathsf{Q}}]}{\{\mathsf{P}\}\ \mathsf{e}\ \{\mathsf{Q}\}}\ \textsc{S-Shift}$$

Fig. 4. Selected rules of the TSO logic.

need to appear in the right-hand side of the flush assumption, like in A-Write and A-CAS-True rules.

Also of interest are some of the proof rules for the read and flush judgments. Note how in rules Rd-Ax and Fl-Ax the *later* operator ($\triangleright$) appears. This arises from the fact that the model is defined using guarded recursion to break circularities, and later is used as a guard. However, since the guardedness is tied to operational semantics through step-indexing and atomic expressions always take one evaluation step, *later* can be removed at the atomic steps of the proof, as expressed by the rules. Moreover, the rules also match the intuition we gave about the store buffer related connectives. First, the judgment means that all the updates in t's store buffer are flushed: thus, it is enough to know $\ulcorner \mathsf{P}\ \textbf{in}\ t\urcorner$ holds to get $\ulcorner \mathsf{P}\urcorner$ in Fl-Ax, and similarly we only look to the right-hand side of $\mathcal{U}$ in the rule Fl-$\mathcal{U}$-Eq. Note also, that we can reason about updates buffered in other threads, as evidenced by the rule Rd-$\mathcal{U}$-NEq, where we "ignore" the buffered update and read from the left-hand side of $\mathcal{U}$.

Stability and stabilization. There is one potentially worrying issue in the definition of the $\mathcal{U}_t$ operator given in this section: since at any point in the program a *flush* action can occur nondeterministically, how can we know that there still exists a buffered update as asserted by $\mathcal{U}_t$? After all, it might have been flushed to the memory. This is the question of *stability*[3] of the until operator — and the answer is that it is unstable by design. The rationale behind this choice is simple: Suppose we had made it stable by allowing the possibility that

[3] Recall an assertion is stable, if it cannot be invalidated by the environment.

the buffered update has already been flushed. Then, if we were to read a field that had a buffered write to it in a different thread, we would not know whether the write was still buffered or had been flushed, and so we would not know what value we read. With the current definition, when we read, we know that the update is still buffered and so the result of the read is known. However, we only allow reasoning with unstable assertions in the *atomic* triples, i.e., when reasoning about a single read, write or compare-and-swap expression. Hence, we need a way to make $\mathcal{U}$ stable. For this reason, we define an explicit *stabilization* operator, $\llparenthesis - \rrparenthesis$. It is a closure operator, which means we have $\mathsf{P} \vdash \llparenthesis \mathsf{P} \rrparenthesis$. Moreover, for stable assertions, the other direction, $\llparenthesis \mathsf{P} \rrparenthesis \vdash \mathsf{P}$, also holds. The important part, however, is how stabilization behaves with respect to $\mathcal{U}$: provided P and Q are stable, we have $\llparenthesis \mathsf{P}\, \mathcal{U}_t\, \mathsf{Q} \rrparenthesis \dashv\vdash (\mathsf{P}\, \mathcal{U}_t\, \mathsf{Q}) \vee \mathsf{Q}$. This does indeed correspond to our intuition — even for stable assertions P and Q, the interference can flush the buffered update that is asserted in the definition of $\mathcal{U}$, which would transition to a state in which Q holds. However, since P and Q are stable, this is also the *only* problem that the interference could cause.

Explicit stabilization has been explored before in separation logic, most often in connection with rely-guarantee reasoning. In particular, Wickerson studies explicit stabilization in RGSep in his PhD thesis [27, Chapter 3], and Ridge [19] uses it to reason about x86-TSO.

Semantically, stability is defined through the notion of *interference*, which expresses the effect that the environment can have on a state. In iCAP-TSO there are two classes of interference. Firstly, other threads can concurrently change the state of shared regions. This source of interference is inherited from iCAP; we reason about it by considering the states of shared regions, and the transitions the environment is allowed to make. As an example, after releasing a lock we cannot be certain it remains unlocked, since other threads could concurrently acquire it. The protocol for a lock is described in Section 5. A second class of interference is related to the TSO nature of our semantics, and includes the interference that arises in the memory system: we refer to this class as *store-buffer* interference. It is defined through three possible actions of the memory system: allocation of a new store-buffer (which happens when a fork command gets executed), adding a new buffered update to a location outside the assertion's footprint to a store-buffer, and committing the oldest buffered update from one of the buffers. Stability under allocation of new store-buffers and under buffering new updates is never a problem. Most of the connectives we use are also stable under flushing — both embeddings are specifically designed in this way. As we mentioned above, $\mathcal{U}$ is unstable under flushing by design, and we stabilize it explicitly. For the formal definition of the interference relation we refer the reader to the technical appendix [22].

Interpretation of the SC logic. As we have already mentioned, the intuition that lies behind the SC logic, discussed in the previous section, is precisely expressed by the $\ulcorner - \textbf{ in } t \urcorner$ embedding. This is more formally captured by the rule S-SHIFT in Figure 4 (recall $\overline{\mathsf{P}}$ is syntactic sugar for $\lambda t.\ \ulcorner \mathsf{P} \textbf{ in } t \urcorner$), which states that the two ways of expressing that a triple holds from the perspective of

```
Lock {
  bool locked;
  Lock() = this.locked := false; fence; this
  unit acquire() =
    let x = CAS(this.locked, true, false) in
      if x then () else acquire()
  unit release() = this.locked := false
}
```

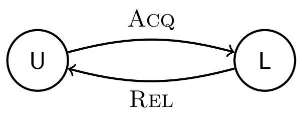

Fig. 5. Left: spin-lock implementation. Right: lock protocol.

the current thread are equivalent. In fact, we take this rule as the *definition* of the SC triples, and so we can prove that the SC triples actually form a standard separation logic by proving that the proof rules of the SC logic correspond to admissible rules in the TSO logic. This is expressed by the following theorem:

Theorem 1 (Soundness of SC logic). *The SC logic is sound wrt. its interpretation within the TSO logic, i.e., the proof rules of the SC logic composed with the rule* S-SHIFT *are admissible rules of the TSO logic.*

For most of the proof rules, the soundness follows directly; the only ones that require additional properties to be proved are the frame, consequence, and standard quantifier rules, which additionally require the following property:

Lemma 1. *The embeddings* $\ulcorner - \urcorner$ *and* $\ulcorner - \textbf{ in } t \urcorner$ *distribute over quantifiers and separating conjunction, and preserve entailment.*

The formal statement of this property, along with the proof, can be found in the accompanying technical report.

5 Reasoning in the TSO logic

In Section 3 we illustrated that the fiction of sequential consistency provided by the lock specification allows us to reason about shared mutable data structures shared through locks, without explicitly reasoning about the underlying relaxed memory model. Of course, to verify a lock implementation against this lock specification, we *do* have to reason about the relaxed memory model. In this section we illustrate how to achieve this using our TSO logic. We focus on the use of the TSO connectives introduced in Section 4 to describe the machine states of the spin-lock and elide the details related to the use of concurrent abstract predicates.

The spin-lock implementation that we wish to verify is given in Figure 5. It uses a compare-and-swap (CAS) instruction to attempt to acquire the lock, but only a primitive write instruction to release the lock. While CAS flushes the store buffer of the thread that executes the CAS, a plain write does not. To verify this implementation, we thus have to explicitly reason about the possibility of buffered releases in store buffers.

Specification. In the Introduction we introduced a lock specification expressed in our SC logic. When verifying the spin-lock implementation, we actually verify the implementation against the following slightly stronger specification, from which we can easily derive the SC specification.

$$\begin{aligned}
&\exists \mathrm{isLock}, \mathrm{locked} : \mathsf{Prop}_{\mathsf{sc}} \times \mathsf{Val} \to \mathsf{Prop}_{\mathsf{sc}}.\ \forall \mathsf{R} : \mathsf{Prop}_{\mathsf{sc}}.\ \mathsf{stable}(\mathsf{R}) \ \Rightarrow \\
&\qquad [\overline{\mathsf{R}}]\ \mathsf{Lock}()\ [\mathsf{r}.\ \overline{\mathrm{isLock}(\mathsf{R}, \mathsf{r})}] \\
&\quad \wedge\ [\overline{\mathrm{isLock}(\mathsf{R}, \mathsf{x})}]\ \mathsf{Lock.acquire}()\ [\overline{\mathrm{locked}(\mathsf{R}, \mathsf{x})} * \ulcorner \mathsf{R} \urcorner] \\
&\quad \wedge\ [\overline{\mathrm{locked}(\mathsf{R}, \mathsf{x}) * \mathsf{R}}]\ \mathsf{Lock.release}()\ [\top] \\
&\quad \wedge\ \mathsf{valid}(\forall \mathsf{x} : \mathsf{Val}.\ \mathrm{isLock}(\mathsf{R}, \mathsf{x}) \Leftrightarrow \mathrm{isLock}(\mathsf{R}, \mathsf{x}) * \mathrm{isLock}(\mathsf{R}, \mathsf{x})) \\
&\quad \wedge\ \forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathrm{isLock}(\mathsf{R}, \mathsf{x})) \wedge \mathsf{stable}(\mathrm{locked}(\mathsf{R}, \mathsf{x}))
\end{aligned}$$

Note that this stronger specification is expressed using TSO triples. This specification of the acquire method is slightly stronger: this specification asserts that upon termination of acquire, the resource invariant R holds in main memory and there are no buffered writes affecting R in *any* store buffer ($\ulcorner \mathsf{R} \urcorner$). The weaker SC specification only asserts that the resource invariant R holds from the point of view of the acquiring thread and that there are no buffered writes affecting R in *any of the other threads'* store buffers ($\overline{\mathsf{R}}$).

Lock protocol. To verify the spin-lock implementation against the above specification, we first need to define the abstract representation predicates isLock and locked. Following CAP [11] and iCAP [23], to reason about sharing iCAP-TSO extends separation logic with shared regions, with protocols governing the resources owned by each shared region. In the case of the spin-lock, upon allocation of a new spin-lock the idea is to allocate a new shared region governing the state of the spin-lock and ownership of the resource invariant.

Conceptually, a spin-lock can be in one of two states: locked and unlocked. In iCAP-TSO we express this formally using the transition system in Figure 5. This labeled transition system specifies an abstract model of the lock. To relate it to the concrete implementation, for each abstract state (L and U), we choose an assertion that describes the resources the lock owns in that state.

Since acquiring the lock flushes the store buffer of the acquiring thread, the locked state is fairly simple. In the locked state the spin-lock owns the locked field, which contains the value true in main memory and there are no buffered writes to locked in any store buffer. The spin-lock x with resource invariant R thus owns the resources described by $I_{\mathsf{L}}(\mathsf{x}, \mathsf{R}, \mathsf{n})$ in the abstract locked state.

$$I_{\mathsf{L}}(\mathsf{x}, \mathsf{R}, \mathsf{n}) = \ulcorner \mathsf{x.locked} \mapsto \mathsf{true} \urcorner$$

Due to the possibility of buffered releases in store buffers, the unlocked state is more complicated. In the unlocked state,

– either locked is false in main memory and there are no buffered writes to locked in any store buffer
– or locked is true in main memory, and there is *exactly one* store buffer with a buffered write to locked, and the value of this buffered write is false

Furthermore, in case there is a buffered write to locked that changes its value from true to false, then, once the buffered write reaches main memory, the resource invariant holds in main memory. Since the resource invariant must hold from the point of view of the releasing thread before the lock is released any buffered writes affecting the resource invariant must reach main memory before the buffered release. We can express this ordering dependency using the until operator:

$$I_{\mathsf{U}}(\mathsf{x}, \mathsf{R}, \mathsf{n}) = \exists \mathsf{t} : \mathsf{TId}.\ (\!| \ulcorner \mathsf{x.locked} \mapsto \mathsf{true} \urcorner\, \mathcal{U}_{\mathsf{t}}\, \ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner |\!)$$

Here $[\textsc{Rel}]_1^{\mathsf{n}}$ is a CAP action permission used to ensure that only the current holder of the lock can release it. Since this is orthogonal to the underlying memory model, we refer the interested reader to the technical report [22] for details.

Since both arguments of $\mathcal{U}_t$ are stable, as explained in Section 4, $I_{\mathsf{U}}(\mathsf{x}, \mathsf{R}, \mathsf{n})$ is equivalent to the following assertion.

$$\begin{aligned} &\exists \mathsf{t} : \mathsf{TId}.\ \ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner \ \vee \\ &\qquad\qquad (\ulcorner \mathsf{x.locked} \mapsto \mathsf{true} \urcorner\, \mathcal{U}_t\, \ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner) \end{aligned}$$

The first disjunct corresponds to the case where the release has made its way to main memory and the second disjunct to the case where it is still buffered.

The definition of isLock in terms of I_{L} and I_{U} now follows iCAP.[4] The isLock predicate asserts the existence of a shared region governed by the above labeled transition system, where the resources owned by the shared region in the two abstract states are given by I_{L} and I_{U}. It further asserts that the abstract state of the shared region is either locked or unlocked and also a non-exclusive right to acquire the lock.

Proof outline. To verify the spin-lock implementation, it remains to verify each method against the specification instantiated with the concrete isLock and locked predicates. To illustrate the reasoning related to the relaxed memory model, we focus on the verification of the $\mathsf{acquire}$ method and the compare-and-swap instruction in particular. The full proof outline is given in the accompanying technical report.

As the name suggests, the resources owned by a shared region are shared between all threads. Atomic instructions are allowed to access and modify resources owned by shared regions, provided they follow the protocol imposed by the region. In the case of the spin-lock, the spin-lock region owns the shared locked field and we thus need to follow the spin-lock protocol to access and modify the locked field. Since the precondition of $\mathsf{acquire}$ asserts that the lock is either in the locked or unlocked state, we need to consider two cases.

If the spin-lock region is already locked, then the compare-and-swap fails and we remain in the locked state. This results in the following proof obligation:

$$\langle \triangleright I_{\mathsf{L}}(\textbf{this}, \mathsf{R}, \mathsf{n}) * \mathsf{iam}(t) \rangle\ \textbf{CAS}(\textbf{this}.\mathsf{locked},\ \textbf{true},\ \textbf{false})\ \langle \mathsf{r}.\ \triangleright I_{\mathsf{L}}(\textbf{this}, \mathsf{R}, \mathsf{n}) * \mathsf{iam}(t) * \mathsf{r} = \mathsf{false} \rangle$$

That is, if locked contains the value true from the point of view of a thread t, then CAS'ing from false to true in thread t will fail. This is easily shown to hold by rule A-CAS-False.

[4] See the accompanying technical report for a formal definition of isLock.

If the spin-lock region is unlocked, then the compare-and-swap may or may not succeed, depending on whether the buffered release has made it to main memory and which thread performed the buffered release. If it succeeds, the acquiring thread transitions the shared region to the locked state and takes ownership of the resource invariant; otherwise, the shared region remains in the unlocked state. This results in the following proof obligation:

$$\langle \triangleright I_{\mathsf{U}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * \mathsf{iam}(t)\rangle\ \mathbf{CAS}(\mathbf{this}.\mathsf{locked},\ \mathbf{true},\ \mathbf{false})\ \langle \mathsf{r}.\ \exists \mathsf{y}.\ \triangleright I_{\mathsf{y}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * \mathsf{iam}(t) * Q(\mathsf{y}, \mathsf{r}, \mathsf{n})\rangle$$

where $Q(\mathsf{y}, \mathsf{r}, \mathsf{n}) \overset{\text{def}}{=} (\mathsf{y} = \mathsf{U} * \mathsf{r} = \mathsf{false}) \vee (\mathsf{y} = \mathsf{L} * [\textsc{Rel}]_1^{\mathsf{n}} * \ulcorner \mathsf{R} \urcorner * \mathsf{r} = \mathsf{true})$. Rewriting the explicit stabilization to a disjunction and commuting in $\triangleright$, this reduces to the following proof obligation:

$$\begin{array}{l}\langle \mathsf{iam}(t) * (\exists \mathsf{t}' : \mathsf{TId}.\ \triangleright \ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner \vee \\ \quad (\triangleright \ulcorner \mathsf{x.locked} \mapsto \mathsf{true} \urcorner\, \mathcal{U}_{\mathsf{t}'}\, \triangleright \ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner))\rangle \\ \quad \mathbf{CAS}(\mathbf{this}.\mathsf{locked},\ \mathbf{true},\ \mathbf{false}) \\ \langle \mathsf{r}.\ \exists \mathsf{y} \in \{\mathsf{U}, \mathsf{L}\}.\ \triangleright I_{\mathsf{y}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * \mathsf{iam}(t) * Q(\mathsf{y}, \mathsf{r}, \mathsf{n})\rangle\end{array}$$

In case the second disjunct holds and there exist buffered releases in the store buffer of t', the CAS will succeed if executed by thread t' and fail if executed by any other thread. To prove this obligation, we thus do case analysis on whether t' is our thread or not, i.e., whether $\mathsf{t} = \mathsf{t}'$. This leaves us with three proof obligations (after strengthening the post-condition):

– either the buffered release is in our store buffer

$$\begin{array}{l}\langle (\triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner\, \mathcal{U}\, \triangleright \ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner) * \mathsf{iam}(t)\rangle \\ \quad \mathbf{CAS}(\mathbf{this}.\mathsf{locked},\ \mathbf{true},\ \mathbf{false}) \\ \langle \mathsf{r}.\ \triangleright I_{\mathsf{L}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * [\textsc{Rel}]_1^{\mathsf{n}} * \ulcorner \mathsf{R} \urcorner * \mathsf{iam}(t) * \mathsf{r} = \mathsf{true}\rangle\end{array}$$

– or in some other thread's store buffer

$$\begin{array}{l}\langle (\triangleright \ulcorner \mathbf{this}.\mathsf{locked} \mapsto \mathsf{true} \urcorner\, \mathcal{U}^{o}\, \triangleright \ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner) * \mathsf{iam}(t)\rangle \\ \quad \mathbf{CAS}(\mathbf{this}.\mathsf{locked},\ \mathbf{true},\ \mathbf{false}) \\ \langle \mathsf{r}.\ \triangleright I_{\mathsf{U}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * \mathsf{iam}(t) * \mathsf{r} = \mathsf{false}\rangle\end{array}$$

– or it has already been flushed

$$\begin{array}{l}\langle \triangleright \ulcorner \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]_1^{\mathsf{n}} \urcorner * \mathsf{iam}(t)\rangle \\ \quad \mathbf{CAS}(\mathbf{this}.\mathsf{locked},\ \mathbf{true},\ \mathbf{false}) \\ \langle \mathsf{r}.\ \triangleright I_{\mathsf{L}}(\mathbf{this}, \mathsf{R}, \mathsf{n}) * [\textsc{Rel}]_1^{\mathsf{n}} * \ulcorner \mathsf{R} \urcorner * \mathsf{iam}(t) * \mathsf{r} = \mathsf{true}\rangle\end{array}$$

These three proof obligations are easily discharged using rules A-CAS-True and A-CAS-False.

Note that our logic makes us consider exactly those four cases that intuitively one has to consider when reasoning operationally in TSO.

Logical atomicity and relaxed memory

Although shared-memory concurrency introduces opportunity for threads to interfere, concurrent data structures are often written to ensure that all operations provided by the library are *observably*, or *logically atomic*. That is, for clients of the concurrent data structure, any concurrent execution of operations provided by the library should behave *as if* it occurred in *some* sequential order. This property immensely simplifies client-side reasoning, since the clients need not reason about any internal states of the library. One way of ensuring logical atomicity is by using coarse-grained synchronization, for instance by wrapping the whole data structure in a lock. However, this is far from efficient, and many real-life concurrent data structures opt to use fine-grained synchronization, such as compare-and-swap, while still being logically atomic. Since the simplification of the client-side reasoning one can obtain by exploiting the logical atomicity can be significant, any truly modular proof system that supports fine-grained concurrency should support logical atomicity. This is a known and well-researched problem in the sequentially consistent setting; here we discuss its interplay with relaxed memory and sketch how our system tackles it.

One of the approaches to express logical atomicity is to develop a program logic that *internalizes* the concept, i.e., in which one can express atomicity as a specification and prove that implementations satisfy such a spec within the logic. Several of the more recent program logics go this route, in particular TaDA and iCAP [9, 23]. In this work we follow iCAP, which uses a reasonably simple specification pattern to encode abstract atomicity. The crux of the idea is for the data structure to provide an *abstract* mathematical model of its state, and to model the (possibly non-atomic) updates of the concrete state with an atomic update of the abstract state. Since the abstract state is only a model, it can be updated after *any* atomic step of the program, and thus any update of the abstract state can be considered atomic.

Since iCAP-TSO inherits most of the properties of iCAP, one could imagine that we inherit iCAP's specification pattern for logical atomicity verbatim. This, however, would lead to problems. If we ported the pattern to the relaxed setting directly, we would gain a way to express logical atomicity, but lose all the information about the flushing behavior of the data structure—and in effect we would not be able to derive an SC specification, even if the data structure provided a fiction of sequential consistency. The idea for how one can adjust the specification pattern hinges on using the SC assertions and the embeddings described earlier to describe the state of the store-buffers when the abstract update happens. We refer the interested reader to Appendix A for an explanation of how to extend the iCAP pattern to the TSO setting and the accompanying technical report [22] for the formal proofs and technical details.

Other case studies

In addition to the spin-lock, we have verified several other algorithms in the TSO logic against SC specifications. Below we discuss the challenges of each case-study. Full proofs are included in the accompanying technical report.

Treiber's stack Treiber's stack is a classic fine-grained concurrent stack implementation. We verify this data structure against a specification that provides logical atomicity, based on the one given in [23]. From this general specification we derive two classic specifications: a single-owner stack and a shared-bag that provides a fiction of sequential consistency. The challenge, as explained in the preceding section, is to provide a specification pattern that provides both logical atomicity and fiction of sequential consistency.

Double-checked initialization. Double-checked initialization [20] is a design pattern that reduces the cost of lazy initialization by having clients only use a lock if the wrapped object has not been initialized yet, to their knowledge. We verify this algorithm against a specification that ensures that the wrapped object is initialized only once. The challenge is to capture the fact that holding the lock ensures that there are no buffered updates to the object.

Ticket lock. A bounded ticket lock [17] is a fair locking algorithm where threads obtain a ticket number and wait for it to be served, and where the ticket number goes back to zero when it reaches its bound. We verify this algorithm against a specification that allows a bounded number of clients to transfer resources. The challenge is to ensure that a thread's ticket will not be skipped and reissued to another thread, despite the fact that in TSO, the increment to the serving number in the release can be buffered, as for the spinlock.

Circular buffer. A circular buffer [14] is a single-writer single-reader resource ownership transfer mechanism based on an array viewed circularly. This algorithm is interesting in TSO because it does not need any synchronisation: the FIFO behavior of store buffers is enough. Because there are no synchronisation operations, a thread can be ahead of main memory in the array, and the challenge is to ensure that despite that, the writer does not overtake the reader.

6 Soundness

We prove soundness of iCAP-TSO with respect to the TSO model of section 2. Soundness is proven by relating the machine semantics to an instrumented semantics that, for instance, enforces that clients obey the chosen protocols when accessing shared state. This relation is expressed through an erasure function, $\lfloor - \rfloor$, that erases an instrumented state to a set of machine states.

The soundness theorem is stated in terms of the following $\mathit{eval}(\mu, T, q)$ predicate, which asserts that for any terminating execution of the thread pool T from initial state μ, the predicate q must hold for the terminal state and thread pool. The *eval* predicate is defined as a guarded recursive predicate (the recursive occurrence of *eval* is guarded by $\triangleright$), to express that each step of evaluation in the machine semantics corresponds to a step in the topos of trees.

$$\begin{aligned}\mathit{eval}(\mu, T, q) \stackrel{\text{def}}{=} & (\mathrm{irr}(\mu, T) \Rightarrow (\mu, T) \in q) \wedge \\ & (\forall T', \mu'.\ (\mu, T) \to (\mu', T') \Rightarrow \triangleright \mathit{eval}(\mu', T', q))\end{aligned}$$

Here $\mathrm{irr}(\mu, T)$ means that (μ, T) is irreducible. We can now state the soundness of iCAP-TSO.

Theorem 2 (Soundness). *If* $[\mathsf{P}]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q}]$ *and* $\mu \in \lfloor [\![\mathsf{P}]\!](t) \rfloor$ *then*

$$\mathit{eval}(\mu, [t \mapsto e], \lambda(\mu', T).\ \mu' \in \lfloor [\![\mathsf{Q}]\!](t)(T(t)) \rfloor)$$

This theorem expresses that if a specification $[\mathsf{P}]\ \mathsf{e}\ [\mathsf{r}.\ \mathsf{Q}]$ holds and the execution of the thread pool $[t \mapsto e]$ with a single thread t from an initial state μ in the erasure of P terminates (including threads spawned by t), then the execution has finished in a proper terminal state (i.e., did not fault), which is in the erasure of Q instantiated with the return value $T(t)$ of thread t.

7 Related work

Our work builds directly on iCAP [23], which is an extension of separation logic for modular reasoning about concurrent higher-order programs with shared mutable state. Our work extends the model of iCAP with store buffers to implement a TSO memory model, extends the iCAP logic with TSO-connectives for reasoning about these store buffers and crucially, it reduces to standard concurrent separation logic for sequentially consistent clients.

Rely/Guarantee reasoning over operational models. Conceptually, iCAP-TSO is a Rely/Guarantee-based proof system for reasoning about an operational semantics with a relaxed memory model. This approach has also been explored by Ridge [19], Wehrman [26], and Jacobs [15].

Ridge [19] and Wehrman [26] both propose proof systems for low-level reasoning about racy TSO programs based on Rely/Guarantee reasoning. In Ridge's system [19] the Rely/Guarantee is explicit, while in Wehrman's system [26] it is expressed implicitly through a separation logic. To reason in the presence of a relaxed memory model, both systems enforce a rely that includes possible interference from write buffers. Consequently, both systems support reasoning about racy code. However, in the case where a library includes sufficient synchronization, neither system is able to take advantage of the stronger rely provided by this synchronization to simplify client proofs. This is exactly what our fiction of sequential consistency allows.

Jacobs [15] proposes to extend separation logic with "TSO spaces" for reasoning about shared resources in a TSO setting. While the exact goals of his approach remain a bit unclear, it seems that Jacobs is also aiming for a system that reduces to standard separation logic reasoning when possible. However, to ensure soundness Jacobs' proof system lacks the usual structural rules for disjunction and existentials. This results in non-standard reasoning even for non-racy clients.

Recovering sequential consistency. There are several other approaches for recovering sequentially consistent reasoning about clients in the presence of a relaxed memory model.

Cohen and Schirmer [8] propose a programming discipline based on ownership, which ensures that all TSO program behaviors can be simulated by a sequentially consistent machine. Unfortunately, the proposed discipline enforces too much synchronization. In particular, an efficient spin-lock implementation with a buffered release, like the one we verify in Section 5, does not obey their programming discipline. Their approach is thus unable to deal with such code without introducing additional synchronization.

Owens [18] defines a trace property on the set of SC behaviors of a program which ensures that all TSO behaviors can be simulated by an SC machine. Owens shows how this property allows clients of synchronization primitives to reason using SC semantics, despite *racy* implementations of these synchronization primitives. However, in contrast to our appraoch, Owens' approach is non-compositional: while Owens proves similar results for multiple synchronization primitives *in isolation*, these results do not apply to clients that *combine* two or more of these synchronization primitives.

Gotsman et al. [13] propose another approach for providing clients with a fiction of sequential consistency, based on linearizability. By relating racy library implementations on a TSO architecture with abstract specifications on an SC architecture, they can reason about data-race free clients that call racy libraries using an SC memory model. Their approach is only compositional for *non-interacting* libraries (libraries that do not interact through the heap) and further requires libraries and clients to be non-interacting. Their approach can also relate fine-grained implementations with coarse-grained implementations, which provides similar advantages to our logical atomicity.

Our approach does not suffer from the compositionality problems of [18, 13] or the need for unnecessary and potentially expensive synchronization required by [8]. In particular, iCAP-TSO allows *racy* libraries that interact through the heap to be verified *independently.*

Reasoning over axiomatic models. Relaxed memory models are often defined using relations over read and write events that enforce certain consistency/visibility constraints.

Alglave et al. [4] proposes the use of such axiomatic models to support efficient model-checking in the context of relaxed memory models. The use of an axiomatic semantics avoids the need to consider all the possible interleavings introduced by operational models with explicit buffers and caches. Alglave et al.'s approach supports fully automatic verification of simple correctness properties of realistic C code. Alglave et al.'s approach is non-modular in the sense that it only supports whole-program verification and thus lacks support for verifying modules independently.

While our logic is based on an operational model with explicit buffers, we use Rely/Guarantee reasoning to avoid the explosion in interleavings observed by Alglave et al. Our fiction of sequential consistency is specifically designed to strengthen the rely (and implicitly, reduce the number of possible interleavings that have to be considered) when the code enforces sufficient synchronization.

More recently, Turon et al. [25] has proposed GPS, a proof system over the axiomatic C11 memory model. GPS extends separation logic with per-location protocols which internalize some of the properties of the underlying visibility properties between read and write events. GPS supports two of the C11 access modes: non-atomics and release/acquire. Reasoning about non-atomics reduces to standard separation logic. However, ownership transfer requires the use of release/acquire and explicit reasoning about visibility of memory events. GPS lacks support for logical atomicity and thus cannot express canonical specifications for concurrent data structures such as the specification of Treiber's stack in Appendix A.

8 Conclusion and future work

We have presented a new proof system, iCAP-TSO, to support modular and scalable reasoning for a language with a TSO memory model. The proof system consists of two logics. The TSO logic supports reasoning about libraries with low-level racy code. In cases where the libraries provide sufficient synchronization, they can be verified against SC specifications. Clients that only do resource transfer through such libraries can then be verified entirely within the SC logic, which uses standard separation logic rules.

We use the TSO logic to verify an efficient spin-lock implementation against an SC specification. We use this to verify a shared bag library, implemented using a spin-lock, in the SC logic. We also verify a double-checked initialization wrapper, a bounded ticket lock, and a circular buffer against SC specifications. Lastly, we verify Treiber's stack against a specification that showcases how logical atomicity can be extended to TSO.

We think of iCAP-TSO as a first step towards more automated/interactive tools for reasoning about the TSO memory model. In this paper we have focused on the foundational issues of constructing a logic that allows simple reasoning for well-behaved code. As future work it would be interesting to try to extend tools like [7, 12] to support mostly automated verification in the SC logic and interactive verification in the TSO logic. We believe that the fiction of sequential consistency could be really beneficial in this area: one could imagine that the lock-free concurrency libraries would be verified by hand, while automated tools could verify properties of client programs. Since the rules of the SC logic are standard, the whole range of techniques developed for automating separation logic could be applicable. The open question here is how one could infer the instantiations of higher-order specifications, for instance the invariants for locks, and this should be investigated.

Acknowledgements

We thank Mark Batty, Aleš Bizjak, Susmit Sarkar, and Peter Sewell for helpful discussions on this work. This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

References

1. Intel threading building blocks documentation: Fenced data transfer. `https://software.intel.com/en-us/node/506122`. Accessed: 25 June 2014.
2. java.util.concurrent API. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`. Accessed: 25 June 2014.
3. Linux kernel mailing list, Nov. 1999. spin_unlock optimization(i386).
4. J. Alglave, D. Kroening, and M. Tautschnig. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *Proc. of CAV*, 2013.

5. B. Biering, L. Birkedal, and N. Torp-Smith. BI-Hyperdoctrines, Higher-order Separation Logic, and Abstraction. *ACM TOPLAS*, 2007.
6. L. Birkedal, R. Møgelberg, J. Schwinghammer, and K. Støvring. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *Proc. of LICS*, 2011.
7. A. Chlipala. Mostly-automated Verification of Low-level Programs in Computational Separation Logic. In *Proc. of PLDI*, 2011.
8. E. Cohen and B. Schirmer. From Total Store Order to Sequential Consistency: A Practical Reduction Theorem. In *Proc. of ITP*, 2010.
9. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A logic for time and data abstraction. In *Proc. of ECOOP*, 2014.
10. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proc. of POPL*, 2013.
11. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *Proc. of ECOOP*, 2010.
12. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular Shape Analysis. In *Proc. of PLDI*, 2007.
13. A. Gotsman, M. Musuvathi, and H. Yang. Show No Weakness: Sequentially Consistent Specifications of TSO Libraries. In *Proc. of DISC*, 2012.
14. D. Howells and P. E. McKenney. Circular buffers. Available at `https://www.kernel.org/doc/Documentation/circular-buffers.txt`.
15. B. Jacobs. Verifying TSO Programs. Technical report, May 2014. Report CW660.
16. B. Jacobs and F. Piessens. Expressive Modular Fine-Grained Concurrency Specification. In *Proc. of POPL*, 2011.
17. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9(1):21–65, Feb. 1991.
18. S. Owens. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *Proc. of ECOOP*, 2010.
19. T. Ridge. A Rely-Guarantee proof system for x86-TSO. In *Proc. of VSTTE*, 2010.
20. D. C. Schmidt and T. Harrison. Double-checked locking - an optimization pattern for efficiently initializing and accessing thread-safe objects, 1997. Available at `http://www.dre.vanderbilt.edu/~schmidt/PDF/DC-Locking.pdf`.
21. P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A Rigorous and Usable Programmers Model for x86 Multiprocessors. In *Comm. ACM*, 2010.
22. F. Sieczkowski, K. Svendsen, L. Birkedal, and J. Pichon-Pharabod. A Separation Logic for Fictional Sequential Consistency. Technical report, Aarhus University, 2014. `http://cs.au.dk/~filips/icap-tso-tr.pdf`.
23. K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *Proc. of ESOP*, 2014.
24. K. Svendsen, L. Birkedal, and M. Parkinson. Modular Reasoning about Separation of Concurrent Data Structures. In *Proc. of ESOP*, 2013.
25. A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *Proc. of OOPSLA*, 2014.
26. I. Wehrman. *Weak-Memory Local Reasoning.* PhD thesis, University of Texas, 2012. Dissertation draft.
27. J. Wickerson. *Concurrent verification for sequential programs.* PhD thesis, University of Cambridge, 2012.

A Logical atomicity

In Section 5 we explained how reasoning about clients of concurrent libraries can benefit if one is able to express that the library methods are *logically* atomic. In this appendix, we discuss the specification pattern adapted from iCAP that provides us this expressivity. We also discuss the technical challenges of adapting the pattern to relaxed-memory setting and reconciling logical atomicity with fiction of sequential consistency. In the following, we use Treiber's stack as a running example, in particular we show how we can derive a shared-bag specification used in Section 3 from a specification that supports both logical atomicity and fiction of sequential consistency.

Specification pattern. To formally capture the notion of *logical atomicity* in the SC setting, Svendsen et al. [24] introduced a specification pattern for concurrent data structures (under the name of granularity abstraction), inspired by previous work of Jacobs et al. [16]. The idea is to relate the non-atomic operation on the concrete data structure with an atomic operation on an abstraction of the data structure. This atomic operation is expressed as a logical update of a *phantom field*. Phantom fields are similar to auxiliary variables, in that they are only used for specification purposes; however, unlike auxiliary variables, they are updated logically rather than through assignments. Hence, they are not subject to the underlying relaxed memory model and every thread will agree on the current value of phantom fields.

By storing the current abstract state of the concurrent data structure in a phantom field we can separate reasoning about the data structure and its clients: the implementor of the data structure proves that every intermediate state within each library method is related to the current abstract state stored in the phantom field. This establishes a simulation between the concrete state and the abstract state stored in the phantom field. By sharing the phantom field between the data structure and clients, clients can thus reason about the state of the shared data structure by reasoning about the state of the phantom field. For instance, in the case of Treiber's stack, the abstract state of the stack would be the list of elements on the stack. In the case of a push operation, the implementor would thus have to prove that for each intermediate state in the execution of push there exists an abstract state — a list of elements — that corresponds to the concrete state of the data structure.

In this setup, we can say that an operation appears atomic if the abstract state changes directly from the initial abstract state to the terminal abstract state — such as in the example below.

$$\underbrace{\overset{\alpha}{\longrightarrow}\overset{\alpha}{\longrightarrow}\overset{\mathsf{x} :: \alpha}{\longrightarrow}\overset{\mathsf{x} :: \alpha}{\longrightarrow}\,\mathsf{x} :: \alpha}_{\mathsf{push}(\mathsf{x})}$$

We express this notion formally by specifying that the concrete operation simulates an arbitrary atomic update of the phantom field from the initial abstract state to the terminal abstract state. We express these atomic updates using *view shifts*. A view shift from P to Q, written $\mathsf{P} \sqsubseteq \mathsf{Q}$, expresses a logical update of the instrumented state that does not change the underlying machine state. In particular, for a phantom field $\mathsf{x_f}$, one can set it to any value, provided one has exclusive ownership of the field:

$$\mathsf{x_f} \overset{1}{\mapsto} \mathsf{v_1} \sqsubseteq \mathsf{x_f} \overset{1}{\mapsto} \mathsf{v_2}$$

Here we use $\mathsf{x_f} \overset{\pi}{\mapsto} \mathsf{v}$ to assert fractional ownership ownership of the phantom field $\mathsf{x_f}$ with fraction $\pi \in (0, 1]$ and that the field currently contains the value v. A fraction $\pi = 1$ corresponds to exclusive ownership. Ownership can be split and joined arbitrarily: $\mathsf{x_f} \overset{\pi_1+\pi_2}{\mapsto} \mathsf{v} \Leftrightarrow \mathsf{x_f} \overset{\pi_1}{\mapsto} \mathsf{v} * \mathsf{x_f} \overset{\pi_2}{\mapsto} \mathsf{v}$. In our specification pattern we use this law to split ownership of the phantom field that contains the abstract state of the data structure between the data structure itself and its clients – they each own $1/2$ of the phantom field.

The constructor returns a stack resource along with a $1/2$ fractional permission for the phantom field containing abstract state of the stack:

$$[\mathsf{emp}] \;\; \mathsf{new\ Stack}(-) \;\; [\mathsf{r}.\ \mathrm{stack}(\mathsf{r}) * \mathsf{r_{cont}} \xmapsto{1/2} \varepsilon]$$

The $\mathrm{stack}(\mathsf{x})$ resource expresses that x refers to a stack and is freely duplicable ($\mathrm{stack}(\mathsf{x}) \Leftrightarrow \mathrm{stack}(\mathsf{x}) * \mathrm{stack}(\mathsf{x})$) to allow an arbitrary number of clients to use the stack simultaneously.

Now, we have all the ingredients we need to formally express that an operation is logically atomic, or, that it simulates an arbitrary view shift that performs the corresponding operation on the phantom field that contains the abstract state. In the case of a push operation, it appears atomic if it simulates an arbitrary view shift that updates the abstract state from the list α to $\mathsf{y} :: \alpha$, when pushing y[5]:

$$\begin{gathered}\forall \mathsf{P}, \mathsf{Q}, \alpha, \mathsf{x}, \mathsf{y}.\ \ \mathsf{stable}(\mathsf{P}) \wedge \mathsf{stable}(\mathsf{Q}) \;\; \Rightarrow \\ \triangleright\, \mathsf{x_{cont}} \xmapsto{1/2} \alpha * \mathsf{P} \sqsubseteq \triangleright(\mathsf{x_{cont}} \xmapsto{1/2} (\mathsf{y} :: \alpha) * \mathsf{Q}) \;\; \Rightarrow \\ [\mathrm{stack}(\mathsf{x}) * \mathsf{P}] \;\; \mathsf{x.push(y)} \;\; [\mathrm{stack}(\mathsf{x}) * \mathsf{Q}]\end{gathered}$$

The view shift is taken as an assumption and parameterized with assertions P and Q, to allow callers to relate their local resources with the abstract state α, before and after the operation appears to take effect. Note, however, that *this specification does not express the flushing behavior of* push: we know nothing about the state of the store buffers at the time when the update of the abstract state happens. This is crucial information: to use Treiber's stack to transfer resources associated with an element y we need to know that all updates to y have

[5] Formally, the specification must also ensure that the client does not update the $\mathrm{stack}(\mathsf{x})$ resource within the provided view shift. Since these conditions are orthogonal to the underlying memory model, we elide them. See the technical appendix for the complete specification.

been flushed when the abstract state changes from α to $\mathsf{y} :: \alpha$. While the above specification is valid, it is not strong enough to entail all desired SC specifications; in particular, it does not entail the SC specification of a shared bag. Thus, we need to refine the specification pattern to express flushing behavior.

The crucial observation is that we need to express abstractly the ordering constraints that arise between the beginning of the method and the point at which the logical update takes place. We already introduced logical connectives to describe this kind of constraints on store buffers in Section 4. To express the flushing behavior of Treiber's stack, it is enough to assert that when the logical update happens, P holds in main memory. Thus, we can update the specification to read as follows:

$$\begin{gathered}\forall \mathsf{P}, \mathsf{Q}, \alpha, \mathsf{x}, \mathsf{y}.\ \ \mathsf{stable}(\mathsf{P}) \wedge \mathsf{stable}(\mathsf{Q}) \ \Rightarrow \\ \triangleright \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \alpha * \ulcorner \mathsf{P} \urcorner \sqsubseteq \triangleright(\mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} (\mathsf{y} :: \alpha) * \ulcorner \mathsf{Q} \urcorner) \ \Rightarrow \\ [\mathrm{stack}(\mathsf{x}) * \overline{\mathsf{P}}] \ \ \mathsf{x.push}(\mathsf{y}) \ [\mathrm{stack}(\mathsf{x}) * \ulcorner \mathsf{Q} \urcorner]\end{gathered}$$

In this version, we express that if P holds from the point of view of the calling thread initially ($\overline{\mathsf{P}}$), then by the time the abstract effects of the operation appear to take effect, P holds objectively, in main memory ($\ulcorner \mathsf{P} \urcorner$). Thus, using embeddings we can extend the specification pattern with an abstract way of expressing the synchronization behavior provided by a concurrent data structure.

Verifying an implementation. One of the motivations for logical atomicity is that it separates reasoning about clients from reasoning about internal data structure invariants, thereby avoiding reproving internal invariants for every client.

To verify a stack implementation against the abstract stack specification sketched above, we thus have to prove that the concrete stack representation is related to the current abstract state of the stack. Reasoning about clients can then proceed entirely in terms of the abstract state, independently of internal data structure invariants. To illustrate, we sketch how to prove that Treiber's stack satisfies the abstract stack specification.

Figure 6 defines an implementation of Treiber's stack. Since both push and pop use a CAS operation to push and pop elements, there will never be a buffered update to the head pointer or any Node object reachable from head. The invariant thus expresses that in abstract state α, the concrete state of the stack is a list representing the mathematical list α, which holds objectively in main memory. The stack(x) resource thus asserts the existence of a shared region governed by a labelled transition system with a single state s and the following interpretation function:

$$I(s) = \exists \mathsf{y} : \mathsf{Val}.\ \exists \alpha : \mathrm{list}\ \ \mathsf{Val}.\ \ulcorner \mathsf{x.head} \mapsto \mathsf{y} * \mathit{lst}_r(\mathsf{y}, \alpha) * \triangleright \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \alpha \urcorner,$$

where $\mathit{lst}_r(\mathsf{y}, \alpha)$ is a read-only list resource representing the list α. This last predicate is definable as follows by induction on the second argument:

$$\begin{aligned}\mathit{lst}_r(\mathsf{x}, \varepsilon) &= \mathsf{x} =_{\mathsf{Val}} \mathbf{null} \\ \mathit{lst}_r(\mathsf{x}, \mathsf{y} :: \alpha) &= \exists \mathsf{z} : \mathsf{Val}.\ \mathsf{x.next} \overset{r}{\mapsto} \mathsf{z} * \mathsf{x.val} \overset{r}{\mapsto} \mathsf{y} * \mathit{lst}_r(\mathsf{z}, \alpha)\end{aligned}$$

```
class Node { Object val; Node next; }
class Stack {
  Node head;

  Stack() = this.head := null; fence; this

  unit push'(Node nHead) =
    let oHead = this.head in
      nHead.next := oHead;
      let t = CAS(this.head, nHead, oHead) in
        if t then () else push'(nHead)

  unit push(Object x) =
    let nHead = new Node() in
      nHead.val := x; push'(nHead)

  Object pop() =
    let oHead = this.head in if oHead = null then ()
      else let nHead = oHead.next in
              if CAS(this.head, nHead, oHead) then oHead.val else pop()
}
```

Fig. 6. Treiber's stack.

The list is read-only, since the val and next fields of a Node object never change once the node has been inserted into the list.

The proof outline for the push method and the push′ helper method is shown in Figure 7. Note that the view-shift actually happens in the push′ method, after the successful execution of the compare-and-swap.

Deriving an SC specification. The specification pattern sketched above provides an abstract way of expressing the synchronization provided by the data structure. If the data structure provides sufficient synchronization, clients can thus derive SC specifications that support transfer of resources through the data structure, without adding additional client-side synchronization. To illustrate, we sketch how one can derive the SC shared bag specification from Section 3 from the abstract stack specification. Since Treiber's stack satisfies the abstract stack specification, it follows that Treiber's stack also satisfies the SC shared bag specification.

To derive the shared bag specification, the idea is to allocate a new shared region that will own the clients' half of the phantom field that describes the abstract state and all the resources associated with each element currently in the bag. Furthermore, these resources should hold objectively in main memory. This ensures that when a client pops an element, the resources associated with that element will also hold from that client's point of view. The bag(x, P) resource thus asserts ownership of stack(x) and the existence of a shared region governed by a labelled transition system with a single state s and the following interpretation

```
  unit push(Object x) =
$[\mathsf{stack}(\mathbf{this}) * \overline{\mathsf{P}(\mathbf{this}, \mathsf{x})}]$
    let nHead = new Node(x) in
$[\mathsf{stack}(\mathbf{this}) * \overline{\mathsf{P}(\mathbf{this}, \mathsf{x}) * \mathsf{nHead.val} \mapsto \mathsf{x} * \mathsf{nHead.next} \mapsto \_}]$
      push'(nHead)
$[\mathsf{stack}(\mathbf{this}) * \ulcorner \mathsf{Q}(\mathbf{this}, \mathsf{x}) \urcorner]$

  unit push'(Node nHead) =
$[\mathsf{stack}(\mathbf{this}) * \overline{\mathsf{P}(\mathbf{this}, \mathsf{x}) * \mathsf{nHead.val} \mapsto \mathsf{x} * \mathsf{nHead.next} \mapsto \_}]$
    let oHead = this.head in
$[\mathsf{stack}(\mathbf{this}) * \overline{\mathsf{P}(\mathbf{this}, \mathsf{x}) * \mathsf{nHead.val} \mapsto \mathsf{x} * \mathsf{nHead.next} \mapsto \_}]$
      nHead.next := oHead;
$[\mathsf{stack}(\mathbf{this}) * \overline{\mathsf{P}(\mathbf{this}, \mathsf{x}) * \mathsf{nHead.val} \mapsto \mathsf{x} * \mathsf{nHead.next} \mapsto \mathsf{oHead}}]$
      let t = CAS(this.head, nHead, oHead) in
$[\mathsf{stack}(\mathbf{this}) *$
    $((t = \mathbf{true} * \triangleright \ulcorner \mathsf{Q}(\mathbf{this}, \mathsf{x}) \urcorner) \;\; \vee$
     $(t = \mathbf{false} * \overline{\mathsf{P}(\mathbf{this}, \mathsf{x}) * \mathsf{nHead.val} \mapsto \mathsf{x} * \mathsf{nHead.next} \mapsto \mathsf{oHead}}))]$
        if t then () else push'(nHead)
$[\mathsf{stack}(\mathbf{this}) * \ulcorner \mathsf{Q}(\mathbf{this}, \mathsf{x}) \urcorner]$
```

Fig. 7. Proof outline of the push method.

function:

$$I_{\text{bag}}(s) = \exists \alpha : \text{list } \mathsf{Val}.\ \ulcorner \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \alpha * (\circledast_{\mathsf{y} \in mem(\alpha)} \mathsf{P}(\mathsf{y})) \urcorner$$

To update the abstract state of the stack we thus have to update the phantom field $\mathsf{x}_{\mathsf{cont}}$ partially owned by this region. This forces us to transfer ownership of $\ulcorner \mathsf{P}(\mathsf{y}) \urcorner$ to this shared region when the abstract state changes from α to $\mathsf{y} :: \alpha$ and allows us to take ownership of $\ulcorner \mathsf{P}(\mathsf{y}) \urcorner$ when the abstract state changes from $\mathsf{y} :: \alpha$ to α. In the case of $\underline{\mathsf{push}(\mathsf{y})}$, the SC specification of the shared bag only requires the client to provide $\overline{\mathsf{P}(\mathsf{y})}$. However, the abstract stack specification expresses that by the time the abstract push operation appears to take effect, the calling threads buffer will have been flushed and $\ulcorner \mathsf{P}(\mathsf{y}) \urcorner$ thus holds. *This is* what allows us to transfer $\ulcorner \mathsf{P}(\mathsf{y}) \urcorner$ to the region, and the reason *why we needed to express the synchronization behavior of the stack in the logically atomic specification.*