Cosmo: A Concurrent Separation Logic for Multicore OCaml

GLEN MÉVEL, Inria, Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, France
JACQUES-HENRI JOURDAN, Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, France
FRANÇOIS POTTIER, Inria, France

Multicore OCaml extends OCaml with support for shared-memory concurrency. It is equipped with a weak memory model, for which an operational semantics has been published. This begs the question: what reasoning rules can one rely upon while writing or verifying Multicore OCaml code? To answer it, we instantiate Iris, a modern descendant of Concurrent Separation Logic, for Multicore OCaml. This yields a low-level program logic whose reasoning rules expose the details of the memory model. On top of it, we build a higher-level logic, Cosmo, which trades off some expressive power in return for a simple set of reasoning rules that allow accessing nonatomic locations in a data-race-free manner, exploiting the sequentially-consistent behavior of atomic locations, and exploiting the release/acquire behavior of atomic locations. Cosmo allows both low-level reasoning, where the details of the Multicore OCaml memory model are apparent, and high-level reasoning, which is independent of this memory model. We illustrate this claim via a number of case studies: we verify several implementations of locks with respect to a classic, memory-model-independent specification. Thus, a coarse-grained application that uses locks as the sole means of synchronization can be verified in the Concurrent-Separation-Logic fragment of Cosmo, without any knowledge of the weak memory model.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Program verification**.

Additional Key Words and Phrases: separation logic, program verification, concurrency, weak memory

ACM Reference Format:
Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: A Concurrent Separation Logic for Multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 96 (August 2020), 29 pages. https://doi.org/10.1145/3408978

We are all in the gutter, but some of us are looking at the stars. — *Oscar Wilde*

1 INTRODUCTION

Multicore OCaml [Dolan et al. 2018a] extends the OCaml programming language [Leroy et al. 2019] with support for shared-memory concurrency. It is an ongoing experimental project: at the time of writing, although preparations are being made for its integration into mainstream OCaml, this has not yet been done. Nevertheless, Multicore OCaml has a well-defined semantics: in particular, its *memory model*, which specifies how threads interact through shared memory locations, has been published by Dolan et al. [2018b]. Therefore, one may already ask: what reasoning rules can or should a Multicore OCaml programmer rely upon in order to verify their code?

Authors' addresses: Glen Mével, Inria, Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, 91405, Orsay, France, glen.mevel@inria.fr; Jacques-Henri Jourdan, Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, 91405, Orsay, France, jacques-henri.jourdan@lri.fr; François Pottier, Inria, France, francois.pottier@inria.fr.

Multicore OCaml's memory model is *weak*: it does not enforce sequential consistency [Lamport 1979]. That is, the execution of a program is not necessarily an interleaving of the actions performed by its threads while interacting with a central shared memory. Many mainstream programming languages, including lower-level languages such as C11 [Batty et al. 2011; Lahav et al. 2017] and higher-level languages such as Java [Manson et al. 2005; Lochbihler 2012; Bender and Palsberg 2019], adopt weak memory models, because this allows compilers to perform more aggressive software optimizations and to better exploit hardware optimizations. Multicore OCaml follows this trend. Although it is expected that most application programmers will not need to worry about weak memory, because they will rely on a library of concurrent data structures written by domain experts, adopting a weak memory model allows said experts to write more efficient code—if they know what they are doing, that is!

We believe that there is a need for a set of reasoning rules—in other words, a program logic—that both populations of programmers can rely upon. Concurrency experts, who wish to implement efficient concurrent data structures, must be able to verify that their code is correct, even though they have removed as many synchronization operations as they thought possible. Furthermore, they must be able to verify that their code implements a simple, high-level abstraction, thereby allowing application programmers, in turn, to use it as a black box and reason about application code in a simple manner. In short, the system must allow compositional reasoning, and must allow both low-level reasoning, where the details of the Multicore OCaml memory model are apparent, and high-level reasoning, which is independent of this memory model.

Others before us have made a similar argument. For instance, Sieczkowski et al. [2015] propose iCAP-TSO, a variant of Concurrent Separation Logic that is sound with respect to the TSO memory model. While iCAP-TSO allows explicit low-level reasoning about weak memory, it also includes a high-level fragment, the "SC logic", whose reasoning rules are the traditional rules of Concurrent Separation Logic. These rules, which are independent of the memory model, require all direct accesses to memory to be data-race-free. Therefore, they require synchronization to be performed by other means. A typical means of achieving synchronization is to implement a lock, a concurrent data structure whose specification can be expressed in the SC logic, but whose proof of correctness must be carried out at the lower level of iCAP-TSO. As another influential example, Kaiser et al. [2017] follow a similar approach: they instantiate Iris [Jung et al. 2018b], a modern descendant of Concurrent Separation Logic, for a fragment of the C11 memory model. This yields a low-level "base logic", on top of which Kaiser et al. proceed to define several higher-level logics, whose reasoning rules are easier to use. Our aim, in this paper, is analogous. We wish to allow the verification of a low-level concurrent data structure implementation, such as the implementation of a lock. Such a verification effort must take place in a program logic that exposes the details of the Multicore OCaml memory model. At the same time, we would like the program logic to offer a high-level fragment that is independent of the memory model and in which data-race-free accesses to memory, mediated by locks or other concurrent data structures, are permitted.

Compared with other memory models in existence today, the Multicore OCaml memory model is relatively simple. Only two kinds of memory locations, known as "nonatomic" and "atomic" memory locations, are distinguished. Every program has a well-defined set of permitted behaviors. In particular, data races on nonatomic memory locations are permitted, and have "limited effect in space and in time" [Dolan et al. 2018b]. This is in contrast with the C11 memory model, where racy programs are deemed to have "undefined behavior", therefore must be ruled out by a sound program logic. Dolan et al. [2018b] propose two definitions of the Multicore OCaml memory model, and provide an informal proof that these definitions are equivalent. One definition takes the form of an operational semantics, where the execution of the program is in fact an interleaving of the executions of its threads. These threads interact with a memory whose behavior is more complex than usual, and whose description involves concepts such as *time stamps*, *histories*, *views*, and so on (§2). The other definition is an axiomatic semantics, where a "candidate execution" takes the form of a set of memory events and relations between them, and an actual execution is a candidate execution that respects certain conditions, such as the acyclicity of certain relations.

In this paper, we take Dolan *et al.*'s operational semantics, which we recall in Section 2, as a foundation. We instantiate Iris for this operational semantics. This yields a low-level logic, BaseCosmo (§3), whose reasoning rules expose the details of the Multicore OCaml memory model. Because of this, these rules are not very pleasant to use. In particular, the rules that govern access to nonatomic memory locations are rather unwieldy, as they expose the fact that the store maps each nonatomic location to a history, a set of write events. In order to facilitate reasoning, on top of BaseCosmo, we build a higher-level logic, Cosmo (§4), whose main features are as follows:

- Cosmo forbids data races on nonatomic locations. Data races on atomic locations remain permitted: atomic locations are in fact the sole primitive means of synchronization. This design decision allows Cosmo to offer a simplified set of reasoning rules, including:
  - standard, straightforward rules for data-race-free access to nonatomic locations;
  - standard, straightforward rules for possibly racy access to atomic locations, with the ability of exploiting the sequentially-consistent behavior of these locations; and
  - nonstandard yet arguably simple rules for reasoning about the release/acquire behavior of atomic locations.

  The last group of rules allow transferring a "view" of the nonatomic memory from one thread to another. By exploiting this mechanism, one can arrange to transfer an arbitrary assertion P from one thread to another, even when the footprint of P involves nonatomic memory locations. This feature is used in all of our case studies (§5).

  Although views have appeared in several papers [Kaiser et al. 2017; Dang et al. 2020], letting the user reason about release/acquire behavior in terms of abstract views seems novel and simpler than previous approaches. In particular, we claim that combining objective invariants and the axiom Split-Subjective-Objective yields a simple reasoning scheme, which could be exploited in logics for other weak memory models.
- Cosmo offers a high-level fragment where reasoning is independent of the memory model, that is, a fragment whose reasoning rules are those of traditional Concurrent Separation Logic. In this fragment, there is no notion of view. This fragment consists at least of the rules that govern access to nonatomic locations and can be extended by allowing the use of concurrent data structures whose specification is independent of the memory model and whose correctness has been verified using full Cosmo. The spin lock and ticket lock (§5) are examples of such data structures: their specification in Cosmo is the traditional specification of a lock in Concurrent Separation Logic [Gotsman et al. 2007; Hobor et al. 2008]. Thus, Cosmo contains traditional Concurrent Separation Logic, and allows reasoning about coarse-grained concurrent programs where all accesses to memory locations are mediated via locks.

We illustrate the use of Cosmo via several case studies (§5), including a spin lock, a ticket lock, and a lock based on Peterson's algorithm. All of our results, including the soundness of Cosmo and the verification of our case studies, are machine-checked in Coq. Our proofs can be browsed in the accompanying artifact and are also available from our repository [Mével et al. 2020].

2 MULTICORE OCAML

In this section, we recall the Multicore OCaml memory model and present the syntax and operational semantics of a core calculus that is representative of the Multicore OCaml programming language.

$$
\begin{aligned}
a &\in \textsc{Loc}_{\text{NA}} \\
A &\in \textsc{Loc}_{\text{AT}} \\
t &\in \textsc{Time} \triangleq \mathbb{Q} \cap [0, \infty) \\
h &\in \textsc{Hist} \triangleq \textsc{Time} \xrightarrow{\text{fin}} \textsc{Val} \\
\mathcal{V}, \mathcal{W}, \mathcal{G} &\in \textsc{View} \triangleq \textsc{Loc}_{\text{NA}} \xrightarrow{\text{aez}} \textsc{Time} \\
\sigma &\in \textsc{Store} \triangleq (\textsc{Loc}_{\text{NA}} \xrightarrow{\text{fin}} \textsc{Hist}) \times (\textsc{Loc}_{\text{AT}} \xrightarrow{\text{fin}} \textsc{Val} \times \textsc{View})
\end{aligned}
$$

Fig. 1. Semantic objects: locations, time stamps, histories, views, stores.

We start by describing interactions with the memory by the memory subsystem (§2.1), and then we present the reduction rules of the programming language itself (§2.2).

2.1 Memory Model

In this section, we recall Dolan et al.'s definition of the Multicore OCaml memory model (2018b) and extend it with memory allocation and CAS, which do not appear in Dolan et al.'s paper. We first define a number of semantic objects (Figure 1), then describe the behavior of the memory subsystem (Figure 2). At this point, we assume a set Val of values, which is defined later on (§2.2).

2.1.1 Locations. A memory location has one of two possible kinds, fixed throughout its lifetime: it is either *nonatomic* or *atomic*. We write a for a nonatomic location and A for an atomic one.

2.1.2 Time Stamps. We use *time stamps* t to describe the behavior of reads and writes at nonatomic memory locations. They belong to $\mathbb{Q} \cap [0, \infty)$, the set of the nonnegative rational numbers. This set is infinite, totally ordered, and admits a minimum element 0, which is used for initial writes, and only for initial writes. It is worth noting that the meaning of time stamps is purely "per location": that is, the time stamps associated with two write events at two distinct locations are never compared.

2.1.3 Histories. A *history* h is a finite map of time stamps to values. It represents the history of the write events at a single nonatomic memory location. If desired, a history can be also be viewed as a set of pairs of a time stamp and a value, that is, a set of write events, under the constraint that no two events have the same time stamp. Naturally, a read instruction at a nonatomic memory location must read from one of the write events in the history of this location, and a write instruction must extend the history of this location with a write event at a previously-unused time stamp. In either case, which time stamp may be chosen depends on the "view" of the active thread, described next.

2.1.4 Views. A *view* $\mathcal{V}$, referred to as a "frontier" by Dolan et al. [2018b], is a total mapping of nonatomic memory locations to time stamps that is *almost everywhere zero* ("aez" for short), that is, whose value is zero everywhere except a finite set of locations. Each thread has a view, which evolves over time. The view of a thread maps each nonatomic memory location to (the time stamp of) the most recent write event at this location this thread is aware of. The view of the active thread imposes a constraint on the behavior of reads and writes at nonatomic memory locations (§2.2.2).

The order on time stamps gives rise to a partial order on views: by definition, the view inequality $\mathcal{V}_1 \sqsubseteq \mathcal{V}_2$ holds if and only if, for every nonatomic memory location a, the time stamp inequality $\mathcal{V}_1(a) \leq \mathcal{V}_2(a)$ holds. This order can be thought of as an *information* order: if $\mathcal{V}_1 \sqsubseteq \mathcal{V}_2$ holds, then a thread with view $\mathcal{V}_2$ has more information (is aware of more write events) than a thread with view $\mathcal{V}_1$. Equipped with this order, views form a bounded semilattice. Its minimum element

$$\textsc{mem-na-Alloc}\quad \frac{a \notin \operatorname{dom} \sigma \qquad h = \{0 \mapsto v\}}{\sigma; \mathcal{W} \xrightarrow{\mathsf{alloc}(a,v)} \sigma[a \mapsto h]; \mathcal{W}} \qquad \textsc{mem-at-Alloc}\quad \frac{A \notin \operatorname{dom} \sigma}{\sigma; \mathcal{W} \xrightarrow{\mathsf{alloc}(A,v)} \sigma[A \mapsto \langle v, \mathcal{W}\rangle]; \mathcal{W}}$$

$$\textsc{mem-na-Read}\quad \frac{\begin{array}{cc} h = \sigma(a) & t \in \operatorname{dom} h \\ \mathcal{W}(a) \le t & v = h(t) \end{array}}{\sigma; \mathcal{W} \xrightarrow{\mathsf{rd}(a,v)} \sigma; \mathcal{W}} \qquad \textsc{mem-at-Read}\quad \frac{\sigma(A) = \langle v, \mathcal{V}\rangle}{\sigma; \mathcal{W} \xrightarrow{\mathsf{rd}(A,v)} \sigma; \mathcal{W} \sqcup \mathcal{V}}$$

$$\textsc{mem-na-Write}\quad \frac{\begin{array}{cc} h = \sigma(a) & t \notin \operatorname{dom} h \\ \mathcal{W}(a) < t & h' = h[t \mapsto v] \end{array}}{\sigma; \mathcal{W} \xrightarrow{\mathsf{wr}(a,v)} \sigma[a \mapsto h']; \mathcal{W}[a \mapsto t]}$$

$$\textsc{mem-at-Write}\quad \frac{\sigma(A) = \langle v, \mathcal{V}\rangle \qquad \mathcal{V}' = \mathcal{W}' = \mathcal{W} \sqcup \mathcal{V}}{\sigma; \mathcal{W} \xrightarrow{\mathsf{wr}(A,v')} \sigma[A \mapsto \langle v', \mathcal{V}'\rangle]; \mathcal{W}'} \qquad \textsc{mem-at-Read-Write}\quad \frac{\sigma(A) = \langle v, \mathcal{V}\rangle \qquad \mathcal{V}' = \mathcal{W}' = \mathcal{W} \sqcup \mathcal{V}}{\sigma; \mathcal{W} \xrightarrow{\mathsf{rdwr}(A,v,v')} \sigma[A \mapsto \langle v', \mathcal{V}'\rangle]; \mathcal{W}'}$$

Fig. 2. Operational behavior of the memory subsystem: $\sigma; \mathcal{W} \xrightarrow{m} \sigma'; \mathcal{W}'$

is the *empty view* $\perp$, which maps every nonatomic memory location to the time stamp 0. Its join operation is pointwise maximum: that is, $\mathcal{V}_1 \sqcup \mathcal{V}_2$ is $\lambda a.\ \max(\mathcal{V}_1(a), \mathcal{V}_2(a))$.

2.1.5 Stores. A *store* σ is a pair of a nonatomic store and an atomic store. A *nonatomic store* σ_{na} is a finite map of nonatomic memory locations to histories: as suggested earlier, each nonatomic memory location is mapped to a history of all write events at this location. An *atomic store* σ_{at} is a finite map of atomic memory locations to pairs of a value and a view. Indeed, atomic memory locations in Multicore OCaml serve two purposes, which are conceptually independent of one another. On the one hand, each atomic memory location stores a value. A single value, as opposed to a history, suffices, because atomic memory locations have sequentially consistent semantics. On the other hand, each atomic memory location stores a view, that is, a certain amount of information about the nonatomic store. This view is read and updated by atomic read and write instructions, thereby giving atomic memory locations a “release/acquire” semantics.

2.1.6 The Memory Subsystem. To complete the definition of the Multicore OCaml memory model, there remains to give an operational description of the behavior of the memory subsystem. This is done via a labelled transition system, that is, a relation $\sigma; \mathcal{W} \xrightarrow{m} \sigma'; \mathcal{W}'$, which describes how the shared global store σ and the view $\mathcal{W}$ of the active thread evolve through a memory event m, yielding an updated store σ' and an updated view $\mathcal{W}'$. The syntax of memory events is as follows:

$$m ::= \varepsilon \mid \mathsf{alloc}(a, v) \mid \mathsf{rd}(a, v) \mid \mathsf{wr}(a, v) \mid \mathsf{alloc}(A, v) \mid \mathsf{rd}(A, v) \mid \mathsf{wr}(A, v) \mid \mathsf{rdwr}(A, v, v')$$

These memory events also appear in the definition of the semantics of Multicore OCaml expressions (§2.2.2). Thus, they form a language by which the expression and memory subsystems communicate.

The rules that define the relation $\sigma; \mathcal{W} \xrightarrow{m} \sigma'; \mathcal{W}'$ (Figure 2) can be briefly described as follows.

When a fresh nonatomic memory location is allocated (mem-na-Alloc), its history consists of a single write event at time stamp 0. This guarantees that a read of this location will succeed, even if the reading thread has not synchronized with the thread where allocation was performed.

As suggested earlier (§2.1.4), the view of the active thread imposes a constraint on the behavior of read and write instructions at nonatomic memory locations. A read instruction at location a (mem-na-Read) cannot read from an outdated write event, therefore must read from an event whose time stamp t is no less than $\mathcal{W}(a)$. Both $\mathcal{W}(a) = t$ and $\mathcal{W}(a) < t$ are permitted: the latter case allows a thread to read from a write event of which it is not yet aware. A nonatomic read instruction

$b \in \{\texttt{false}, \texttt{true}\}$	— Boolean values
$n \in \mathbf{Z}$	— integer values
$v ::= \mu f.\lambda x.\, e \mid () \mid b \mid n \mid a \mid A \mid (v, \ldots, v)$	— values
$\odot \in \{\ \&\&\ ,\ \Vert\ , +, -, \ldots\}$	— primitive operations (of arity 2)
$e ::=$	— expressions:
$x \mid \mu f.\lambda x.\, e \mid e\ e$	— λ-calculus with recursive functions
$\mid () \mid b \mid n \mid e \odot e$	— primitive values and operations
$\mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	— conditional
$\mid \texttt{let}\ x = e\ \texttt{in}\ e$	— sequencing
$\mid (e, \ldots, e) \mid \texttt{let}\ (x, \ldots, x) = e\ \texttt{in}\ e$	— tuples
$\mid \texttt{fork}\ e$	— spawning a new thread
$\mid a \mid \texttt{new}_{\mathsf{na}}\ e \mid !_{\mathsf{na}}\, e \mid e :=_{\mathsf{na}} e$	— operations on nonatomic memory locations
$\mid A \mid \texttt{new}_{\mathsf{at}}\ e \mid !_{\mathsf{at}}\, e \mid e :=_{\mathsf{at}} e \mid \texttt{CAS}\ e\ e\ e$	— operations on atomic memory locations

Fig. 3. Multicore OCaml: Syntax

does *not* update the view of the active thread. A write instruction at location a (mem-na-Write) cannot produce an outdated write event, therefore must use a time stamp t greater than $\mathcal{W}(a)$. The new time stamp must be fresh (that is, not in the domain of h), but there is no requirement for t to be greater than every time stamp in the domain of h. The history of location a and the view of the active thread are updated so as to include the new write event.

Allocating a fresh atomic memory location (mem-at-Alloc) causes it to be initialized with the value v provided by the allocation instruction and with the view $\mathcal{W}$ of the active thread. When reading an atomic memory location (mem-at-Read) the view $\mathcal{V}$ stored in this location is merged into the view of the active thread, reflecting the idea that the active thread acquires information via this read operation. Writing an atomic memory location (mem-at-Write) overwrites the value v stored at this location with the new value v' and updates the view $\mathcal{V}$ stored at this location by merging the view $\mathcal{W}$ of the active thread into it, reflecting the idea that the active thread releases information via this write operation. At the same time, the view of the active thread is updated; indeed, an atomic write has both a "release" and an "acquire" effect. The rule mem-at-Read-Write models a successful CAS instruction. It which combines the effects of a read and a write in a single atomic step. An unsuccessful CAS instruction behaves like an atomic read.

As Multicore OCaml is equipped with a garbage collector, there is no explicit deallocation event.

2.2 Programming Language

We now present a core calculus that is representative of the Multicore OCaml programming language. It is equipped with a dynamic thread creation operation and with the two kinds of memory locations presented earlier. For simplicity, we refer to this calculus as Multicore OCaml.

2.2.1 Syntax. The syntax of the calculus appears in Figure 3. It is a call-by-value λ-calculus, equipped with recursive functions, primitive operations on Boolean and integer values, tuples, and standard control constructs, including conditionals and sequencing. This calculus is extended with shared-memory concurrency, as follows. First, the construct `fork` e spawns a new thread. In the parent thread, this operation immediately returns the unit value `()`. In the new thread, the expression e is executed, and its result is discarded. Second, the language supports the standard

$$\mathsf{new}_{\mathsf{na}}\ v \xrightarrow{\mathsf{alloc}(a,v)} a \qquad !_{\mathsf{na}}\ a \xrightarrow{\mathsf{rd}(a,v)} v \qquad a :=_{\mathsf{na}} v \xrightarrow{\mathsf{wr}(a,v)} () \qquad \mathsf{new}_{\mathsf{at}}\ v \xrightarrow{\mathsf{alloc}(A,v)} A$$

$$!_{\mathsf{at}}\ A \xrightarrow{\mathsf{rd}(A,v)} v \qquad A :=_{\mathsf{at}} v \xrightarrow{\mathsf{wr}(A,v)} () \qquad \frac{v_0 \neq v_1}{\mathsf{CAS}\ A\ v_1\ v_2 \xrightarrow{\mathsf{rd}(A,v_0)} \mathsf{false}} \qquad \mathsf{fork}\ e \xrightarrow{\varepsilon} (), e$$

$$\mathsf{CAS}\ A\ v_1\ v_2 \xrightarrow{\mathsf{rdwr}(A,v_1,v_2)} \mathsf{true}$$

Fig. 4. Thread-local reduction: $e \xrightarrow{m} e', e'_1, \ldots, e'_n$ (selected rules)

$$\frac{e \xrightarrow{m} e' \qquad \sigma; \mathcal{W} \xrightarrow{m} \sigma'; \mathcal{W}'}{\begin{array}{ll} & \sigma; p_1, \langle e, \mathcal{W} \rangle, p_2 \\ \Longrightarrow & \sigma'; p_1, \langle e', \mathcal{W}' \rangle, p_2 \end{array}} \qquad \frac{e \xrightarrow{\varepsilon} e', e'_1, \ldots, e'_n}{\begin{array}{ll} & \sigma; p_1, \langle e, \mathcal{W} \rangle, p_2 \\ \Longrightarrow & \sigma; p_1, \langle e', \mathcal{W} \rangle, p_2, \langle e'_1, \mathcal{W} \rangle, \langle e'_n, \mathcal{W} \rangle \end{array}}$$

Fig. 5. Thread-pool reduction: $\sigma; p \Longrightarrow \sigma'; p'$

operations of allocation, reading, and writing, on both nonatomic and atomic memory locations. In addition, atomic memory locations support a compare-and-set (CAS) operation.[1]

2.2.2 Operational Semantics. The "per-thread" reduction relation $e \xrightarrow{m} e', e'_1, \ldots, e'_n$, describes how an expression e takes a step and reduces to a new expression e', possibly interacting with the memory subsystem via an event m, and possibly spawning a number of new threads $e'_1, \ldots, e'_n$. The bulk of the rules that define this reduction relation form a standard small-step reduction semantics for call-by-value λ-calculus. These rules do not involve an interaction with the memory subsystem and do not spawn new threads. We omit them. In Figure 4, we present the reduction rules for the memory access operations, which interact with the memory subsystem via a memory event, and the reduction rule for "fork", which spawns one new thread.

In Figure 5, we define the "thread-pool" reduction relation $\sigma; p \Longrightarrow \sigma'; p'$, which describes how the machine steps between two configurations $\sigma; p$ and $\sigma'; p'$. In such a configuration, σ is a store, while p is a thread pool, that is, a list of threads, where each thread $\langle e, \mathcal{W} \rangle$ is a pair of an expression e and a view $\mathcal{W}$. The left-hand rule allows the per-thread execution system and the memory subsystem to synchronize on an event m. The right-hand rule shows how new threads are spawned; a newly-spawned thread inherits the view of its parent thread. Because a per-thread reduction step cannot both access memory and spawn new threads, these two rules suffice.

The operational semantics of Multicore OCaml is the reflexive transitive closure of the thread-pool reduction relation. It is therefore an interleaving semantics, albeit not a sequentially consistent semantics, as it involves a store whose behavior is nonstandard.

A machine configuration $\sigma; p$ is considered *stuck* if the thread pool p contains at least one thread that cannot take a step yet has not reached a value. A stuck configuration represents an undesirable event, a runtime error. There are many ways of constructing stuck configurations: examples include applying a primitive operation to arguments of incorrect nature, attempting to call a value other than a function, and so on. All such errors are ruled out by the OCaml type system:[2] the execution

[1]In the surface language, nonatomic locations have type `'a ref`. Their operations are `ref`, `!`, and `:=`. Atomic locations have type `'a Atomic.t`. Their operations are `Atomic.make`, `Atomic.get`, `Atomic.set`, and `Atomic.compare_and_set`.
[2]No changes to the type system are required by the move from OCaml to Multicore OCaml. The type `'a Atomic.t` is a new (primitive) abstract type.

of a well-typed program cannot lead to a stuck configuration. Although this claim does not appear to have been formally established, it seems clear that a syntactic proof of type soundness for ML with references [Wright and Felleisen 1994] can be adapted to the semantics of Multicore OCaml.

A careful reader might note that a nonatomic read instruction (MEM-NA-READ) *could* potentially be stuck under certain circumstances. This could be the case, for instance, if the history h is empty, or if the time stamp $\mathcal{W}(a)$ is too high, causing all write events in h to be considered outdated. In reality, though, neither of these situations can arise, because only a subset of *well-formed* machine configurations can be reached. In a well-formed configuration, every nonatomic location ever allocated must have a nonempty history, and no thread can get hold of an unallocated location, thereby removing the concern that h might be empty. Furthermore, a well-formed configuration must satisfy the following *global view invariant*: there exists a *global view* $\mathcal{G}$ such that:

(1) every thread's view $\mathcal{W}$ is contained in $\mathcal{G}$, that is, $\forall \langle e, \mathcal{W}\rangle \in p.\ \mathcal{W} \sqsubseteq \mathcal{G}$ holds;
(2) the view of an atomic location is contained in $\mathcal{G}$, that is, $\forall (A, \langle v, \mathcal{V}\rangle) \in \sigma_{\text{at}}.\ \mathcal{V} \sqsubseteq \mathcal{G}$ holds;
(3) the global view maps every nonatomic location to a time stamp that exists in the history of this location, that is, $\forall (a, h) \in \sigma_{\text{na}}.\ \mathcal{G}(a) \in \text{dom}\ h$ holds;
(4) the global view maps every currently unallocated location to the time stamp 0, that is, $\forall a \notin \text{dom}\ \sigma_{\text{na}}.\ \mathcal{G}(a) = 0$ holds.

One can check that this is an indeed an invariant (i.e., it is preserved by every reduction step) and that this invariant implies that a nonatomic read instruction cannot be stuck. In fact, in the next section, we must prove this claim, and exploit this invariant, otherwise we would be unable to prove that our logic is sound: a sound program logic must guarantee that a verified program cannot get stuck. Thus, when we instantiate Iris for Multicore OCaml, we build the global view invariant into our "state interpretation" invariant (§3.1), and when we establish the reasoning rule for nonatomic read instructions (rule BASE-NA-READ in Figure 6), we exploit it (§3.3).

3 A LOW-LEVEL LOGIC: BASECOSMO

In this section, we set up a program logic for Multicore OCaml, based on the operational semantics presented in the previous section. To do so, we rely on Iris [Jung et al. 2018b], a generic framework for building program logics. Iris is not tied to a particular programming language or calculus. Its lower layer, the Iris base logic [Jung et al. 2018b, §3–5], is a purely logical construction. Its upper layer, the Iris program logic [Jung et al. 2018b, §6–7], is parameterized with a programming language. In order to instantiate it, a client must provide the following information:

- A set of "expressions".
- A subset of "values".
- A set of machine "states". For instance, a state might be a store, that is, a map whose domain is the set of all currently allocated memory locations.
- An operational semantics, in the form of a "per-thread step relation". This relation relates an expression and a state to an expression and a state and a list of expressions, which represent newly-spawned threads.
- A "state interpretation" predicate $S : \text{State} \rightarrow \text{iProp}$.[3] This predicate represents a global invariant about the machine state. It typically relates the state with a piece of ghost state whose monoid structure[4] is chosen by the client so as to justify splitting the ownership of the machine state in certain ways.

Once the client has provided this information, the framework yields a program logic, that is,

[3] iProp is the type of Iris assertions.
[4] For simplicity, we use the word "monoid" everywhere, although "partial commutative monoid" would be more explicit, and an even more accurate term would be "resource algebra" [Jung et al. 2018b, §2.1] or "camera" [Jung et al. 2018b, §4.4].

- A weakest-precondition predicate $\mathrm{wp}\ e\ \{\Phi\}$.
- A Hoare triple $\{P\}\ e\ \{\Phi\}$, which is just sugar for $\square\,(P \twoheadrightarrow \mathrm{wp}\ e\ \{\Phi\})$.
- An adequacy theorem, which states that if a closed program e satisfies the triple $\{\mathsf{True}\}\ e\ \{\Phi\}$ then it is safe to run, that is, its execution will not lead to a stuck configuration.
- A set of programming-language-independent deduction rules for triples. These include the consequence rule, the frame rule, rules for allocating and updating ghost state, rules for setting up and exploiting invariants, and so on.

It is then up to the client to perform extra programming-language-specific work, namely:

- Define programming-language-specific assertions, such as "points-to" assertions.
- Prove entailment laws describing, e.g., how points-to assertions can be split and combined.
- Establish programming-language-specific deduction rules for triples, e.g., axioms that give triples for reading and writing memory locations.

We now apply this recipe to Multicore OCaml. This yields BaseCosmo, a logic for Multicore OCaml.

3.1 Instantiating Iris for Multicore OCaml

We begin instantiating Iris as follows:

- An "expression" is a pair $\langle e, \mathcal{V}\rangle$ of a Multicore OCaml expression and a view.
- Accordingly, a "value" is a pair $\langle v, \mathcal{V}\rangle$ of a Multicore OCaml value and a view.
- A "state" is a store σ, that is, a pair $(\sigma_{\mathrm{na}}, \sigma_{\mathrm{at}})$ of a nonatomic store σ_{na} and an atomic store σ_{at}.
- The "per-thread step relation" is as defined in Figure 4.

To complete this instantiation, there remains to define a suitable "state interpretation" invariant S. In choosing this definition, we have a great deal of freedom. Our choice is guided by several considerations, including the manner in which we wish to allow splitting the ownership of the state, and the invariants about the state that we wish to keep track of. In the present case, we have the following two independent concerns in mind:

— We wish to allow splitting the ownership of memory locations (including nonatomic and atomic memory locations) under a standard "fractional permissions" regime.

— We need to keep track of the global view invariant (§2.2.2) enjoyed by the operational semantics of Multicore OCaml, because this invariant is required to justify that a nonatomic read instruction cannot be stuck.

With these goals in mind, we define our state interpretation invariant as follows. We give the definition first, then recall the Iris-specific notation used in this definition. We delay an explanation of the definition to the next subsection (§3.2).

- Let γ_{na} be a ghost location storing an element of $\textsc{Auth}(\textsc{Loc}_{\textsc{na}} \xrightarrow{\mathrm{fin}} \textsc{Ag}(\textsc{Hist}) \times \textsc{Frac})$.
- Let γ_{at} be a ghost location storing an element of $\textsc{Auth}(\textsc{Loc}_{\textsc{at}} \xrightarrow{\mathrm{fin}} \textsc{Ag}(\textsc{Val} \times \textsc{View}) \times \textsc{Frac})$.
- Let γ_{gv} be a ghost location storing an element of $\textsc{Auth}(\textsc{View})$.[5]
- We define the state interpretation invariant as follows:

$$S\ (\sigma_{\mathrm{na}}, \sigma_{\mathrm{at}}) \quad \triangleq \quad \boxed{\bullet\, \sigma_{\mathrm{na}}}^{\gamma_{\mathrm{na}}} \ * \ \boxed{\bullet\, \sigma_{\mathrm{at}}}^{\gamma_{\mathrm{at}}} \ * \ \exists \mathcal{G}.\ \mathop{\Large *}\begin{cases} \boxed{\bullet\, \mathcal{G}}^{\gamma_{\mathrm{gv}}} \\ \lceil \forall (A, \langle v, \mathcal{V}\rangle) \in \sigma_{\mathrm{at}}.\ \mathcal{V} \sqsubseteq \mathcal{G} \rceil \\ \lceil \forall (a, h) \in \sigma_{\mathrm{na}}.\ \mathcal{G}(a) \in \mathrm{dom}\ h \rceil \\ \lceil \forall a \notin \mathrm{dom}\ \sigma_{\mathrm{na}}.\ \mathcal{G}(a) = 0 \rceil \end{cases}$$

Let us briefly recall some Iris concepts and notation. The monoids mentioned above are built out of the standard Iris toolkit. An authoritative monoid $\textsc{Auth}(\cdot)$ [Jung et al. 2018b, §6.3.3] has

[5] $\textsc{View}$ is the monoid of views equipped with the operation $\sqcup$. It is an idempotent monoid.

both *authoritative elements* of the form $\bullet\, a$ and *fragmentary elements* of the form $\circ\, b$. An authoritative element is intended to represent the full knowledge of something, and cannot be split: the composition $(\bullet\, a_1) \cdot (\bullet\, a_2)$ is never valid. A fragmentary element is intended to represent partial knowledge, and can be split and joined: the composition $(\circ\, b_1) \cdot (\circ\, b_2)$ is defined as $\circ\,(b_1 \cdot b_2)$. Because a fragment must be a part of the whole, the composition $(\bullet\, a) \cdot (\circ\, b)$ is valid only if $b \preccurlyeq a$ holds, where $\preccurlyeq$ is the ordering induced by the monoid law. The finite map monoid $\cdot \xrightarrow{\text{fin}} \cdot$ has finite maps as its elements; composition is pointwise. The agreement monoid [Jung et al. 2018b, §3.1, §4.3] on histories $\textsc{Ag}(\textsc{Hist})$ has histories h as elements. Its composition law requires agreement: that is, $h \cdot h$ is h, and $h_1 \cdot h_2$ is invalid if h_1 and h_2 differ. The fractional permission monoid $\textsc{Frac}$ has the fractions $q \in \mathbb{Q} \cap (0, 1]$ as its elements and addition as its composition law. Note that, because the fraction 0 is excluded, the fraction 1 cannot be composed with any fraction. This reflects the intuitive idea that the fraction 1 represents exclusive ownership of something.

3.2 Multicore OCaml-Specific Assertions

We have performed the first step described at the beginning of this section (§3): we have instantiated Iris for the operational semantics of Multicore OCaml and for a state interpretation of our choosing. This yields a weakest-precondition predicate $\text{wp}\ \langle e, \mathcal{V}\rangle\ \{\Phi\}$; a triple $\{P\}\ \langle e, \mathcal{V}\rangle\ \{\Phi\}$; an adequacy theorem, which guarantees that this triple is sound; and a set of programming-language-independent deduction rules for triples. There remains to perform programming-language-specific work. We define three custom assertions, namely the nonatomic and atomic "points-to" assertions and a "valid-view" assertion. Next (§3.3), we give a set of reasoning rules where these assertions appear.

3.2.1 Nonatomic Points-to. We wish to be able to split up the ownership of the nonatomic store under a fractional permission regime. As usual [Boyland 2003; Bornat et al. 2005], the fraction 1 should represent exclusive read-write access, while a fraction $q < 1$ should represent shared read-only access. Furthermore, we wish to ensure that whoever owns a share of a nonatomic location has exact knowledge of its history.

For this purpose, we have placed the ghost-state-ownership assertion $\boxed{\bullet\, \sigma_{\text{na}}}^{\gamma_{\text{na}}}$ in the state interpretation invariant (§3.1), and we now define the nonatomic points-to assertion as follows:

$$q \cdot a \rightsquigarrow_{\text{na}} h \;\triangleq\; \boxed{\circ\, \{a \mapsto (h, q)\}}^{\gamma_{\text{na}}}$$

What is going on here? On the one hand, the points-to assertion claims the ownership of a fragmentary element of the monoid $\textsc{Auth}(\textsc{Loc}_{\textsc{na}} \xrightarrow{\text{fin}} \textsc{Ag}(\textsc{Hist}) \times \textsc{Frac})$. The notation $\{a \mapsto (h, q)\}$ describes an element of the monoid $\textsc{Loc}_{\textsc{na}} \xrightarrow{\text{fin}} \textsc{Ag}(\textsc{Hist}) \times \textsc{Frac}$: it is a singleton map, which maps the location a to the pair (h, q). On the other hand, the state interpretation invariant owns the authoritative element $\bullet\, \sigma_{\text{na}}$ of the monoid $\textsc{Auth}(\ldots)$.[6] This ties the nonatomic store σ_{na}, which is part of the physical machine state, with the state of the ghost location γ_{na}.

Now, recall that, from the coexistence of the elements $\bullet\, a$ and $\circ\, b$, one can deduce $b \preccurlyeq a$ (§3.1). Thus, when a points-to assertion $q \cdot a \rightsquigarrow_{\text{na}} h$ is at hand, one can unfold the definition of this assertion and get a fragmentary element $\circ\, \{a \mapsto (h, q)\}$. One can also briefly "open" the state interpretation invariant and see an authoritative element $\bullet\, \sigma_{\text{na}}$. From their coexistence, one deduces $\{a \mapsto (h, q)\} \preccurlyeq \sigma_{\text{na}}$, that is, the singleton map of a to the pair (h, q) is indeed a fragment of the physical nonatomic store σ_{na}.

Here, $\preccurlyeq$ is the ordering induced by the composition law of the monoid $\textsc{Loc}_{\textsc{na}} \xrightarrow{\text{fin}} \textsc{Ag}(\textsc{Hist}) \times \textsc{Frac}$. Because this law requires agreement of the history components and addition of the fraction

[6]The notation $\bullet\, \sigma_{\text{na}}$ involves an abuse of notation. In it, σ_{na} must be understood as an element of the monoid $\textsc{Loc}_{\textsc{na}} \xrightarrow{\text{fin}} \textsc{Ag}(\textsc{Hist}) \times \textsc{Frac}$, which maps every location a in the domain of σ_{na} to a pair of the history $\sigma_{\text{na}}(a)$ and the fraction 1.

components, one can read the assertion $q \cdot a \rightsquigarrow_{\mathrm{na}} h$ as representing the knowledge that the history of this location is h and the ownership of a q-share of the nonatomic memory location a.

This technique is not original: we follow the pattern presented by Jung et al. [2018b, §6.3.3, §7.3]. We omit the fraction q when it is 1. Thus, we write $a \rightsquigarrow_{\mathrm{na}} h$ for $1 \cdot a \rightsquigarrow_{\mathrm{na}} h$.

3.2.2 Atomic Points-to. Regarding the atomic points-to assertion, we proceed essentially in the same manner. Earlier (§3.1), we have placed the assertion $\boxed{\bullet\, \sigma_{\mathrm{at}}}^{\gamma_{\mathrm{at}}}$ in the state interpretation invariant. We now define the atomic points-to assertion as follows:

$$q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \rangle \;\triangleq\; \exists \mathcal{W}.\; \lceil \mathcal{V} \sqsubseteq \mathcal{W} \rceil \,*\, \boxed{\circ \{A \mapsto (\langle v, \mathcal{W} \rangle, q)\}}^{\gamma_{\mathrm{at}}}$$

As a result of these definitions, the assertion $q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \rangle$ claims the ownership of a q-share of the atomic memory location A, guarantees that the value stored at this location is v, and guarantees that the view stored at this location is at least $\mathcal{V}$.

By requiring the view $\mathcal{W}$ stored at A to satisfy $\mathcal{V} \sqsubseteq \mathcal{W}$, as opposed to the equality $\mathcal{V} = \mathcal{W}$, we make the points-to assertion reverse-monotonic in its view parameter: that is, if $\mathcal{V}_1 \sqsubseteq \mathcal{V}_2$ holds, then $q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}_2 \rangle$ entails $q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}_1 \rangle$. This seems convenient in practice, as it gives the user a concise way of retaining partial knowledge of the view that is stored at location A.

As with nonatomic points-to assertions, we omit the fraction q when it is 1.

3.2.3 Validity of a View. The last part of the state interpretation invariant (§3.1) asserts the existence of a view $\mathcal{G}$ such that items (2), (3) and (4) of the global view invariant hold (§2.2.2). It also includes the ghost-state-ownership assertion $\boxed{\bullet\, \mathcal{G}}^{\gamma_{\mathrm{gv}}}$. We now define the "valid-view" assertion as follows:

$$\mathit{valid}\, \mathcal{V} \;\triangleq\; \boxed{\circ\, \mathcal{V}}^{\gamma_{\mathrm{gv}}}$$

The assertion $\mathit{valid}\, \mathcal{V}$ guarantees that $\mathcal{V}$ is a fragment of the global view, that is, $\mathcal{V} \sqsubseteq \mathcal{G}$ holds. Because the View monoid is idempotent (that is, $\mathcal{V} \sqcup \mathcal{V} = \mathcal{V}$), this assertion is duplicable. Thus, it does not represent the ownership of anything. It represents only the knowledge that $\mathcal{V}$ is a fragment of the global view.

A validity assertion appears in almost all of the reasoning rules that we present in the next subsection (§3.3). In other words, the rules effectively require (and allow) every thread to keep track of the fact that its current view is valid. This encodes item (1) of the global view invariant.

3.3 Multicore OCaml-Specific Axioms

There remains to give a set of Multicore OCaml-specific deduction rules that allow establishing Hoare triples. We present just the rules that govern the memory access operations; the rest is standard [Jung et al. 2018b, §6.2]. These rules appear in Figure 6.[7] They are "small axioms" [O'Hearn 2019], that is, triples that describe the minimum resources required by each operation. Each of them is just a single triple $\{P\}\ \langle e, \mathcal{W} \rangle\ \{\Phi\}$, which, for greater readability, we display vertically:

$$\frac{\dfrac{P}{\langle e, \mathcal{W} \rangle}}{\Phi}$$

The precondition P describes the resources required in order to safely execute the expression e on a thread whose view is $\mathcal{W}$. The postcondition Φ, which takes a pair $\langle v', \mathcal{W}' \rangle$ of a value and a new view as an argument, describes the updated resources that exist at the end of this execution, if it terminates. This is a logic of partial correctness.

[7]For the sake of readability, we hide the "later" modalities that appear in these rules. See Jung et al. [2018b, §6.2] for details.

BASE-NA-ALLOC

$$\begin{array}{c} \mathsf{True} \\ \hline \langle \mathsf{new}_{\mathsf{na}}\ v, \mathcal{W}\rangle \\ \hline \lambda \langle v', \mathcal{W}'\rangle .\ \exists a.\ \mathop{\ast} \begin{cases} v' = a \\ \mathcal{W}' = \mathcal{W} \\ a \rightsquigarrow_{\mathrm{na}} \{0 \mapsto v\} \end{cases} \end{array}$$

BASE-AT-ALLOC

$$\begin{array}{c} \mathit{valid}\,\mathcal{W} \\ \hline \langle \mathsf{new}_{\mathsf{at}}\ v, \mathcal{W}\rangle \\ \hline \lambda \langle v', \mathcal{W}'\rangle .\ \exists A.\ \mathop{\ast} \begin{cases} v' = A \\ \mathcal{W}' = \mathcal{W} \\ A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{W}\rangle \end{cases} \end{array}$$

BASE-NA-READ

$$\begin{array}{c} q \cdot a \rightsquigarrow_{\mathrm{na}} h \ \ast\ \mathit{valid}\,\mathcal{W} \\ \hline \langle !_{\mathsf{na}}\, a, \mathcal{W}\rangle \\ \hline \lambda \langle v', \mathcal{W}'\rangle .\ \exists t.\ \mathop{\ast} \begin{cases} t \in \mathrm{dom}\ h \\ \mathcal{W}(a) \le t \\ v' = h(t) \\ \mathcal{W}' = \mathcal{W} \\ q \cdot a \rightsquigarrow_{\mathrm{na}} h \end{cases} \end{array}$$

BASE-NA-WRITE

$$\begin{array}{c} a \rightsquigarrow_{\mathrm{na}} h \ \ast\ \mathit{valid}\,\mathcal{W} \\ \hline \langle a :=_{\mathsf{na}} v, \mathcal{W}\rangle \\ \hline \lambda \langle v', \mathcal{W}'\rangle .\ \exists t.\ \mathop{\ast} \begin{cases} t \notin \mathrm{dom}\ h \\ \mathcal{W}(a) < t \\ v' = () \\ \mathcal{W}' = \mathcal{W}[a \mapsto t] \\ a \rightsquigarrow_{\mathrm{na}} h[t \mapsto v] \\ \mathit{valid}\,\mathcal{W}' \end{cases} \end{array}$$

BASE-AT-READ

$$\begin{array}{c} q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle \ \ast\ \mathit{valid}\,\mathcal{W} \\ \hline \langle !_{\mathsf{at}}\, A, \mathcal{W}\rangle \\ \hline \lambda \langle v', \mathcal{W}'\rangle .\ \mathop{\ast} \begin{cases} v' = v \\ \mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}' \\ q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle \\ \mathit{valid}\,\mathcal{W}' \end{cases} \end{array}$$

BASE-AT-WRITE

$$\begin{array}{c} A \rightsquigarrow_{\mathrm{at}} \langle \_, \mathcal{V}\rangle \ \ast\ \mathit{valid}\,\mathcal{W} \\ \hline \langle A :=_{\mathsf{at}} v, \mathcal{W}\rangle \\ \hline \lambda \langle v', \mathcal{W}'\rangle .\ \mathop{\ast} \begin{cases} v' = () \\ \mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}' \\ A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \sqcup \mathcal{W}\rangle \\ \mathit{valid}\,\mathcal{W}' \end{cases} \end{array}$$

BASE-CAS-FAILURE

$$\begin{array}{c} v_0 \ne v_1 \ \ast\ A \rightsquigarrow_{\mathrm{at}} \langle v_0, \mathcal{V}\rangle \ \ast\ \mathit{valid}\,\mathcal{W} \\ \hline \langle \mathsf{CAS}\ A\ v_1\ v_2, \mathcal{W}\rangle \\ \hline \lambda \langle v', \mathcal{W}'\rangle .\ \mathop{\ast} \begin{cases} v' = \mathtt{false} \\ \mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}' \\ A \rightsquigarrow_{\mathrm{at}} \langle v_0, \mathcal{V}\rangle \\ \mathit{valid}\,\mathcal{W}' \end{cases} \end{array}$$

BASE-CAS-SUCCESS

$$\begin{array}{c} A \rightsquigarrow_{\mathrm{at}} \langle v_1, \mathcal{V}\rangle \ \ast\ \mathit{valid}\,\mathcal{W} \\ \hline \langle \mathsf{CAS}\ A\ v_1\ v_2, \mathcal{W}\rangle \\ \hline \lambda \langle v', \mathcal{W}'\rangle .\ \mathop{\ast} \begin{cases} v' = \mathtt{true} \\ \mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}' \\ A \rightsquigarrow_{\mathrm{at}} \langle v_2, \mathcal{V} \sqcup \mathcal{W}\rangle \\ \mathit{valid}\,\mathcal{W}' \end{cases} \end{array}$$

Fig. 6. BaseCosmo: triples for the memory access operations

The word "axiom" is used because each rule is just a fact of the form "triple", as opposed to an implication of the form "triple implies triple". This does *not* mean that we accept these rules without justification. Each "axiom" in Figure 6 is in reality a lemma that we prove.

Allocating a nonatomic memory location (BASE-NA-ALLOC) returns a memory location a and does not change the view of the current thread. The points-to assertion $a \rightsquigarrow_{\mathrm{na}} \{0 \mapsto v\}$ in the postcondition represents the full ownership of the newly-allocated memory location and guarantees that its history contains a single write of the value v at time stamp 0.

Reading a nonatomic memory location (BASE-NA-READ) requires (possibly shared) ownership of this memory location, which is why the points-to assertion appears in the precondition $q \cdot a \rightsquigarrow_{\mathrm{na}} h$. What can be said of the value v' produced by this instruction? A nonatomic read can read from any write whose time stamp is high enough, according to this thread's view of the location a. Thus, for some time stamp t such that $t \in \text{dom}\ h$ holds (i.e., t is a valid time stamp) and $\mathcal{W}(a) \leq t$ holds (i.e., the view $\mathcal{W}$ allows reading from this time stamp), the value v' must be the value that was written at time t, that is, $h(t)$. The thread's view is unaffected, and the points-to assertion is preserved.

In order to justify a nonatomic read, the validity of the current view is required: the assertion $\mathit{valid}\,\mathcal{W}$ appears in the precondition of base-na-Read. Without this requirement, we would not be able to establish this triple. Indeed, we must prove that this read instruction can make progress, that is, it cannot be stuck. In other words, we must prove that the history h contains a write event whose time stamp is at least $\mathcal{W}(a)$. Quite obviously, in the absence of any hypothesis about the view $\mathcal{W}$, it would be impossible to prove such a fact. Thanks to the validity hypothesis $\mathit{valid}\,\mathcal{W}$, we find that that $\mathcal{W}$ must be a fragment of the global view $\mathcal{G}$. This implies $\mathcal{W}(a) \leq \mathcal{G}(a)$. Furthermore, the state interpretation invariant guarantees $\mathcal{G}(a) \in \text{dom}\ h$. Therefore, this read instruction can read (at least) from the write event whose time stamp is $\mathcal{G}(a)$. Therefore, it is not stuck.

Writing a nonatomic memory location (base-na-Write) requires exclusive ownership of this memory location, which is expressed by the points-to assertion $a \rightsquigarrow_{\text{na}} h$. It also requires the validity of the current view $\mathcal{W}$, as this information is needed to prove that the updated view $\mathcal{W}'$ is valid. In accordance with the operational semantics, the history h is extended with a write event at some time stamp t that is both fresh for the history h and permitted by the view $\mathcal{W}$. The updated points-to assertion $a \rightsquigarrow_{\text{na}} h[t \mapsto v]$ reflects this updated history. The thread's new view $\mathcal{W}'$ is obtained by updating $\mathcal{W}$ with a mapping of a to the time stamp t.

The axioms that govern atomic memory locations are also a little heavy, due to the fact that atomic memory locations play two independent roles: they store a value and a view. Regarding the "value" aspect, these axioms are identical to the standard axioms of Concurrent Separation Logic with fractional permissions. This reflects the sequentially consistent behavior of atomic memory locations. Regarding the "view" aspect, these axioms describe how views are written and read by the atomic memory instructions. This reflects the "release/acquire" behavior of these instructions.

When an atomic memory location is allocated (base-at-Alloc), it is initialized with the view $\mathcal{W}$ of the current thread. This is expressed by the points-to assertion $A \rightsquigarrow_{\text{at}} \langle v, \mathcal{W} \rangle$ in the postcondition.

When an atomic memory location is read (base-at-Read), the view stored at this location is merged into the view of the current thread: this is an "acquire read". This is expressed by the inequality $\mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}'$. We cannot expect to obtain an equality $\mathcal{V} \sqcup \mathcal{W} = \mathcal{W}'$ because, according to our definition of the atomic points-to assertion (§3.2), $\mathcal{V}$ is only a fragment of the view that is stored at location A. Thus, we get only a lower bound on the thread's new view $\mathcal{W}'$. Nevertheless, we can prove that $\mathcal{W}'$ is valid.

When performing an atomic write (base-at-Write), the same "acquisition" phenomenon occurs: we get $\mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}'$. Furthermore, this thread's view is merged into the view stored at this location: this is a "release write". This is expressed by the updated points-to assertion $A \rightsquigarrow_{\text{at}} \langle v, \mathcal{V} \sqcup \mathcal{W} \rangle$.

The two axioms associated with the CAS instruction respectively describe the case where this instruction fails (base-cas-Failure) and the case where it succeeds (base-cas-Success). These axioms express the idea that a CAS behaves as a read followed (in case of success) with a write.

3.4 Soundness of BaseCosmo

By relying on Iris's generic soundness theorem, we establish the following result. Recall that $\bot$ is the empty view and $\varnothing$ is the empty store. The theorem states that, if the user can prove anything about e using BaseCosmo, then e is safe, that is, its execution cannot lead to a stuck configuration.

Theorem 3.1 (Soundness of BaseCosmo). *If the entailment "*$\mathit{valid}\,\bot \vdash \text{wp}\ \langle e, \bot \rangle\ \{\mathsf{True}\}$*" holds, then the configuration* $\varnothing; \langle e, \bot \rangle$ *is safe to execute.*

4 A HIGHER-LEVEL LOGIC: COSMO

The program logic that we have just presented, BaseCosmo, is definitely low-level, as it exposes the details of the Multicore OCaml memory model. Several aspects of it can be criticized:

(1) The assertion $q \cdot a \rightsquigarrow_{\text{na}} h$ exposes the fact that a nonatomic memory location stores a history h, as opposed to a single value v. The axioms BASE-NA-READ and BASE-NA-WRITE paraphrase the operational semantics and reveal the time stamp machinery. This makes it difficult to reason about nonatomic memory locations. Yet, at least in the absence of data races on these locations, one would like to reason in a simpler way. Is it possible to offer higher-level points-to assertions and axioms, so that a nonatomic location appears to store a single value?

(2) The view $\mathcal{W}$ of the current thread is explicitly named in every triple $\{P\}\ \langle e, \mathcal{W}\rangle\ \{\Phi\}$, and its validity is typically explicitly asserted as part of every pre- and postcondition. This seems heavy. Is it possible to make this view everywhere implicit by default, and to have a way of referring to it only where needed?

In this section, we answer these questions in the affirmative. On top of the low-level logic BaseCosmo, we build a higher-level logic, Cosmo, where the points-to assertion for nonatomic locations takes the traditional form $q \cdot a \rightsquigarrow_{\text{na}} v$, as in Concurrent Separation Logic with fractional permissions [Boyland 2003; Bornat et al. 2005]. This assertion is strong enough to guarantee that reading a will yield the value v. Therefore, at an intuitive level, one can take it to mean that "the location a currently contains v", or slightly more accurately, "in the eyes of this thread, the location a currently contains v". In more technical terms, this assertion guarantees that *the most recent write to the location a is a write of the value v* and that *this thread is aware of this write.*

The meaning of this simplified points-to assertion is relative to "this thread", that is, to a certain thread about which one is presently reasoning. More precisely, its meaning depends on "this thread's view", as the assertion claims that a certain write event is part of this thread's view. Therefore, to give meaning to this simplified points-to assertion, we find that we must parameterize every assertion with a view: in other words, a Cosmo assertion must denote *a function of a view to a BaseCosmo assertion.* This change in perspective not only seems required in order to address item 1 above, but also addresses item 2 at the same time.

Following Kaiser et al. [2017] and Dang et al. [2020], we require every Cosmo assertion to denote a *monotone* function with respect to the information ordering $\sqsubseteq$ (§2.1.4). This guarantees that, as new memory events become visible to this thread, the validity of every assertion is preserved. This condition makes the frame rule sound at the level of Cosmo. In summary, the type vPROP of Cosmo assertions is defined as the type of monotone functions of VIEW to IPROP (Figure 7).

In this section, we describe how Cosmo is constructed on top of BaseCosmo. A user of Cosmo need not be aware of this construction: the assertions and reasoning rules of Cosmo can be presented to her directly. Nevertheless, a Cosmo proof is a BaseCosmo proof, and it is therefore easy to combine proofs carried in the two logics, should Cosmo alone not be expressive enough.

4.1 Cosmo: Syntax and Meaning of Assertions

Figure 7 defines a number of ways of constructing Cosmo assertions. The assertion $\uparrow\mathcal{V}$, pronounced "*I have* $\mathcal{V}$", asserts that this thread's view contains the view $\mathcal{V}$. It is defined as the function $\lambda\mathcal{W}.\, \mathcal{V} \sqsubseteq \mathcal{W}$, where the formal parameter $\mathcal{W}$ represents this thread's view. It is easy to see that this is a monotone function of $\mathcal{V}$. The assertion $\uparrow\mathcal{V}$ is the archetypical example of a *subjective* assertion, that is, an assertion whose meaning depends on this thread's view. It satisfies the first two axioms in Figure 8. SEEN-ZERO states that to have nothing is the same as to have the empty view. SEEN-TWO states that to have two views $\mathcal{V}_1$ and $\mathcal{V}_2$ separately is the same as to have their

$$\begin{aligned}
\text{vProp} &\triangleq \text{View} \xrightarrow{\text{mon}} \text{iProp} \\
\uparrow\mathcal{V} &\triangleq \lambda\mathcal{W}.\, \mathcal{V} \sqsubseteq \mathcal{W} \\
P * Q &\triangleq \lambda\mathcal{W}.\, P(\mathcal{W}) * Q(\mathcal{W}) \\
P \mathrel{-\!\!*} Q &\triangleq \lambda\mathcal{W}.\, \forall\mathcal{V} \sqsupseteq \mathcal{W}.\, P(\mathcal{V}) \mathrel{-\!\!*} Q(\mathcal{V}) \\
\exists x.\, P &\triangleq \lambda\mathcal{W}.\, \exists x.\, P(\mathcal{W}) \\
\forall x.\, P &\triangleq \lambda\mathcal{W}.\, \forall x.\, P(\mathcal{W}) \\
\lceil P \rceil &\triangleq \lambda\mathcal{W}.\, P \qquad \text{where } P : \text{iProp} \\
P@\mathcal{V} : \text{iProp} &\triangleq P(\mathcal{V}) \qquad \text{where } P : \text{vProp}
\end{aligned}
\qquad
q \cdot a \rightsquigarrow_{\text{na}} v \triangleq \exists h.\, \exists t.\; \mathop{*}\begin{cases}
\lceil q \cdot a \rightsquigarrow_{\text{na}} h \rceil \\
\lceil t = \max\,(\text{dom}\ h) \rceil \\
\lceil h(t) = v \rceil \\
\exists\mathcal{V}.\ \lceil \mathcal{V}(a) = t \rceil * \uparrow\mathcal{V}
\end{cases}$$

Fig. 7. Cosmo: assertions and their meaning

join $\mathcal{V}_1 \sqcup \mathcal{V}_2$. This implies that the assertion $\uparrow\mathcal{V}$ is duplicable and that it is anti-monotone with respect to $\mathcal{V}$: to have a larger view implies to have a smaller view.

The nature of views is hidden from a user of Cosmo. That is, views can be presented to the user as an abstract type, equipped with a distinguished element $\bot$ and with a join operation $\sqcup$ that is associative, commutative, idempotent, and admits $\bot$ as a unit. This means that the user need not (and must not) think of views as "functions of nonatomic locations to time stamps", or as "sets of write events". Instead, the user should think of a view as a certain amount of "information" about the (nonatomic part of the) shared memory. The deduction rules of the logic allow the user to reason abstractly about the manner in which this information is acquired and transmitted and about the places where it is needed.

The next four lines in Figure 7 lift the standard connectives of Separation Logic from BaseCosmo up to Cosmo. In the definition of the magic wand, a universal quantification over a future view $\mathcal{V}$ that contains $\mathcal{W}$ is used so as to obtain a monotone function of $\mathcal{W}$; this is a standard technique.

The following line defines how to lift a BaseCosmo assertion P up to a Cosmo assertion $\lceil P \rceil$. This provides a means of communication between BaseCosmo and Cosmo. The definition is simple: $\lceil P \rceil$ is the constant function $\lambda\mathcal{W}.\, P$. It is the archetypical example of an *objective* assertion, that is, an assertion whose meaning is independent of this thread's view. We often write just P for $\lceil P \rceil$.

The last line in Figure 7 (left) defines $P@\mathcal{V}$ as sugar for the function application $P(\mathcal{V})$. This provides a means of communication in the reverse direction: if P is a Cosmo assertion, then $P@\mathcal{V}$ is a BaseCosmo assertion, which can be read as "*P holds at $\mathcal{V}$*".

Thanks to the constructs that have just been introduced, it is possible to express the idea that every Cosmo assertion P can be split into a subjective component and an objective component. This is stated by the axiom Split-Subjective-Objective in Figure 8. When read from left to right, the axiom splits P into a subjective component $\uparrow\mathcal{V}$ and an objective component $P@\mathcal{V}$, for some view $\mathcal{V}$. (The witness for $\mathcal{V}$ is this thread's view at the time of splitting.) When read from right to left, the axioms reunites these components and yields P again.

Here are several typical examples of BaseCosmo assertions that can be usefully lifted up, yielding objective Cosmo assertions. (In each case, we omit the brackets $\lceil\cdot\rceil$.)

- A Cosmo assertion at a fixed view $P@\mathcal{V}$.
- The fractional ownership of an atomic memory location $q \cdot A \rightsquigarrow_{\text{at}} \langle v, \mathcal{V} \rangle$.
- The ownership of a piece of ghost state $\boxed{m}^{\gamma}$.
- The knowledge of an invariant $\boxed{I}$.

We emphasize that, in the last case above, $\boxed{I}$ is an ordinary BaseCosmo assertion. This means that the assertion I that appears inside the box must also be a BaseCosmo assertion. That is, the logic Cosmo is subject to a restriction: *in every invariant* $\boxed{I}$*, the assertion I must be objective.* A subjective assertion, such as $\uparrow\mathcal{V}$, cannot appear in an invariant. In short, because an invariant represents knowledge that is shared by all threads, it cannot depend on some thread's view.

At first, this restriction might seem problematic: in general, an arbitrary Cosmo assertion P cannot appear in an invariant, therefore cannot be transmitted from one thread to another. Fortunately, in many situations, it is possible to work around this limitation. The idea is to decompose P into a subjective part and an objective part, via Split-Subjective-Objective; to let the objective part appear in the invariant, thereby allowing it to be shared between threads; and to transmit the subjective part via explicit synchronization operations, typically writes and reads of atomic locations. Our spin lock implementation (§5.2) offers a typical example of this idiom.

The last key definition is that of the nonatomic points-to assertion (Figure 7, right). As announced earlier, this assertion takes the traditional form $q \cdot a \rightsquigarrow_{\text{na}} v$, and means that *the most recent write to the location a is a write of the value v* and that *this thread is aware of this write.* Its definition, whose right-hand side is a conjunction of four assertions, reflects this:

(1) $q \cdot a \rightsquigarrow_{\text{na}} h$ claims a fraction q of the location a and guarantees that its history is h.
(2) $t = \max\,(\text{dom}\ h)$ guarantees that t is the time stamp of the most recent write event at a.
(3) $h(t) = v$ indicates that v is the value written by this write event.
(4) $\exists \mathcal{V}.\ \ulcorner \mathcal{V}(a) = t \urcorner\ * \uparrow\mathcal{V}$ guarantees that this write event is visible to this thread.

Because $\uparrow\mathcal{V}$ is subjective, the nonatomic points-to assertion $q \cdot a \rightsquigarrow_{\text{na}} v$ is itself subjective. Therefore, it cannot appear in an invariant. This is the price to pay for the apparent simplicity of this predicate.

4.2 Cosmo: Syntax and Meaning of Weakest-Precondition Assertions

We have just presented the universe of Cosmo assertions, which have type vProp, in contrast with BaseCosmo assertions, which have type iProp. We now wish to define a Cosmo weakest-precondition assertion $\text{wp}\ e\ \{\Psi\}$ whose postcondition Ψ is a function of a value to a Cosmo assertion. This is in contrast with BaseCosmo's weakest-precondition assertion $\text{wp}\ \langle e, \mathcal{V}\rangle\ \{\Phi\}$ (§3.2) where Φ is a function of a value and a view to a BaseCosmo assertion. We define the former on top of the latter, as follows:

$$\begin{aligned}\text{wp}\ e\ \{\Psi\} \quad \triangleq \quad & \lambda \mathcal{W}. \\ & \quad \forall \mathcal{V} \sqsupseteq \mathcal{W}. \\ & \qquad \mathit{valid}\,\mathcal{V} \twoheadrightarrow \text{wp}\ \langle e, \mathcal{V}\rangle\ \{\lambda\,\langle v', \mathcal{V}'\rangle\,.\,\Psi(v')(\mathcal{V}')\ *\ \mathit{valid}\,\mathcal{V}'\}\end{aligned}$$

Because $\text{wp}\ e\ \{\Psi\}$ must have type vProp, it must be a monotone function of this thread's view $\mathcal{W}$. In order to make it monotone, we quantify over a future view $\mathcal{V}$ that contains $\mathcal{W}$. We use a BaseCosmo weakest-precondition assertion to require that, from the view $\mathcal{V}$, executing the expression e must be safe and must yield a value and a view that satisfy Ψ. As a final touch, we place validity assertions in the hypothesis and in the postcondition so as to maintain the invariant that "this thread's view is valid", thus removing from the user the burden of keeping track of this information.

The Cosmo triple is derived from the Cosmo weakest-precondition assertion in the usual way: $\{P\}\ e\ \{\Psi\}$ stands for $\Box(P \twoheadrightarrow \text{wp}\ e\ \{\Psi\})$.

4.3 Soundness of Cosmo

Cosmo, equipped with the weakest-precondition assertion that was just defined, is sound. This follows straightforwardly from the soundness of BaseCosmo (§3.4).

Theorem 4.1 (Soundness of Cosmo). *If the entailment* $\vdash \text{wp}\ e\ \{\text{True}\}$ *holds then the configuration* $\varnothing; \langle e, \bot\rangle$ *is safe to execute.*

Seen-Zero
$$\vdash \uparrow\bot$$

Seen-Two
$$\uparrow\mathcal{V}_1 \ast \uparrow\mathcal{V}_2 \dashv\vdash \uparrow(\mathcal{V}_1 \sqcup \mathcal{V}_2)$$

Split-Subjective-Objective
$$P \dashv\vdash \exists\mathcal{V}.\,(\uparrow\mathcal{V} \ast P@\mathcal{V})$$

Fig. 8. Cosmo: axioms

na-Alloc
$$\begin{array}{c} \mathsf{True} \\ \hline \mathtt{new}_{\mathtt{na}}\ v \\ \hline \lambda v'.\, \exists a.\, v' = a \ast a \rightsquigarrow_{\mathrm{na}} v \end{array}$$

na-Read
$$\begin{array}{c} q \cdot a \rightsquigarrow_{\mathrm{na}} v \\ \hline !_{\mathrm{na}}\, a \\ \hline \lambda v'.\, v' = v \ast q \cdot a \rightsquigarrow_{\mathrm{na}} v \end{array}$$

na-Write
$$\begin{array}{c} a \rightsquigarrow_{\mathrm{na}} \_ \\ \hline a :=_{\mathrm{na}} v \\ \hline \lambda ().\, a \rightsquigarrow_{\mathrm{na}} v \end{array}$$

at-Alloc
$$\begin{array}{c} \uparrow\mathcal{V} \\ \hline \mathtt{new}_{\mathtt{at}}\ v \\ \hline \lambda v'.\, \exists A.\, \ast\begin{cases} v' = A \\ A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle \end{cases} \end{array}$$

at-Read
$$\begin{array}{c} q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle \\ \hline !_{\mathrm{at}}\, A \\ \hline \lambda v'.\, \ast\begin{cases} v' = v \\ q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle \\ \uparrow\mathcal{V} \end{cases} \end{array}$$

at-Write
$$\begin{array}{c} A \rightsquigarrow_{\mathrm{at}} \langle \_, \mathcal{V}\rangle \ast \uparrow\mathcal{V}' \\ \hline A :=_{\mathrm{at}} v \\ \hline \lambda ().\, \ast\begin{cases} A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \sqcup \mathcal{V}'\rangle \\ \uparrow\mathcal{V} \end{cases} \end{array}$$

cas-Failure
$$\begin{array}{c} v_0 \neq v_1 \ast A \rightsquigarrow_{\mathrm{at}} \langle v_0, \mathcal{V}\rangle \ast \uparrow\mathcal{V}' \\ \hline \mathsf{CAS}\ A\ v_1\ v_2 \\ \hline \lambda v'.\, \ast\begin{cases} v' = \mathtt{false} \\ A \rightsquigarrow_{\mathrm{at}} \langle v_0, \mathcal{V}\rangle \\ \uparrow\mathcal{V} \end{cases} \end{array}$$

cas-Success
$$\begin{array}{c} A \rightsquigarrow_{\mathrm{at}} \langle v_1, \mathcal{V}\rangle \ast \uparrow\mathcal{V}' \\ \hline \mathsf{CAS}\ A\ v_1\ v_2 \\ \hline \lambda v'.\, \ast\begin{cases} v' = \mathtt{true} \\ A \rightsquigarrow_{\mathrm{at}} \langle v_2, \mathcal{V} \sqcup \mathcal{V}'\rangle \\ \uparrow\mathcal{V} \end{cases} \end{array}$$

Fig. 9. Cosmo: triples for the memory access operations

4.4 Cosmo: Axioms

Each of the BaseCosmo axioms described in §3.3 can now be used as a basis to establish a higher-level Cosmo axiom, whose statement is often simpler. The resulting axioms appear in Figure 9.

The first three axioms describe the operations of allocating, reading, and writing nonatomic memory locations. Allocation (na-Alloc) requires nothing and produces the points-to assertion $a \rightsquigarrow_{\mathrm{na}} v$, which is sugar for $1 \cdot a \rightsquigarrow_{\mathrm{na}} v$, and represents the exclusive ownership of the fresh memory location a. Reading (na-Read) requires $q \cdot a \rightsquigarrow_{\mathrm{na}} v$, which represents possibly-shared ownership of the memory location a, and preserves this assertion. Writing (na-Write) requires $a \rightsquigarrow_{\mathrm{na}} \_$, which represents exclusive ownership of a, and updates it to $a \rightsquigarrow_{\mathrm{na}} v$.

We expect the reader to find these axioms unsurprising: indeed, they are identical to the axioms that govern access to memory in Concurrent Separation Logic with fractional permissions [Bornat et al. 2005]. In the absence of a mechanism that allows a points-to assertion to be shared between several threads,[8] these axioms forbid data races. In Cosmo, as explained earlier, a nonatomic points-to assertion cannot appear in an invariant, as it is not an objective assertion. Therefore, Cosmo forbids data races on nonatomic memory locations. In other words, for a program to be verifiable

[8]Sharing an assertion between several threads is usually permitted either via a runtime synchronization mechanism, such as a critical region [O'Hearn 2007, Section 4] or a lock [Gotsman et al. 2007; Hobor et al. 2008], or by a purely static mechanism, such as an invariant that can be accessed for the duration of an atomic instruction, as in some variants of Concurrent Separation Logic [Parkinson et al. 2007] and in Iris [Jung et al. 2018b].

$$\begin{array}{c}\textsc{at-Alloc-SC}\\ \mathsf{True}\\ \hline \mathtt{new_{at}}\ v\\ \hline \lambda v'.\, \exists A.\, v' = A \ast A \rightsquigarrow_{\mathrm{at}} v\end{array}\qquad\begin{array}{c}\textsc{at-Read-SC}\\ q \cdot A \rightsquigarrow_{\mathrm{at}} v\\ \hline !_{\mathrm{at}}\, A\\ \hline \lambda v'.\, v' = v \ast q \cdot A \rightsquigarrow_{\mathrm{at}} v\end{array}\qquad\begin{array}{c}\textsc{at-Write-SC}\\ A \rightsquigarrow_{\mathrm{at}} \_\\ \hline A :=_{\mathrm{at}} v\\ \hline \lambda ().\, A \rightsquigarrow_{\mathrm{at}} v\end{array}$$

$$\begin{array}{c}\textsc{cas-Failure-SC}\\ v_0 \neq v_1 \ast A \rightsquigarrow_{\mathrm{at}} v_0\\ \hline \mathsf{CAS}\ A\ v_1\ v_2\\ \hline \lambda v'.\, v' = \mathtt{false} \ast A \rightsquigarrow_{\mathrm{at}} v_0\end{array}\qquad\begin{array}{c}\textsc{cas-Success-SC}\\ A \rightsquigarrow_{\mathrm{at}} v_1\\ \hline \mathsf{CAS}\ A\ v_1\ v_2\\ \hline \lambda v'.\, v' = \mathtt{true} \ast A \rightsquigarrow_{\mathrm{at}} v_2\end{array}$$

Fig. 10. Cosmo: derived triples for atomic memory locations ignoring views

in Cosmo, all accesses to nonatomic memory locations must be properly synchronized via other means, such as reads and writes of atomic memory locations.

The next five axioms describe the operations on atomic memory locations, namely allocation, reading, writing, and CAS. They are analogous to their BaseCosmo counterparts (Figure 6), yet simpler, as the validity assertions have vanished, and this thread's view is no longer named: instead, assertions of the form $\uparrow\mathcal{V}$ are used to indicate partial knowledge of this thread's view.

These axioms deal with two aspects of atomic memory locations, namely the fact that an atomic memory location holds a value, and the fact that an atomic memory location holds a view. Fortunately, these two aspects are essentially independent of one another. Furthermore, the second aspect can be ignored when it is not relevant: indeed, from the axioms of Figure 9, one can easily derive a set of simplified axioms, shown in Figure 10. These derived axioms are the standard axioms that govern access to memory in Concurrent Separation Logic with fractional permissions. In Figure 10, we omit the view $\mathcal{V}$ carried by atomic locations since it is not relevent: we write $q \cdot A \rightsquigarrow_{\mathrm{at}} v$ for $\exists \mathcal{V}.\, q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle$, which is logically equivalent to $q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \bot\rangle$.

In Cosmo, because an atomic points-to assertion is objective, it *can* be shared between several threads, via an Iris invariant. Therefore, the axioms of both Figure 9 and Figure 10 allow data races on atomic memory locations. By using just the derived axioms of Figure 10, one can reason in Cosmo about atomic memory locations exactly in the same way as one reasons in Concurrent Separation Logic under the assumption of sequential consistency [Parkinson et al. 2007].

Allocating an atomic memory location (at-Alloc) requires this thread to have some view $\mathcal{V}$, as witnessed by the precondition $\uparrow\mathcal{V}$. In the postcondition, one obtains an atomic points-to assertion $A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle$, which is sugar for $1 \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle$, and represents the exclusive ownership of the fresh memory location A. This assertion states that the memory location A holds the value v and a view that is at least as good as the view $\mathcal{V}$. This axiom offers flexibility regarding the choice of $\mathcal{V}$. Indeed, $\mathcal{V}$ need not reflect all of the information that is currently available to this thread: it can be a partial view. (The rule Seen-Two, exploited from right to left, weakens $\uparrow(\mathcal{V}_1 \sqcup \mathcal{V}_2)$ to $\uparrow\mathcal{V}_1$.) In fact, if desired, one can always take $\mathcal{V}$ to be the empty view. (The rule Seen-Zero allows creating $\uparrow\bot$ out of thin air.) This is how the derived axiom at-Alloc-SC is obtained.

Reading an atomic memory location (at-Read) requires a fractional points-to assertion $q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle$ and preserves it. The last conjunct of the postcondition, $\uparrow\mathcal{V}$, reflects the fact that the view held at location A becomes part of this thread's view: this is an "acquire read". The derived axiom at-Read-SC is obtained by dropping this information.

Writing an atomic memory location (at-Write) requires an exclusive points-to assertion $A \rightsquigarrow_{\mathrm{at}} \langle \_, \mathcal{V}\rangle$ as well as (possibly partial) information about this thread's view $\uparrow\mathcal{V}'$. In the postcondition,

$$\frac{\dfrac{P}{\texttt{make ()}}}{\lambda\ell.\,\exists \mathsf{locked}.\, \mathop{\Large *}\begin{cases}\{\mathsf{True}\}\ \texttt{acquire}\ \ell\ \{\lambda().\,P * \mathsf{locked}\}\\ \{P * \mathsf{locked}\}\ \texttt{release}\ \ell\ \{\lambda().\,\mathsf{True}\}\end{cases}}$$

Fig. 11. A specification of the “lock” data structure

$$\begin{aligned}\texttt{make} &\triangleq \lambda().\,\texttt{new}_{\mathsf{at}}\ \texttt{false}\\ \texttt{acquire} &\triangleq \mu\mathsf{acquire}.\lambda\ell.\ \texttt{if CAS}\ \ell\ \texttt{false true then () else acquire}\ \ell\\ \texttt{release} &\triangleq \lambda\ell.\,\ell :=_{\mathsf{at}} \texttt{false}\end{aligned}$$

Fig. 12. Implementation of a spin lock

the points-to assertion is updated to $A \rightsquigarrow_{\mathsf{at}} \langle v, \mathcal{V} \sqcup \mathcal{V}'\rangle$, which reflects the fact that both the value v and the view $\mathcal{V}'$ are written to the memory location A: this is a “release write”. Furthermore, the second conjunct of the postcondition, $\uparrow\mathcal{V}$, indicates that the view $\mathcal{V}$ is acquired by this thread; indeed, an atomic write has both “release” and “acquire” effects. Again, it is possible to ignore these details when they are irrelevant. In particular, the rule AT-WRITE-SC is obtained by letting $\mathcal{V}$ remain undetermined, by letting $\mathcal{V}'$ be the empty view, and by dropping $\uparrow\mathcal{V}$ from the postcondition.

CAS-FAILURE and CAS-SUCCESS combine the axioms for reading and writing an atomic location. A failed CAS instruction does not affect the content of the memory location, but still acquires a view $\mathcal{V}$ from it. A successful CAS instruction writes a value and a view to the memory location and acquires a view from it. Again, if one does not care about these information transfers, then one can use the simplified axioms CAS-FAILURE-SC and CAS-SUCCESS-SC in Figure 10.

5 CASE STUDIES

We now illustrate the use of Cosmo by proving the functional correctness of several simple concurrent data structures: a spin lock (§5.2), a ticket lock (§5.3), and a lock based on Peterson’s algorithm (§5.4), which supports at most two threads.

5.1 Specification of Locks

The lock is perhaps the most basic and best known concurrent data structure. It supports three operations, namely `make`, `acquire`, and `release`. One way of describing its purpose is to say that it allows achieving *mutual exclusion* between several threads: that is, the `acquire` and `release` operations delimit *critical sections* in the code, and the purpose of the lock is to guarantee that no two threads can be in a critical section “at the same time”. However, this is not a very good description, especially in a weak memory setting, where the intuitive notions of “time” and “simultaneity” do not make much sense. What matters, really, is that a lock mediates access to a shared resource, and does so in a correct manner. A thread that successfully acquires the lock expects the resource to be in a consistent state, and expects to be allowed to affect this state in arbitrary ways, as long as it brings the resource back to a consistent state by the time it releases the lock.

This idea is expressed in Concurrent Separation Logic in a simple and elegant manner [O’Hearn 2007]. The classic specification of dynamically-allocated locks in Concurrent Separation Logic [Gotsman et al. 2007; Hobor et al. 2008; Buisse et al. 2011; Sieczkowski et al. 2015; Birkedal and Bizjak 2018], a version of which appears in Figure 11, allows the user to choose an *invariant* P when a lock is allocated. This invariant is a Separation Logic assertion. It appears in the postcondition of `acquire` and in the precondition of `release`, which means that a thread that acquires the lock gets access to the invariant, and can subsequently break this invariant if desired, while a thread that releases the lock must restore and relinquish the invariant.

In Figure 11, the entire specification is contained in a triple for `make`, whose precondition requires the invariant to hold initially, and whose postcondition contains triples for `acquire` and `release`. Because a triple is persistent [Jung et al. 2018b], therefore duplicable, several threads can share the use of the newly-created lock.

In addition to the invariant P, an abstract 0-ary predicate "locked" appears in the postcondition of `acquire` and in the precondition of `release`. Because it is abstract, it must be regarded as nonduplicable by the user. Therefore, it can be thought of as a "token", a witness that the lock is currently held. This witness is required and consumed by `release`. There are two reasons why it is desirable to let such a token appear in the specification. First, from a user's point of view, this is a useful feature, as it forbids releasing a lock that one does not hold, an error that could arise in the (somewhat unlikely) situation where several copies of the invariant P can coexist. Second, from an implementor's point of view, this makes the specification easier to satisfy. In a ticket lock implementation (§5.3), for instance, only the thread that holds the lock can safely release it. Thus, a ticket lock does not satisfy a stronger specification of locks where the "locked" token is omitted.

If one can prove that an implementation of locks satisfies this specification, then (because Separation Logic is compositional) a user can safely rely on this specification to *reason* about her use of locks in an application program. This is a necessary and sufficient condition for an implementation of locks to be considered correct. Proving something along the lines of "no two threads can be in a critical section at the same time" is not necessary, nor would it be sufficient.

We emphasize that, when this specification is understood in the setting of Cosmo, the user invariant P is an arbitrary Cosmo assertion, thus possibly a subjective assertion: no restriction bears on P. Indeed, the synchronization performed by the lock at runtime ensures that every thread has a consistent view of the shared resource. This is in contrast with Iris invariants, which involve no runtime synchronization, and therefore must be restricted to objective assertions (§4.1).

We must also emphasize that, because we work in a logic of partial correctness, this specification does not guarantee deadlock freedom, nor does it guarantee any form of fairness. As an extreme example, an implementation where `acquire` diverges satisfies the specification in Figure 11; to see this, let "locked" be False.

5.2 A Spin Lock

A spin lock is a simple implementation of a lock. It relies on a single atomic memory location ℓ, which holds a Boolean value. Its implementation in Multicore OCaml appears in Figure 12. The implementation of `acquire` involves busy-waiting in a loop, whence the name "spin lock".

An Iris proof of correctness of a spin lock [Birkedal and Bizjak 2018, Example 7.36], in a traditional sequentially consistent setting, would rely on the following Iris invariant, which states that either the lock is currently available and the user assertion P holds, or the lock is currently held:

$$\mathsf{isSpinLockSC} \triangleq \boxed{(\ell \leadsto \mathtt{false} * P) \vee (\ell \leadsto \mathtt{true})}$$

In our setting, however, such an invariant does not make sense, and cannot even be expressed, because the assertion P is an arbitrary (possibly subjective) assertion of type vProp, whereas Cosmo requires every Iris invariant $\boxed{I}$ to be formed with an *objective* assertion I of type iProp (§4.1).

We work around this restriction by reformulating this invariant under an objective form: "either the lock is currently available and the user assertion P holds *in the eyes of the thread that last* *released the lock*, or the lock is currently held". Cosmo allows expressing this easily:

$$\mathsf{isSpinLock} \triangleq \boxed{(\exists \mathcal{V}.\, \ell \rightsquigarrow_{\mathrm{at}} \langle \mathtt{false}, \mathcal{V} \rangle \ast P@\mathcal{V}) \vee (\ell \rightsquigarrow_{\mathrm{at}} \mathtt{true})}$$

The left-hand disjunct, which describes the situation where the lock is available, now involves an existential quantification over a view $\mathcal{V}$. The atomic points-to assertion $\ell \rightsquigarrow_{\mathrm{at}} \langle \mathtt{false}, \mathcal{V} \rangle$ indicates that $\mathcal{V}$ is the view currently stored at location ℓ. The objective assertion $P@\mathcal{V}$ indicates that P holds at this view. That is, the assertion P holds in the eyes of whomever last wrote the location ℓ. (We use the present-tense "holds", as opposed to the past-tense "held", because every Cosmo assertion is monotonic in its implicit view parameter.) In other words, P holds in the eyes of the thread that last released the lock.

The right-hand disjunct, which describes the case where the lock is held, uses the simplified points-to assertion $\ell \rightsquigarrow_{\mathrm{at}} \mathtt{true}$, which is sugar for $\ell \rightsquigarrow_{\mathrm{at}} \langle \mathtt{true}, \bot \rangle$ (§4.4). In this case, the view stored at location ℓ is irrelevant.

In a spin lock, a "locked" token is of no use, from the implementor's point of view, so we let "locked" be True. The proofs of the triples for `acquire` and `release` are sketched in the following.

The proof of `acquire` hinges mainly on establishing the following triple for the CAS instruction:

$$\{\mathsf{isSpinLock}\}\ \mathsf{CAS}\ \ell\ \mathtt{false}\ \mathtt{true}\ \{\lambda b.\, b = \mathtt{true} \implies P\}$$

This triple guarantees that if CAS succeeds then one can extract the assertion P. To establish it, we must open the invariant isSpinLock for the duration of the CAS instruction [Jung et al. 2018b, §2.2] and reason separately about the case where the lock is available and the case where it is held. In the latter case, we apply the reasoning rule CAS-FAILURE (Figure 9) and conclude easily. In the former case, we have $\ell \rightsquigarrow_{\mathrm{at}} \langle \mathtt{false}, \mathcal{V} \rangle$ and $P@\mathcal{V}$, for an unknown view $\mathcal{V}$. The rule CAS-SUCCESS (instantiated with the empty view for $\mathcal{V}'$) shows that the outcome of the CAS instruction in this case is $\ell \rightsquigarrow_{\mathrm{at}} \mathtt{true} \ast {\uparrow}\mathcal{V}$. In other words, the CAS instruction achieves the double effect of writing `true` to the location ℓ and acquiring the view $\mathcal{V}$ that was stored at this location by the last write. This is exactly what we need. Indeed, the axiom SPLIT-SUBJECTIVE-OBJECTIVE lets us combine $P@\mathcal{V}$ and ${\uparrow}\mathcal{V}$ to obtain P. By performing an "acquire" read, we have ensured that P holds in the eyes of the thread that has just acquired the lock. Furthermore, the points-to assertion $\ell \rightsquigarrow_{\mathrm{at}} \mathtt{true}$ allows us to establish the right-hand disjunct of the invariant isSpinLock and to close it again.

The proof of `release` exploits SPLIT-SUBJECTIVE-OBJECTIVE in the reverse direction. Per the precondition of `release`, we have P. We split it into two assertions $P@\mathcal{V}$ and ${\uparrow}\mathcal{V}$, where $\mathcal{V}$ is technically a fresh unknown view, but can be thought of as this thread's view. Then, we open the invariant isSpinLock for the duration of the write instruction. We do not know which of the two disjuncts is currently satisfied (indeed, we have no guarantee that the lock is currently held), but we find that, in either case, we have $\ell \rightsquigarrow_{\mathrm{at}} \_$. This allows us to apply the reasoning rule AT-WRITE (Figure 9), which guarantees that, after the write instruction, we have $\ell \rightsquigarrow_{\mathrm{at}} \langle \mathtt{false}, \mathcal{V} \rangle$. In other words, because we have performed a "release" write, we know that, after this write, our view of memory is stored at location ℓ. Because we have $\ell \rightsquigarrow_{\mathrm{at}} \langle \mathtt{false}, \mathcal{V} \rangle$ and $P@\mathcal{V}$, we are able to prove that the left-hand disjunct of isSpinLock holds and to close the invariant.

5.3 A Ticket Lock

The ticket lock is a variant of the spin lock where a "ticket dispenser" is used to serve threads in the order that they arrive, thereby achieving a certain level of fairness. A simple implementation of the ticket lock appears in Figure 13. A lock is a pair (`served`, `next`) of two atomic locations `served` and `next`, each of which stores an integer value. The counter `served` displays the number of the ticket that is currently being served, or ready to be served. The counter `next` displays the number of the next available ticket.

```
    make ≜ λ().
                let served = new_at 0 in let next = new_at 0 in (served, next)
 acquire ≜ μacquire.λℓ.
                let (served, next) = ℓ in
                let $n$ = !_at next in if CAS next $n$ $(n + 1)$ then wait $n$ ℓ else acquire ℓ
    wait ≜ μwait.λn. λℓ.
                let (served, next) = ℓ in if !_at served = $n$ then () else wait $n$ ℓ
 release ≜ λℓ.
                let (served, next) = ℓ in served :=_at (!_at served) + 1
```

Fig. 13. Implementation of a ticket lock

$$
\begin{array}{rcl}
\textit{served} & : & \textsc{Auth}(\textsc{Ex}(\mathbb{N})) \\
\mathsf{locked} & \triangleq & \exists s.\, \boxed{\circ\, s}^{\textit{served}}
\end{array}
\qquad
\begin{array}{rcl}
\textit{issued} & : & \textsc{Auth}(\textsc{DisjointSet}(\mathbb{N})) \\
\mathsf{ticket}\ n & \triangleq & \boxed{\circ\, \{n\}}^{\textit{issued}}
\end{array}
$$

$$
\mathsf{isTicketLock} \triangleq \boxed{\exists s, n, \mathcal{V}.\, \mathop{\Huge *} \begin{cases} \mathsf{served} \rightsquigarrow_{\mathsf{at}} \langle s, \mathcal{V} \rangle \;*\; \mathsf{next} \rightsquigarrow_{\mathsf{at}} n \\ \boxed{\bullet\, s}^{\textit{served}} \;*\; \boxed{\bullet\, [0, n)}^{\textit{issued}} \\ (\mathsf{locked} \;*\; P @ \mathcal{V}) \vee \mathsf{ticket}\ s \end{cases}}
$$

Fig. 14. Internal invariant of a ticket lock

A thread that wishes to acquire the lock first obtains a unique number n from the counter `next`, which it increments at the same time. This number is known as a *ticket*. Then, the thread waits for its number to be called: that is, it waits for the counter `served` to contain the value n. When it observes that this is the case, it concludes that it has acquired the lock.

A thread that wishes to release the lock simply increments the counter `served`, so that the next thread in line is allowed to proceed and take the lock. To do so, a CAS instruction is unnecessary: a sequence of a read instruction, an addition, and a write instruction suffices. Indeed, because the lock is held and can be released by only one thread, no interference from other threads is possible.

We now sketch a proof of this ticket lock implementation. Our proof requires a straightforward modification of the proof carried out by Birkedal and Bizjak in a sequentially consistent setting (2018, §9). That is precisely our point: it is our belief and our hope that, in many cases, a traditional Iris proof can be easily ported to Cosmo. The process is mostly a matter of identifying which atomic memory locations also serve as information transmission channels (thanks to the "release/acquire" semantics of atomic writes and reads) and of decorating every Iris invariant with explicit views, where needed, so as to meet the requirement that every invariant must be objective. Here, a moment's thought reveals that only the view stored at the location `served` matters; the view stored at `next` is irrelevant.

The ghost state and the Iris invariant used in the verification of the ticket lock appear in Figure 14.[9]

- The ghost location *served* stores an element of the monoid $\textsc{Auth}(\textsc{Ex}(\mathbb{N}))$. The exclusive assertion $\boxed{\circ\, s}^{\textit{served}}$ represents both the knowledge that ticket s is currently being served and a permission to change who is being served. The exclusive assertion "locked", defined as

[9]This ghost state and this invariant are allocated in the body of `make`, in the scope of the local definitions of `served` and `next`. The proofs of the desired triples for `acquire`, `wait`, and `release` also carried out in this scope. This explains why we are able to get away with an unparameterized definition of isTicketLock: it is a local definition.

$$\begin{array}{c} P \\ \hline \texttt{make ()} \\ \hline \lambda(\ell_0, \ell_1).\ \exists \mathsf{canLock}, \mathsf{locked}.\ \mathop{\Huge\ast} i \in \{0,1\}.\ \mathop{\Huge\ast} \left\{ \begin{array}{c} \mathsf{canLock}\ i \\ \{\mathsf{canLock}\ i\}\ \texttt{acquire}\ \ell_i\ \{\lambda().\, P \ast \mathsf{locked}\ i\} \\ \{P \ast \mathsf{locked}\ i\}\ \texttt{release}\ \ell_i\ \{\lambda().\, \mathsf{canLock}\ i\} \end{array} \right. \end{array}$$

Fig. 15. A specification of a lock that can be used by two threads

```
   make ≜ λ().
              let (turn, flag_0, flag_1) = (new_at 0, new_at false, new_at false) in
              ((turn, 0, flag_0, flag_1), (turn, 1, flag_1, flag_0))
acquire ≜ μacquire.λℓ.
              let (turn, myTid, myFlag, otherFlag) = ℓ in
              myFlag :=_at true;
              turn :=_at 1 − myTid;
              wait turn myTid otherFlag
   wait ≜ μwait.λturn. λmyTid. λotherFlag.
              if !_at otherFlag = true && !_at turn = 1 − myTid then
                wait turn myTid otherFlag
release ≜ λℓ.
              let (turn, myTid, myFlag, otherFlag) = ℓ in
              myFlag :=_at false
```

Fig. 16. Implementation of Peterson's algorithm for two threads

$\exists s.\, \boxed{\circ\, s}^{\mathit{served}}$, represents a permission to change who is being served, therefore a permission to release the lock.

- The ghost location *issued* keeps track of which tickets have been issued since the lock was created. It stores an element of $\textsc{Auth}(\textsc{DisjointSet}(\mathbb{N}))$, where $\textsc{DisjointSet}\ \mathbb{N}$ is the monoid whose elements are finite sets of integers and whose partial composition law is disjoint set union. The exclusive assertion ticket n represents the ownership of the ticket numbered n.
- The invariant isTicketLock synchronizes the physical state and the ghost state by mentioning the auxiliary variables s and n both in points-to assertions and in ghost state ownership assertions. The same technique as in the previous subsection (§5.2) is used to make this invariant objective. The last conjunct in its definition states that either no thread holds the lock and the user assertion P holds *in the eyes of the thread that last released the lock*, or the invariant owns the ticket numbered s. This implies that, in order to acquire the lock while maintaining the invariant, a thread must present and relinquish the ticket numbered s.

We do not sketch the proof further, as it is analogous to Birkedal and Bizjak's proof (2018, §9), with the added detail that the read of `served` in `wait` acquires a view and the write of `served` in `release` releases a view.

5.4 Peterson's Algorithm

Peterson's algorithm [Raynal 2013, §2.1.2] is a mutual exclusion algorithm for two threads, whose implementation requires only read and write operations on sequentially consistent registers: no

$$\begin{tikzcd}[column sep=huge, row sep=large]
\mathbf{O} \arrow[d, "\begin{array}{l}\text{// acquire:}\\ \texttt{myFlag} \mathrel{:=_{\mathrm{at}}} \texttt{true}\end{array}"'] & & \mathbf{C} \arrow[ll, "\begin{array}{l}\text{// release:}\\ \texttt{myFlag} \mathrel{:=_{\mathrm{at}}} \texttt{false}\end{array}"'] \\
\mathbf{A} \arrow[rr, "\begin{array}{l}\text{// acquire:}\\ \texttt{turn} \mathrel{:=_{\mathrm{at}}} 1 - \texttt{myTid}\end{array}"'] & & \mathbf{W} \arrow[u, "\begin{array}{l}\text{// wait:}\\ \text{when } !_{\mathrm{at}}\,\texttt{otherFlag} \text{ returns } \texttt{false}\\ \text{or } !_{\mathrm{at}}\,\texttt{turn} \text{ returns } \texttt{myTid}\end{array}"'] \arrow[loop right, "\begin{array}{l}\text{// wait:}\\ \text{when } !_{\mathrm{at}}\,\texttt{otherFlag} \text{ returns } \texttt{true}\\ \text{and } !_{\mathrm{at}}\,\texttt{turn} \text{ returns } 1 - \texttt{myTid}\end{array}"]
\end{tikzcd}$$

Fig. 17. Internal state of each thread participating in Peterson's algorithm

$$\begin{array}{rcl}
s_0, s_1 & \in & \textsc{PetersonState} \triangleq \{\mathrm{O}, \mathrm{A}, \mathrm{W}, \mathrm{C}\} \\
\gamma_0, \gamma_1 & : & \textsc{Auth}(\textsc{Ex}\ \textsc{PetersonState}) \\
\omega_0, \omega_1 & : & \textsc{Auth}(\textsc{Ex}\ \textsc{View}) \\
\mathsf{petersonInv} & \triangleq & \boxed{\begin{array}{l}
\exists t \in \{0,1\}, f_0, f_1, s_0, s_1, \mathcal{V}, \mathcal{V}_0, \mathcal{V}_1. \\
\quad \ast \left\{ \begin{array}{l}
\texttt{turn} \leadsto_{\mathrm{at}} \langle t, \mathcal{V} \rangle \\
\texttt{flag}_0 \leadsto_{\mathrm{at}} \langle f_0, \mathcal{V}_0 \rangle \ast \boxed{\bullet\, s_0}^{\gamma_0} \ast \boxed{\bullet\, \mathcal{V}_0}^{\omega_0} \ast (s_0 = \mathrm{O} \lor f_0 = \texttt{true}) \\
\texttt{flag}_1 \leadsto_{\mathrm{at}} \langle f_1, \mathcal{V}_1 \rangle \ast \boxed{\bullet\, s_1}^{\gamma_1} \ast \boxed{\bullet\, \mathcal{V}_1}^{\omega_1} \ast (s_1 = \mathrm{O} \lor f_1 = \texttt{true}) \\
(t = 0 \land s_0 = \mathrm{W}) \implies (s_1 = \mathrm{W} \land \mathcal{V} = \mathcal{V}_1) \\
(t = 1 \land s_1 = \mathrm{W}) \implies (s_0 = \mathrm{W} \land \mathcal{V} = \mathcal{V}_0) \\
(s_0 \neq \mathrm{C} \land s_1 \neq \mathrm{C}) \implies P@(\mathcal{V}_0 \sqcup \mathcal{V}_1)
\end{array} \right.
\end{array}} \\
\mathsf{canLock}\ i & \triangleq & \mathsf{petersonInv} \ast \exists \mathcal{V}. \boxed{\circ\, \mathrm{O}}^{\gamma_i} \ast \boxed{\circ\, \mathcal{V}}^{\omega_i} \ast {\uparrow}\mathcal{V} \\
\mathsf{locked}\ i & \triangleq & \mathsf{petersonInv} \ast \exists \mathcal{V}. \boxed{\circ\, \mathrm{C}}^{\gamma_i} \ast \boxed{\circ\, \mathcal{V}}^{\omega_i}
\end{array}$$

Fig. 18. Internal invariant of Peterson's algorithm

CAS is needed. Peterson's algorithm remains valid in Multicore OCaml, provided atomic locations are used in its implementation. This is not an entirely trivial claim, as the user invariant P may involve nonatomic locations.

Like an ordinary lock, this data structure offers three operations, namely `make`, `acquire` and `release`. This time, `make` returns a pair of two handles: each participant thread is expected to use one of these handles. The specification of this data structure (Figure 15) is analogous to the specification of locks (Figure 11), but introduces a new token, canLock i, so as to limit the number of participant threads to two: `make` produces only two tokens canLock 0 and canLock 1. The code, shown in Figure 16, uses three atomic locations:

- Each of the two threads $i \in \{0, 1\}$ uses a Boolean register $\texttt{flag}_i$ to indicate it currently holds the lock or intends to acquire it. Both threads read $\texttt{flag}_i$ but only thread i writes it.
- An integer register `turn` indicates which thread has priority, should both threads simultaneously attempt to acquire the lock. It is read and written by both threads.

We now outline our proof that the code in Figure 16 satisfies the specification in Figure 15. As we use a logic of partial correctness, we do not prove deadlock-freedom or fairness. We only verify that this data structure behaves like a lock.

The algorithm's possible states are summarized in Figure 17. At any time, each of the participant threads is in one of four possible states: either it is outside of a critical section (O), or it has just entered `acquire` and set its flag to `true` (A), or it has written `turn` and is now waiting (W), or it is inside a critical section (C). Figure 18 presents the algorithm's invariant. We use two ghost variables γ_0 and γ_1 to keep track of the logical state of each thread.

The key novelty, with respect to the proof that one could carry out in a sequentially consistent setting, is that the invariant must record the view $\mathcal{V}_i$ that was the view of thread i when this thread last released the lock. This view is stored at the atomic location $\mathtt{flag}_i$. When the lock is available, because the lock could have been last released by either thread, the assertion P holds either at $\mathcal{V}_0$ or at $\mathcal{V}_1$. Because Cosmo assertions are monotonic, this implies that P holds at the combined view $\mathcal{V}_0 \sqcup \mathcal{V}_1$. The invariant records this fact.

Thus, in the proof of `wait`, when `wait` terminates, the invariant can give us $P @ (\mathcal{V}_0 \sqcup \mathcal{V}_1)$. A key step is to argue that we also have $\uparrow \mathcal{V}_1$ and $\uparrow \mathcal{V}_2$, so as to then combine the pieces via Split-Subjective-Objective and obtain P. Let us briefly sketch why this is the case.

Establishing $\uparrow \mathcal{V}_i$ is easy enough, because $\mathcal{V}_i$ is a past view of this very thread. One way of recording this information is to let the token canLock i carry it; indeed, the role of this token is precisely to carry information from a `release` to the next `acquire`. We include in the definition of canLock i an assertion $\uparrow \mathcal{V}$ and set up a ghost variable ω_i whose purpose is to allow us to prove that $\mathcal{V}$ is in fact $\mathcal{V}_i$.

There remains to argue that we have $\mathcal{V}_{1-i}$ when `wait` terminates. Because of the conjunction in the conditional statement, the loop in `wait` has two exit points:

- The loop can terminate immediately after reading $\mathtt{flag}_{1-i}$ yields the value `false`. In this case, the thread acquires $\mathcal{V}_{1-i}$ via this read.
- Or, the loop terminates after reading `turn` yields the value i. We must prove that the thread acquires $\mathcal{V}_{1-i}$ via this read. This is nonobvious: it requires us to argue that the view $\mathcal{V}$ stored at location `turn` must in fact contain $\mathcal{V}_{1-i}$. The intuitive reason why this is true is that the two participants write `turn` in a polite way, always giving priority to the other thread. Thus, thread 0 writes only the value 1 to `turn`, and vice versa. Thus, when thread i reads i from `turn`, this value must have been written by thread $1 - i$, which implies that this read allows thread i to acquire the view $\mathcal{V}_{1-i}$. Technically, this argument is reflected in the invariant by the proposition $(t = 0 \land s_0 = \mathrm{W}) \implies (s_1 = \mathrm{W} \land \mathcal{V} = \mathcal{V}_1)$ and its symmetric counterpart.

6 RELATED WORK

There is a wide variety of work on weak memory models and on approaches to program verification with respect to weak memory models. We restrict our attention to program verification based on extensions of Concurrent Separation Logic, because this is the most closely related work and because we believe that the abstraction and compositionality of Separation Logic are features that will be absolutely essential in the long run.

The first instance of Separation Logic for weak memory appears to be RSL [Vafeiadis and Narayan 2013]. It is based on an axiomatic semantics of a fragment of the C11 memory model. It supports nonatomic accesses, where it enforces the absence of data races; release/acquire accesses, with reasoning rules that allow ownership transfers from writer to reader; and relaxed accesses, without any ownership transfer. The logic involves a permission $Rel(\ell, Q)$ to perform a release write at location ℓ of a value v while relinquishing the assertion $Q\ v$. Symmetrically, there is a permission

$Acq(\ell, Q)$ to perform an acquire read at location ℓ of a value v and obtain the assertion $Q\ v$. The release and acquire permissions are created, and the predicate Q is fixed, when the location is allocated. RSL can verify simple concurrent data structures, such as a spin lock. However, because it lacks invariants and ghost state, its expressive power is limited.

FSL [Doko and Vafeiadis 2016] extends RSL with support for release/acquire fences. A release write can be replaced with a release fence followed with a relaxed write; symmetrically, an acquire read can be replaced with a relaxed read followed with an acquire fence. Two new assertions ΔP and ∇P witness that P has been released by the last release fence or will be acquired by the next acquire fence, respectively. In Cosmo, both of these assertions would be replaced by $P @ \mathcal{V}$, for a well-chosen view $\mathcal{V}$. Multicore OCaml has no fences; instead, atomic read and write instructions are used to transmit views. FSL++ [Doko and Vafeiadis 2017] extends FSL with shared read permissions for nonatomic accesses, support for CAS, and ghost state.

GPS [Turon et al. 2014] supports a fragment of C11 that includes nonatomic accesses and release/acquire accesses. Like the papers cited above, it is based on an axiomatic presentation of the C11 memory model. It introduces ghost state and a notion of per-location protocol that governs a single atomic memory location. At the cost of rather complex reasoning rules, it offers good expressive power. The case studies described in the paper include a spin lock, a bounded ticket lock, a Michael-Scott queue, and a circular FIFO queue. In comparison, Cosmo does not need per-location protocols. Because atomic memory locations in Multicore OCaml have sequentially consistent behavior, our "atomic points-to" predicate is objective. Therefore, in Cosmo, an invariant can refer to one or more atomic memory locations if desired. This has been illustrated in §5.3 and §5.4.

Vafeiadis [2017] offers a good survey of the papers cited thus far.

Sieczkowski et al. [2015], already cited earlier (§1), present a variant of Concurrent Separation Logic that is sound with respect to the TSO memory model. The logic includes a high-level fragment whose reasoning rules are those of Concurrent Separation Logic. The logic is proved sound with respect to an operational semantics where store buffers are explicit. This work and ours seem quite close in spirit, but differ due to the choice of a different memory model. In particular, Sieczkowski et al. have no notion of view. In order to reason about the behavior of store buffers, they propose a few ad hoc logical constructs, such as an "until" modality $P\ \mathcal{U}_t\ Q$, which means that P holds until an update from thread t is flushed to main memory, at which point Q holds.

Kaiser et al. [2017] propose the first instantiation of Iris in a weak-memory setting. This involves abandoning the axiomatic memory models used by the papers cited above and switching to an operational semantics. The paper proposes such a semantics for a fragment of C11 that includes two kinds of memory locations, namely nonatomic locations and release/acquire locations. Like Dolan et al.'s semantics of Multicore OCaml (2018b), this semantics involves time stamps and histories. It also includes a "race detector" which ensures that data races on nonatomic memory locations lead to undefined behavior. In comparison, Dolan et al.'s semantics does not need a race detector: every Multicore OCaml program has a well-defined set of permitted behaviors. Several aspects of our work are modeled after Kaiser et al.'s paper. Indeed, in a first step, they instantiate Iris, yielding a low-level "base logic". Then, in a second step, they define a higher-level logic, whose reasoning rules are easier to use, and whose assertions are implicitly parameterized with the thread's view. We follow this approach. Kaiser et al. construct not one, but two high-level logics, iGPS and iRSL, which are inspired by GPS and RSL, and benefit from the power of Iris. iGPS introduces a new feature, namely single-writer protocols. Cosmo follows a different route: as explained above, it does not need per-location protocols. Furthermore, it puts emphasis on explicit user-level reasoning with abstract views, via assertions like $P @ \mathcal{V}$ and $\uparrow \mathcal{V}$.

iRC11 [Dang et al. 2020] extends iGPS with additional features of the C11 memory model, namely relaxed accesses and release/acquire fences. It is based on ORC11, an operational presentation of the

Repaired C11 memory model [Lahav et al. 2017]. One of its key features is support for cancellable invariants, an abstraction whose implementation in Iris in an SC setting has been well-understood for some time [10], but whose implementation in a weak-memory setting is significantly more challenging. In particular, the tokens that represent a fraction of the ownership of an invariant are not objective, and therefore cannot appear in an invariant; they must be transmitted from one thread to another via a synchronization operation. Dang et al.'s implementation of cancellable invariants involves explicit reasoning about views, yet this is not apparent in the cancellable invariant API, a remarkable achievement.

On top of iRC11, Dang et al. reconstruct Lifetime Logic and the model of the Rust type system previously built by Jung et al. in an SC setting (2018a). Furthermore, they prove the soundness of Rust's atomic reference counter library (ARC).

7 CONCLUSION AND FUTURE WORK

In this paper, we have laid the groundwork for verifying Multicore OCaml programs in a suitable variant of Concurrent Separation Logic. We have instantiated Iris [Jung et al. 2018b] for Multicore OCaml, yielding a low-level logic, BaseCosmo, which exposes all of the details of the Multicore OCaml memory model [Dolan et al. 2018b], including time stamps, histories, and views. BaseCosmo can in principle verify arbitrary Multicore OCaml programs, including programs that involve data races on nonatomic memory locations. However, BaseCosmo is too low-level to be pleasant and convenient. In the higher-level logic Cosmo, we have aimed to provide simpler reasoning principles. In order to achieve this result, we have removed the ability of reasoning about programs that race on nonatomic memory locations. This has allowed us to offer the illusion that a nonatomic memory location stores a single value, to remove all mention of time stamps and histories, and to present views to the user as an abstract type, equipped with the structure of a bounded semilattice. We believe that the manner in which Cosmo allows reasoning about weak memory is original and relatively simple and natural. A key mechanism is the ability to split an arbitrary assertion P into an objective fragment $P @ \mathcal{V}$ and a subjective fragment $\uparrow \mathcal{V}$ and to later reassemble these fragments. The two fragments are transmitted from one thread to another by different means: whereas sharing an objective assertion requires no runtime synchronization and is typically achieved via an Iris invariant, transmitting a view is typically achieved by relying on the "release/acquire" behavior of atomic writes and reads.

In the immediate future, we wish to extend our work with support for (nonatomic and atomic) arrays. We also would like to propose suitable specifications for Multicore OCaml's `Domain` API, which offers a set of primitive low-level synchronization operations, including monitors equipped with `wait` and `notify` operations. On this basis, we would like to verify some of the concurrent data structures that have been developed for Multicore OCaml, namely a multiword compare-and-swap, a suite of lock-free queues, bags, and hash tables, and an implementation of Turon's reagents (2012).

In order to write satisfactory specifications for lock-free data structures, one must express the idea of logical atomicity [Jung et al. 2015]. This can be done, with some limitations, in Iris [Jung 2019]. We have not yet explored how well this works in Cosmo, but are planning to do so.

In the somewhat more distant future, we would like to find concrete examples of Multicore OCaml programs where data races on nonatomic memory locations are correctly exploited and to discover what rules can be proposed to reason about these programs without abandoning the simplicity of Cosmo.

[10] For example, in RustBelt [Jung et al. 2018a], nonatomic persistent borrows are a form of cancellable invariant.

REFERENCES

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Principles of Programming Languages (POPL)*. 55–66. https://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf

John Bender and Jens Palsberg. 2019. A formalization of Java's concurrent access modes. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 142:1–142:28. https://johnbender.us/assets/oopsla-2019.pdf

Lars Birkedal and Aleš Bizjak. 2018. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. (Dec. 2018). https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf Lectures notes.

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Principles of Programming Languages (POPL)*. 259–270. http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/permissions_paper.pdf

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis Symposium (SAS) (Lecture Notes in Computer Science, Vol. 2694)*. Springer, 55–72. http://www.cs.uwm.edu/~boyland/papers/permissions.pdf

Alexandre Buisse, Lars Birkedal, and Kristian Støvring. 2011. A Step-Indexed Kripke Model of Separation Logic for Storable Locks. *Electronic Notes in Theoretical Computer Science* 276 (Sept. 2011), 121–143. https://cs.au.dk/~birke/papers/locks.pdf

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 34:1–34:29. https://hal.inria.fr/hal-02351793/

Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *Verification, Model Checking and Abstract Interpretation (VMCAI) (Lecture Notes in Computer Science, Vol. 9583)*. Springer, 413–430. https://plv.mpi-sws.org/fsl/base/paper.pdf

Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 448–475. https://plv.mpi-sws.org/fsl/ARC/paper.pdf

Stephan Dolan, Anil Madhavapeddy, and KC Sivaramakrishnan. 2018a. The Multicore OCaml development wiki. https://github.com/ocaml-multicore/ocaml-multicore/wiki

Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. 2018b. Bounding data races in space and time. In *Programming Language Design and Implementation (PLDI)*. 242–255. http://kcsrk.info/papers/pldi18-memory.pdf

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science, Vol. 4807)*. Springer, 19–37. http://dx.doi.org/10.1007/978-3-540-76637-7_3

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 4960)*. Springer, 353–367. http://www.cs.princeton.edu/~appel/papers/concurrent.pdf

Ralf Jung. 2019. Logical Atomicity in Iris: the Good, the Bad, and the Ugly. Iris Workshop. https://people.mpi-sws.org/~jung/iris/logatom-talk-2019.pdf

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 66:1–66:34. https://people.mpi-sws.org/~dreyer/papers/rustbelt/paper.pdf

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In *Principles of Programming Languages (POPL)*. 637–650. http://plv.mpi-sws.org/iris/paper.pdf

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *European Conference on Object-Oriented Programming (ECOOP)*. 17:1–17:29. https://people.mpi-sws.org/~dreyer/papers/iris-weak/paper.pdf

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Programming Language Design and Implementation (PLDI)*. 618–632. https://plv.mpi-sws.org/scfix/paper.pdf

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://www.microsoft.com/en-us/research/uploads/prod/2016/12/How-to-Make-a-Multiprocessor-Computer-That-Correctly-Executes-Multiprocess-Programs.pdf

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2019. The OCaml system: documentation and user's manual. http://caml.inria.fr/pub/docs/manual-ocaml/

Andreas Lochbihler. 2012. Java and the Java Memory Model – A Unified, Machine-Checked Formalisation. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7211)*. Springer, 497–517. https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/people/andreloc/lochbihler12esop.pdf

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Principles of Programming Languages (POPL)*. 378–391. http://rsim.cs.uiuc.edu/Pubs/popl05.pdf

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Coq proofs for Cosmo. https://gitlab.inria.fr/gmevel/cosmo.

Peter W. O'Hearn. 2007. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science* 375, 1–3 (May 2007), 271–307. http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/concurrency.pdf

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. 2007. Modular verification of a non-blocking stack. In *Principles of Programming Languages (POPL)*. 297–302. https://doi.org/10.1145/1190216.1190261

Michel Raynal. 2013. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer. https://doi.org/10.1007/978-3-642-32027-9

Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. 2015. A Separation Logic for Fictional Sequential Consistency. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 736–761. https://www.cl.cam.ac.uk/~jp622/a_separation_logic_for_fictional_sequential_consistency.pdf

Aaron Turon. 2012. Reagents: expressing and composing fine-grained concurrency. In *Programming Language Design and Implementation (PLDI)*. 157–168. https://aturon.github.io/academic/reagents.pdf

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 691–707. http://plv.mpi-sws.org/gps/paper.pdf

Viktor Vafeiadis. 2017. Program Verification Under Weak Memory Consistency Using Separation Logic. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 10426)*. Springer, 30–46. https://people.mpi-sws.org/~viktor/papers/cav2017-invited.pdf

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 867–884. https://people.mpi-sws.org/~viktor/papers/oopsla2013-rsl.pdf

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94. http://www.cs.rice.edu/CS/PLT/Publications/Scheme/ic94-wf.ps.gz