Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency

Aaron Turon
MPI-SWS
turon@mpi-sws.org

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Lars Birkedal
Aarhus University
birkedal@cs.au.dk

Abstract

Modular programming and modular verification go hand in hand, but most existing logics for concurrency ignore two crucial forms of modularity: *higher-order functions*, which are essential for building reusable components, and *granularity abstraction*, a key technique for hiding the intricacies of fine-grained concurrent data structures from the clients of those data structures. In this paper, we present CaReSL, the first logic to support the use of granularity abstraction for modular verification of higher-order concurrent programs. After motivating the features of CaReSL through a variety of illustrative examples, we demonstrate its effectiveness by using it to tackle a significant case study: the first formal proof of (partial) correctness for Hendler *et al.*'s "flat combining" algorithm.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Refinement, fine-grained concurrency, linearizability, separation logic, logical relations, data abstraction, local state

1. Introduction

Over the past decade, a number of Hoare logics have been developed to cope with the complexities of concurrent programming [20, 32, 7, 3, 18, 4, 27]. Unsurprisingly, the name of the game in these logics is improving support for *modular* reasoning along a variety of dimensions. Concurrent Abstract Predicates (CAP) [3], for example, utilizes a deft combination of *separation logic* [26] (for spatially-modular reasoning about resources and ownership), *rely-guarantee* [15] (for thread-modular reasoning in the presence of interference), and *abstract predicates* [21] (for hiding invariants about a module's private data structures from its clients).

At the same time, of course, the importance of modularity is not restricted to verification. Programmers use modularity in the *design* of their concurrent programs, precisely to enable reasoning about individual program components in relative isolation. And indeed, certain aspects of advanced concurrency logics, such as their aforementioned use of abstract predicates, are geared toward building proofs that reflect the data abstraction inherent in well-designed programs.

ICFP '13, September 25–27, 2013, Boston, MA, USA.

ACM 978-1-4503-2326-0/13/09. . . $15.00.
http://dx.doi.org/10.1145/2500365.2500600

We contend, however, that existing concurrency logics are not exploiting the modular design of sophisticated concurrent programs to full effect. In particular, we observe that there are two crucial dimensions of modular concurrent programming that existing logics provide no way to leverage in the construction of proofs: namely, **higher-order functions** and **granularity abstraction**.

Higher-order concurrency Higher-order functional abstraction is of course one of the most basic hammers in the modern programmer's toolkit for writing reusable and modular code. Moreover, a number of concurrent programming patterns rely on it: work stealing [2], Concurrent ML-style events [25], concurrent iterators [16], parallel evaluation strategies [29], and monadic approaches to concurrent programming [10], just to name a few.

Yet, verification of higher-order concurrent programs remains a largely unexplored topic. To our knowledge, only a few existing logics can handle higher-order concurrent programs [27, 17, 14], but these logics are limited in other ways—in particular, they do not presently support verification of "fine-grained" concurrent ADTs. This leads us directly to the second limitation we observe of the state of the art, concerning granularity abstraction.

Granularity abstraction via contextual refinement An easy way to adapt a sequential mutable data structure for concurrent access is to employ *coarse-grained* synchronization: use a single global lock, and instrument each of the operations on the data structure so that they acquire the lock before they begin and release it after they complete. On the other hand, more sophisticated implementations of concurrent data structures employ *fine-grained* synchronization: they protect different parts of a data structure with different locks, or avoid locking altogether, so that threads can access disjoint pieces of the data structure in parallel.

There may seem at first glance to be a fundamental trade-off here. Fine-grained synchronization enables parallelism, but makes the data structures that use it very tricky to reason about directly, due to their complex internal coordination between threads. Coarse-grained synchronization sequentializes access to the data structure, which is bad for parallelism but perfect for client-side reasoning, since it enables clients to reason about concurrent accesses as if each operation takes effect atomically.

Fortunately, modular programming comes to the rescue. In particular, so long as tricky uses of fine-grained synchronization are confined to the hidden state of a carefully crafted ADT, it is possible to prove that the fine-grained implementation of the ADT is a *contextual refinement* of some coarse-grained implementation. Contextual refinement means that, assuming clients only access the ADT through its abstract interface (so that the state really is hidden), every behavior that clients can observe of the fine-grained implementation is also observable of the coarse-grained one. Thus, clients can pretend, for the purpose of simplifying their own verification, that they are using the coarse-grained version, yet be sure that their code will still be correct when linked with the more efficient fine-grained version. This is what we call *granularity abstraction*. (Note: gran-

ularity abstraction is similar to *atomicity abstraction* [19], but is more general in that, as we will see in the iterator example later in this section, it applies even if the target of the abstraction is only *somewhat* coarse-grained.)

To illustrate this point more concretely, let us consider a simple motivating example of reasoning about Treiber's stack [28]. (We will in fact use this very example as part of a larger case study later in the paper.) Treiber's stack is a fine-grained implementation of a concurrent stack ADT. Instead of requiring concurrently executing push and pop operations to contend for a global lock on the whole stack (as a coarse-grained implementation would), Treiber's implementation allows them to race to access the head of the stack using compare-and-set (CAS). (The implementation of Treiber's stack is shown in Figure 9, and discussed in detail in §3.3.)

Now, the reader may expect that stacks should admit a canonical, principal specification (spec, for short), perhaps something like the following "precise" spec which tracks the exact contents s of the stack using the abstract predicate $\mathsf{Con}(s)$:

$$\{\mathsf{Con}(s)\}\ \mathsf{push}(x)\ \{\mathsf{Con}(x :: s)\}$$
$$\{\mathsf{Con}(s)\}\ \mathsf{pop}()\ \{\mathsf{ret}.\ (\mathsf{ret} = \mathsf{none} \wedge s = \mathsf{nil} \wedge \mathsf{Con}(s)) \vee (\exists x, s'.\mathsf{ret} = \mathsf{some}(x) \wedge s = x :: s' \wedge \mathsf{Con}(s'))\}$$

The trouble with using this spec in a concurrent setting is that knowledge about the exact contents of the stack is not stable under interference from other threads. As a result, some concurrency logics prohibit this spec altogether. Others permit the spec, but force the $\mathsf{Con}(s)$ predicate to be treated as a resource that only one thread can own at a time, thus effectively preventing any concurrent access to the stack and defeating the point of using Treiber's stack in the first place!

We want instead to capture the idea that the client threads of a data structure interact with it according to some (application-specific) protocol. Take, for example, the following "per-item" spec, which abstracts away from the LIFO nature of the stack and instead imposes an item-level protocol:

$$\begin{array}{ll} & \forall x.\ \{p(x)\}\ \mathsf{push}(x)\ \{\mathbf{true}\} \\ \wedge & \{\mathbf{true}\}\ \mathsf{pop}()\ \{\mathsf{ret}.\ \mathsf{ret} = \mathsf{none} \vee (\exists x.\ \mathsf{ret} = \mathsf{some}(x) \wedge p(x))\} \end{array}$$

Given an arbitrary predicate p of the client's choice, this per-item spec asserts that the stack contains only elements that satisfy the predicate p. It is pleasantly simple, and sufficient for the purposes of the case study we present later in the paper. It should be clear, however, that this "per-item" spec is far from a canonical or principal specification of stacks: the same spec would also be satisfied, for instance, by queues.

This brings us to our key point: different clients have different needs, and so we may want to verify the stack ADT against a range of different Hoare-style specs, but we do not want to have to re-verify *Treiber's implementation* each time. Ideally, we would like to modularly decompose the proof effort into two parts:

1. Prove that Treiber's implementation is a contextual refinement of a coarse-grained, lock-based implementation, which serves as a simple *reference implementation* of the stack ADT.
2. Use Hoare-style reasoning to verify that this reference implementation satisfies the various specs of interest to clients.

This decomposition engenders a clean separation of concerns, confining the difficulty of reasoning about Treiber's particular implementation[1] to the proof of refinement, and simplifying the verification of the stack ADT against different client specs.

The story we have just told is, *in principle*, nothing new. The ability to prove granularity abstraction via contextual refinement has been accepted in the concurrency literature as a useful correctness criterion for tricky concurrent data structures, precisely because of the modular decomposition of proof effort that it ought to facilitate [13, 8, 9]. Unfortunately, despite the utility of such a modular decomposition in theory, *no existing concurrency logic* actually supports it in practice. In particular, very few systems support proofs of contextual refinement at all, and those few that do support refinement proofs—such as the recent work of Liang and Feng [18]—do not provide a means of composing refinement with client-side Hoare-style verification in a unified logic.

Granularity abstraction for higher-order functions Although supporting granularity abstraction is already a challenge for first-order concurrent programs, it becomes even more interesting for higher-order concurrent programs.

Suppose, for instance, we wished to add a higher-order iterator, iter, to the concurrent stack ADT. (Such concurrent iterators are already commonplace [16].) Adding iter to Treiber's implementation is trivial: $\mathsf{iter}(f)$ will (without acquiring any lock) just read the current head pointer of the stack and apply f to each element of the stack accessible from it. But how do we specify the behavior of this iterator? What reference implementation should we use in proving granularity abstraction? Unlike for push and pop, it does not make sense for the reference implementation of $\mathsf{iter}(f)$ to be "maximally" coarse-grained (*i.e.,* to execute entirely within a critical section) because that will not correspond to the reality of the implementation we just described. In particular, the Treiber implementation does *not* freeze modifications to the stack while f is being applied to each element, so it does not contextually refine the maximally coarse-grained implementation of iteration.

Rather, what the Treiber iterator guarantees is that f will be applied to all the elements that were accessible from some node that was the head of the stack *at some point in time*. A clean way to specify this behavior is via a reference implementation that (1) acquires the lock, (2) takes a "snapshot" (*i.e.,* makes a copy) of the stack, (3) releases the lock, and (4) iterates f over the snapshot. (For the code of this reference implementation, see Figure 9.) What makes this reference implementation so intriguing is that it is only *somewhat* coarse-grained, yet as we will show later in the paper, it is nonetheless quite useful as a target for granularity abstraction.

This example demonstrates the flexibility of refinement, a flexibility which has not heretofore been tested, since no one has previously applied granularity abstraction to higher-order ADTs.

CaReSL: A logic for higher-order concurrency

In this paper, we present CaReSL (pronounced "carousel"), the first logic to support the use of granularity abstraction for modular verification of higher-order concurrent programs.

In providing both refinement and Hoare-style reasoning, CaReSL builds on ideas from two distinct lines of research:

- *Kripke logical relations* [22, 1, 5]. Logical relations are a fundamental technique for proving refinement in languages with *e.g.,* higher-order functions, polymorphism, and recursive types. Logical relations explain observable behavior in terms of the logical interpretation of each type. For example, two functions of type $\tau \to \tau'$ are logically related if giving them related inputs of type τ *implies* that they will produce related results of type τ'. *Kripke* logical relations are defined relative to a "possible world" that describes the relationship between the internal, hidden state of a program and that of its spec.
- *Concurrent separation logics* [20, 32, 3]. Separation logic is a reimagining of Hoare logic in which assertions are understood relative to some *portion* of the heap, with the separating conjunction $P * Q$ dividing up the heap between assertions P and Q. The motivation for separation is *local* reasoning: pre- and post-conditions need only mention the relevant portion of the

[1] Treiber's stack is actually one of the easiest fine-grained data structures to reason about, but our general line of argument applies equally well to more sophisticated implementations, such as the HSY elimination stack [12].

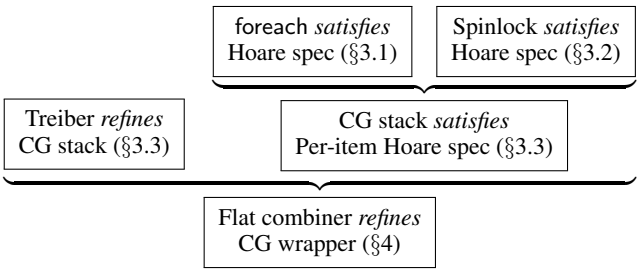


Figure 1. The structure of the case study—and paper

heap, and the rest of the heap can be presumed invariant. Most concurrent separation logics also add some form of protocols (*e.g.,* rely-guarantee), which facilitate compositional reasoning by constraining the potential interference between threads operating concurrently on some *shared* portion of the heap.

Rather than a combination of these approaches, CaReSL is an attempt to unify them. So, for example, the logic does *not* include refinement as a primitive notion. Instead, refinement is derived from Hoare-style reasoning: to prove that e refines e', one merely proves a particular Hoare triple about e, allowing the tools of concurrent separation logic to be exploited in the proof of logical relatedness. In the other direction, CaReSL is a modal logic with the possible worlds of Kripke logical relations, which can in turn be used to express the shared-state protocols of concurrent separation logics. The unification yields surprising new techniques, such as the use of modal *necessity* ($\Box$) to hide resources that an ADT's client would otherwise have to thread through their reasoning (§3.2).[2]

The semantics of CaReSL is derived directly from the model of Turon *et al.* [31], who presented the first formal proofs of refinement for fine-grained data structures in an ML-like setting (although they did not use it to reason about higher-order examples). Compared with Turon *et al.*'s work, which was purely semantic, CaReSL provides a *syntactic* theory for carrying out refinement proofs at a much higher level of abstraction. This proof theory, in turn, is inspired by Dreyer *et al.*'s LADR [5], a modal logic for reasoning about contextual equivalence of (sequential) stateful ML programs. CaReSL exemplifies how the key ideas of LADR are just as relevant to—and arguably even more compelling when adapted to—the concurrent setting.

To demonstrate the effectiveness of CaReSL, we use it to tackle a significant case study: namely, the first formal proof of (partial) correctness for Hendler *et al.*'s "flat combining" algorithm [11]. Flat combining provides a generic way to turn a sequential ADT into a relatively efficient concurrent one, by having certain threads perform—in one fell swoop—the combined actions requested by a whole bunch of other threads. The flat combining algorithm is interesting not only because it itself is a rather sophisticated higher-order function, but also because it is modularly assembled from other higher-order components, including a fine-grained stack with concurrent iterator. We therefore take the opportunity to verify flat combining in a modular way that mirrors the modular structure of its implementation.

Figure 1 illustrates the high-level structure of our proof, which involves an intertwined application of refinement and Hoare-style verification. For example, we prove that Treiber's stack refines the coarse-grained (CG) reference implementation of stacks; we then use Hoare-style reasoning to prove that the reference implementation satisfies the per-item spec; and finally we rely on the per-item spec in the proof that the flat combining algorithm refines an even higher-level abstraction. Every component of the case study involves higher-order code and therefore higher-order specifications.

[2] Previously, such hiding required the subtle *anti-frame* rule [24].

Syntax

$$\begin{array}{lll}
\text{Val} & v ::= () \mid \mathsf{true} \mid \mathsf{false} \mid (v,v) \mid \mathsf{inj}_i\ v \mid \mathsf{rec}\ f(x).e \mid \Lambda.e \mid \ell \\
\text{Exp} & e ::= v \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e\ e \mid e\ _ \mid (e,e) \mid \mathsf{prj}_i\ e \mid \mathsf{inj}_i\ e \\
& \quad \mid \mathsf{case}(e, \mathsf{inj}_1\ x \Rightarrow e, \mathsf{inj}_2\ y \Rightarrow e) \mid \mathsf{new}\ e \mid \mathsf{get}\ e \mid e := e \\
& \quad \mid \mathsf{CAS}(e,e,e) \mid \mathsf{newLcl} \mid \mathsf{getLcl}(e) \mid \mathsf{setLcl}(e,e) \mid \mathsf{fork}\ e \\
\text{CType} & \sigma ::= \mathsf{B} \mid \mathsf{ref}\ \tau \mid \mathsf{refLcl}\ \tau \mid \mu\alpha.\sigma \\
\text{Type} & \tau ::= \sigma \mid \mathbf{1} \mid \alpha \mid \tau \times \tau \mid \tau + \tau \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \tau \to \tau \\
\text{ECtx} & K ::= [\,] \mid \mathsf{if}\ K\ \mathsf{then}\ e\ \mathsf{else}\ e \mid K\ e \mid v\ K \mid \cdots
\end{array}$$

Typing contexts

$$\Delta ::= \cdot \mid \Delta, \alpha \qquad \Gamma ::= \cdot \mid \Gamma, x:\tau \qquad \Omega ::= \Delta; \Gamma$$

Well-typed expressions $\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{\Omega \vdash e : \mathsf{ref}\ \sigma \qquad \Omega \vdash e_o : \sigma \qquad \Omega \vdash e_n : \sigma}{\Omega \vdash \mathsf{CAS}(e, e_o, e_n) : \mathsf{B}} \qquad \frac{\Omega \vdash e : \forall\alpha.\tau}{\Omega \vdash e\ \_ : \tau[\tau'/\alpha]}$$

Machine configurations

$$\begin{array}{ll}
\text{TLV} \quad L \in \mathbb{N} \xrightarrow{\text{fin}} \text{Value} & \text{TPool}\ \ \mathcal{E} \in \mathbb{N} \xrightarrow{\text{fin}} \text{Expression} \\
\text{Heaps}\ h \in \mathbb{N} \xrightarrow{\text{fin}} (\text{Val} \cup \text{TLV}) & \text{Config}\ \varsigma ::= h; \mathcal{E}
\end{array}$$

Thread-local lookup

$$\lfloor L \rfloor(i) \triangleq \begin{cases} \mathsf{none} & i \notin \mathrm{dom}(L) \\ \mathsf{some}(L(i)) & \text{otherwise} \end{cases}$$

Pure reduction $\boxed{e \xrightarrow{\text{pure}} e'}$

$$(\mathsf{rec}\ f(x).e)\ v \xrightarrow{\text{pure}} e[\mathsf{rec}\ f(x).e/f, v/x] \qquad (\Lambda.e)\ \_ \xrightarrow{\text{pure}} e$$

Per-thread reduction $\boxed{h; e \xrightarrow{i} h'; e'}$

$$\begin{array}{rll}
h; e & \xrightarrow{i} h; e' & \text{when } e \xrightarrow{\text{pure}} e' \\
h; \mathsf{newLcl} & \xrightarrow{i} h \uplus [\ell \mapsto \emptyset]; \ell & \\
h \uplus [\ell \mapsto L]; \mathsf{setLcl}(\ell, v) & \xrightarrow{i} h \uplus [\ell \mapsto L[i := v]]; () & \\
h; \mathsf{getLcl}(\ell) & \xrightarrow{i} h; v & \text{when } h(\ell) = L,\ \lfloor L \rfloor(i) = v \\
h; \mathsf{CAS}(\ell, v_o, v_n) & \xrightarrow{i} h; \mathsf{false} & \text{when } h(\ell) \neq v_o \\
h \uplus [\ell \mapsto v_o]; \mathsf{CAS}(\ell, v_o, v_n) & \xrightarrow{i} h \uplus [\ell \mapsto v_n]; \mathsf{true} &
\end{array}$$

General reduction $\boxed{h; \mathcal{E} \to h'; \mathcal{E}'}$

$$\frac{h; e \xrightarrow{i} h'; e'}{h; \mathcal{E} \uplus [i \mapsto K[e]] \to h'; \mathcal{E} \uplus [i \mapsto K[e']]}$$

$$h; \mathcal{E} \uplus [i \mapsto K[\mathsf{fork}\ e]] \to h; \mathcal{E} \uplus [i \mapsto K[()]] \uplus [j \mapsto e]$$

Figure 2. The calculus: $F^{\mu!}$ with **fork**, **CAS**, and thread-local refs

2. The programming model

The programming language for our program logic is sketched in Figure 2; the full details are in the appendix [30]. It has a standard core: the polymorphic lambda calculus with sums, products, references, and equi-recursive types. We elide all type annotations in terms, but polymorphism is nevertheless introduced and eliminated by explicit type abstraction ($\Lambda.e$) and application ($e\ \_$). To this standard core, we add concurrency (through a **fork** construct), atomic compare-and-set (**CAS**) and thread-local references (type **refLcl** τ).

While normal references provide shared state through which threads can communicate, thread-local references allow multiple threads to access disjoint values in memory (a different one for each thread) using a common location. Formally, each thread-local reference has an associated *thread ID-indexed* table L, which is initially empty. Thus for **refLcl** τ, the **getLcl** operation returns an

"option" of type $1+\tau$, reflecting whether the thread-local reference has been initialized for the thread that invoked it.[3] Thread-local references are commonly used when a single data structure needs mutable, thread-local storage for an unbounded number of threads; the flat combining algorithm presented in §4 embodies this technique.

We define a small-step, call-by-value operational semantics, using evaluation contexts K to specify a left-to-right evaluation order within each thread. The $\xrightarrow{\text{pure}}$ relation gives the reductions that do not interact with the heap, while $\xrightarrow{i}$ gives general, single-thread reductions for the thread with ID i. These reductions are then lifted to general reduction $\to$ of *machine configurations* ς, which consist of a heap h together with an ID-indexed pool $\mathcal{E}$ of threads.

The type system imposes two important restrictions. First, recursive types $\mu\alpha.\tau$ must be *guarded*, meaning that all free occurrences of α in τ must appear under a non-μ type constructor. Second, because **CAS** exposes physical equality comparison of word-sized values at the hardware level, it can only be applied to references of *comparable type* σ, which we take to be base types and locations.

If $\Omega \vdash e_{\text{I}} : \tau$ and $\Omega \vdash e_{\text{S}} : \tau$, we say e_{I} *contextually refines* e_{S}, written $\Omega \models e_{\text{I}} \preceq e_{\text{S}} : \tau$, if, for every pair of thread IDs i and j,

$$\forall C : (\Omega, \tau) \rightsquigarrow (\emptyset, \mathbf{B}).\ \forall v. \forall \mathcal{E}_{\text{I}}.\ \ \emptyset; [i \mapsto C[e_{\text{I}}]] \to^* h_{\text{I}}; [i \mapsto v] \uplus \mathcal{E}_{\text{I}} \implies \exists \mathcal{E}_{\text{S}}.\ \ \emptyset; [j \mapsto C[e_{\text{S}}]] \to^* h_{\text{S}}; [j \mapsto v] \uplus \mathcal{E}_{\text{S}}$$

where $C : (\Omega, \tau) \rightsquigarrow (\Omega', \tau')$ is the standard typing judgment for contexts (guaranteeing that $\Omega \vdash e : \tau$ implies $\Omega' \vdash C[e] : \tau'$). This definition resembles the standard one for sequential languages, but take note that only the termination behavior of the main thread is observed; as with most languages in practice, we assume that once the main thread has finished, any forked threads are aborted.

3. CaReSL

While CaReSL ultimately provides mixed refinement and Hoare-style reasoning about higher-order concurrent programs, it is easiest to understand by first considering less powerful "sublogics", and then gradually incorporating more powerful features:

- We begin with a core logic that provides separation-logic reasoning for sequential (but higher-order) programs (§3.1). In this core logic, both Hoare triples and heap assertions live in a single syntactic class: they are propositions. Thus propositions include *e.g.,* implications between Hoare triples, which are ubiquitous when reasoning about higher-order code. Propositions also include first- and second-order quantification, guarded recursion, separating conjunction, and modal necessitation—all standard concepts whose interplay we explain in §3.1.
- We then add concurrency, leading to a concurrent separation logic for higher-order programs (§3.2). The new concept needed to deal with concurrency is that of a *protocol* governing some shared region of the heap. We express protocols through transition systems whose states provide an abstraction of the heap region's actual state. Threads can gain or lose *roles* in the protocol, which determine the transitions those (and other) threads are permitted to take. Traditional rely/guarantee reasoning can be recovered as a particular, restricted way of using protocols.
- Finally, we show how to reason about refinement, yielding a **C**oncurrent **a**nd **Re**fined **S**eparation **L**ogic (**CaReSL**) for higher-order programs (§3.3). As explained in the introduction, CaReSL does not simply add refinement as a primitive notion on top of a concurrent separation logic. Instead, CaReSL treats refinement as a derived notion, expressed as a particular Hoare-style specification. In order to do so, however, CaReSL does require one further extension: the notion of "spec resources", first introduced by Turon *et al.* [31], which allow pieces of a program configuration (heap and threads) to be manipulated as logical resources within Hoare-style proofs.

While CaReSL is the main contribution of the paper, its sublogics are of independent interest. We explain them by way of idiomatic higher-order examples, each of which serves as a component of the case study in §4.

3.1 Core logic: Sequential higher-order programs

To motivate the basic ingredients of core CaReSL, we begin with the quintessential higher-order stateful combinator, foreach:

$$\mathsf{foreach} : \forall\alpha.(\alpha \to \mathbf{1}) \to \mathsf{list}(\alpha) \to \mathbf{1} \qquad \mathsf{list}(\alpha) \triangleq \mu\beta.\mathbf{1} + (\alpha \times \beta)$$
$$\mathsf{foreach} \triangleq \Lambda.\ \lambda f.\ \mathbf{rec}\ \mathsf{loop}(l).\ \mathbf{case}\ l\ \mathbf{of}\ \mathsf{none} \Rightarrow () \mid \mathsf{some}(x, n) \Rightarrow f(x);\ \mathsf{loop}(n)$$

A specification of foreach should explain that foreach f l "lifts" the imperative behavior of f (which works on elements) to l (a list of elements). But to do so, it needs to *quantify* over the unknown behavior of an unknown function in a way that can be lifted to lists.

One possibility is to characterize f by means of a Hoare triple, while ultimately quantifying over the pre- and post-conditions of that triple. Suppose that $p(x)$ and $q(x)$ are predicates on values x, and that r is an arbitrary proposition, such that

$$\forall x.\ \{p(x) * r\}\ f(x)\ \{q(x) * r\}$$

The idea is that when f is applied to an element x of the list, it may assume both that $p(x)$ holds and that an invariant r holds of a disjoint portion of the heap. If $f(x)$ transforms $p(x)$ to $q(x)$ while maintaining the invariant, then foreach lifts that behavior to an entire list—regardless of what the predicates actually are. To capture the assumption that $p(x)$ holds of each element of a list l, we need something like the following recursive predicate:[4]

$$\text{Map}(p, l) \triangleq l = \mathsf{none} \vee (\exists x, n.\ l = \mathsf{some}(x, n) * p(x) * \text{Map}(p, n))$$

Note that, thanks to the use of the separating conjunction $*$, for each x in l there is a *disjoint region* of the heap satisfying $p(x)$.

These ideas lead to the following spec for foreach:

$$\forall p, q, r.\ \forall f.\ (\forall x.\{p(x) * r\}\ f(x)\ \{q(x) * r\}) \Rightarrow \forall l.\ \{\text{Map}(p, l) * r\}\ \mathsf{foreach}\ \_\ f\ l\ \{\text{Map}(q, l) * r\}$$

Formalizing this sketch will require a logic with a number of basic features: we need to be able to mix Hoare triples with other kinds of connectives (*e.g.,* implication), to quantify over both values (*e.g.,* f and x) and predicates (*e.g.,* p and q), and to define recursive predicates (*e.g.,* Map). With these goals in mind, we now explore CaReSL's design more systematically.

Syntax Figure 3 presents core CaReSL—the fragment of the logic appropriate for reasoning about sequential, higher-order code. CaReSL is a multi-sorted second-order logic, meaning that its syntax is stratified into the following three layers.

First, there are *terms* M, N, which come in a variety of *sorts* Σ, including the values and expressions of the language, as well as thread-local storage (with operations $\lfloor - \rfloor$, $\uplus$ and $:=$). The judgment $\mathcal{X} \vdash M : \Sigma$ gives the sort of a term (where $\mathcal{X}$ is a variable context giving the sorts of term and predicate variables).

Second, there are *propositions* P, Q, R, which include the connectives of multi-sorted first-order logic (*e.g.,* $\forall X \in \Sigma.P$) and second-order logic (*e.g.,* $\exists p \in \mathbb{P}(\Sigma).P$) with their standard meanings. The judgment $\mathcal{X} \vdash P : \mathbb{B}$ asserts that the proposition P is well-sorted.

Third, there are *predicates* φ, ψ, which are just propositions parameterized by a term: a predicate of sort $\mathbb{P}(\Sigma)$ can be introduced by $(X \in \Sigma).P$, which binds an unknown term X of sort Σ in P,

[3] Throughout, we let $\mathsf{none} \triangleq \mathbf{inj}_1\ ()$ and $\mathsf{some}(e) \triangleq \mathbf{inj}_2\ e$.

[4] Throughout, names of functions and types are written in lower case sans-serif, while predicate names are Capitalized Serif.

Syntax

Terms $M ::= X \mid (M, M) \mid n \mid e \mid M \uplus M \mid \lfloor M \rfloor(M) \mid M[M := M]$

Sorts $\Sigma ::= \mathbf{1} \mid \Sigma \times \Sigma \mid \text{Nat} \mid \text{Val} \mid \text{Exp} \mid \text{AtomicExp} \mid \text{LclStorage}$

Preds $\varphi ::= p \mid (X \in \Sigma).P \mid \mu p \in \mathbb{P}(\Sigma).\varphi$

Propositions $P ::=$

$\text{True} \mid P \Rightarrow P \mid P \wedge P \mid P \vee P \mid \exists X \in \Sigma.P \mid \forall X \in \Sigma.P \mid \triangleright P \mid$
$\varphi(M) \mid \exists p \in \mathbb{P}(\Sigma).P \mid \forall p \in \mathbb{P}(\Sigma).P \mid P * P \mid M \hookrightarrow_{\text{I}} M \mid A$

Necessary ("always") propositions $A ::=$

$\Box P \mid M \overset{\text{pure}}{\to} M \mid M{=}M \mid (\!|P|\!)\ M \Mapsto_{\text{IS}} M\ (\!|\varphi|\!) \mid \{P\}\ M \Mapsto M\ \{\varphi\}$

Contexts

$\mathcal{X} ::= \cdot \mid \mathcal{X}, X : \Sigma \mid \mathcal{X}, p : \mathbb{P}(\Sigma) \qquad \mathcal{P} ::= \cdot \mid \mathcal{P}, P \qquad \mathcal{C} ::= \mathcal{X}; \mathcal{P}$

Well-sorted terms $\boxed{\mathcal{X} \vdash M : \Sigma}$

$$\frac{}{\mathcal{X}, X : \Sigma \vdash X : \Sigma} \qquad \frac{\mathcal{X} \vdash M : \text{LclStorage} \qquad \mathcal{X} \vdash N : \text{Nat}}{\mathcal{X} \vdash \lfloor M \rfloor(N) : \text{Val}} \qquad \ldots$$

Well-sorted props. $\boxed{\mathcal{X} \vdash P : \mathbb{B}}$ ***Well-sorted preds.*** $\boxed{\mathcal{X} \vdash \varphi : \mathbb{P}(\Sigma)}$

Figure 3. Core CaReSL: Syntax

and eliminated by $\varphi(M)$ (also written $M \in \varphi$) which substitutes M for the parameter of φ. Because sorts include unit and products, predicates can express *relations* of arbitrary arity. The judgment $\mathcal{X} \vdash \varphi : \mathbb{P}(\Sigma)$ asserts that φ is a well-sorted predicate over terms of sort Σ.

In general, term variables are written X. But to avoid cluttering our rules and proofs with sort annotations, we use variables x, y, z for sort Val and i, j, k for sort Nat. We abuse notation, writing *e.g.,* v, e or L to stand for a term of sort Val, Exp or LclStorage, respectively.

In addition to this standard logical setup, CaReSL adopts key connectives from separation logic. In general, these connectives will refer to a variety of "resources" that will be introduced as we go along. In *core* CaReSL, however, the only resource is the heap. The *points-to* assertion $i \hookrightarrow_{\text{I}} v$ holds of any heap containing a location i that points to the value v. (Ignore the I subscript for now; we will return to it in §3.3). The *separating conjunction* $P * Q$ holds if the currently-owned resources (here, a portion of the heap) can be split into two disjoint parts satisfying P and Q respectively.

While the truth of some propositions (*e.g.,* $i \hookrightarrow_{\text{I}} v$) is *contingent* on the presence of certain resources, others (*e.g.,* $M = N$) are *necessary*: if they hold, they do so regardless of the currently-owned resources, and therefore will continue to hold in any future state. (Such propositions are called "pure" in separation logic parlance.) The syntactic subcategory A of necessary propositions includes claims about term equality, about the operational semantics ($M \overset{\text{pure}}{\to} N$), and Hoare triples. Arbitrary propositions can be made necessary via the $\Box$ ("always") modality, where $\Box P$ holds if P holds for all possible resources. As we will see shortly, necessary propositions play by special rules: they can move freely through Hoare triples and separating conjunctions.

Ultimately, CaReSL distinguishes between triples about general expressions e and those about *atomic* expressions a (which execute in a single step). Since this distinction is motivated by concurrency, we postpone its explanation to §3.2. We include the distinction syntactically in core CaReSL, but it can be safely ignored for now.

Atomic expressions a have the following grammar:

$$\mathsf{new}\ v \mid \mathsf{get}\ v \mid v\ :=\ v \mid \mathsf{CAS}(v, v, v) \mid \mathsf{newLcl} \mid \mathsf{getLcl}(v) \mid \mathsf{setLcl}(v, v)$$

The propositions for atomic triples[5] $(\!|P|\!)\ i \Mapsto_{\text{IS}} a\ (\!|\varphi|\!)$ and general triples $\{P\}\ i \Mapsto e\ \{\varphi\}$ are both parameterized by a *thread ID* i; the expression may access thread-local storage, in which case its behavior is ID-dependent. (When the thread ID doesn't matter, we write $\{P\}\ e\ \{\varphi\}$ as short for $\forall i.\ \{P\}\ i \Mapsto e\ \{\varphi\}$.) In addition, since we are working with an expression language (as opposed to a command language), postconditions are *predicates* over the return value of the expression, rather than simply propositions about the final state of the heap. But when an expression returns unit, we often abuse notation and write a proposition instead.

[5] Again, ignore the IS subscript until §3.3.

Logical axioms and rules

$$\frac{\mathcal{X}; A \vdash P}{\mathcal{X}; \mathcal{P}, A \vdash \Box P}\ \Box\text{I} \qquad \frac{\mathcal{C} \vdash P}{\mathcal{C} \vdash \triangleright P}\ \textsc{Mono} \qquad \frac{\mathcal{C}, \triangleright P \vdash P}{\mathcal{C} \vdash P}\ \textsc{Löb}$$

$$\begin{array}{rcl@{\qquad\qquad}rcl} \Box(P \Rightarrow Q) & \Rightarrow & \Box P \Rightarrow \Box Q & A * P & \Leftrightarrow & A \wedge P \\ A & \Rightarrow & \Box A & \text{True} * P & \Leftrightarrow & P \\ \Box P & \Rightarrow & P & \triangleright(P * Q) & \Leftrightarrow & \triangleright P * \triangleright Q \end{array}$$

$$\frac{\mathcal{C}, p : \mathbb{P}(\Sigma) \vdash P}{\mathcal{C} \vdash \forall p \in \mathbb{P}(\Sigma).\ P}\ \forall_2\text{I} \qquad \begin{array}{r} M \in (X \in \Sigma).P \Leftrightarrow P[M/X] \\ M \in (\mu p.\varphi) \Leftrightarrow M \in \varphi[\mu p.\varphi/p] \end{array}$$

Figure 4. Core CaReSL: Selected logical axioms and rules

In order to support recursive assertions, the logic includes *guarded recursion* ($\mu p \in \mathbb{P}(\Sigma).\ P$), which entails the following tradeoff. On the one hand, guarded recursion allows occurrences of the recursive predicate p to appear negatively, which is crucial for modelling recursive types (§3.3) but is usually prohibited for lack of monotonicity. On the other hand, the recursion is "guarded" in that references to p in P must appear under the $\triangleright$ modality. A proposition $\triangleright Q$ represents the present knowledge that Q holds *later*, *i.e.,* after at least one step of computation. Guarded recursion supports a coinductive style of reasoning: to prove P one can assume $\triangleright P$, but this assumption is only useful after at least one step of computation. As we explain in §5, our use of guarded recursion descends from a line of work on *step-indexed* logical relations, but the interaction with Hoare triples is a novelty of CaReSL.

Proof rules The main judgment of CaReSL is written $\mathcal{C} \vdash P$, where $\mathcal{C} = \mathcal{X}; \mathcal{P}$ is a *combined* context of annotated term/predicate variables $\mathcal{X}$ and propositional assumptions $\mathcal{P}$. The meaning is the usual one: for any way of instantiating the variables in $\mathcal{X}$, if the hypotheses $\mathcal{P}$ are true for a given resource (*i.e.,* for the moment, a given heap), then P is true of the same resource. We implicitly assume $\mathcal{X} \vdash Q : \mathbb{B}$ for every proposition Q in $\mathcal{P}$ and likewise that $\mathcal{X} \vdash P : \mathbb{B}$ holds.

A few basic logical axioms and proof rules for core CaReSL are shown in Figure 4. Axioms hold in an arbitrary well-sorted context. The axioms include all the standard ones for a multi-sorted second-order logic (we show only $\forall_2$I), as well as several characterizing separating conjunction and the $\Box$ and $\triangleright$ modalities. We'll just mention the highlights:

- Because True is a unit for separating conjunction (and every proposition implies True), propositions are *affine*: we can "throw away" resources, because $(P * Q) \Rightarrow (\text{True} * Q) \Rightarrow Q$.
- The two conjunctions $*$ and $\wedge$ are identical if at least one of their operands is a necessary proposition. Consequently, necessary propositions can be "freely copied": $A \Rightarrow A * A$.
- The LÖB rule provides a coinductive reasoning principle: to prove P, you may assume P—but only under the $\triangleright$ modality, which guards use of the assumption until at least one step of computation has taken place. On the other hand, MONO says that any proposition can be *weakened* to one that is guarded. (Both MONO and LÖB are inherited from LADR [5].) We will momentarily see how $\triangleright$ is eliminated in Hoare-style reasoning.
- The $\triangleright$ modality distributes over $*$, as it does with most other connectives, except implication and recursion.

Atomic Hoare logic (where IS ::= I | S as explained in §3.3)

$$
\begin{array}{ll}
(\!|\text{True}|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{IS}} \textsf{new}\ v & (\!|\text{ret. ret} \hookrightarrow_{\text{IS}} v|\!) \\
(\!|v \hookrightarrow_{\text{IS}} v'|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{IS}} \textsf{get}\ v & (\!|\text{ret. ret} = v' * v \hookrightarrow_{\text{IS}} v'|\!) \\
(\!|v \hookrightarrow_{\text{IS}} -|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{IS}} v := v' & (\!|\text{ret. ret} = () * v \hookrightarrow_{\text{IS}} v'|\!) \\
(\!|\text{True}|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{IS}} \textsf{newLcl} & (\!|\text{ret. ret} \hookrightarrow_{\text{IS}} \emptyset|\!) \\
(\!|v \hookrightarrow_{\text{IS}} L|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{IS}} \textsf{getLcl}(v) & (\!|\text{ret. ret} = \lfloor L \rfloor(i) * v \hookrightarrow_{\text{IS}} L|\!) \\
(\!|v \hookrightarrow_{\text{IS}} L|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{IS}} \textsf{setLcl}(v, v') & (\!|\text{ret. ret} = () * v \hookrightarrow_{\text{IS}} L[i := v']|\!) \\
(\!|v \hookrightarrow_{\text{IS}} v'|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{IS}} & (\!|\text{ret. } (v' = v_o * \text{ret} = \textsf{true} * v \hookrightarrow_{\text{IS}} v_n)|\!) \\
\quad \textsf{CAS}(v, v_o, v_n) & \quad \vee\ (v' \neq v_o * \text{ret} = \textsf{false} * v \hookrightarrow_{\text{IS}} v')
\end{array}
$$

$$\frac{\mathcal{X}; P \vdash P' \qquad \mathcal{C} \vdash (\!|P'|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{IS}} a\ (\!|x.\ Q'|\!) \qquad \mathcal{X}, x; Q' \vdash Q}{\mathcal{X}; \mathcal{P} \vdash (\!|P|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{IS}} a\ (\!|x.\ Q|\!)}\ \textsc{ACsq}$$

$$\frac{\mathcal{C} \vdash (\!|P|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{IS}} a\ (\!|x.\ Q|\!)}{\mathcal{C} \vdash (\!|P * R|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{IS}} a\ (\!|x.\ Q * R|\!)}\ \textsc{AFrame} \qquad (\textsc{ADisj}, \textsc{AEx}\ \text{elided})$$

General Hoare logic

$$\frac{\mathcal{C} \vdash (\!|P|\!)\ i \mathrel{\Rightarrow\!\!\!\mapsto}_{\text{I}} a\ (\!|Q|\!)}{\mathcal{C} \vdash \{\triangleright P\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} a\ \{Q\}}\ \textsc{Private} \qquad \frac{\mathcal{C} \vdash e \overset{\text{pure}}{\to} e' \qquad \mathcal{C} \vdash \{P\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} e'\ \{\varphi\}}{\mathcal{C} \vdash \{\triangleright P\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} e\ \{\varphi\}}\ \textsc{Pure}$$

$$\frac{\mathcal{C} \vdash \{P\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} e\ \{x.\ Q\} \qquad \mathcal{C}, x \vdash \{Q\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} K[x]\ \{R\}}{\mathcal{C} \vdash \{P\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} K[e]\ \{R\}}\ \textsc{Bind}$$

$\mathcal{C} \vdash \{\text{True}\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} v\ \{x.\ x = v\}$ RET (CSQ, FRAME, DISJ, EX elided)

$$\frac{\mathcal{C} \vdash A}{\mathcal{C} \vdash \{P * A\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} e\ \{\varphi\}}\ \textsc{AIn} \qquad \frac{\mathcal{C}, A \vdash \{P\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} e\ \{\varphi\}}{\mathcal{C} \vdash \{P * A\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} e\ \{\varphi\}}\ \textsc{AOut}$$

Derivable rules

$$\frac{\mathcal{C}, f, \forall x.\ \{\triangleright P\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} f\ x\ \{\varphi\} \vdash \forall x.\ \{P\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} e\ \{\varphi\}}{\mathcal{C} \vdash \forall x.\ \{\triangleright P\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} (\textsf{rec}\ f(x).e)\ x\ \{\varphi\}}\ \textsc{Rec}$$

$$\frac{\mathcal{C} \vdash \{P\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} e\ \{x.\ \exists y.\ (x = \textsf{inj}_1\ y * \triangleright Q_1) \vee (x = \textsf{inj}_2\ y * \triangleright Q_2)\} \qquad \forall k \in \{1,2\}:\ \mathcal{C}, x, y \vdash \{x = \textsf{inj}_k\ y * Q_k\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} e_k\ \{\varphi\}}{\mathcal{C} \vdash \{P\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} \textsf{case}(e, \textsf{inj}_1\ y \Rightarrow e_1, \textsf{inj}_2\ y \Rightarrow e_2)\ \{\varphi\}}$$

Figure 5. Core CaReSL: Hoare logic

The rules for atomic triples (Figure 5) are formulated in the standard style of separation logic. They transcribe the operational semantics of atomic expressions, mentioning only the part of the heap relevant to the expression. Atomic Hoare logic supports the usual rules—consequence, framing, disjunction, $\exists$-elimination—with one important exception: it does not support a sequencing rule, since a sequence of atomic expressions is not atomic.

All atomic expressions take exactly one step to execute, and in so doing allow us to peel off a layer of the $\triangleright$ modality. To cut down on clutter, the precondition in an atomic triple is *implicitly* understood as being under one $\triangleright$ modality. The rule PRIVATE, which lifts an atomic triple to a general one, makes this implicit assumption explicit. (In core CaReSL, *all* resources are private; §3.2 adds shared resources.) To handle pure reduction steps (like β-reduction), the PURE rule appeals directly to the operational semantics using the necessary proposition $e \overset{\text{pure}}{\to} e'$. The rest of the rules for general Hoare triples are mostly standard; we show only the nonstandard rules in Figure 5.

In an expression language, the monadic nature of Hoare logic becomes visible: the BIND rule replaces the usual rule of sequencing, while RET is used to inject a value into a Hoare triple.

We also have a rule allowing necessary propositions to move freely from the proof context into preconditions (AIN), and vice versa (AOUT). In general, any contingent assumptions like $x \hookrightarrow_{\text{I}} 3$ need to be given explicitly in the precondition of a Hoare triple, because the truth of such statements can change over time; the triple says that it is only usable at times when its precondition holds. But in the specific case of necessary propositions, we can do better: we know that if such a proposition happens to be true now, it will be true forever, and so it does not need to be given explicitly in the precondition. As we will see in §3.2, these rules will allow us to completely hide pieces of state that are known to always obey a certain protocol.

Finally, using LÖB and PURE, together with standard Hoare rules, we can derive specialized rules for constructs like recursive functions and pattern matching—the two derived rules in Figure 5.

***Verifying* foreach** Having seen core CaReSL in detail, we can now return to the foreach example. First, we need to rewrite our sketch of the specification more formally. The Map predicate needs to employ guarded recursion:

$$\begin{array}{l}\text{Map}(\varphi) \triangleq \mu m \in \mathbb{P}(\text{Val}).\ (l \in \text{Val}). \\ \qquad l = \textsf{none} \vee (\exists x, n.\ l = \textsf{some}(x, n) * \varphi(x) * \triangleright m(n))\end{array}$$

while the foreach spec should be annotated with sorts:

$$\begin{array}{l}\forall p, q \in \mathbb{P}(\text{Val}). \forall r \in \mathbb{P}(\mathbf{1}).\ \forall f.\ (\forall x. \{p(x) * r\}\ f(x)\ \{q(x) * r\}) \\ \quad \Rightarrow \quad \forall l.\ \{\text{Map}(p)(l) * r\}\ \textsf{foreach}\ \_\ f\ l\ \{\text{Map}(q)(l) * r\}\end{array}$$

To prove foreach correct, we use a kind of Hoare "proof outline", annotating each program point with a proposition:

$\textsf{foreach} \triangleq \Lambda.\ \lambda f.\ \textsf{rec}\ \textsf{loop}(l).$

Prop context:	**Variables:** $f, i, p, q, r, l, \textsf{loop}$
$\forall x.\ \{p(x) * r\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} f(x)\ \{q(x) * r\}$	
$\forall n.\ \{\triangleright(\text{Map}(p)(n) * r)\}\ i \mathrel{\Rightarrow\!\!\!\mapsto} \textsf{loop}\ n\ \{\text{Map}(q)(n) * r\}$	

$\{\text{Map}(p)(l) * r\}$
$\textsf{case}\ l\ \textsf{of}\ \textsf{none} \Rightarrow \{l = \textsf{none} * \text{Map}(p)(l) * r\}\ ()\ \{\text{Map}(q)(l) * r\}$
$\quad \mid \textsf{some}(x, n) \Rightarrow \{l = \textsf{some}(x, n) * \text{Map}(p)(l) * r\}$
$\qquad \{l = \textsf{some}(x, n) * p(x) * \triangleright\text{Map}(p)(n) * r\}$
$\qquad f(x);$
$\qquad \{l = \textsf{some}(x, n) * q(x) * \triangleright\text{Map}(p)(n) * r\}$
$\qquad \textsf{loop}(n)$
$\qquad \{l = \textsf{some}(x, n) * q(x) * \text{Map}(q)(n) * r\}$
$\qquad \{\text{Map}(q)(l) * r\}$

Proof outlines implicitly apply the Hoare logic rule corresponding to each language construct: for recursive functions and pattern matching we use the derived rules shown earlier; for an atomic expression we use the corresponding axiom; when going under a Λ or λx we use the PURE rule and prove a triple about the function as applied to $\_$ or x respectively. The frame rule is applied implicitly as needed, while uses of consequence are shown by writing a sequence of assertions (each one implying the next). We do not explicitly write the thread ID in a proof outline, but it is always clear from context (and, for our examples, always the variable i).

We supplement traditional proof outlines with boxed *context assertions* spelling out an extension to both the variable context $\mathcal{X}$ and proposition context $\mathcal{P}$. Extending the context with new variables introduces a universal quantification (using *e.g.,* the rule $\forall_2$I in Figure 4), while adding propositions introduces an implication. We use the AIN rule implicitly to bring necessary consequences of the proof context into the Hoare-style outline.

The proof for foreach is quite straightforward: the initial case analysis on the input list allows us to expand the definition of the Map predicate, which for the nonempty case gives us the necessary knowledge to execute f on an element of the list; note the heavy use of the frame rule to add invariant propositions. The use of $\triangleright$ in Map is neatly dispatched by the use of the REC rule for foreach.

3.2 Adding concurrency: Reasoning about shared protocols

To reason about concurrency, we need to reason about the protocols governing shared (and often hidden) state. Take, for example, the following higher-order spinlock:

$$\begin{array}{lcl}
\mathsf{tryAcq} & \triangleq & \lambda x.\ \mathbf{CAS}(x, \mathbf{false}, \mathbf{true}) \\
\mathsf{acq} & \triangleq & \mathbf{rec}\ \mathsf{loop}(x).\ \mathbf{if}\ \mathsf{tryAcq}(x)\ \mathbf{then}\ ()\ \mathbf{else}\ \mathsf{loop}(x) \\
\mathsf{rel} & \triangleq & \lambda x.\ x\ :=\ \mathbf{false}
\end{array}$$

$$\begin{array}{l}
\mathsf{mkSync}\ :\ \mathbf{1} \to \forall\alpha.\forall\beta.(\alpha \to \beta) \to (\alpha \to \beta) \\
\mathsf{mkSync}\ \triangleq\ \lambda().\ \mathbf{let}\ \mathsf{lock} = \mathbf{new}\ \mathbf{false}\ \mathbf{in} \\
\qquad\qquad \Lambda.\Lambda.\lambda f.\lambda x.\ \mathsf{acq}(\mathsf{lock}); \mathbf{let}\ z = f(x)\ \mathbf{in}\ \mathsf{rel}(\mathsf{lock});\ z
\end{array}$$

Each invocation mkSync() creates a new *wrapper* that closes over a fresh, hidden lock. The wrapper can then be used to add simple synchronization to an arbitrary function. There are, of course, a variety of ways to use synchronization, but a particularly common one is to protect access to some shared resource characterized by an invariant p—an idea that leads to the following specification:

$$\forall p \in \mathbb{P}(\mathbf{1}).\ \{p\}\ i \Mapsto \mathsf{mkSync}\ ()\ \left\{s.\ \mathrm{Syncer}_p(s)\right\}$$
$$\textit{where}\quad \mathrm{Syncer}_p \triangleq (s \in \mathrm{Val}).\ \square\forall f, x, q, r.\ \{p * q\}\ f\ x\ \{z.\ p * r(z)\} \Rightarrow \{q\}\ s\ \_\ \_\ f\ x\ \{z.\ r(z)\}$$

The spec says that for each invocation of mkSync(), the client can choose some invariant resource p, giving up control over the resource in exchange for a synchronization wrapper s. When s is later applied to a function f, it provides f with exclusive access to the resource p (seemingly out of thin air), which f can use however it pleases, so long as the invariant is reestablished on completion.

Intuitively, the reason that mkSync satisfies its specification is that the lock itself is *hidden*: all access to it is mediated through the wrapper s, which the client can only apply to invariant-preserving functions. Hiding enables mkSync to maintain an internal *protocol* in which, whenever the lock is free, the invariant p holds. To express this protocol, as well as the more sophisticated protocols needed for fine-grained concurrency (§3.3, §4), CaReSL provides a syntactic treatment of the *local protocols* given in Turon *et al.* [31].

Protocols A local protocol π governs a shared resource abstractly, by means of a set of protocol states (S) equipped with a transition relation ($\rightsquigarrow$). Each state $s \in S$ has an associated proposition $\varphi(s)$ giving its concrete interpretation in terms of *e.g.,* heap resources. The idea is then that any thread is allowed to update the shared resource, so long as at each atomic step those concrete updates correspond to a permitted abstract transition.

In general, although all threads must abide by a given protocol, not all of them play the same role within it. For example, a protocol governing a lock might have two states, Locked and Unlocked, with transitions in both directions, but we usually want to allow only the thread that acquired the lock to be allowed to release it (as is the case in the mkSync example). To allow threads to take on or relinquish roles dynamically, local protocols employ *tokens* that individual threads may own. By taking a transition, a thread may "earn" a token from the protocol; conversely, certain transitions require a thread to "pay" by giving up a previously-earned token. For the locking protocol, we use a single token *Lock*:

In the Unlocked state, the protocol itself owns the *Lock*, which we show by annotating the state with the token. A thread that transitions from Unlocked to Locked then earns (takes ownership of) the *Lock*. Conversely, to transition back to the Unlocked state, a thread must transfer the *Lock* back to the protocol—a move only possible for the thread that owns the *Lock*.

Formally, a local protocol $\pi = (S, \rightsquigarrow, \mathcal{T}, \varphi)$ is given by a transition system (S and $\rightsquigarrow$), a function $\mathcal{T}$ giving the set of tokens owned by the protocol at each state, and a predicate φ on states giving their interpretations. To be able to talk about states and tokens in the logic, we add sorts State and TokSet, for which we will use the metavariables s and T respectively; see Figure 6. We leave the grammar of terms for these sorts open-ended, implicitly

Protocols

$$\text{Protocol}\ \ \pi ::= (S, \rightsquigarrow, \mathcal{T}, \varphi)\quad \text{where}\quad S \subseteq \text{State},\ \ \rightsquigarrow\ \subseteq S \times S,\ \ \mathcal{T} \in S \to \text{TokSet},\ \ \varphi\ \text{token-pure}$$

$$(s, T) \rightsquigarrow_\pi (s', T') \triangleq s\ (\pi.\rightsquigarrow)\ s', \quad \pi.\mathcal{T}(s) \uplus T = \pi.\mathcal{T}(s') \uplus T'$$
$$\mathrm{frame}_\pi(s, T) \triangleq (s,\ \mathrm{AllTokens} - (\pi.\mathcal{T}(s) \uplus T))$$

$$(s, T) \sqsubseteq^{\mathrm{guar}}_\pi (s', T') \triangleq (s, T) \rightsquigarrow^*_\pi (s', T')$$
$$(s, T) \sqsubseteq^{\mathrm{rely}}_\pi (s', T') \triangleq \mathrm{frame}_\pi(s, T) \rightsquigarrow^*_\pi \mathrm{frame}_\pi(s', T')$$

Syntax

$$\begin{array}{lll}
\text{Sort} & \Sigma ::= \cdots \mid \text{State} \mid \text{TokSet} \\
\text{Prop} & P ::= \cdots \mid \boxed{M}^M_\pi \mid \mathit{Tid}(M) \\
\text{AbProp} & A ::= \cdots \mid M \sqsubseteq^{\mathrm{rely}}_\pi M \mid M \sqsubseteq^{\mathrm{guar}}_\pi M \mid \mathrm{TokPure}(P)
\end{array}$$

Hoare logic (*where* $\pi[\![b]\!] \triangleq \exists s.b = (s, -) \wedge \pi.\varphi(s)$)

$$\frac{\mathcal{C} \vdash \{P\}\ i \Mapsto e\ \{x.\ Q * \pi[\![b]\!]\}}{\mathcal{C} \vdash \{P\}\ i \Mapsto e\ \left\{x.\ Q * \exists n.\ \boxed{b}^n_\pi\right\}}\ \textsc{NewIsl}$$

$$\frac{\mathcal{C} \vdash \forall b \overset{\mathrm{rely}}{\sqsupseteq}_\pi b_0.\ (\!|\pi[\![b]\!] * P|\!)\ i \mapsto_\iota a\ (\!|x.\ \exists b'\ \overset{\mathrm{guar}}{\sqsupseteq}_\pi b.\ \pi[\![b']\!] * Q|\!)}{\mathcal{C} \vdash \left\{\boxed{b_0}^n_\pi * \triangleright P\right\}\ i \Mapsto a\ \left\{x.\ \exists b'.\ \boxed{b'}^n_\pi * Q\right\}}\ \textsc{UpdIsl}$$

$$\frac{\mathcal{C} \vdash \{P * \mathit{Tid}(j)\}\ j \Mapsto e\ \{\mathsf{ret}.\ \mathsf{ret} = ()\}}{\mathcal{C} \vdash \{P\}\ i \Mapsto \mathbf{fork}\ e\ \{\mathsf{ret}.\ \mathsf{ret} = ()\}}\ \textsc{Fork}$$

Logical axioms and rules

$$\boxed{s_1, T_1}^n_\pi * \boxed{s_2, T_2}^n_\pi\ \Leftrightarrow\ \exists s.\ \boxed{s, T_1 \uplus T_2}^n_\pi \wedge (s, T_1) \overset{\mathrm{rely}}{\sqsupseteq}_\pi (s_1, T_1) \wedge (s, T_2) \overset{\mathrm{rely}}{\sqsupseteq}_\pi (s_2, T_2)\ \textsc{SplitIsl}$$

Figure 6. CaReSL: Incorporating concurrency

extending it as needed for particular transition systems.[6] Thus, we can give an interpretation $\mathrm{LockInterp}_p$ for the lock protocol that is appropriate for an instance of mkSync protecting an invariant p:

$$\mathrm{LockInterp}_p \triangleq (s \in \text{State}).\ s = \mathsf{Locked} \vee (s = \mathsf{Unlocked} * p)$$

The combination of transition systems and tokens gives rise to *token-sensitive transitions*. A transition from state/tokens (s, T) to state/tokens (s', T') is permitted by π, written $(s, T) \rightsquigarrow_\pi (s', T')$, so long as the *law of token conservation* holds: the disjoint union of the thread's tokens and the protocols tokens $T \uplus \pi.\mathcal{T}(s)$ before the transition must be the same as the disjoint union $T' \uplus \pi.\mathcal{T}(s')$ afterwards.[7] Token-sensitive transitions constrain both what a thread can do (the *guarantee* moves $\sqsubseteq^{\mathrm{guar}}_\pi$ enabled by its tokens) and what its environment can do (the *rely* moves $\sqsubseteq^{\mathrm{rely}}_\pi$ enabled by the tokens owned by other threads). See Figure 6.

Island assertions In CaReSL, all resources are either privately owned by a thread, or else governed by a shared protocol. When a heap assertion like $x \hookrightarrow_\iota 3$ appears in the pre- or postcondition of a triple, it is understood as asserting private ownership over the corresponding portion of the heap; no other thread is allowed to access it. Thus the rules of core CaReSL are immediately sound in a concurrent setting: there is no interference to account for.

To talk about shared resources, CaReSL includes *island assertions* $\boxed{b}^n_\pi$ (similar to region assertions in RGSep/CAP). As the name suggests, each island is an independent region of the heap governed by its own laws (the protocol π). The number n is the "name" of the island, which is used to distinguish multiple islands with the same protocol; we often leave off the name when it is exis-

[6] To be completely formal, we could allow each transition system to come equipped with its own explicit grammar of states and tokens.

[7] We use notation like $\pi.\mathcal{T}$ to extract named components from a tuple.

tentially quantified. The term b (of sort State $\times$ TokSet) asserts private ownership of a set of tokens,[8] and acts as a "rely-lower bound" on the state of the protocol: the current state of the protocol is some $b' \sqsupseteq_\pi^{\text{rely}} b$. Thus, in CaReSL *every assertion about shared resources is automatically stable under interference*.

While a thread's island assertions cannot be invalidated by a change its environment makes, they *can* be invalidated by a change the thread itself makes. For example, if a thread owns *Lock* in Locked state of the mkSync protocol, the environment cannot change the state at all—but the thread itself can move to the Unlocked state. There is, however, a special class of island assertions which are "completely" stable, *i.e.,* that act as necessary propositions: island assertions that do not claim any tokens. To understand why, consider that when no tokens are owned, the $\sqsubseteq_\pi^{\text{rely}}$ relation *degenerates* to the underlying transition system of the protocol, which means it contains all possible moves that any thread can make. Formally, we have $\boxed{(M,\emptyset)}_\pi^n \;\Rightarrow\; \Box\,\boxed{(M,\emptyset)}_\pi^n$.

When island assertions *do* claim ownership over tokens (a form of resource), they can be meaningfully combined by separating conjunction; see the SPLITISL rule in Figure 6, which takes into account the fact that the assertions give only rely-lower bounds.[9]

There are two Hoare logic rules for working with islands (Figure 6). The rule NEWISL creates a new island starting with an initial state and token set bound b; the resources necessary to satisfy the protocol's initial state must be present as private resources, and afterwards will be shared. (The shorthand $\pi[\![b]\!]$ just gives the interpretation at the state in b.) Once an island is established, the only way to access the shared resources it governs is through the UPDISL rule, which can only be applied to an *atomic* expression. Starting from an initial bound b_0, the atomic expression a might be (instantaneously) executed in any rely-reachable b; for each such state, the expression is granted exclusive access to the shared resource, but in its single atomic step, it must make a guarantee move in the protocol. This rule reveals the important semantic difference between atomic and general triples: atomic triples have access to the concrete resources owned by shared islands, while general triples can only work with island assertions.

Thread creation In a local protocol, threads gain and lose roles (tokens) by making deliberate moves within the protocol. But there is also a role that every thread plays *automatically*: the role of being a thread with a certain ID. To support protocols that use thread IDs (such as the one in §4), CaReSL builds in a proposition $\mathit{Tid}(j)$ that asserts the existence of a thread j, and acts as an *uncopyable* resource: $\mathit{Tid}(j) * \mathit{Tid}(j) \Leftrightarrow$ False. The resource is introduced by the FORK rule, which also allows the parent thread to transfer an arbitrary P to the newly-forked thread. (The parent can keep resources for itself via the FRAME rule.) The trivial postconditions in FORK may seem alarming, but they reflect the fact that the *only* form of communication between threads is shared state, which must be mediated by a shared protocol.

***Verifying* mkSync** The interpretations $\pi.\varphi(s)$ of protocol states in CaReSL inherit a limitation from their semantic treatment by Turon *et al.* [31]: they must be token-pure, *i.e.,* they cannot assert ownership of island tokens. The necessary proposition TokPure(P) can be used to assert that, for example, an unknown proposition is token-pure and thus safe to use in an interpretation, and we need such a restriction on p in mkSync:

$$\forall p \in \mathbb{P}(\mathbf{1}).\ \text{TokPure}(p) \Rightarrow \{p\}\ \ i \Vdash \mathsf{mkSync}\ ()\ \left\{s.\ \text{Syncer}_p(s)\right\}$$
$$\textit{where}\ \ \text{Syncer}_p \triangleq (s \in \text{Val}).\ \ \Box\forall f,x,q,r.\ \ \{p*q\}\ f\ x\ \{z.\ p*r(z)\} \Rightarrow\ \{q\}\ s\ \_\ \_\ f\ x\ \{z.\ r(z)\}$$

[8] An island assertion is only satisfied if the asserted token set is disjoint from the tokens owned by the protocol at the asserted state.

[9] We assume that terms M include disjoint union on token sets.

The hidden protocol LockProt_p for mkSync just puts together the pieces we have already seen:

$$\text{LockProt}_p \triangleq (\{\mathsf{Locked}, \mathsf{Unlocked}\}, \rightsquigarrow, \mathcal{T}, \text{LockInterp}_p)$$

where $\rightsquigarrow$ and $\mathcal{T}$ are given by the transition system for locking shown above.

The high-level proof outline for mkSync is straightforward:

$\lambda().$ **Prop context:** TokPure(p) **Variables:** p

$\{p\}$ **let** lock $=$ **new false** $\{p * \mathsf{lock} \hookrightarrow_{\mathrm{I}} \mathbf{false}\}$

$\{\exists n.\ \boxed{\mathsf{Unlocked}, \emptyset}^n_{\text{LockProt}_p}\}$

$\Lambda.\Lambda.\lambda f.\lambda x.$

Prop context: $\Box\ \boxed{\mathsf{Unlocked}, \emptyset}_{\text{LockProt}_p}$ **Variables:** f, x, q, r

$\{p*q\}\ f\ x\ \{z.\ p*r(z)\}$

$\{q\}$	$\{\boxed{\mathsf{Unlocked}, \emptyset}_{\text{LockProt}_p} * q\}$
acq(lock);	$\{\boxed{\mathsf{Locked}, \mathit{Lock}}_{\text{LockProt}_p} * p * q\}$
let $z = f(x)$	$\{\boxed{\mathsf{Locked}, \mathit{Lock}}_{\text{LockProt}_p} * p * r(z)\}$
rel(lock);	$\{\boxed{\mathsf{Unlocked}, \emptyset}_{\text{LockProt}_p} * r(z)\}$
z	$\{z.\ r(z)\}$

After allocating the hidden lock, we move it into a fresh island using NEWISL, and then move that island assertion (wrapped with $\Box$) into the proof context (using AOUT) before reasoning about the returned wrapper function. In this way the protocol is completely hidden from the client, yet available to the closure we return to the client—mirroring the fact that lock is hidden from the client but available in the closure. Since the contents of the Syncer spec is within the $\Box$ modality, the client may move the spec into its proof context, which means that the client can freely use the synchronization wrapper without threading *any* assertion about it through its proof. (Previous logics, like CAP, require at least that the client thread through an abstract predicate standing for the lock.)

When verifying the closure, we begin with a precondition q of the client's choice, but then apply AIN to load the hidden island assertion into the precondition. The subsequent lines use the locking protocol to acquire the resource p, execute the client's function f, and then return p to the protocol. The tokens in the island assertions, which say what the thread owns at each point, are the mirror image to those labelling the corresponding states in the protocol.

The triples for acq and rel must ultimately be proved by appeal to the UPDISL rule. For acq, the *bound* on the protocol is just $(\mathsf{Unlocked}, \emptyset)$, which means that the *actual* state might be either Locked or Unlocked. The **CAS** within acq will return:

- **true** if successful; the state must have been Unlocked. We make a guarantee move to $(\mathsf{Locked}, \mathit{Lock})$, gaining the token.
- **false** if it fails; the state must have been Locked. We do not change the state, and the acq function retries by calling loop.

On the other hand, for rel, the bound $(\mathsf{Locked}, \mathit{Lock})$ means that the protocol *must* be in state Locked: the environment cannot move to Unlocked because it does not own *Lock*. But since rel *does* own *Lock*, it can simply update the lock to **false**, corresponding to a guarantee move to the Unlocked state in the protocol.

3.3 Adding refinement: Reasoning about specification code

At this point, we have seen the fragment of CaReSL providing Hoare-style specs and proofs for higher-order concurrent programs, as exemplified (in a simple way) by the mkSync example. This section explains the other major component of the logic: higher-order granularity abstraction, exemplified (in a simple way) by the Treiber stack with iterator.

Syntax

$$\Sigma ::= \cdots \mid \text{EvalCtx} \qquad P ::= \cdots \mid M \hookrightarrow_{\text{s}} M \mid M \Mapsto_{\text{s}} M$$
$$M ::= \cdots \mid K \mid M[M] \qquad A ::= \cdots \mid P \to_{\text{s}} P \mid M \bowtie M$$

Spec rewriting

$$\frac{\mathcal{C} \vdash e \overset{\text{pure}}{\to} e'}{\mathcal{C} \vdash (P * j \Mapsto_{\text{s}} K[e]) \to_{\text{s}} (P * j \Mapsto_{\text{s}} K[e'])}\ \textsc{SPure}$$

$$\frac{\mathcal{C} \vdash (\!|P|\!)\ j \Mapsto_{\text{s}} a\ (\!|x.\ Q|\!)}{\mathcal{C} \vdash (P * j \Mapsto_{\text{s}} K[a]) \to_{\text{s}} (\exists x.\ Q * j \Mapsto_{\text{s}} K[x])}\ \textsc{SPrim}$$

$$\mathcal{C} \vdash (j \Mapsto_{\text{s}} K[\mathsf{fork}\ e]) \to_{\text{s}} (j \Mapsto_{\text{s}} K[()] * k \Mapsto_{\text{s}} e)\ \textsc{SFork}$$

Hoare logic (AExecSpec elided)

$$\frac{\mathcal{C} \vdash \{P\}\ i \Mapsto e\ \{x.\ Q\} \qquad \mathcal{C} \vdash Q \to_{\text{s}} R}{\mathcal{C} \vdash \{P\}\ i \Mapsto e\ \{x.\ R\}}\ \textsc{ExecSpec}$$

$$\frac{\mathcal{C}, j \bowtie j' \vdash \left\{P * \mathit{Tid}(j) * j' \Mapsto_{\text{s}} e_{\text{s}}\right\}\ j \Mapsto e_{\text{I}}\ \{x.\ x = ()\}}{\mathcal{C} \vdash \left\{P * i' \Mapsto_{\text{s}} K[\mathsf{fork}\ e_{\text{s}}]\right\}\ i \Mapsto \mathsf{fork}\ e_{\text{I}}\ \left\{i' \Mapsto_{\text{s}} K[()]\right\}}\ \textsc{ForkS}$$

Figure 7. CaReSL: Incorporating spec resources

Spec resources While it is possible to formulate a relational version of Hoare logic (with Hoare *quadruples* [33]), or to develop special-purpose logics for refinement [5], our goal with CaReSL is to support *both* standard Hoare-style reasoning and refinement reasoning in a single unified logic. In particular, we want a treatment of refinement that re-uses as much Hoare-style reasoning as possible. To this end, we adapt the idea of *specification resources*, first proposed by Turon *et al.* [31] as a way of proving refinement when threads engage in "cooperation" (a point we return to in §4). We will show how spec resources make it possible to state and prove refinements as an *entirely* derived concept on top of the Hoare logic we have already built up, in particular allowing protocols and triples to serve double-duty when reasoning about spec code.

To prove that e_{I} refines e_{s}, one must (intuitively) show that every behavior observable of e_{I} is observable of e_{s} as well. Our strategy is to encode these proof obligations into certain Hoare triples about the execution of e_{I}, but with pre- and postconditions instrumented with pieces of the corresponding spec state—both heap and code—which we treat as resources in CaReSL. These spec resources are entirely a fiction of the logic: they do not reflect anything about the *physical* state of e_{I}, but they must be used through logical rules that enforce the operational semantics for e_{s}.

There are two basic spec resource assertions: the *(spec) points-to* assertion $i \hookrightarrow_{\text{s}} v$, which claims ownership of a fragment of the spec heap containing a location i that points to the value v, and the *spec thread* assertion $i \Mapsto_{\text{s}} e$, which claims ownership of a spec thread with ID i executing expression e. The separating conjunction $P * Q$ works in the usual way with these resources, dividing up the spec heap and thread pool between the propositions P and Q. We also add a final sort, EvalCtx, and terms K and $M[N]$ for expressing and combining them, which makes it possible to *abstract* over the evaluation context for some spec thread. (By convention, the variable κ is always of sort EvalCtx.)

In addition, CaReSL provides a necessary proposition $P \to_{\text{s}} Q$, which says that the spec resources owned by P can, according to the operational semantics, take a step to those owned by Q. All other resources must be left invariant. The rule SPure lifts the pure stepping relation from the operational semantics directly. The rule SPrim, on the other hand, *re-uses* atomic triples to support reasoning about atomic spec expressions. (The subscript IS in the laws of atomic triples allows them to be used in *either* implementation mode I or spec mode S.) The ExecSpec Hoare rule (and identical AExecSpec rule for atomic triples) allows the specification to be

Relating expressions

$$(e_{\text{I}}, e_{\text{s}}) \downarrow \varphi \triangleq \forall (i \bowtie j), \kappa.\ \{\mathit{Tid}(i) * j \Mapsto_{\text{s}} \kappa[e_{\text{s}}]\}\ i \Mapsto e_{\text{I}}\ \{x_{\text{I}}.\ \exists x_{\text{s}}.\ \varphi(x_{\text{I}}, x_{\text{s}}) * \mathit{Tid}(i) * j \Mapsto_{\text{s}} \kappa[x_{\text{s}}]\}$$

Relating values

$$[\![\mathbf{1}]\!] \triangleq (x_{\text{I}}, x_{\text{s}}).\ x_{\text{I}} = x_{\text{s}} = () \qquad [\![\mu\alpha.\tau]\!] \triangleq \mu\alpha.[\![\tau]\!]$$
$$[\![\mathbf{B}]\!] \triangleq (x_{\text{I}}, x_{\text{s}}).\ (x_{\text{I}} = x_{\text{s}} = \mathbf{true}) \vee (x_{\text{I}} = x_{\text{s}} = \mathbf{false}) \qquad [\![\alpha]\!] \triangleq \alpha$$

$$[\![\tau_1 \times \tau_2]\!] \triangleq (x_{\text{I}}, x_{\text{s}}).\ (\mathbf{prj}_1\ x_{\text{I}}, \mathbf{prj}_1\ x_{\text{s}}) \downarrow [\![\tau_1]\!] \wedge (\mathbf{prj}_2\ x_{\text{I}}, \mathbf{prj}_2\ x_{\text{s}}) \downarrow [\![\tau_2]\!]$$
$$[\![\tau_1 + \tau_2]\!] \triangleq (x_{\text{I}}, x_{\text{s}}).\quad (\triangleright \exists (y_{\text{I}}, y_{\text{s}}) \in [\![\tau_1]\!].\ x_{\text{I}} = \mathbf{inj}_1\ y_{\text{I}} \wedge x_{\text{s}} = \mathbf{inj}_1\ y_{\text{s}})$$
$$\vee\ (\triangleright \exists (y_{\text{I}}, y_{\text{s}}) \in [\![\tau_2]\!].\ x_{\text{I}} = \mathbf{inj}_2\ y_{\text{I}} \wedge x_{\text{s}} = \mathbf{inj}_2\ y_{\text{s}})$$
$$[\![\tau \to \tau']\!] \triangleq (x_{\text{I}}, x_{\text{s}}).\ \Box \forall y_{\text{I}}, y_{\text{s}}. (\triangleright (y_{\text{I}}, y_{\text{s}}) \in [\![\tau]\!]) \Rightarrow (x_{\text{I}}\ y_{\text{I}}, x_{\text{s}}\ y_{\text{s}}) \downarrow [\![\tau']\!]$$
$$[\![\forall \alpha.\tau]\!] \triangleq (x_{\text{I}}, x_{\text{s}}).\ \Box \forall \alpha. \text{Type}(\alpha) \Rightarrow (x_{\text{I}}\ \_, x_{\text{s}}\ \_) \downarrow [\![\tau]\!]$$
$$[\![\mathsf{ref}\ \tau]\!] \triangleq (x_{\text{I}}, x_{\text{s}}).\ \text{Inv}(\exists (y_{\text{I}}, y_{\text{s}}) \in [\![\tau]\!].\ x_{\text{I}} \hookrightarrow_{\text{I}} y_{\text{I}} * x_{\text{s}} \hookrightarrow_{\text{s}} y_{\text{s}})$$
$$[\![\mathsf{refLcl}\ \tau]\!] \triangleq (x_{\text{I}}, x_{\text{s}}).\ \text{Inv}\begin{pmatrix}\exists L_{\text{I}}, L_{\text{s}}.\ x_{\text{I}} \hookrightarrow_{\text{I}} L_{\text{I}} * x_{\text{s}} \hookrightarrow_{\text{s}} L_{\text{s}} * \\ \forall (i \bowtie j).\ (\lfloor L_{\text{I}} \rfloor(i), \lfloor L_{\text{s}} \rfloor(j)) \in [\![\mathbf{1} + \tau]\!]\end{pmatrix}$$

Invariant protocols $\text{Inv}(P) \triangleq \exists n.\ \boxed{0, \emptyset}^{n}_{(\{0\}, \emptyset, [0 \mapsto \emptyset], (s).P)}$

Type interpretations $\text{Type}(\varphi) \triangleq \Box \forall (x_{\text{I}}, x_{\text{s}}) \in \varphi.\ \Box (x_{\text{I}}, x_{\text{s}}) \in \varphi$

Logical relatedness (i.e., refinement)

$$\overline{\alpha}; \overline{x : \tau} \vdash e_{\text{I}} \preceq e_{\text{s}} : \tau \triangleq \overline{\alpha}, \overline{x_{\text{I}}, x_{\text{s}}}; \overline{\text{Type}(\alpha)}, \overline{(x_{\text{I}}, x_{\text{s}}) \in [\![\tau]\!]} \vdash (e_{\text{I}}\overline{[x_{\text{I}}/x]}, e_{\text{s}}\overline{[x_{\text{s}}/x]}) \downarrow [\![\tau]\!]$$

Figure 8. Encoding refinement in CaReSL

"executed" within any postcondition, *i.e.,* at any point in a proof. Since ExecSpec can be applied repeatedly, a single step of some implementation code—*e.g.,* an atomic expression—can correspond to an arbitrary number of spec steps.

Putting these pieces together, a triple like

$$\{j \Mapsto_{\text{s}} e_{\text{s}}\}\ i \Mapsto e_{\text{I}}\ \{x_{\text{I}}.\ \exists x_{\text{s}}.\ j \Mapsto_{\text{s}} x_{\text{s}} * \varphi(x_{\text{I}}, x_{\text{s}})\}$$

says that if e_{I} produces some value x_{I}, then e_{s} can be executed to produce some value x_{s} such that $\varphi(x_{\text{I}}, x_{\text{s}})$ holds—exactly the kind of observational claim we set out to make.

Encoding refinement To give a full account of refinement, we also need to ensure that visible updates to the heap by the implementation are matched in lock-step by updates by the spec, which we will do by imposing a protocol governing visible (shared) reference cells. And to account for thread-local references, we must also track a *correspondence* between implementation and specification thread IDs, which allows us to state invariants connecting the storage on either side. The (necessary) assertion $i \bowtie j$ asserts that the implementation thread i and spec thread j exist, and obeys the law $(i \bowtie j) \wedge (i' \bowtie j') \Rightarrow (i = i' \Leftrightarrow j = j')$. Correspondences are introduced using the rule ForkS, a variant of Fork that jointly creates fresh implementation and spec threads.

And that's all: using these ingredients, we can *encode* refinement by simply writing down a particular predicate in CaReSL.

The encoded predicate expresses a logical relation—a variant of the relation given semantically by Turon *et al.* [31]—following the approach first laid out by Plotkin and Abadi [23]. We define the relation in stages (see Figure 8).

First, we have the proposition $(e_{\text{I}}, e_{\text{s}}) \downarrow \varphi$, which is a more general version of the triple we gave earlier: it includes assumptions about thread IDs, and permits compositional reasoning about specs by quantifying over an unknown evaluation context κ. The expressions do not begin with private ownership of any heap resources because, when proving refinement, we must assume that any pre-existing state is shared with the context. As we will see in a moment, this shared state is governed by an extremely liberal protocol; all we can assume about the context is that it is well-typed.

Second, we have the binary predicate $[\![\tau]\!]$, which is satisfied by $(v_{\text{I}}, v_{\text{s}})$ when the observations a context can make of v_{I} can

$\mathsf{stack_S} \;:\; \forall\alpha.\ (\alpha \to \mathbf{1}) \times (\mathbf{1} \to (1+\alpha)) \times ((\alpha \to \mathbf{1}) \to \mathbf{1})$
$\mathsf{stack_S} \triangleq \Lambda.\ \mathbf{let}\ \mathsf{sync} = \mathsf{mkSync}(), \mathsf{hd_S} = \mathbf{new}\ (\mathsf{none})$
  $\mathbf{let}\ \mathsf{push} = \lambda x.\ \mathsf{hd_S} := \mathsf{some}(x, \mathbf{get}\ \mathsf{hd_S})$
  $\mathbf{let}\ \mathsf{pop} = \lambda().\ \mathbf{case}\ \mathbf{get}\ \mathsf{hd_S}\ \mathbf{of}\ \mathsf{none} \Rightarrow \mathsf{none} \mid \mathsf{some}(x,n) \Rightarrow \mathsf{hd_S} := n;\ \mathsf{some}(x)$
  $\mathbf{let}\ \mathsf{snap} = \mathsf{sync}\ \_\ \_\ (\lambda().\mathbf{get}\ \mathsf{hd_S})$
  $\mathbf{let}\ \mathsf{iter} = \lambda f.\ \mathsf{foreach}\ \_\ f\ (\mathsf{snap}())$
  $\mathbf{in}\ (\mathsf{sync}\ \_\ \_\ \mathsf{push},\ \mathsf{sync}\ \_\ \_\ \mathsf{pop},\ \mathsf{iter})$

$\mathsf{stack_I} \triangleq \Lambda.\ \mathbf{let}\ \mathsf{hd_I} = \mathbf{new}\ \mathsf{nil}$ *where* $\mathsf{nil} \triangleq \mathbf{new}\ \mathsf{none}$,
  $\mathbf{let}\ \mathsf{push} = \mathbf{rec}\ \mathsf{try}(x).$ $\mathsf{cons}\ e \triangleq \mathbf{new}\ \mathsf{some}(e)$
    $\mathbf{let}\ c = \mathbf{get}\ \mathsf{hd_I}, n = \mathsf{cons}(x, c)$
    $\mathbf{in\ if}\ \mathbf{CAS}(\mathsf{hd_I}, c, n)\ \mathbf{then}\ ()\ \mathbf{else}\ \mathsf{try}(x),$
  $\mathbf{let}\ \mathsf{pop} = \mathbf{rec}\ \mathsf{try}().$
    $\mathbf{let}\ c = \mathbf{get}\ \mathsf{hd_I}\ \mathbf{in\ case\ get}\ c\ \mathbf{of}$
      $\mathsf{none} \Rightarrow \mathsf{none}$
    $\mid \mathsf{some}(d,n) \Rightarrow \mathbf{if}\ \mathbf{CAS}(\mathsf{hd_I}, c, n)\ \mathbf{then}\ \mathsf{some}(d)\ \mathbf{else}\ \mathsf{try}()$
  $\mathbf{let}\ \mathsf{iter} = \lambda f.\ \mathbf{let\ rec}\ \mathsf{loop}(c) = \mathbf{case\ get}\ c\ \mathbf{of}$
      $\mathsf{none} \Rightarrow () \mid \mathsf{some}(d,n) \Rightarrow f(d); \mathsf{loop}(n)$
    $\mathbf{in}\ \mathsf{loop}(\mathbf{get}\ \mathsf{hd_I})$
  $\mathbf{in}\ (\mathsf{push},\ \mathsf{pop},\ \mathsf{iter})$

Figure 9. Coarse- and fine-grained stacks, with iterators

also be made of v_S—the heart of the logical relation. It is defined by induction on the structure of τ, since the ways a value can be observed depend on its type:

For **ground types**, the context can observe the exact value, so only equal values are in the interpretation. For **product types**, the context can project both sides of the product, so two values are related only if each of their projections are related; similarly for **sums**. The context can observe **functions** only by applying them, so one function is related to another only if, when applied to related values, they produce related results.

Recursive and **polymorphic** types are interpreted via guarded recursion and second-order quantification, respectively; we pun type variables α, β as predicate variables, which are implicitly assumed to be of sort $\mathbb{P}(\mathrm{Val} \times \mathrm{Val})$, and for quantification explicitly required to satisfy $\mathrm{Type}(\alpha)$. These two constraints guarantee that the relation $[\![\tau]\!]$ is a resource-insensitive, binary predicate on values. The resource-insensitivity reflects the fact that refinement between values must hold in an arbitrary context of observation/usage—including contexts that freely copy the values in question—which means that we can assume nothing about privately-owned resources.

The context can interact with **reference types** by either reading or writing at any time. Thus, references assert the existence of a particularly simple protocol with a single state, whose interpretation says that related locations must continuously point to related values. For **thread-local references**, the statement is qualified by indexing the storage table by related thread IDs.

The uses of $\triangleright$ throughout reflect both the guardedness of our equi-recursive types *and* the guardedness of recursion in CaReSL; it takes one step of computation to eliminate values of any of the types in which $\triangleright$ appears. Uses of $(e_I, e_S) \downarrow \varphi$ are implicitly guarded, because postconditions are implicitly guarded for any non-value expression.

Finally, e_I is *logically related* to e_S at some context Ω and type τ if $\Omega \vdash e_I \preceq e_S : \tau$, which is shorthand for a use of the $\mathcal{C} \vdash P$ judgment of CaReSL. The proof context closes all type variables α with arbitrary type interpretations and all term variables x with *pairs* of term variables related at the appropriate types. Thus, a term with a free variable of type **ref** τ will gain access in its proof context to a shared protocol governing the corresponding location. The protocol in turn forces updates to the reference in the implementation to occur in lock-step with those in the spec. *Private* references, *i.e.,* those allocated within the implementation or spec, face no such requirement—unless or until they are exposed.

Treiber's stack, with an iterator We now sketch a simple refinement proof. Figure 9 gives two stack implementations. The first, $\mathsf{stack_S}$, is a coarse-grained reference implementation (which we think of as a specification). Its representation includes a mutable reference $\mathsf{hd_S}$ to an immutable list, as well as a synchronization wrapper sync provided by mkSync. The exported functions to push and pop from the stack simply wrap the corresponding updates to $\mathsf{hd_S}$ with synchronization. But the iterator is *not* fully synchronized: it takes an atomic snapshot of the stack, but then calls its argument f on each element of the snapshot without holding any locks. By the time f is called, the contents of the stack may have changed.

The second implementation, $\mathsf{stack_I}$, is Treiber's stack supplemented with an iterator. Treiber's stack [28] is one of the simplest lock-free data structures—a kind of "Hello World" for concurrent program logics. Like $\mathsf{stack_S}$, Treiber's stack maintains a reference $\mathsf{hd_I}$ to a list that it uses to represent the stack. Instead of using a lock, however, it updates $\mathsf{hd_I}$ directly (and atomically) using **CAS**, which requires the contents of $\mathsf{hd_I}$ to have a *comparable* type, *i.e.,* to be testable for pointer equality. Thus $\mathsf{hd_I}$ is of type **ref** $\mathsf{clist}(\alpha)$, where $\mathsf{clist}(\alpha) \triangleq \mu\beta.\ \mathbf{ref}\ (\mathbf{1} + (\alpha \times \beta))$. The push and pop functions employ *optimistic concurrency*: instead of acquiring a lock to inform other threads that they are going to make a change, they just take a snapshot of the current $\mathsf{hd_I}$, compute a new value, and use **CAS** to install the new value if the identity of the $\mathsf{hd_I}$ has not changed. Intuitively, this strategy works because if the $\mathsf{hd_I}$'s identity has not changed, then *nothing* about the stack has changed: *all* mutation is performed by identify-changing updates to $\mathsf{hd_I}$. The iterator simply walks over the stack's contents starting from a (possibly stale!) snapshot of $\mathsf{hd_I}$.

To prove that $\mathsf{stack_I}$ refines $\mathsf{stack_S}$, we begin as follows:

$\mathsf{stack_I} \triangleq \Lambda.$ **Prop context:** $\mathrm{Type}(\alpha),\ i \bowtie j$ **Variables:** α, i, j, κ

$\{j \Mapsto_S \kappa[\mathsf{stack_S}\ \_]\}\ \mathbf{let}\ \mathsf{hd_I} = \mathbf{new}\ \mathsf{nil}$
$\{j \Mapsto_S \kappa[\mathsf{stack_S}\ \_] * \exists x.\ \mathsf{hd_I} \hookrightarrow_I x * x \hookrightarrow_I \mathsf{none}\}$ (ExecSpec)
$\left\{ \begin{array}{l} \exists\ \mathsf{hd_S}, \mathsf{lock_S}.\ \mathsf{hd_S} \hookrightarrow_S \mathsf{none} * \mathsf{lock} \hookrightarrow_S \mathbf{false} * j \Mapsto_S \kappa[\mathsf{stack}'_S] * \\ \quad \exists x.\ \mathsf{hd_I} \hookrightarrow_I x * x \hookrightarrow_I \mathsf{none} \end{array} \right\}$

We have executed the "preambles" of both the implementation and spec—the code that allocates their respective hidden state. (Let stack'_S denote the rest of the spec code after the **let**-binding of sync and $\mathsf{hd_S}$.) At this point, we will use NewIsl to move *all* of this state into a hidden island, with a protocol that we can use when proving refinement for each exported function.

The states s of the protocol are maps $\mathrm{Loc} \xrightarrow{\mathrm{fin}} \mathrm{Val}$ which simply record the identity and contents of every node added to $\mathsf{stack_I}$. The transition relation $s \rightsquigarrow s' \triangleq s \subseteq s'$ captures the idea that, once a node has appeared in the stack, its contents never change. There are no tokens. We interpret a state s as follows:

$$\varphi \triangleq (s).\ \mathsf{lock} \hookrightarrow_S \mathbf{false} * \mathop{\Large *}_{\ell \in \mathrm{dom}(s)} \ell \hookrightarrow_I s(\ell) \\ * \exists x_I, x_S.\ \mathsf{hd_I} \hookrightarrow_I x_I * \mathsf{hd_S} \hookrightarrow_S x_S * \mathrm{Link}(s, x_I, x_S)$$

Because the lock is protected by the protocol, we can only execute the exported spec functions with AExecSpec, as part of an application of UpdIsl; the interpretation furthermore requires that we run the spec in "big steps", starting and ending in unlocked states. It also says that the implementation and spec stack contents are *linked*, in the following sense:

$$\mathrm{Link} \triangleq \mu p.\ (s, x_I, x_S).\ \exists x'_I, s'.\ s = [x_I \mapsto x'_I] \uplus s' * \\ (x'_I = \mathsf{none} * x_S = \mathsf{none}) \ \vee \left(\begin{array}{l} \exists y_I, z_I.\ x'_I = \mathsf{some}(y_I, z_I) * \\ \exists y_S, z_S.\ x_S = \mathsf{some}(y_S, z_S) * \\ (y_I, y_S) \in \alpha * \triangleright p(s', z_I, z_S) \end{array} \right)$$

Linking requires that the stack state s contains an entire linked list starting from $\mathsf{hd_I}$, and that each element of the list is related (at type α) to the corresponding element of the list in $\mathsf{hd_S}$. *But it does so without mentioning the heap at all!* This is the key: it means that $\mathrm{Link}(s, x_I, x_S) \Rightarrow \Box\mathrm{Link}(s, x_I, x_S)$, so the Link assertion can

be freely copied in the proof for iter when the traversal begins. Since the abstract state s of the stack can only grow, all of the nodes mentioned in this "snapshot" of Link are guaranteed to still be available in the region of the heap governed , with their original values, as traversal proceeds—even if, in the meantime, the nodes are no longer reachable from hd_I.

The full proof outlines are available in the appendix [30].

Combining refinement and Hoare-style reasoning Suppose a client of Treiber's stack wishes to use it in only a very weak way, as if it were a lock-free *bag* with the following spec:

$$\begin{array}{l}\text{PerItem} \triangleq (e \in \text{Expr}).\ \forall p, q \in \mathbb{P}(\text{Val}).\\ \quad (\forall x.\ \text{TokPure}(p(x)) \land (p(x) \Rightarrow \Box q(x))) \Rightarrow\\ \qquad \{\text{True}\}\ e\ \{\mathsf{bag}.\ \mathsf{bag} = (\mathsf{add}, \mathsf{rem}, \mathsf{iter}) \land P\}, \quad \textit{where}\\ P \triangleq \forall x.\ \{p(x)\}\ \mathsf{add}\ \{\mathsf{ret}.\ \mathsf{ret} = ()\}\\ \quad \land\ \{\text{True}\}\ \mathsf{rem}\ \{\mathsf{ret}.\ \mathsf{ret} = \mathsf{none} \lor (\exists x.\ \mathsf{ret} = \mathsf{some}(x) * p(\mathsf{ret}))\}\\ \quad \land\ \forall f, r.\ (\forall x.\ \{q(x) * r\}\ f(x)\ \{r\}) \Rightarrow \{r\}\ \mathsf{iter}(f)\ \{r\}\end{array}$$

This "per-item" spec associates a resource p with each element of the stack, which is transferred to and from the data structure when adding or removing elements. If, in addition, each per-item resource entails some permanent knowledge $\Box q$, then that knowledge can be safely assumed by a function concurrently iterating over the stack, even if the resources originally supporting it have been consumed. As we will see in the case study (§4), this spec for iteration is useful for associating permanent, token-free knowledge about some other protocol with each item that appears in the stack.

While the per-item spec could in principle be applied directly to Treiber's stack (*i.e.,* we could prove $\text{Bag}(\mathsf{stack}_\mathsf{I})$), doing so entails repeating much of the verification we just performed. On the other hand, proving $\text{PerItem}(\mathsf{stack}_\mathsf{S})$ is nearly trivial: we just instantiate the lock specification given in §3.2 with the representation invariant $\text{Rep}_p \triangleq \exists l.\ \mathsf{hd}_\mathsf{S} \hookrightarrow_\mathsf{I} l * \text{Map}(p)(l)$. To verify iter, we need only notice that $\text{Map}(p)(l) \Rightarrow \Box\text{Map}(q)(l)$, *i.e.,* the snapshot of the stack provides a list of necessarily true associated facts. (The details are in the appendix [30].) By mixing refinement and Hoare logic, we can significantly modularize our verification effort.

3.4 Soundness

The model theory of CaReSL is based directly on the semantic model of Turon *et al.* [31], with minor adjustments to accommodate the assertions $\mathit{Tid}(i)$ and $i \bowtie j$ necessary for reasoning about thread-local storage. Since this paper is focused on the complementary aim of giving a syntactic proof theory, we do not delve into the model here; it can be found in full in the appendix [30]).

Soundness for CaReSL encompasses two theorems. **First**, that $\mathcal{C} \vdash P$ implies $\mathcal{C} \models P$, where the latter is the semantic entailment relation of the model. Proving this theorem requires validating, in particular, the key Hoare logic rules, which we do in the appendix; these proofs resemble the proofs of key lemmas supporting Turon *et al.*'s model. **Second**, that $\cdot \vdash (\Omega \vdash e_\mathsf{I} \preceq e_\mathsf{S} : \tau)$ implies $\Omega \models e_\mathsf{I} \preceq e_\mathsf{S} : \tau$, *i.e.,* that the logical relation is sound for contextual refinement. Again, this proof follows the soundness proof of Turon *et al.*, except that we can carry it out at a much higher level of abstraction using CaReSL's proof rules.

4. Case study: Flat combining

Using CaReSL's combination of Hoare-style and refinement reasoning, we can verify higher-order concurrent algorithms in layers of abstraction—and this section shows how to do it.

We study the recent *flat combining* construction of Hendler *et al.* [11], which takes an arbitrary sequential data structure and transforms it into a concurrent one in which all operations appear to take effect atomically. One can do so by merely wrapping each operation with synchronization on a global lock—indeed, this is what our *spec* does—but flat combining takes a cache-friendly approach intended for hard-to-parallelize data structures.

$\mathsf{flat}_\mathsf{S} \triangleq \mathsf{mkSync}() \quad : \quad [\forall\alpha.\ \forall\beta.\ (\alpha \to \beta) \to (\alpha \to \beta)]$

$\mathsf{flat}_\mathsf{I} \triangleq \Lambda[\alpha].\ \Lambda[\beta].\ \lambda f : [\alpha \to \beta].$ *(Types annotated for clarity)*

```
  let lock : [ref B] = new (false)
  let lclOps : [refLcl op] = newLcl            (where op ≜ ref (α + β))
  let (add, _, iter) = stack_I [op]
  let install : [α → op] = λ req. case getLcl(lclOps)
    of some(o) ⇒ o := inj_1 req; o
     | none ⇒ let o = new (inj_1 req) in setLcl(lclOps, o); add(o); o
  let doOp : [op → 1] = λo. case get(o)
    of inj_1 req ⇒ o := inj_2 f(req)
     | inj_2 res ⇒ ()
  in λx : [α]. let o = install(x)
                let rec loop() = case get(o)
                  of inj_1 _ ⇒ if not(get lock) and tryAcq(lock) then
                                 iter doOp; rel(lock); loop()
                               else loop()
                   | inj_2 res ⇒ res
                in loop()
```

Figure 10. Flat combining: Spec and implementation

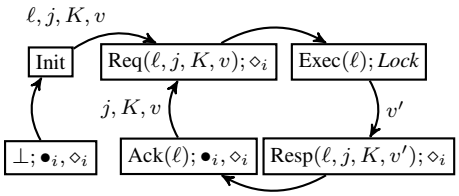
The basic idea is to allow concurrent threads to advertise, through a lock-free side channel, their intent to perform an operation on the data structure. One of the threads then becomes the *combiner*: it locks the data structure and services the requests of all the other threads, one at a time (though new requests may be published concurrently with the combiner's execution). The algorithm exhibits relatively good cache behavior for two reasons: (1) most of the time, operations do not need to execute *any* **CAS**es to complete and (2) the combining thread enjoys good cache locality when executing the operations of the sequential data structure. In practice, flat combining yields remarkably strong performance, even when compared against completely lock-free data structures.

The algorithm and its spec The flat combining algorithm was originally given as informal prose, so our first task is to formalize its implementation and find an appropriate spec, while making its higher-order structure explicit. We do so in Figure 10.

We model an "arbitrary sequential data structure" as a closure of some type $\alpha \to \beta$, where α represents an operation to perform, β represents a result, and calling the function performs the operation by imperatively updating some state hidden within the closure. In other words, $\alpha \to \beta$ represents an *object*. The goal of flat combining, then, is to wrap this object with (cache-efficient) synchronization. Its spec, flat_S, simply uses mkSync to provide generic synchronization via global locking.

Our flat combining implementation flat_I uses three data structures to control synchronization.[10] First, it has a global lock that is used to ensure that only one thread acts as the combiner at a time. Second, it has an instance of Treiber's stack, through which threads *enroll* in the data structure. To enroll, a thread inserts an *operation record* (type **ref** $(\alpha + \beta)$) through which it can advertise requests. The combiner can then perform these requests by iterating over the stack, executing each record that is in the α state and updating it to the β state with its result. Finally, the thread-local reference lclOps allows threads to re-use their operation record once they have enrolled. Note that operation records can be added to the stack or reset from β to α at *any* time—even when the combiner holds the lock.

[10] This implementation simplifies Hendler *et al.*'s description in three respects: it uses a stack rather than a queue, operation records are never de-enrolled, and the combiner does not coalesce multiple operations into a single step. The first simplification makes little difference, because ultimately we give a modular proof using our per-item spec, which could be applied to a queue as well. The other two need only minor changes to the protocol.



Note that, once ℓ has been established, it never changes when going around the cycle.

$$
\begin{array}{rcl}
[\![\mathrm{Init}]\!]_i^L & \triangleq & \mathit{Tid}(i) \\
[\![\mathrm{Req}(\ell, j, \kappa, x_{\mathsf{I}})]\!]_i^L & \triangleq & L(i) = \ell * \ell \hookrightarrow_{\mathsf{I}} \mathbf{inj}_1\ x_{\mathsf{I}} * \mathit{Tid}(i) * \exists x_{\mathsf{S}}.\ (x_{\mathsf{I}}, x_{\mathsf{S}}) \in \alpha * j \mapsto_{\mathsf{S}} \kappa[f'_{\mathsf{S}}\ x_{\mathsf{S}}] \\
[\![\mathrm{Exec}(\ell)]\!]_i^L & \triangleq & L(i) = \ell * \ell \hookrightarrow_{\mathsf{I}} \mathbf{inj}_1 - * \mathit{Tid}(i) \\
[\![\mathrm{Resp}(\ell, j, \kappa, y_{\mathsf{I}})]\!]_i^L & \triangleq & L(i) = \ell * \ell \hookrightarrow_{\mathsf{I}} \mathbf{inj}_2\ y_{\mathsf{I}} * \mathit{Tid}(i) * \exists y_{\mathsf{S}}.\ (y_{\mathsf{I}}, y_{\mathsf{S}}) \in \beta * j \mapsto_{\mathsf{S}} \kappa[y_{\mathsf{S}}] \\
[\![\mathrm{Ack}(\ell)]\!]_i^L & \triangleq & L(i) = \ell * \ell \hookrightarrow_{\mathsf{I}} \mathbf{inj}_2 -
\end{array}
$$

where $f'_{\mathsf{S}} \triangleq \lambda x.\ \mathsf{acq}(\mathsf{lock}_{\mathsf{S}}); \mathbf{let}\ r = f_{\mathsf{S}}(x)\ \mathbf{in}\ \mathsf{rel}(\mathsf{lock}_{\mathsf{S}});\ r$

Figure 11. The protocol for the operation record of thread i

Unfortunately, if we set out to prove the refinement

$$\cdot \vdash \mathsf{flat}_{\mathsf{I}} \preceq \mathsf{flat}_{\mathsf{S}} : \forall \alpha.\ \forall \beta.\ (\alpha \to \beta) \to (\alpha \to \beta)$$

we will run headlong into a problem: it does not hold! To wit: let

$$C \triangleq \mathbf{let}\ r = \mathbf{newLcl},\ f = [\,]\ \_\ \_\ (\lambda().\ \mathbf{getLcl}(r))\ \mathbf{in\ fork}\ (\mathbf{setLcl}(r, \mathbf{true});\ f());\quad \mathbf{setLcl}(r, \mathbf{false});\ f()$$

When this context is applied to $\mathsf{flat}_{\mathsf{S}}$, it always returns $\mathsf{some}(\mathbf{false})$, because the final $f()$ is always executed on the thread whose local r is **false**. But when applied to $\mathsf{flat}_{\mathsf{I}}$, it can also return $\mathsf{some}(\mathbf{true})$: the forked thread might act as the combiner, performing *both* applications of f and thus using its own thread-local value for r.

The crux of the problem is that thread-local storage allows functions to observe the identity of the thread executing them. We need to rule out this kind of side-effect. We do so by unrolling the definition of refinement for functions, and simply removing access to ID-related knowledge, leading to a *qualified refinement*:

$$\forall \alpha, \beta, f_{\mathsf{I}}, f_{\mathsf{S}}.\mathrm{Type}(\alpha) \wedge \mathrm{Type}(\beta) \wedge \Box(\forall (x_{\mathsf{I}}, x_{\mathsf{S}}) \in \alpha.(f_{\mathsf{I}}\ x_{\mathsf{I}}, f_{\mathsf{S}}\ x_{\mathsf{S}}) \downarrow^{\mathrm{pure}} \beta) \Rightarrow \cdot \vdash \mathsf{flat}_{\mathsf{I}}\ \_\ \_\ f_{\mathsf{I}} \preceq \mathsf{flat}_{\mathsf{S}}\ \_\ \_\ f_{\mathsf{S}} : \alpha \to \beta$$

where $(e_{\mathsf{I}}, e_{\mathsf{S}}) \downarrow^{\mathrm{pure}} \varphi \triangleq \forall i, j, \kappa.\ \{j \mapsto_{\mathsf{S}} \kappa[e_{\mathsf{S}}]\}\ i \Rightarrow e_{\mathsf{I}}\ \{x_{\mathsf{I}}.\ \exists x_{\mathsf{S}}.\ \varphi(x_{\mathsf{I}}, x_{\mathsf{S}}) * j \mapsto_{\mathsf{S}} \kappa[x_{\mathsf{S}}]\}$

This "pure" notion of related expressions embodies a semantic version of purity in a type-and-effect system where the only effect is "uses thread-local storage". It coincides *exactly* with the expression relation in Turon *et al.* [31], where thread-local storage was not present. It is easy to prove, using CaReSL, that for any well-typed $f : \tau \to \tau'$ that does not use thread-local storage (*i.e.,* an "effect-free" f) we have $\Box\forall(x_{\mathsf{I}}, x_{\mathsf{S}}) \in [\![\tau]\!].\ (f\ x_{\mathsf{I}}, f\ x_{\mathsf{S}}) \downarrow^{\mathrm{pure}} [\![\tau']\!]$.

The protocol To prove the qualified refinement, we need to formulate a protocol characterizing the algorithm. We do so by giving a local protocol governing the operation record of each thread, and then tying these local protocols together into a single global one. Figure 11 gives the local protocol for a thread with ID i, where the states have the following informal meanings:

$\bot$	thread i has not yet created its operation record.
Init	thread i is creating its operation record.
Req	thread i has an active request.
Exec	the combiner is executing thread i's operation.
Resp	thread i's operation is complete.
Ack	thread i has acknowledged the completion.

The cyclic nature of the protocol reflects that, once thread i has enrolled an operation record, it can reuse that record indefinitely. The labels on transitions signify *branching* that depends on the chosen values of the listed variables. Note that, while ℓ (the location of the operation record) is chosen arbitrarily during initialization, it must remain fixed thereafter.

Movement through the local protocol encompasses two roles: that of thread i, and that of the combiner. The protocol guarantees: (1) when thread i begins execution of the flat combining algorithm, the current state is either $\bot$ or Ack, meaning that either there is no operation record associated with the thread, or that the associated record is ready to be re-used; (2) only thread i can move to or from Init and Ack; and (3) only the combiner can move to or from Exec. To fully understand how these constraints are enforced, we must take into account both the tokens and the state interpretations of the protocol. The tokens include $\bullet_i$ and $\diamond_i$ (one for each thread i, used only in i's local protocol) and a single global token *Lock* representing ownership of the lock.

The interpretations $[\![-]\!]_i^L$ of the non-$\bot$ states are shown in the figure; the parameter L represents the contents of lclOps. With the exception of $\bot$ and Ack, all states assert ownership of $\mathit{Tid}(i)$. On the other hand, when thread i begins executing the algorithm, *it* will have ownership of $\mathit{Tid}(i)$ (recall the definition of refinement, §3.3). Thus, thread i can assume that the protocol is in state $\bot$ or Ack—and only thread i can take a step away from these states. For thread i to take such a step, it must "prove" that it *is* thread i by giving up $\mathit{Tid}(i)$, but in return it gains the token $\bullet_i$ as a "receipt" that can later be traded back for $\mathit{Tid}(i)$ (by later moving to Ack). All told, the ownership of $\mathit{Tid}(i)$ and the corresponding token $\bullet_i$ account for the first two of the guarantees listed above.

The third guarantee is achieved using a similar strategy: to move to the Exec state, the combiner must "prove" that it holds the lock by temporarily giving up the (global) *Lock* token, but it receives the (local) token $\diamond_i$ as a receipt. Subsequently, only the combiner can move to Resp, making the opposite trade.

After initialization, each state also asserts that the location ℓ of the operation record for thread i both corresponds to $L(i)$ and points to an appropriate sum injection (depending on the state).

A final aspect of the local protocol is capturing the *cooperation* inherent in flat combining: the combiner executes code on behalf of other threads. Cooperation is difficult for compositional refinement techniques to handle, because such techniques usually require proving that for *each piece* of implementation code the corresponding piece of spec code behaves the same way—and thus they typically provide no way to account for the *mismatch* between implementation and spec that cooperation entails. While our definition of refinement (§3.3) likewise imposes a one-to-one relationship between implementation and spec code in its pre- and post-conditions, our treatment of spec code as a resource allows ownership to be transferred to other threads *during* the execution of the implementation.

Thus, the Req and Resp states the flat combining protocol assert *shared* ownership of the spec code for thread i. In moving from Req to Exec the combiner thread is forced to take ownership of thread i's spec code, and to subsequently move to Resp, the combiner must actually execute this code on thread i's behalf. On the other hand, by moving to the Ack state, thread i regains ownership of both $\mathit{Tid}(i)$ *and* its spec code, by giving up $\bullet_i$, its "reciept" token.

The global transition system is then a *product construction*:

$$
\begin{array}{ll}
\text{State space} & \left\{ (S, s) \;\middle|\; \begin{array}{l} S \in \mathbb{N} \xrightarrow{\mathrm{fin}} \mathrm{OpState},\quad s \in \mathrm{LockState}, \\ \text{at most one } S(i) \text{ or } s \text{ owns } \mathit{Lock} \end{array} \right\} \\
\text{Transitions} & (S, s) \rightsquigarrow (S', s') \triangleq s \rightsquigarrow^?_{\mathsf{lock}} s' \wedge \forall i.\ S(i) \rightsquigarrow^?_i S'(i) \\
\text{Owned tokens} & \mathcal{T}(S, s) \triangleq \mathcal{T}_{\mathsf{lock}}(s) \cup \bigcup_i \mathcal{T}_i(S(i))
\end{array}
$$

A global state includes a collection S of local states from the operation record protocol (Figure 11), one for each thread ID. In giving the state space, we pun the $\bot$ state with an "undefined"

input to a partial function—and thus, we require that only a finite number of threads i have a non-$\bot$ state in S. Global states also have a single state s drawn from the standard locking protocol (§3.2), reflecting the state of the global lock. The global *Lock* token is *shared* amongst all of the combined protocols: in a valid global state, at most one of the local states $S(i)$ or s may claim the *Lock* token. A transition in the global protocol allows at most one transition in any of the local protocols. Finally, the tokens owned in a global state are just the union of all the tokens claimed by the local states.

We interpret states of the locking protocol as follows:

$$\begin{array}{rcl} [\![\mathsf{Unlocked}]\!] & \triangleq & \mathsf{lock} \hookrightarrow_{\mathrm{I}} \textbf{false} * \mathsf{lock}_{\mathrm{S}} \hookrightarrow_{\mathrm{S}} \textbf{false} \\ [\![\mathsf{Locked}]\!] & \triangleq & \mathsf{lock} \hookrightarrow_{\mathrm{I}} \textbf{true} \end{array}$$

Thus, the combiner gains private ownership of both the *Lock* token and the spec lock (*i.e.,* the internal lock used by mkSync; see f'_{S} in Figure 11) when it acquires the implementation lock. Finally, we lift these local interpretations to interpret global states via the following predicate:

$$(S, s).\ [\![s]\!] * \exists L.\ \mathsf{lclOps} \hookrightarrow_{\mathrm{I}} L * \mathop{\Large *}_{i \in \mathrm{dom}(S)} [\![S(i)]\!]^L_i$$

The proof We close with a high-level view of the proof itself; details, as usual, are in the appendix. Unrolling the statement of qualified refinement, we need to prove

$$\{\mathit{Tid}(i) * j \Mapsto_{\mathrm{S}} \kappa[\mathsf{flat}_{\mathrm{S}}\ \_\ \_\ f_{\mathrm{S}}]\}\ \ i \Mapsto \mathsf{flat}_{\mathrm{I}}\ \_\ \_\ f_{\mathrm{I}}\ \ \{x_{\mathrm{I}}.\ \exists x_{\mathrm{S}}.\ (x_{\mathrm{I}}, x_{\mathrm{S}}) \in [\![\alpha \to \beta]\!] * \mathit{Tid}(i) * j \Mapsto_{\mathrm{S}} \kappa[x_{\mathrm{S}}]\}$$

under the assumptions

$$i \bowtie j,\ \ \mathrm{Type}(\alpha),\ \ \mathrm{Type}(\beta),\ \ \square(\forall (x_{\mathrm{I}}, x_{\mathrm{S}}) \in \alpha.(f_{\mathrm{I}}\ x_{\mathrm{I}}, f_{\mathrm{S}}\ x_{\mathrm{S}}) \downarrow^{\mathrm{pure}} \beta)$$

The proof begins in much the same way as the refinement proof for Treiber's stack (§3.3): we execute the **let**-bound expressions that allocate the hidden state in both the implementation and spec, *i.e.,*

$$\mathsf{lock} \hookrightarrow_{\mathrm{I}} \textbf{false} * \mathsf{lclOps} \hookrightarrow_{\mathrm{I}} \emptyset * \exists\, \mathsf{lock}_{\mathrm{S}}.\ j \Mapsto_{\mathrm{S}} \kappa[f'_{\mathrm{S}}] * \mathsf{lock}_{\mathrm{S}} \hookrightarrow_{\mathrm{S}} \textbf{false}$$

where $\mathsf{lock}_{\mathrm{S}}$ is the lock allocated by mkSync and f'_{S} is as in Figure 11. These resources are precisely what we need to apply NEWISL for our global protocol, moving them from private to shared ownership. Letting π be the global protocol defined above, we can claim $\square\ \boxed{(\emptyset, \mathsf{Unlocked}); \emptyset}^n_\pi$, *i.e.,* permanent knowledge that the global protocol exists. We must then, in the context of this island and our previous assumptions, show that the closure returned by $\mathsf{flat}_{\mathrm{I}}$ refines the one returned by $\mathsf{flat}_{\mathrm{S}}$, *i.e.,* f'_{S}, at type $\alpha \to \beta$.

We first verify a version $\mathsf{flat}'_{\mathrm{I}}$ of the flat combining algorithm that is identical to the one in Figure 10, except that it uses the coarse-grained stack, allowing us to use the per-item spec of §3.3. We instantiate the per-item spec using the same predicate Op for per-item resources and iteration knowledge (p and q, respectively, in the per-item spec): $\mathrm{Op} \triangleq (\ell).\ \exists k.\ \boxed{[k \mapsto \mathrm{Req}(\ell, -, -, -)]}^n_\pi$.[11] This is a *local* assertion about the global protocol, in that the states of threads other than k can be in any rely-future state of $\bot$, *i.e.,* any state whatsoever. The Op predicate just claims that location ℓ is an initialized operation record for *some* thread. By the per-item spec, all locations inserted into the stack must satisfy this property—and since we have $\mathrm{Op}(\ell) \Rightarrow \square\mathrm{Op}(\ell)$, the property can be assumed even when iterating over stale elements of the stack. Operation records are created via install, whose specification consumes $\mathit{Tid}(i)$ and associated spec resources in exchange for the "receipt" $\bullet_i$:

$$\left\{\mathit{Tid}(i) * j \Mapsto_{\mathrm{S}} \kappa[f'_{\mathrm{S}}\ x_{\mathrm{S}}]\right\} i \Mapsto \mathsf{install}\ x \left\{o. \boxed{[i \mapsto \mathrm{Req}(o, j, \kappa, x)]; \bullet_i}^n_\pi\right\}$$

The combiner actually performs operations via doOp,

$$\{\mathrm{Op}(o) * P\}\ \mathsf{doOp}\ o\ \{P\} \quad \text{where} \quad P \triangleq \boxed{\emptyset; \mathit{Lock}}^n_\pi * \mathsf{lock}_{\mathrm{S}} \hookrightarrow_{\mathrm{S}} \textbf{false}$$

which assumes that the given location o is a valid operation record, and that the invoking thread owns the *Lock* token as well as the spec lock. The shape of the spec for doOp exactly matches that required for iteration in the per-item spec (§3.3), with P serving as the loop invariant, thus allowing us to deduce $\{P\}\ \mathsf{iter}\ \mathsf{doOp}\ \{P\}$. These auxiliary specs make it quite straightforward to prove refinement for the closures returned by $\mathsf{flat}'_{\mathrm{I}}$ (the version using the coarse-grained stack) and $\mathsf{flat}_{\mathrm{S}}$.

Finally, suppose we plug the combiner into a client context C that provides a suitable argument f. The above proof allows (together with soundness, §3.4) us to deduce $\models C[\mathsf{flat}'_{\mathrm{I}}] \preceq C[\mathsf{flat}_{\mathrm{S}}] : \tau$. Likewise, refinement for the Treiber stack allows us to deduce $\models C[\mathsf{flat}_{\mathrm{I}}] \preceq C[\mathsf{flat}'_{\mathrm{I}}] : \tau$. Because contextual refinement is *transitive*, we can conclude $\models C[\mathsf{flat}_{\mathrm{I}}] \preceq C[\mathsf{flat}_{\mathrm{S}}] : \tau$.[12] Hence, we have given a modular proof for flat combining, layering refinement and Hoare-style reasoning.

5. Related work

In many respects, CaReSL builds directly on the semantic foundation that we and colleagues laid in our prior work [31]. There, we developed a relational Kripke model of a language very similar to the one considered here, and showed how granularity abstraction for several sophisticated (but structurally simple) fine-grained data structures could be established by direct appeal to the model. The present work is a natural continuation of that work. First, we provide a *logic* that greatly simplifies reasoning compared to working with the model; such a proof theory is essential for scaling the verification method to large examples. Second, we use our logic not just to prove granularity abstraction results in isolation (as we did before), but also to facilitate the modular verification of a higher-order, structurally complex program. Along the way, we also extend our prior model to handle thread-local storage.

The logic itself draws inspiration from LADR [5], which in turn provided a proof theory for reasoning about a *sequential* relational Kripke model (ADR [1]). Aside from incorporating concurrency, CaReSL provides a deeper unification of refinement and Hoare logic by treating refinement as a mode of use of Hoare logic.

Higher-order functions and concurrency While there has been stunning progress in logics for concurrency over the last decade, very few of these logics meaningfully support *higher-order* programming, and among those that do, none supports reasoning about fine-grained synchronization or granularity abstraction.

The logic that comes closest is *higher-order concurrent abstract predicates* (HOCAP), first proposed for reasoning about first-order code [4] (the "higher-order" refers to the logic of predicates) and very recently applied to higher-order code as well [27]. HOCAP, like its predecessor CAP [3], accounts for concurrency by way of "shared region" assertions that constrain the possible updates to a shared resource. Our island assertions resemble shared region assertions—indeed, we have adopted notation suggesting as much—but work at a higher level of abstraction (*i.e.,* protocol states), separating *knowledge* bounding the state of the protocol (treated as a copyable assertion) from the *rights* to change the state (treated as a linear resource: tokens); see [31] for a more detailed comparison. Because CAP lacks granularity abstraction, it is difficult to give a single "principal" specification for a data structure. Instead, one gives specialized specs (like the per-item spec for stacks) reflecting particular usages envisioned for a client—which means new client scenarios necessitate new proofs. HOCAP attempts to address this shortcoming by explicitly quantifying over the way a client uses a data structure, but (1) this introduces the potential for

[11] We leave off the locking state when it is Unlocked.

[12] We are appealing to *semantic* refinement here to take advantage of transitivity. This reasoning can be internalized in CaReSL by adding a proposition for semantic refinement and axioms for refinement soundness and transitivity, but there is little benefit to doing so.

a problematic circularity, which clients must explicitly prove does not arise, and (2) it is not clear how to scale the approach to handle *cooperation* between threads (as in the flat combiner).

Other concurrency logics that support higher-order functions—such as Hobor *et al.*'s extension of concurrent separation logic [14], or the very recent *Subjective Concurrent Separation Logic* [17]—support only reasoning about simple lock-based synchronization, and do not enable the kinds of refinement proofs we have presented.

Granularity abstraction Herlihy and Wing's seminal notion of *linearizability* [13] has long been held as the gold standard of correctness for concurrent data structures, but as Filipović *et al.* argue [8], what clients of these data structures *really* want is a contextual refinement property. Filipović *et al.* go on to show that, under certain (strong) assumptions about a programming language, linearizability implies contextual refinement for that language. More recently, Gotsman and Yang generalized both linearizability and this result (the so-called *abstraction theorem*) to include potential ownership transfer of memory between concurrent data structures and their clients in a first-order setting [9]. While in principle proofs of linearizability can be composed with these results to support granularity abstraction, no existing logic has provided support for doing so. We found it simpler to work with refinement directly (in particular, to *encode* it directly into a Hoare logic). CaReSL enables clients to layer ownership-transferring protocols *on top* of a coarse-grained reference implementation, *after* applying granularity abstraction, as we showed with the per-item spec (§3.3).

The only Hoare logic we are aware of that can (formally) prove refinement results is Liang and Feng's new logic [18] (extending LRG [7]), which is inspired by the use of ghost state in Vafeiadis's thesis [32]. The logic is powerful enough to deal with a range of sophisticated, fine-grained concurrent algorithms, but it is limited to *first-order* code. In addition, the specifications used in refinement are not reference implementations, but are instead essentially atomic Hoare triples. While such specifications are appealingly abstract, they present two limitations: (1) they do not model the more general notion of granularity abstraction (supporting only *atomicity* abstraction, as we explained in footnote 1) and (2) they do not support the kind of transitive composition of proofs that we used in our case study. As a result, it is unclear how to use the logic to build modular proofs layering Hoare logic and refinement.

A radically different approach to atomicity abstraction is Lipton's method of *reduction* [19], which is based on showing that an atomic step *commutes* with all concurrent activity, and can therefore be coalesced into a larger atomic block with *e.g.,* code that is sequenced after it. Elmas *et al.* developed a logic for proving linearizability via a combination of reduction and abstraction [6], which in some ways resembles our interleaved application of refinement and Hoare-style reasoning, but with a rather different way of *proving* refinement. It is limited, however, to first-order code and atomicity refinement, and like HOCAP it is not powerful enough to handle fine-grained algorithms that employ cooperation. Moreover, it does not allow linearizability proofs to be composed transitively.

Acknowledgements We would like to thank David Swasey for his careful reading of both the paper and its appendix, and the anonymous reviewers for their suggestions and encouragement. This work was partially funded by the EC FET project ADVENT.

References

[1] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *JPDC*, 37(1):55–69, Aug. 1996.

[3] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[4] M. Dodds, S. Jagannathan, and M. Parkinson. Modular reasoning for deterministic parallelism. In *POPL*, 2011.

[5] D. Dreyer, G. Neis, A. Rossberg, and L. Birkedal. A relational modal logic for higher-order stateful ADTs. In *POPL*, 2010.

[6] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS*, 2010.

[7] X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

[8] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411, 2010.

[9] A. Gotsman and H. Yang. Linearizability with ownership transfer. In *CONCUR*, 2012.

[10] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPOPP*, 2005.

[11] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

[12] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, 2004.

[13] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[14] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

[15] C. B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS*, 5(4):596–619, 1983.

[16] D. Lea. The java.util.concurrent ConcurrentHashMap.

[17] R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, 2013.

[18] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, 2013.

[19] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[20] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[21] M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, 2005.

[22] A. M. Pitts and I. Stark. Operational reasoning for functions with local state. In *HOOTS*, 1998.

[23] G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *TLCA*, 1993.

[24] F. Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *LICS*, 2008.

[25] J. H. Reppy. *Higher-order concurrency*. PhD thesis, Cornell University, 1992.

[26] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[27] K. Svendsen, L. Birkedal, and M. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, 2013.

[28] R. Treiber. Systems programming: coping with parallelism. Technical report, Almaden Research Center, 1986.

[29] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *JFP*, 8(1):23–60, Jan. 1998.

[30] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency: Appendix. `http://www.mpi-sws.org/~turon/caresl/appendix.pdf`.

[31] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, 2013.

[32] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

[33] H. Yang. Relational separation logic. *TCS*, 375(1-3):308–334, 2007.