Logical Relations for Fine-Grained Concurrency

Aaron Turon
Northeastern University
turon@ccs.neu.edu

Jacob Thamsborg
IT University of Copenhagen
thamsborg@itu.dk

Amal Ahmed
Northeastern University
amal@ccs.neu.edu

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

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

Abstract

Fine-grained concurrent data structures (or FCDs) reduce the granularity of critical sections in both time and space, thus making it possible for clients to access different parts of a mutable data structure in parallel. However, the tradeoff is that the implementations of FCDs are very subtle and tricky to reason about directly. Consequently, they are carefully designed to be *contextual refinements* of their coarse-grained counterparts, meaning that their clients can reason about them as if all access to them were sequentialized.

In this paper, we propose a new semantic model, based on Kripke logical relations, that supports direct proofs of contextual refinement in the setting of a type-safe high-level language. The key idea behind our model is to provide a simple way of expressing the "local life stories" of individual pieces of an FCD's hidden state by means of *protocols* that the threads concurrently accessing that state must follow. By endowing these protocols with a simple yet powerful transition structure, as well as the ability to assert invariants on both heap states and specification *code*, we are able to support clean and intuitive refinement proofs for the most sophisticated types of FCDs, such as conditional compare-and-set (CCAS).

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

Suppose you want to take a sequential mutable data structure and adapt it to a concurrent setting, so that multiple threads can safely access it in parallel. The simplest way to do it is to treat all of the operations on the data structure as critical sections governed by a common lock. This *coarse-grained* approach to concurrency is easy for clients to reason about, since it essentially sequentializes all access to the data structure, but by the same token it also thwarts any speedup one might hope to gain from parallelism. In contrast, *fine-grained* concurrent data structures (or FCDs) reduce the granularity of critical sections in both time and space, often down to a single primitive atomic instruction like "compare-and-set" (CAS), so that clients can exploit parallelism by having different threads manipulate different parts of the data structure simultaneously.

POPL'13, January 23–25, 2013, Rome, Italy.

As FCDs are very tricky to reason about directly, they are carefully designed to be *contextual refinements* of their course-grained counterparts. This means essentially that, performance gains aside, no client can tell they are working with the fine-grained version of a data structure instead of the coarse-grained version. Put another way, the FCD is a faithful *implementation* of its coarse-grained *specification*. Thus, clients can safely reason about the FCD as if all access to it were sequentialized, while at the same time reaping the efficiency benefits of parallelism.

Contextual refinement is clearly an essential property that clients of an FCD expect to hold [20]. The question is how to prove it. In this paper, we propose a new semantic model that supports **direct** proofs of contextual refinement in the setting of a type-safe **high-level language**. The key idea behind our model is to provide a simple way of expressing the **protocols** that govern the hidden state of an FCD and that the threads concurrently accessing it must follow. By endowing these protocols with a simple yet powerful transition structure, as well as the ability to assert invariants on both heap states and specification *code*, we are able to support intuitive refinement proofs for **the most sophisticated types of FCDs**. We now examine these selling points in more detail.

Direct: Rather than prove contextual refinement directly, most prior approaches have focused on proving a related property on traces called *linearizability* [21]. Linearizability is often viewed as being synonymous with contextual refinement, but in fact this has only recently been shown by Filipovic *et al.* [14] to be the case (and only for a particular class of languages). As Filipovic *et al.* argue, refinement is the property that clients of an FCD actually want, and linearizability is one technique for proving it. We instead provide a direct proof technique for contextual refinement, which sidesteps any discussion of linearizability.

High-level language: Most prior work has only considered FCDs coded in first-order C-like languages. However, one of the most widely-used FCD libraries, `java.util.concurrent`, is written in a type-safe high-level language (Java) and indeed *depends* on the abstraction facilities of Java to ensure that the private state of its FCDs is hidden from clients. Our approach is the first to prove contextual refinement for `java.util.concurrent`-style FCDs in the setting of a higher-order language with abstract types, recursive types, and general mutable references, thus establishing the correctness of these FCDs when linked with unknown well-typed client code.

How? To achieve these first two aims, we employ a *step-indexed Kripke logical relations (SKLR)* model [2, 3, 9, 4]. SKLRs have been actively developed in recent years as an effective tool for reasoning about representation independence for higher-order stateful ADTs, and thus provide a solid foundation for reasoning about contextual refinement of concurrent objects in a realistic, high-level setting. To adapt SKLRs to reasoning about FCDs, we follow the approach of recent work by Birkedal *et al.* [5] and employ a "small-step-style" model, which accounts properly for the possibility that threads are

preempted after every step of computation. Birkedal *et al.*'s model, however, is limited in its ability to reason about interference between threads, which is ubiquitous in FCDs. We therefore generalize their model with support for *protocols*.

Protocols: To understand how an FCD works, it helps to think of each *piece* of the data structure (*e.g.,* each node of a linked list) as being subject to a *protocol* that tells its "life story": how it came to be allocated, how its contents evolve over time, and how it eventually "dies" by being disconnected (or deleted) from the data structure. This protocol describes the rules by which all threads must play as they access the shared state of the FCD.

A number of FCDs additionally require their protocols to support *role-playing*—that is, a mechanism by which different threads participating in the protocol can dynamically acquire certain "roles". These roles may enable them to make certain transitions that other threads cannot. A simple example of this is a locking protocol, under which the thread that acquires the lock adopts the unique role of "lock-holder" and thus knows that no other thread has the ability to release the lock.

How? Following recent work by Dreyer *et al.* in the setting of SKLRs [9], our model supports a direct encoding of protocols as *state transition systems (STSs)* of a certain kind. In comparison to Dreyer *et al.*'s work, we deploy STSs at a much finer granularity and use them to tell *local life stories* about individual nodes of a data structure. Of course, there are also "global" constraints connecting up the life stories of the individual nodes, but to a large extent we are able to reason about FCDs at the level of these local life stories and their local interactions with one another.

In order to account for role-playing, we enrich our STSs with a notion of *tokens*. Intuitively, the idea is that, while the STS defines the basic roadmap for the possible changes to the state of the FCD, some of the roads on that map are toll roads that may only be traversed by threads owning certain tokens. This idea is highly reminiscent of recent work on "concurrent abstract predicates" [7], but we believe our approach is simpler and more direct.

The most sophisticated types of FCDs: In proving refinement for each operation of an FCD, a key step is identifying its *linearization point*, the point during its execution at which the operation can be considered to have "committed", *i.e.,* the point at which its coarse-grained spec can be viewed as having executed atomically. What sets the most sophisticated FCDs apart from the pack, and makes them so challenging to verify, is that their linearization points are hard to identify in a thread-local and temporally-local way.

For example, the "elimination stack" FCD [19] provides side channels by which a "push" and "pop" operation can mutually decide to cancel each other out without touching the stack itself. This works by having one operation (say, push) use the side channel to offer its argument to be pushed; if a thread running the pop operation sees this offer, it can commit both the push and pop at once. Although it is very kind of the pop thread to *cooperate* with the push thread by helping it complete its operation in this way, it also means that the linearization point for push occurs during the execution of pop, thus making thread-local verification difficult.

In other algorithms like CCAS [18, 15], the *nondeterminism* induced by shared-state concurrency has the effect that it is impossible to determine where the linearization point has occurred until after the fine-grained operation has completed its execution. This in turn makes it challenging to reason about refinement in a temporally-local way, *i.e.,* showing that each step of the algorithm, considered in isolation, obeys the FCD's shared-state protocol.

How? The whole point of SKLRs is to provide a way of describing local knowledge about the hidden resources of an abstract data type, but in prior work those "hidden resources" have been synonymous with "local variables" or "a private piece of the heap". To support reasoning about thread cooperation and nondeterminism, we make two orthogonal generalizations to the notion of resources.

First, to model cooperation, we extend resources to also include *specification code*. This extension makes it possible for "the right to commit an operation" (*e.g.,* push, in the example above) to be treated as a shareable resource, which one thread may pass to other "helper" threads to run on its behalf. Second, to model nondeterminism, we extend resources to include *sets of specification states*. This extension makes it possible to *speculate* about all the possible specification states that our FCD implementation could be viewed as refining, so that we can wait until the implementation has finished executing to decide which one we want to choose.

Both of these extensions are formally and conceptually simple—especially in comparison to prior approaches relying on ghost and prophecy variables [1]—and we will demonstrate their utility on both illustrative toy examples (Sections 2.5 and 2.6) and a more realistic FCD example toward the end of the paper (Section 4).

2. The main ideas

2.1 The language

We study FCDs within a variant of the polymorphic lambda calculus, extended with tagged sums, general mutable references (higher-order state), equi-recursive types, CAS, and **fork**. These features suffice for modeling the kinds of FCDs used within java.util.concurrent, which (a) are polymorphic, (b) use recursive, linked data structures, and (c) rely on the abstraction facilities of the language for data hiding.[1] Figure 1 presents the syntax of the language together with excerpts of the static and dynamic semantics. In examples, we will employ a few other features that trivially extend the language defined here, *e.g.,* immutable pairs and records.

While the language is essentially standard, there are a few unusual aspects that help keep our treatment of FCDs concise. Terms are not annotated with types, but polymorphism is nevertheless introduced and eliminated by explicit type abstraction ($\Lambda.e$) and application ($e\ \_$). Reference and tuple types are combined into the type $\mathbf{ref}(\overline{\tau})$, useful for constructing objects with many mutable fields. The term $e[i]$ reads and projects the i-th component from a tuple reference e, while $e[i] := e'$ assigns a new value to that component. When e is a single-cell reference, we will usually write $e := e'$ instead of $e[1] := e'$. Finally, the type $\mathbf{ref}_?(\overline{\tau})$ of "option references" provides an *untagged union* of the unit and reference types, with explicit coercions **null** and $\mathbf{some}(e)$. Because reading and writing operations work on references, and not option references (which must be separately eliminated by cases), there are no null-pointer errors. The net effect of flattening these types is fewer layers of indirection, which simplifies verification.

The type system imposes two important restrictions. First, recursive types $\mu\alpha.\tau$ are required to be *productive*, meaning that all free occurrences of α in τ must appear under a non-μ type constructor. More subtle is the restriction on CAS, which can only be used on components of *comparable type* σ. This constraint is needed because CAS performs an equality comparison of word-sized values at the hardware level; it is only reasonable to apply it at types whose representation allows such a comparison, *i.e.,* base types and locations. Tagged sums are allocated on the heap and hence represented using locations, making them comparable as well. Figure 1 presents the nonstandard typing rules; the remaining rules are standard (see appendix [35]). Note that we use u to denote values that can be stored in the heap.

A threadpool T is a finite map from thread identifiers to expressions. A program configuration $h; T$ pairs a heap with a thread pool. We define a small-step, call-by-value operational semantics as a re-

[1] We will use closures as our primary means of hiding local state.

$$
\begin{array}{rcl}
\tau & ::= & \mathbf{1} \mid \mathbf{B} \mid \mathbf{N} \mid \tau + \tau \mid \mathbf{ref}(\overline{\tau}) \mid \mathbf{ref}_?(\overline{\tau}) \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \alpha \mid \tau \to \tau \\
\sigma & ::= & \mathbf{1} \mid \mathbf{B} \mid \mathbf{N} \mid \tau + \tau \mid \mathbf{ref}(\overline{\tau}) \mid \mathbf{ref}_?(\overline{\tau}) \mid \mu\alpha.\sigma \\
e & ::= & () \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid n \mid e + e \mid x \mid \Lambda.e \mid e\ \_ \mid e\ e \\
& \mid & \mathbf{rec}\ f(x).e \mid \mathbf{new}\ \overline{e} \mid e[i] \mid e[i]\ :=\ e \mid \mathsf{CAS}(e[i], e, e) \mid \ell \mid \mathbf{fork}\ e \\
& \mid & \mathbf{null} \mid \mathbf{some}(e) \mid \mathbf{case}(e, \mathbf{null} \Rightarrow e, \mathbf{some}(x) \Rightarrow e) \\
& \mid & \mathbf{inj}_i\ e \mid \mathbf{case}(e, \mathbf{inj}_1\ x \Rightarrow e, \mathbf{inj}_2\ y \Rightarrow e) \\
v & ::= & \mathbf{rec}\ f(x).e \mid \Lambda.e \mid () \mid n \mid \mathbf{true} \mid \mathbf{false} \mid \ell \mid x \quad u\ ::=\ (\overline{v}) \mid \mathbf{inj}_i\ v \\
\Gamma & ::= & \cdot \mid \Gamma, x : \tau \qquad \Delta\ ::=\ \cdot \mid \Delta, \alpha \qquad \Omega\ ::=\ \Delta; \Gamma \\
K & ::= & [\,] \mid \mathbf{some}(K) \mid \mathbf{case}(K, \mathbf{null} \Rightarrow e, \mathbf{some}(x) \Rightarrow e) \mid K\ \_ \mid \cdots \\
T & \in & \text{ThreadPool} \triangleq \mathbb{N} \xrightarrow{\text{fin}} \text{Exp} \qquad h \in \text{Heap} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{HeapVal}
\end{array}
$$

Type rules $\boxed{\Omega \vdash e : \tau}$

$$
\frac{\Omega \vdash e : \mathbf{ref}(\overline{\tau}) \quad \tau_i = \sigma \quad \Omega \vdash e_o : \sigma \quad \Omega \vdash e_n : \sigma}{\Omega \vdash \mathsf{CAS}(e[i], e_o, e_n) : \mathbf{B}} \qquad \frac{\Omega \vdash e : \mathbf{1}}{\Omega \vdash \mathbf{fork}\ e : \mathbf{1}}
$$

$$
\frac{\Omega \vdash e : \mathbf{ref}(\overline{\tau})}{\Omega \vdash \mathbf{some}(e) : \mathbf{ref}_?(\overline{\tau})} \qquad \frac{\Omega, \alpha \vdash e : \tau}{\Omega \vdash \Lambda.e : \forall\alpha.\tau} \qquad \frac{\Omega \vdash e : \forall\alpha.\tau}{\Omega \vdash e\ \_ : \tau[\tau'/\alpha]}
$$

Primitive reductions $\boxed{h; e \hookrightarrow h'; e'}$

$$
\begin{array}{rcll}
h; \ell[i] & \hookrightarrow & h; v_i & \text{when } h(\ell) = (\overline{v}) \\
h; \mathsf{CAS}(\ell[i], v_o, v_n) & \hookrightarrow & h[\ell[i] = v_n]; \mathbf{true} & \text{when } h(\ell)[i] = v_o \\
h; \mathsf{CAS}(\ell[i], v_o, v_n) & \hookrightarrow & h; \mathbf{false} & \text{when } h(\ell)[i] \neq v_o
\end{array}
$$

$$
\begin{array}{ll}
h; \mathbf{case}((), \mathbf{null} \Rightarrow e_1, \mathbf{some}(x) \Rightarrow e_2) \hookrightarrow h; e_1 & \\
h; \mathbf{case}(\ell, \mathbf{null} \Rightarrow e_1, \mathbf{some}(x) \Rightarrow e_2) \hookrightarrow h; e_2[\ell/x] & \\
h; \mathbf{new}\ (\overline{v}) \hookrightarrow h \uplus [\ell \mapsto (\overline{v})]; \ell & h; \mathbf{null} \hookrightarrow h; () \\
h; \mathbf{inj}_i\ v \hookrightarrow h \uplus [\ell \mapsto \mathbf{inj}_i\ v]; \ell & h; \mathbf{some}(\ell) \hookrightarrow h; \ell \\
h; \Lambda.e\ \_ \hookrightarrow h; e &
\end{array}
$$

Program reduction $\boxed{h; T \to h'; T'}$

$$
\frac{h; e \hookrightarrow h'; e'}{h; T \uplus [i \mapsto K[e]] \to h'; T \uplus [i \mapsto K[e']]}
$$

$$
h; T \uplus [i \mapsto K[\mathbf{fork}\ e]] \to h; T \uplus [i \mapsto K[()]] \uplus [j \mapsto e]
$$

Figure 1. Our language: $F^{\mu!}$ with **fork** and CAS

lation $\to$ between program configurations, allowing the creation of new threads and the nondeterministic interleaving of existing ones. We use evaluation contexts K to specify a left-to-right evaluation order within each thread.

2.2 A finer look at FCDs

The granularity of a concurrent data structure is a measure of the locality of synchronization between threads accessing it. Coarse-grained data structures provide exclusive, global access for the duration of a critical section: a thread holding the lock can access as much of the data structure as needed, secure in the knowledge that it will encounter a consistent, frozen representation. By contrast, fine-grained data structures localize or eliminate synchronization, forcing threads to do their work on the basis of limited knowledge about its state—sometimes as little as what the contents of a single word are at a single moment.

The local, highly-concurrent nature of FCDs is best understood by example. In Figure 2, we give a variant of Michael and Scott's lock-free queue [29].[2] The queue maintains a reference, head, to a *nonempty* linked list; the first node of the list is considered a "sentinel" whose data does not contribute to the queue.

Nodes are dequeued from the front of the list, so we examine the deq code first. If the queue is *logically* nonempty, it contains at least two nodes: the sentinel (physical head), and its successor (logical head). Intuitively, the deq operation should atomically update the head reference from the sentinel to its successor; after doing so, the old logical head becomes the new sentinel, and the next node, if any, becomes the new logical head. Because there is no lock protecting head, however, a concurrent operation could update it at any time. Thus, deq employs *optimistic concurrency*: after gaining access to the sentinel by dereferencing head, it does some additional work—finding the logical head—while optimistically assuming that head has not changed behind its back. In the end, optimism meets reality through CAS, which performs an atomic update only when head is unchanged. If its optimism was misplaced, deq must start from scratch. After all, the queue's state may have entirely changed in the interim.

The key thing to notice is just how little knowledge deq has as it executes. Immediately after reading head, the most that can be said is that the resulting node *was once* the physical head of the queue. The power of CAS is that it mixes instantaneous knowledge—the head *is now* n—with instantaneous action—the head *becomes* n'. The weakness of CAS is that this potent mixture applies only to a single word of memory. For deq, this weakness is manifested in the lack of knowledge CAS has about the new value n', which *should* still be the successor to the physical head n at the instant of the CAS. Because CAS cannot check this fact, it must be established *pessimistically*, *i.e.,* guaranteed to be true on the basis of the queue's internal protocol. We will see in a moment how to formulate such a protocol, but first, we examine the more subtle enq.

In a singly-linked queue implementation, one would expect to have both head and tail pointers, and indeed the full Michael-Scott queue includes a "tail" pointer. However, because CAS operates on only one word at a time, it is impossible to use a single CAS operation to both link in a new node and update a tail pointer. The classic algorithm allows the tail pointer to lag behind the true tail by at most one node, while the implementation in java.util.concurrent allows multi-node lagging. These choices affect performance, of course, but from a correctness standpoint one needs to make essentially the same argument whether one has a lagging tail, or simply traverses from head as we do.

In all of these cases, it is necessary to *find* the actual tail of the list (whose successor is **null**) by doing some amount of traversal. Clearly, this requires at least that the actual tail be reachable from the starting point of the traversal; the loop invariant of the traversal is then that the tail is reachable from the current node. But in our highly-concurrent environment, we must account for the fact that the data structure is changing under foot, even as we traverse it. The node that was the tail of the list when we *began* the traversal might not even be *in* the data structure by the time we finish.

2.3 The story of a node

We want, ultimately, to prove that MSQ refines its coarse-grained specification CGQ. The latter uses a lock to protect its internal state; $\mathsf{sync}(e)\ \{\ e'\ \}$ is a simple derived form that wraps acquisition/release of the non-reentrant spinlock e around the code e'.[3] Ideally, the proof would proceed in the same way one's intuitive reasoning does, *i.e.,* by considering the execution of a single function by a single thread one line at a time, reasoning about what is known at each program point. To achieve this goal, we must solve two closely-related problems: we must characterize the possible interference from concurrent threads, and we must characterize the knowledge that our thread can gain.

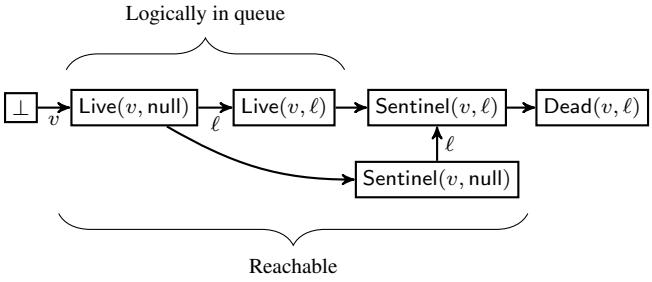
We solve both of these problems by introducing a notion of *protocol*, based on the abstract state transition systems of Dreyer *et al.* [9], but with an important twist: we apply these transition systems at the level of *individual nodes*, rather than as a description

[2] We use the shorthand $\mathsf{cons}(e, e') \triangleq \mathbf{some}(\mathbf{new}\ (e, e'))$.

[3] The definition of sync is given in the appendix [35].

```
MSQ: ∀α.1 → { enq : α → 1,  deq : 1 → ref?(α) }
MSQ ≜ Λ. λ(). let head = new (new (null, null)) in {
  deq = rec try(). let n = head[1] in case n[2]
            of some(n′) ⇒ if CAS(head[1], n, n′)
                          then n′[1] else try()
             | null     ⇒ null,      (* queue is empty *)
  enq = λx. let n = cons(some(new x), null) in
            let rec try(c) = case c[2]
              of some(c′) ⇒ try(c′)
               | null     ⇒ if CAS(c[2], null, n)
                            then () else try(c)
            in try(head[1])
}
CGQ ≜ Λ. λ(). let head = new null, lock = new false in {
  deq = λ(). sync(lock) {
    case head[1] of some(n) ⇒ head[1] := n[2]; some(new n[1])
                  | null    ⇒ null
  },
  enq = λx. sync(lock) { ... }   (* elided *)
}
```

Per-node protocol:



Global interpretation:

$$I(s) \triangleq \mathsf{head}_\mathrm{I} \mapsto_\mathrm{I} \ell_0 * \ell_0 \mapsto_\mathrm{I} (v_0, v_\mathrm{I}) * \exists v_\mathrm{S}.\ \mathsf{head}_\mathrm{S} \mapsto_\mathrm{S} v_\mathrm{S} * \mathsf{lock} \mapsto_\mathrm{S} \textbf{false} * \mathsf{link}(v_\mathrm{I}, v_\mathrm{S}, s_L) * \overline{(s(\ell_i') \neq \bot \wedge \ell_i \mapsto_\mathrm{I} (v_i, \ell_i'))}$$
$$\text{when } s = [\ell_0 \mapsto \mathsf{Sentinel}(v_0, v_\mathrm{I})] \uplus s_L \uplus \overline{[\ell_i \mapsto \mathsf{Dead}(v_i, \ell_i')]}$$

$$\mathsf{link}(\textbf{null}, \textbf{null}, \emptyset) \triangleq \mathsf{emp}$$

$$\mathsf{link}(\ell_\mathrm{I}, \ell_\mathrm{S}, [\ell_\mathrm{I} \mapsto \mathsf{Live}(v_\mathrm{I}, v_\mathrm{I}')] \uplus s) \triangleq \exists v, v_\mathrm{S}, v_\mathrm{S}'.\ v \preceq^{\mathcal{V}} v_\mathrm{S} : \alpha * v_\mathrm{I} \mapsto_\mathrm{I} v * \ell_\mathrm{I} \mapsto_\mathrm{I} (v_\mathrm{I}, v_\mathrm{I}') * \ell_\mathrm{S} \mapsto_\mathrm{I} (v_\mathrm{S}, v_\mathrm{S}') * \mathsf{link}(v_\mathrm{I}', v_\mathrm{S}', s)$$

Figure 2. A variant of Michael and Scott's queue

of the entire data structure. These transition systems describe what we call the *local life stories* of each piece of an FCD. The diagram in Figure 2 is just such a story. Every heap location can be seen as a *potential* node in the queue, but all but finitely many are "unborn" (state $\bot$). After birth, nodes go through a progression of life changes. Some changes are manifested physically. The transition from $\mathsf{Live}(v, \textbf{null})$ to $\mathsf{Live}(v, \ell)$, for example, occurs when the successor field of the node is updated to link in a new node. Other changes reflect evolving relationships. The transition from Live to Sentinel, for example, does not represent an internal change to the node, but rather a change in the node's position in the data structure. Finally, a node "dies" when it becomes unreachable.

The benefit of these life stories is that they account for knowledge and interference together, in a local and abstract way. Knowledge is expressed by asserting that a given node is *at least* at a certain point in its life story. This kind of knowledge is inherently stable under interference, because *all* code must conform to the protocol, and is therefore constrained to a forward march through the STS. The life story gathers together in one place all the knowledge and interference that is relevant to a given node, even knowledge like "reachability" which is ostensibly a global property. This allows us to draw global conclusions from local information, which is precisely what is needed when reasoning about FCDs. For example, notice that no node can die with a **null** successor field. A successful CAS on the successor field from **null** to some location—like the one performed in enq—entails that the successor field was instantaneously **null** (local information), which by the protocol means the node was instantaneously reachable (global information), which entails that the CAS makes a new node reachable. Similarly, the protocol makes it immediately clear that the queue is free from any ABA problems [34], because nodes cannot be reincarnated and their fields, once non-**null**, never change.

To formalize this reasoning, we must connect the abstract account of knowledge and interference provided by the protocol to concrete constraints on the queue's representation. We do this by giving a *state-dependent invariant* I for the data structure, where "state" refers to abstract STS states. For the queue, we have the following set of states for each node's local STS:

$$S_0 \triangleq \{\bot\} \cup \{\mathsf{Live}(v, v') \mid v, v' \in \mathrm{Val}\} \cup \{\mathsf{Sentinel}(v, v') \mid v, v' \in \mathrm{Val}\} \cup \{\mathsf{Dead}(v, \ell) \mid v \in \mathrm{Val}, \ell \in \mathrm{Loc}\}$$

along with the transition relation $\rightsquigarrow_0$ given in the diagram, where the annotated edges denote branches for choosing particular concrete v and ℓ values. The data structure as a whole is governed by a *product* STS with states $S \triangleq \mathrm{Loc} \xrightarrow{\text{fin}} S_0$, where $\xrightarrow{\text{fin}}$ indicates that all but finitely many locations are in the $\bot$ (unborn) state in their local STS. The transition relation $\rightsquigarrow$ for the product STS is just the pointwise lifting of the one for each node's STS: $s \rightsquigarrow s'$ iff $\forall \ell.\ s(\ell) \rightsquigarrow_0 s'(\ell)$. Thus, at the abstract level, the product STS is simply a collection of independent, local STSs.

At the concrete level of the invariant I, however, we record the constraints that tie one node's life story to another's; see Figure 2. The invariant is essentially a *relational* version of the recursive "list" predicate from separation logic [33]. The relational nature is apparent in the fact that the invariant makes assertions about *both* the implementation ($\mapsto_\mathrm{I}$) *and* specification ($\mapsto_\mathrm{S}$) heap, carving out the portion of each corresponding to an instance of MSQ and CGQ, respectively. It is thus a kind of linking invariant (or "refinement map") [22, 1], as one would expect to find in any refinement proof.

At a high level, the invariant says: the state of the product STS must decompose into exactly one sentinel node (at location ℓ_0), a collection of Live nodes s_L, and a collection of Dead nodes at locations $\overline{\ell_i}$ (the overbar notation represents a list). The link assertion recursively asserts that each Live node in the implementation corresponds to some node in the specification, and that a node is Live iff it is reachable from the sentinel. Since the queue is parametric over the type α of its data, the data stored in each live implementation node must *refine* the data stored in the specification node at type α (written $v_1 \preceq^{\mathcal{V}} v_2 : \alpha$; see Section 3). The invariant also accounts for two representation differences between MSQ and CGQ. First, the node data in the implementation is stored in a $\textbf{ref}_?(\alpha)$, while the specification stores the data directly. Second, the specification has a lock. The invariant requires that the lock is always free (**false**) because, as we show that MSQ refines CGQ, we always run entire critical sections of CGQ at once, going from unlocked state to unlocked state. These "big steps" of the CGQ correspond to the linearization points of the MSQ.

Finally—and crucially—Dead nodes must have non-**null** successor pointers whose locations are in a non-$\bot$ state. This property is the key for giving a simple, *local* loop invariant for enq, namely, that the current node c is *at least* in a Live state. It follows that if the successor pointer of c is not **null**, it must be another node at least in the Live state. If, on the other hand, the successor node of c *is* **null**, we know that c is both not Dead, and at least Live, which means that c must be (at that instant) reachable from the implementation's head pointer.

The proof outlines for MSQ and the other examples in this section are given in the appendix [35].

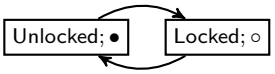
2.4 Role-playing and tokens

Although Michael and Scott's queue is already tricky to verify, there is a specific sense in which its protocol in Figure 2 is simple: it treats all threads equally. All threads see a level playing field with a single notion of "legal" transition, and any thread is free to make any legal transition according to the protocol. Many FCDs, however, require more refined protocols in which different threads can play different *roles*—granting them the rights to make different sets of transitions—and in which threads can acquire and release these roles dynamically as they execute.

In fact, one need not look to complex FCDs for instances of this dynamic role-playing—the simple lock used in the coarse-grained "spec" of the Michael-Scott queue is a perfect and canonical example. In a protocol governing a single lock (*e.g.,* lock, in CGQ), there are two states: Unlocked and Locked. Starting from the Unlocked state, all threads should be able to acquire the lock and transition to the Locked state. But not vice versa: once a thread has acquired the lock and moved to the Locked state, it has adopted the role of "lock-holder" and should know that *it* is the only thread with the right to release the lock and return to Unlocked.

To support this kind of role-playing, we enrich STSs with a notion of *tokens*, which are used to grant authority over certain types of actions in a protocol. Each STS may employ its own appropriately chosen set of tokens, and each thread may *privately own* some subset of these tokens. The idea, then, is that certain transitions are only legal for the thread that privately owns certain tokens. Formally speaking, this is achieved by associating with each state in the STS a set of tokens that are currently *free*, *i.e.,* not owned by any thread.[4] We then stipulate *the law of conservation of tokens*: for a thread to legally transition from state s to state s', the (disjoint) union of its private tokens and the free tokens must be the same in s and in s'.

For instance, in the locking protocol, there is just a single token—call it TheLock. In the Unlocked state, the STS asserts that TheLock must belong to the free tokens and thus that *no* thread owns it privately, whereas in the Locked state, the STS asserts that TheLock does *not* belong to the free tokens and thus that *some* thread owns it privately. Pictorially, • denotes that TheLock is in the free tokens, and ○ denotes that it is not:

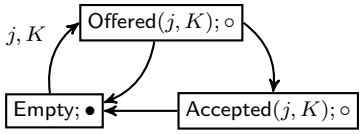


When a thread acquires the physical lock and transitions to the Locked state, it must add TheLock to its private tokens in order to satisfy conservation of tokens. Thereafter, no other thread may transition back to Unlocked because doing so requires putting TheLock back into the free tokens of the STS, which is something only the private owner of TheLock can do. Usually the invariant for the Unlocked state owns all of the hidden state for the data structure, while the Locked invariant owns nothing. Thus, a thread taking the lock also acquires the resources it protects, but must return these resources on lock release (in the style of CSL [31]).

As this simple example suggests, tokens induce very natural *thread-relative* notions of *rely* and *guarantee* relations on states of an STS. For any thread i, the total tokens A of an STS must equal the disjoint union of i's private tokens A_i, the free tokens A_{free} in the current state s, and the "frame" tokens A_{frame} (*i.e.,* the combined private tokens of all other threads but i). The *guarantee* relation says which future states thread i may transition to, namely those that are accessible by a series of transitions that i can "pay for" using its private tokens A_i. Dually, the *rely* relation says which future states other threads may transition to, namely those that are accessible by a series of transitions that can be paid for *without* using i's private tokens A_i (*i.e.,* only using the tokens in A_{frame}). These two relations play a central role in our model (Section 3).

[4] Another perspective is that the free tokens are owned by the STS itself, as opposed to the threads participating in the protocol; *cf.* CSL [31].

```
redFlag ≜ λ(). let flag = new true, chan = new 0 in {
  flip = rec try().
    if CAS(chan, 1, 2) then () else
    if CAS(flag, true, false) then () else
    if CAS(flag, false, true) then () else
    if CAS(chan, 0, 1) then
      if CAS(chan, 1, 0) then try() else chan := 0
    else try(),
  read = λ(). flag[1]
}
blueFlag ≜ λ(). let flag = new true, lock = new false in {
  flip = λ(). sync(lock) { flag := not flag[1] },
  read = λ(). sync(lock) { flag[1] }
}
```



$$
\begin{aligned}
Q &\triangleq \exists x : \mathbf{B}.\ \mathit{flag}_{\mathrm{I}} \mapsto_{\mathrm{I}} x * \mathit{flag}_{\mathrm{S}} \mapsto_{\mathrm{S}} x * \mathsf{lock} \mapsto_{\mathrm{S}} \mathbf{false} \\
I(\mathsf{Empty}) &\triangleq Q * \mathit{chan} \mapsto_{\mathrm{I}} 0 \\
I(\mathsf{Offered}(j,K)) &\triangleq Q * \mathit{chan} \mapsto_{\mathrm{I}} 1 * j \rightarrowtail_{\mathrm{S}} K[\mathsf{flip}_{\mathrm{S}}()] \\
I(\mathsf{Accepted}(j,K)) &\triangleq Q * \mathit{chan} \mapsto_{\mathrm{I}} 2 * j \rightarrowtail_{\mathrm{S}} K[()]
\end{aligned}
$$

Figure 3. Red flags versus blue flags

2.5 Cooperation and specifications-as-resources

As explained in the introduction, some FCDs use side channels, separate from the main data structure, to enable threads executing different operations to *cooperate*. To illustrate this, we use a toy example—inspired specifically by "elimination stacks" [19]—that isolates the essential challenge of reasoning about cooperation, minus the full-blown messiness of a real data structure.

Figure 3 shows the example, in which redFlag is a lock-free implementation of blueFlag. The latter is a very simple data structure, which maintains a hidden boolean *flag*, and provides operations to flip it and read it. One obvious lock-free implementation of flip would be to keep running CAS(*flag*, **true**, **false**) and CAS(*flag*, **false**, **true**) repeatedly until one of them succeeds. However, to demonstrate cooperation, redFlag does something more "clever": in addition to maintaining *flag*, it also maintains a side channel *chan*, which it uses to enable two flip operations to cancel each other out without ever modifying *flag* at all!

More specifically, *chan* adheres to the following protocol, which is visualized in Figure 3 (ignore the K's for now). If $\mathit{chan} \mapsto_{\mathrm{I}} 0$, it means the side channel is not currently being used (it is in the Empty state). If $\mathit{chan} \mapsto_{\mathrm{I}} 1$, it means that some thread j has offered to perform a flip using the side channel and moved it into the Offered$(j, -)$ state. If $\mathit{chan} \mapsto_{\mathrm{I}} 2$, it means that another thread has accepted thread j's offer and transitioned to Accepted$(j, -)$—thus silently performing both flip's at once (since they cancel out)—but that thread j has not yet acknowledged that its offer was accepted.

Like the locking example, this protocol uses a single token—call it Offer—which is free in state Empty but which thread j moves into its private tokens when it transitions to the Offered$(j, -)$ state. After that transition, due to its ownership of Offer, thread j is the only thread that has the right to revoke that offer by setting *chan* back to 0 and returning to Empty. On the other hand, *any* thread may transition from Offered$(j, -)$ to Accepted$(j, -)$, since the two states have identical free tokens, namely, none. Once in the

$\mathsf{Accepted}(j, -)$ state, though, thread j is again the only thread able to Empty the channel.

The implementation of flip in redFlag then works as follows. First, we use CAS to check if another thread has offered to flip (*i.e.,* if *chan* $\mapsto_{\mathrm{I}} 1$), and if so, we accept the offer by setting *chan* to 2. We then immediately return, having implicitly committed both flips right then and there, without ever accessing *flag*. If that fails, we give up temporarily on the side-channel shenanigans and instead try to perform a bona fide flip by doing $\mathsf{CAS}(\mathit{flag}, \mathbf{true}, \mathbf{false})$ and $\mathsf{CAS}(\mathit{flag}, \mathbf{false}, \mathbf{true})$ as suggested above. If *that* fails as well, then we attempt to make an offer on the side channel by changing *chan* from 0 to 1. If our attempt succeeds, then we (rather stupidly[5]) try to immediately revoke the offer and loop back to the beginning. If perchance another thread has preempted us at this point and accepted our offer (*i.e.,* if $\mathsf{CAS}(\mathit{chan}, 1, 0)$ fails, implying that another thread has updated *chan* to 2), then that other thread must have already committed our flip on our behalf, so we simply set *chan* back to 0, thus making the side channel free for other threads to use, and return. Finally, if all else fails, we loop again.

As far as the refinement proof is concerned, there are essentially two interesting points here. The first concerns the $\mathsf{CAS}(\mathit{chan}, 1, 0)$ step. As we observed already, the failure of this CAS implies that *chan* must be 2. Why? Because of the way our protocol uses tokens. After the previous $\mathsf{CAS}(\mathit{chan}, 0, 1)$ succeeded, we knew that we had successfully transitioned to the $\mathsf{Offered}(j, -)$ state, and thus that our thread j now controls the Offer token. Our ownership of Offer tells us that other threads can only transition to a limited set of states via the rely ordering (*i.e.,* without owning Offer): they can either leave the state where it is, or they can transition to $\mathsf{Accepted}(j, -)$. Thus, when we observe that *chan* is not 1, we know it *must* be 2.

The second, more interesting point concerns the *semantics* of cooperation. If we make an offer on *chan*, which is accepted by another thread, it should imply that the other thread performed our flip for us, so we don't have to. At least that's the intuition, but how is that intuition enforced by the protocol? That is, when we observe that our offer has been accepted, we do so merely by inspecting the current value of *chan*. But how do we know that the other thread that updated *chan* from 1 to 2 actually "performed our flip" for us? For example, as perverse as this sounds, what is to prevent redFlag from performing *chan* $:= 2$ as part of its implementation of read?

Our key to enforcing that the semantics of cooperation is respected is to treat *specification code* as a kind of resource. We introduce a new assertion, $j \rightarrowtail_{\mathrm{S}} e$, which describes the knowledge that thread j (on the spec side) is poised to run the term e. Ordinarily, this knowledge is kept private to thread j itself, but in a cooperative protocol, the whole idea is that j should be able to pass control over its spec code e to other threads, so that they may execute some steps of e on its behalf.

Specifically, this assertion is used to give semantic meaning to the $\mathsf{Offered}(j, K)$ and $\mathsf{Accepted}(j, K)$ states in our protocol (see the interpretation of F in Figure 3). In the former state, we know that $j \rightarrowtail_{\mathrm{S}} K[\mathsf{flip}_{\mathrm{S}}()]$, which tells us that thread j has offered its spec code $K[\mathsf{flip}_{\mathrm{S}}()]$ to be run by another thread, whereas in the latter state, we know that $j \rightarrowtail_{\mathrm{S}} K[()]$, which tells us that j's flip has been executed. (The K is present here only because we do not want to place any restrictions on the evaluation context of the flip operation.) These interpretations demand that whatever thread accepts the offer by transitioning from $\mathsf{Offered}(j, K)$ to $\mathsf{Accepted}(j, K)$ must take the responsibility not only of updating *chan* to 2 but also of executing $\mathsf{flip}_{\mathrm{S}}()$—and *only* $\mathsf{flip}_{\mathrm{S}}()$—on j's behalf. When j subsequently moves back to the Empty state, it regains private control over its specification code, so that other threads may no longer execute it.

[5] At this point in a real implementation, it would make sense to wait a while for other threads to accept our offer, but we elide that detail since it is irrelevant for reasoning about correctness.

$$
\begin{array}{rcl}
\mathsf{rand} & \triangleq & \lambda().\ \mathbf{let}\ y = \mathbf{new}\ \mathbf{false}\ \mathbf{in}\ (\mathbf{fork}\ y := \mathbf{true}); y[1] \\
\mathsf{lateChoice} & \triangleq & \lambda x.\ x := 0; \mathsf{rand}() \\
\mathsf{earlyChoice} & \triangleq & \lambda x.\ \mathbf{let}\ r = \mathsf{rand}()\ \mathbf{in}\ x := 0; r
\end{array}
$$

$\langle x_{\mathrm{I}} \preceq^{\mathcal{V}} x_{\mathrm{S}} : \mathbf{ref}(\mathbf{N}) \wedge j \rightarrowtail_{\mathrm{S}} K[\mathsf{earlyChoice}(x_{\mathrm{S}})] \rangle$
$x_{\mathrm{I}} := 0$
$\langle x_{\mathrm{I}} \preceq^{\mathcal{V}} x_{\mathrm{S}} : \mathbf{ref}(\mathbf{N}) \wedge (j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}]) \rangle$
$\mathsf{rand}()$
$\langle \mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} \vee \mathsf{ret} = \mathbf{false}) \wedge (j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}]) \rangle$
$\langle \mathsf{ret}.\ j \rightarrowtail_{\mathrm{S}} K[\mathsf{ret}] \rangle$

Figure 4. Late choice versus early choice

2.6 Nondeterminism and speculation

Another tricky aspect of reasoning about FCDs (like the "conditional CAS" example we consider in Section 4) is dealing with nondeterminism. The problem is that when proving that an FCD refines some coarse-grained spec, we want to reason in a temporally-local fashion—*i.e.,* using something akin to a *simulation* argument, by which the behavior of each step of FCD code is matched against zero or more steps of spec code—but nondeterminism, it would seem, foils this plan.

To see why, consider the "late choice/early choice" example in Figure 4. This example is really simple: it does not maintain any hidden state (hence no protocol), and is not in fact an FCD at all, but it nevertheless illustrates the core difficulty with nondeterminism. We want to show that lateChoice refines earlyChoice. Both functions flip a coin (*i.e.,* use rand() to nondeterministically choose a boolean value) and set a given variable x to 0, but they do so in opposite orders. Intuitively, though, the order shouldn't matter: there is no way to observe the coin flip until the functions return. However, if we try to reason about the refinement using a simulation argument, we run into a problem. The first step of lateChoice is the setting of x to 0. To simulate this step in earlyChoice, we need to match the assignment of x to 0 as well, since the update is an externally observable effect. But to do that we must first flip earlyChoice's coin. While we have the freedom to choose the outcome of the flip,[6] the trouble is that we don't know what the outcome should be: lateChoice's coin flip has yet to be executed.

The solution is simple: speculate! That is, if you don't know which spec states to step to in order to match an implementation step, then keep your options open and maintain a *speculative set of specification states* that are reachable from the initial spec state and consistent with any observable effects of the implementation step. In the case of lateChoice/earlyChoice, this means that we can simulate the first step of lateChoice (the setting of x to 0) by executing the *entire* earlyChoice function *twice*. In both speculative states x is set to 0, but in one the coin flip returns **true**, and in the other it returns **false**.

This reasoning is captured in the Hoare-style proof outline given in Figure 4. The precondition $j \rightarrowtail_{\mathrm{S}} K[\mathsf{earlyChoice}(x_{\mathrm{S}})]$—an instance of the assertions on specification code introduced in the previous section—denotes that initially the spec side is poised to execute earlyChoice. After we execute $x := 0$ in lateChoice, we speculate that the coin flip on the spec side could result in earlyChoice either returning **true** or returning **false**. This is represented by the speculative assertion $(j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}])$ appearing in the postcondition of this first step, in which the $\oplus$ operator provides a *speculative choice* between two subassertions characterizing possible spec states. In the subsequent step, lateChoice flips its coin,

[6] For *every* implementation execution, we must construct *some* specification execution.

yielding a return value ret of either **true** or **false**. We can then refine the speculative set of specification states to whichever one (either $j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}]$ or $j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}]$) matches ret, and simply drop the other state from consideration. In the end, what matters is that we are left with at least one spec state that has been produced by a sequence of steps matching the observable behavior of the implementation's steps.

The idea of speculation is not new: it is implicit in Lynch and Vaandrager's notion of *forward-backward simulation* [28]. What is new here is that we capture speculation assertionally without using prophecy variables, which allows us to *compose* speculations without fuss. For example, we can use speculative reasoning in a private, thread-local way (as in the choice example) while separately using it within the protocol governing some shared state (as in the CCAS example, Section 4). We give a more detailed comparison to related work in Section 5.

3. The formal model

In this section, we develop the syntax and semantics of a logic for concurrent programs in our higher-order, polymorphic language. We leave a proof theory for future work, and instead work "in the model" (semantically) when verifying examples.

3.1 Overview: proving refinement via hiding and protocols

Our logic is ultimately used to prove contextual refinements, so it is important to first make clear what that entails, both formally and practically. To define contextual refinement, we first introduce a standard typing judgment for contexts,

$$C : (\Omega, \tau) \rightsquigarrow (\Omega', \tau')$$

so that whenever $\Omega \vdash e : \tau$, we have $\Omega' \vdash C[e] : \tau'$. Then if $\Omega \vdash e_{\mathrm{I}} : \tau$ and $\Omega \vdash e_{\mathrm{S}} : \tau$, we say e_{I} *contextually refines* e_{S}, written $\Omega \models e_{\mathrm{I}} \preceq e_{\mathrm{S}} : \tau$, if:

$$\begin{array}{l}\text{for every } i, j \text{ and } C : (\Omega, \tau) \rightsquigarrow (\emptyset, \mathbf{N}) \text{ we have} \\ \quad \forall n. \forall T_{\mathrm{I}}. \quad \emptyset; [i \mapsto C[e_{\mathrm{I}}]] \rightarrow^* h_1; [i \mapsto n] \uplus T_{\mathrm{I}} \\ \implies \exists T_{\mathrm{S}}. \quad \emptyset; [j \mapsto C[e_{\mathrm{S}}]] \rightarrow^* h_2; [j \mapsto n] \uplus T_{\mathrm{S}}\end{array}$$

Refinement formalizes "observable behavior" as anything a client of an expression could "test." A context C captures the notion of an unknown (but well-typed!) client, which can interact arbitrarily with the two expressions. If e_{I} behaves in a way that its spec, e_{S}, does not, it should be possible to find a context C whose main thread returns a number with e_{I} that, with e_{S}, it does not.

The only practical way to *prove* refinement is to find some other, more structured way of characterizing observable behavior—one that does not require detailed reasoning about particular contexts. In a high-level language, FCDs hide their internal state within the functions they export (their "methods"), thereby greatly limiting the scope of interaction they have with their clients. In particular, a client context can neither observe nor alter the internal state directly; all interactions are mediated by the methods. From the perspective of an FCD, then, the behavior of a client can be reduced to a collection of possibly-concurrent method invocations. And from the perspective of a client, the behavior of an FCD can be reduced to the answers it returns from those invocations.

How can we prove that an arbitrary collection of concurrent method calls to an FCD will yield the same answers as its spec? Protocols are the key: they capture the effect methods can have on the internal state—and thereby the effect they have on each other—*abstractly*, *i.e.,* without reference to FCD code. Protocols enable us to reason locally, about one method invocation at a time. Instead of considering an arbitrary sequence of prior method invocations, we simply start from an arbitrary protocol state. And instead of considering arbitrary concurrent invocations, we simply force our local reasoning to withstand arbitrary "rely" moves in a protocol.

$$\begin{array}{rcl}
P & ::= & v = v \mid \mathsf{emp} \mid v \mapsto_{\mathrm{I}} u \mid v \mapsto_{\mathrm{S}} u \mid i \rightarrowtail_{\mathrm{S}} e \mid P * P \\
& \mid & P \Rightarrow P \mid P \wedge P \mid P \vee P \mid \exists x.P \mid \forall x.P \\
& \mid & P \oplus P \mid \varphi \mid \iota \mid \triangleright P \mid T@m\ \langle x.\ P \rangle \\
\varphi & ::= & \langle P \rangle\ e\ \langle x.\ Q \rangle \mid v \preceq^{\mathcal{V}} v : \tau \mid \Omega \vdash e \preceq^{\mathcal{E}} e : \tau \\
\iota & ::= & (\theta, I, s, A) \quad \text{where } \begin{cases} I \in \theta.S \to \text{Assert}, & s \in \theta.S, \\ A \subseteq \theta.A, & A \# \theta.F(s) \end{cases} \\
\theta & ::= & (S, A, \rightsquigarrow, F) \text{ where } S, A \text{ sets},\ \rightsquigarrow \subseteq S \times S,\ F \in S \to \wp(A) \\
m & ::= & i \mid \mathsf{none}
\end{array}$$

Figure 5. Syntax of assertions

3.2 Assertions

In program logics for first-order languages, there is a strict separation between assertions about *data* (*e.g.,* heap assertions) and assertions about *code* (*e.g.,* Hoare triples). But the distinction makes less sense for higher-order languages, where code *is* data and hence claims about data must include claims about code. Our logic is therefore built around a single notion of assertion, P, shown in Figure 5, that plays several disparate roles.

Assertions are best understood one role at a time. The first role they play is similar to that of heap assertions in separation logic: they capture knowledge about a part of the (implementation's) heap, *e.g.,* $x \mapsto_{\mathrm{I}} 0$, and support the composition of such knowledge, *e.g.,* $x \mapsto_{\mathrm{I}} 0 * y \mapsto_{\mathrm{I}} 1$. In this capacity, assertions make claims contingent on the current state, which may be invalidated in a later state.

On the other hand, some assertions are *pure*, meaning that if they hold in a given state, they will hold in any possible future state. The syntactic subclass of *code assertions* φ all have this property, and they include *Hoare triples* $\langle P \rangle\ e\ \langle x.\ Q \rangle$. The Hoare triple says: for any future state satisfying P, if the (implementation) expression e is executed until it terminates with a result, the final state will satisfy Q (where x is the value e returned). So, for example, $\langle \mathsf{emp} \rangle\ \mathbf{new}\ 0\ \langle x.\ x \mapsto_{\mathrm{I}} 0 \rangle$ is a *valid* assertion, *i.e.,* it holds in any state. More generally, the usual rules of separation logic apply, including the frame rule, the rule of consequence—and sequencing. The sequencing rule works, even in our concurrent setting, because heap assertions describe a portion of heap that is *privately owned* by the expression in the Hoare triple. In particular, that portion of the heap is guaranteed to be neither observed nor altered by threads concurrent with the expression.

The next role assertions play is expressing knowledge about *shared* resources. All shared resources are governed by a protocol. For hidden state, the protocol can be chosen freely, modulo proving that exported methods actually follow it. For visible state, however, *e.g.,* a reference that is returned directly to the context, the protocol is forced to be a trivial one—roughly, one that allows the state to take on any well-typed value at any time, accounting for the arbitrary interference an unknown context could cause (see Section 3.4).

Claims about shared resources are made through *island assertions* ι (inspired by LADR [10]), which first of all assert the *existence* of said resources. We call each shared collection of resources an *island*, because each collection is disjoint from the others and is governed by an independent protocol. An island assertion gives the protocol governing its resources:

- The component $\theta = (S, A, \rightsquigarrow, F)$ formalizes the STS for the protocol, where S is its set of states, A is its set of possible tokens, $\rightsquigarrow$ is its transition relation, and F is a function telling which tokens are free at each state. (We will use dot notation like $\theta.S$ to project named components from compound objects.)
- The component I tells how each state of the STS is *interpreted* as an assertion characterizing the concrete, hidden resources that are actually owned by the island in that state.

Domains

$$\begin{array}{lcl}
\text{StateSet} & \triangleq & \{\ \Sigma \subseteq \text{Heap} \times \text{ThreadPool} \ \mid\ \Sigma \text{ finite, nonempty}\ \} \\
\text{Resource} & \triangleq & \{\ \eta \in \text{Heap} \times \text{StateSet}\ \} \\
\text{Island}_n & \triangleq & \left\{ (\theta, J, s, A) \;\middle|\; \begin{array}{l} \theta \in \text{STS},\ s \in \theta.S,\ A \subseteq \theta.A,\ A \# \theta.F(s), \\ J \in \theta.S \to \text{UWorld}_n \overset{\text{mon}}{\to} \wp(\text{Resource}) \end{array} \right\} \\
\text{World}_n & \triangleq & \left\{ W = (k, \omega) \;\middle|\; k < n,\ \omega \in \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \text{Island}_k \right\} \\
\text{UWorld}_n & \triangleq & \{\ U \in \text{World}_n \ \mid\ U = |U|\ \} \\
\text{VRel}_n & \triangleq & \left\{ V \in \text{UWorld}_n \overset{\text{mon}}{\to} \wp(\text{Val} \times \text{Val}) \right\}
\end{array}$$

Island and world operations

$$\begin{array}{lcl}
|(\theta, J, s, A)| & \triangleq & (\theta, J, s, \emptyset) \qquad |(k, \omega)| \triangleq (k, \lambda i.|\omega(i)|) \\
\text{frame}(\theta, J, s, A) & \triangleq & (\theta,\ J,\ s,\ \theta.A - \theta.F(s) - A) \\
\text{frame}(k, \omega) & \triangleq & (k,\ \lambda i.\text{frame}(\omega(i))) \\
\lfloor (\theta, J, s_0, A) \rfloor_k & \triangleq & (\theta, \lambda s.I(s) \upharpoonright \text{UWorld}_k, s_0, A) \\
\triangleright(k+1, \omega) & \triangleq & (k, \lambda i.\lfloor \omega(i) \rfloor_k) \\
\text{interp}(\theta, J, s, A) & \triangleq & J(s)
\end{array}$$

Composition

$$\begin{array}{lrcll}
\text{State sets} & \Sigma_1 \otimes \Sigma_2 & \triangleq & \{\ h_1 \uplus h_2; T_1 \uplus T_2 \ \mid\ h_i; T_i \in \Sigma_i\ \} & \text{when } \textit{all} \text{ compositions are defined} \\
\text{Resources} & (h_1, \Sigma_1) \otimes (h_2, \Sigma_2) & \triangleq & (h_1 \uplus h_2, \Sigma_1 \otimes \Sigma_2) & \\
\text{Islands} & (\theta, J, s, A) \otimes (\theta', J', s', A') & \triangleq & (\theta, J, s, A \uplus A') & \text{when } \theta = \theta',\ s = s',\ J = J' \\
\text{Worlds} & (k, \omega) \otimes (k', \omega') & \triangleq & (k, \lambda i.\omega(i) \otimes \omega'(i)) & \text{when } k = k',\ \text{dom}(\omega) = \text{dom}(\omega')
\end{array}$$

Protocol conformance

$$\theta \vdash (s, A) \rightsquigarrow (s', A') \triangleq s \rightsquigarrow_\theta s',\ \theta.F(s) \uplus A = \theta.F(s') \uplus A' \qquad (k, \omega) \overset{\text{guar}}{\sqsubseteq} (k', \omega') \triangleq k \geq k',\ \forall i \in \text{dom}(\omega).\ \lfloor \omega(i) \rfloor_{k'} \overset{\text{guar}}{\sqsubseteq} \omega'(i)$$

$$(\theta, J, s, A) \overset{\text{guar}}{\sqsubseteq} (\theta', J', s', A') \triangleq \theta = \theta',\ J = J',\ \theta \vdash (s, A) \rightsquigarrow^* (s', A') \qquad W \overset{\text{rely}}{\sqsubseteq} W' \triangleq \text{frame}(W) \overset{\text{guar}}{\sqsubseteq} \text{frame}(W')$$

World satisfaction: $\eta : W, \eta' \triangleq W.k > 0 \implies \eta = \eta' \otimes \overline{\eta_i},\ \forall i \in \text{dom}(W.\omega).\ \eta_i \in \text{interp}(W.\omega(i))(\triangleright |W|)$

Figure 6. Semantic structures and operations on them

In addition, island assertions express knowledge about the state of the protocol (the component s) and any privately-owned tokens (the component A). This last bit of knowledge gives a *lower bound* on the actual state of the protocol, which may in fact by in any "rely-future" state of s, *i.e.,* any state that can be reached from s by the environment without using the privately-owned tokens A.

Finally, assertions play two refinement-related roles. The first is to express refinement itself, either between two closed values ($v_\text{I} \preceq^{\mathcal{V}} v_\text{S} : \tau$) or between open expressions ($\Omega \vdash e_\text{I} \preceq^{\mathcal{E}} e_\text{S} : \tau$)—the syntactic counterpart to semantic refinement (Section 3.1). Until this point, we have avoided saying anything about spec terms, but in order to prove refinement we need to show that the observable behavior of an implementation can be mimicked by its spec. This brings us to an essential idea:

$$\begin{array}{l} e_\text{I} \preceq^{\mathcal{E}} e_\text{S} : \tau \quad \approx \quad \text{(roughly!)} \\ \quad \forall j.\ \langle j \rightarrowtail_\text{S} e_\text{S} \rangle\ e_\text{I}\ \langle x_\text{I}.\ \exists x_\text{S}.\ x_\text{I} \preceq^{\mathcal{V}} x_\text{S} : \tau \wedge j \rightarrowtail_\text{S} x_\text{S} \rangle \end{array}$$

By treating spec code as a resource, we can reduce refinement reasoning to Hoare-style reasoning. Thus, the final role assertions play is to express knowledge about—and ownership of—spec resources, which include portions both of the heap (*e.g.,* $x \mapsto_\text{S} 0$) and of the threadpool (*e.g.,* $j \rightarrowtail_\text{S} e_\text{S}$). These resources can be shared and hence governed by protocols, just as implementation-side resources can.

When proving refinement for an FCD, we will prove something like the above Hoare triple for an arbitrary application of each of its methods—usually in the scope of an island assertion giving the protocol for its shared, hidden state. For each method invocation, we start from an arbitrary state of that protocol, and are *given ownership of* the spec code corresponding to the invocation, which we may choose to transfer to the protocol to support cooperation (as explained in Section 2.5). But in the end, when the implementation's invocation has finished and returned a value x_I, we must have regained exclusive control over its spec, which must have mimicked it by producing a value x_S that x_I refines.

The remaining forms of assertions include standard logical connectives, and two more technical forms of assertions—$\triangleright P$ and $T@m\ \langle x.\ P \rangle$—which we explain in the Section 3.4.

3.3 Semantic structures

The semantics of assertions is given using two judgments, one for general assertions ($W, \eta \models^\rho P$) and the other for code assertions ($U \models^\rho \varphi$), where P and φ contain no free term variables but may contain free type variables bound by ρ. To explain these judgments, we begin with the semantic structures of *worlds* W, *resources* η and *environments* ρ, together with operations on them needed to interpret assertions—all defined in Figure 6.

Resources The *resources* $\eta = (h, \Sigma)$ that assertions claim knowledge about and ownership of include both implementation heaps h, and speculative sets Σ of spec configurations. (Recall that a *configuration* $\varsigma = h; T$ consists of a heap and a threadpool.) Resources can be combined at every level, which is necessary for interpreting the $*$ operator on assertions. For heaps and threadpools, composition is done via $\uplus$, the usual disjoint union. The composition of state sets is just the set of state compositions—but it is only defined when *all* such state compositions are defined, so that speculative sets Σ have a single "footprint" consisting of all the locations/threads existing in *any* speculative state. To ensure that this footprint is finite, we require that speculation is itself finite. Finally, composition of resources is the composition of their parts.

Islands and possible worlds All assertions are interpreted in the context of some *possible world* W, which contains a collection ω of islands—this is the "Kripke" in Kripke logical relations. Semantic islands look very much like syntactic island assertions: the only difference is that STS states are interpreted semantically via J, rather than syntactically via I. Unfortunately, this creates a circularity: J is meant to interpret its syntactic counterpart I, and since assertions are interpreted in the contexts of worlds, the interpretation must be relative to the current world—but we are in the middle of *defining* worlds! The step index k in worlds is used to stratify away circularities in the definition of worlds and the logical relation; it and its attendant operators $\triangleright$ and $\lfloor - \rfloor_k$ are completely standard, and so for space reasons we direct the interested reader to earlier work for a detailed explanation [9, 10]. The less interested reader should simply ignore step indices from here on.

There is one additional subtlety, however: it is crucial that all participants in a protocol agree on the protocol's interpretation of a state, which must therefore be insensitive to which tokens a particular participant owns. We guarantee this by giving the interpretation J access to only the *unprivileged* part of a participant's world, $|W|$, which has been stripped of any tokens; see the constraint on the type of J.

To determine the meaning of assertions like $\iota * \iota'$, we must allow islands to be composed. Semantic island composition $\otimes$ is defined only when the islands agree on all aspects of the protocol, including its state; their owned tokens are then (disjointly) combined. Note, however, that because island *assertions* are rely-closed, an assertion like $\iota * \iota'$ does *not* require ι and ι' to assert the same state. It merely requires that there is some state that is in both of their rely-futures. Worlds are composable only when they define the same islands and those islands are composable.

Environments Type variables are interpreted by an *environment* ρ that maps them to relations $V \in \text{VRel}$. This interpretation of types captures the usual relational parametricity [32], in which the interpretation of an abstract type may relate values of potentially different types on the the implementation and specification sides.

Protocol conformance The judgment $\theta \vdash (s, A) \rightsquigarrow (s', A')$ codifies the law of conservation of tokens (Section 2.4) for a single step.[7] We use this judgment in defining a *guarantee* relation governing the changes an expression can make to an island's state, given the tokens it owns. An expression can likewise *rely* on its environment to only change the island ι according to the tokens that the environment owns, *i.e.,* the tokens owned by $\text{frame}(\iota)$.

The rely and guarantee views of a protocol give rise to *two* notions of future worlds. In both cases, the world may grow to include new islands, but any existing islands are constrained by their rely and guarantee relations, respectively. The guarantee relation on islands may include changes to both the state of the STS and the privately-owned tokens. The rely relation only allows the state to change, since only the (implicit) environmental participant making the move may gain or lose tokens. Island interpretations J are required to be monotone with respect to the rely relation on worlds (written $\overset{\text{mon}}{\to}$), which ensures that making a rely move in one island cannot possibly invalidate the interpretation of another.

World satisfaction Worlds describe shared state abstractly, in terms of protocol states. Expressions, on the other hand, are executed against some *concrete* resources. The *world satisfaction relation* $\eta : W, \eta'$ defines when a given collection of concrete resources η "satisfies" a world, meaning that it breaks into a disjoint portion for each island, with each portion satisfying its island's current interpretation. The parameter η' represents additional resources that are *private*, and therefore disjoint from those governed by the world, which is convenient for defining the semantics of assertions below.

3.4 Semantics

The semantics of assertions given in Figure 7 satisfies a fundamental property: if $W, \eta \models^\rho P$ and $W \overset{\text{rely}}{\sqsubseteq} W'$ then $W', \eta \models^\rho P$. All assertions are therefore "stable" under arbitrary interference from other threads. This should not be a surprise: assertions are either statements about private resources (for which interference is impossible) or about shared islands (for which interference is assumed, *e.g.,* we are careful to only assert *lower bounds* on the state of an island). The only subtlety is in the semantics of implication, which must be explicitly rely-closed to ensure stability. The semantics of the basic assertions about private resources and islands are entirely straightforward, as are those for the basic logical connectives (we omit $=$, $\vee$ and $\exists$).

The value refinement assertion $v_1 \preceq^{\mathcal{V}} v_2 : \tau$ requires that any observations a context can make of v_1 at type τ can also be made of v_2. Those readers familiar with Kripke logical relations will recognize it as essentially the standard definition of logical approximation between values. Base type values must be identical. For function types, we check that the bodies of the functions are related when given related arguments, which, due to the semantics of implication, might happen in a rely-future world. For recursive types, we check that the values are related at the unfolded type, which is well-founded due to the productivity requirement on recursive types (and our strategic uses of $\triangleright$ in type constructors). Values that are exposed to the context at heap-allocated type—**ref** and sum types—are forced to be governed by a trivial island allowing *all* type-safe updates (in the case of **ref**'s) and *no* updates (in the case of sums). Hidden state, on the other hand, is by definition state that does not escape directly to the context, and so we need say nothing about it for value refinement.

The fact that refinement is a pure assertion (insensitive to the state of private resources or the ownership of private tokens) is essential for soundness, for a simple reason: once a value has reached the context, it can be copied and used concurrently. We therefore cannot claim that any *one* copy of the value privately owns some resources. Note that, if P is impure, we use $U \models^\rho P$ as shorthand for $\forall \eta.\, U, \eta \models^\rho P$.

For expression refinement $\Omega \vdash e_1 \preceq^{\mathcal{E}} e_2 : \tau$, we first close off any term or type variables bound by Ω with the appropriate universal quantification. Closed expression refinement is defined in terms of a Hoare triple, almost in the way we suggested in Section 3.2. The main difference in the actual definition is that we additionally quantify over the unknown evaluation context K in which a specification is running; this annoyance appears to be necessary for proving that refinement is a precongruence.

Hoare triples are defined via the *threadpool simulation* assertion $T@m\ \langle x.\ P\rangle$, which is the engine that powers our model. Threadpool simulation accounts for the fact that an expression can fork threads as it executes, but that we care about the return value only from the initial thread, m. To satisfy $T@m\ \langle x.\ P\rangle$ at some W and η, the threads in T must first of all continuously obey the protocols of W, given private ownership of η. That is, every atomic step taken by a thread must transform its shared resources in a way that corresponds to a guarantee move in the protocol, and it must preserve as a frame any private resources of its environment, but it may change private resources η in any way it likes. In between each such atomic step, the context might get a chance to run, which we model by quantifying over an arbitrary rely-future world. If at any point the main thread m terminates, it must do so in a state satisfying P, where x is bound to the value the main thread returned. Afterward, any lingering threads are still required to obey the protocol.

That threadpool simulation is, in fact, a simulation is due to its use of the *speculative stepping relation* $\Sigma \rightrightarrows \Sigma'$, which requires any changes to the spec state to represent feasible execution steps: every new state must be reachable from some old state, but we are free to introduce multiple new states originating in the same old state, and we are free to drop irrelevant old states on the floor. As a result of how simulation is defined, such changes to the spec state can only be made to those pieces that are under the threadpool's control, either as part of its private resources (allowing arbitrary feasible updates) or its shared ones (allowing only protocol-permitted updates).

3.5 Soundness for refinement

Our key theorem is:

Theorem 1 (Soundness)**.** If $U \models^\emptyset \Omega \vdash e_1 \preceq^{\mathcal{E}} e_2 : \tau$ for all U then $\Omega \models e_1 \preceq e_2 : \tau$.

[7] We use the more readable notation $s \rightsquigarrow_\theta s'$ in place of $\theta.\rightsquigarrow (s, s')$.

Private and shared-state assertions: $W, \eta \models^\rho R$ iff

R	Requirements
φ	$\lvert W\rvert \models^\rho \varphi$
emp	$W = \lvert W\rvert, \eta = (\emptyset, \{\emptyset; \emptyset\})$
$v \mapsto_{\mathrm{I}} u$	$\eta = ([v \mapsto u], \{\emptyset; \emptyset\})$
$v \mapsto_{\mathrm{S}} u$	$\eta = (\emptyset, \{[v \mapsto u]; \emptyset\})$
$i \rightarrowtail_{\mathrm{S}} e$	$\eta = (\emptyset, \{\emptyset; [i \mapsto e]\})$
$P \wedge Q$	$W, \eta \models^\rho P$ and $W, \eta \models^\rho Q$
$P \Rightarrow Q$	$\forall W' \overset{\text{rely}}{\sqsupseteq} W.\ W', \eta \models^\rho P \implies W', \eta \models^\rho Q$
$\forall x.P$	$\forall v.\ W, \eta \models^\rho P[v/x]$
$\triangleright P$	$W.k > 0 \implies \triangleright W, \eta \models^\rho P$
$P_1 * P_2$	$W = W_1 \otimes W_2,\ \eta = \eta_1 \otimes \eta_2,\ W_i, \eta_i \models^\rho P_i$
$P_1 \oplus P_2$	$\eta.\Sigma = \Sigma_1 \cup \Sigma_2,\ W, (\eta.h, \Sigma_i) \models^\rho P_i$
(θ, I, s, A)	$\exists i.\ W \overset{\text{rely}}{\sqsupseteq} (W.k, [i \mapsto (\theta, [\![I]\!], s, A)])$ where $[\![I]\!] \triangleq \lambda s.\lambda U.\{\eta \mid U, \eta \models^\rho I(s)\}$

Value refinement: $U \models^\rho v_1 \preceq^{\mathcal{V}} v_2 : \tau_0$ iff

τ_0	Requirements
τ_b	$v_1 = v_2,\ \vdash v_i : \tau_b$ for $\tau_b \in \{\mathbf{1}, \mathbf{B}, \mathbf{N}\}$
α	$(v_1, v_2) \in \rho(\alpha)(U)$
$\tau \to \tau'$	$v_i = \mathbf{rec}\ f(x).e_i, U \models^\rho \triangleright(x : \tau \vdash e_1[v_1/f] \preceq^{\mathcal{E}} e_2[v_2/f] : \tau')$
$\forall\alpha.\tau$	$v_i = \Lambda.e_i, \qquad U \models^\rho \triangleright(\alpha \vdash e_1 \preceq^{\mathcal{E}} e_2 : \tau)$
$\mu\alpha.\tau$	$U \models^\rho v_1 \preceq^{\mathcal{V}} v_2 : \tau[\mu\alpha.\tau/\alpha]$
$\mathbf{ref}_?(\overline{\tau})$	$U \models^\rho v_1 \preceq^{\mathcal{V}} v_2 : \mathbf{1} \vee v_1 \preceq^{\mathcal{V}} v_2 : \mathbf{ref}(\overline{\tau})$
$\mathbf{ref}(\overline{\tau})$	$U \models^\rho \mathrm{inv}(\exists\overline{x}, \overline{y}.\ \bigwedge \overline{x \preceq^{\mathcal{V}} y : \tau} \wedge v_1 \mapsto_{\mathrm{I}} (\overline{x}) * v_2 \mapsto_{\mathrm{S}} (\overline{y}))$
$\tau_1 + \tau_2$	$\exists i.\ U \models^\rho \exists x, y.\ x \preceq^{\mathcal{V}} y : \tau_i \wedge \mathrm{inv}(v_1 \mapsto_{\mathrm{I}} \mathbf{inj}_i\ x * v_2 \mapsto_{\mathrm{S}} \mathbf{inj}_i\ y)$

Expression refinement: $U \models^\rho \Omega \vdash e_1 \preceq^{\mathcal{E}} e_2 : \tau$ iff

Ω	Requirements
$\cdot$	$\forall K, j.\ U \models^\rho \langle j \rightarrowtail_{\mathrm{S}} K[e_2]\rangle\ e_1\ \langle x.\ \exists y.\ x \preceq^{\mathcal{V}} y : \tau \wedge j \rightarrowtail_{\mathrm{S}} K[y]\rangle$
$x{:}\tau', \Omega'$	$\forall v_1, v_2.\ U \models^\rho v_1 \preceq^{\mathcal{V}} v_2 : \tau' \Rightarrow \Omega' \vdash e_1[v_1/x] \preceq^{\mathcal{E}} e_2[v_2/x] : \tau$
α, Ω'	$\forall V.\ U \models^{\rho[\alpha \mapsto V]} \Omega' \vdash e_1 \preceq^{\mathcal{E}} e_2 : \tau$

Hoare triples:
$U \models^\rho \langle P\rangle\ e\ \langle x.\ Q\rangle \triangleq \forall i.\ U \models^\rho P \Rightarrow [i \mapsto e]@i\ \langle x.\ Q\rangle$

Invariant protocols: $\mathrm{inv}(P) \triangleq (\{\{1\}, \emptyset, \emptyset, \lambda\_.\emptyset), \lambda\_.P, 1, \emptyset)$

Spec stepping: $\Sigma \rightrightarrows \Sigma' \triangleq \forall \varsigma' \in \Sigma'.\ \exists \varsigma \in \Sigma.\ \varsigma \to^* \varsigma'$

Threadpool simulation:

$W_0, \eta \models^\rho T@m\ \langle x.\ Q\rangle \triangleq \forall W \overset{\text{rely}}{\sqsupseteq} W_0,\ \eta_F \# \eta.$ if $W.k > 0$ and $h, \Sigma : W, \eta \otimes \eta_F$ then:

- if $h; T \to h'; T'$ then $\exists \Sigma', \eta', W' \overset{\text{guar}}{\sqsupseteq} W.\ \ \Sigma \rightrightarrows \Sigma',\ \ h', \Sigma' : W', \eta' \otimes \eta_F,\ \ W'.k = W.k - 1,\ \ W', \eta' \models^\rho T'@m\ \langle x.\ Q\rangle$
- if $T = T_0 \uplus [m \mapsto v]$ then $\exists \Sigma', \eta', W' \overset{\text{guar}}{\sqsupseteq} W.\ \ \Sigma \rightrightarrows \Sigma',\ \ h, \Sigma' : W', \eta' \otimes \eta_F,\ \ W'.k = W.k,\ \ W', \eta' \models^\rho Q[v/x] * T_0@\mathsf{none}\ \langle x.\ \mathsf{tt}\rangle$

Figure 7. The semantics of assertions

The proof of this theorem, given in the appendix [35], is built on novel lemmas expressing key framing properties for the threadpool simulation assertion, the most important being:

Lemma 1 (Framing)**.** If $W_1, \eta_1 \models^\rho T_1@m_1\ \langle x.\ Q_1\rangle$ and $W_2, \eta_2 \models^\rho T_2@m_2\ \langle x.\ Q_2\rangle$ with $m_1 = \mathsf{none}$ or $m_1 \in \mathrm{dom}(T_1)$, then $W_1 \otimes W_2, \eta_1 \otimes \eta_2 \models^\rho T_1 \uplus T_2@m_1\ \langle x.\ Q_1\rangle$.

These lemmas allow us to prove that refinement assertions are *congruent* (*i.e.,* they compose), which is the difficult part of soundness. We can also derive the following inference rules for *valid*, pure assertions (true at every world):

$$\frac{\langle P\rangle\ e\ \langle x.\ P'\rangle \quad \forall x.\langle P'\rangle\ e'\ \langle y.\ P''\rangle}{\langle P\rangle\ \mathbf{let}\ x = e\ \mathbf{in}\ e'\ \langle y.\ P''\rangle} \qquad \frac{\langle P\rangle\ e\ \langle x.\ Q\rangle}{\langle P * R\rangle\ e\ \langle x.\ Q * R\rangle}$$

$$\frac{P \Rightarrow P' \quad \langle P'\rangle\ e\ \langle x.\ Q'\rangle \quad Q' \Rightarrow Q}{\langle P\rangle\ e\ \langle x.\ Q\rangle} \qquad \frac{\triangleright P \Rightarrow P}{P}$$

The first three of these inference rules are the expected ones for a separation logic (our assertions also model intuitionistic BI). The last rule is the Löb rule, which allows us to reason about recursive functions by assuming their specification holds one step later [10]. In giving proof outlines in the next section, we make implicit use of these rules, and also drop the variable binding in the postcondition when it is irrelevant.

4. Case study: conditional CAS

With the details of our model in hand, we are now in a position to tackle, in detail, a rather complex FCD: Harris *et al.*'s *conditional CAS* [18, 15], which performs a compare-and-set on one word of memory, but only succeeds when some other word (the *control flag*) is non-zero at the same instant. This data structure is the workhorse that enables Harris *et al.* to build their remarkable lock-free multi-word CAS from single-word CAS.

As with the Michael-Scott queue, we have boiled down conditional CAS to its essence, retaining its key verification challenges while removing extraneous detail. Thus, we study lock-free conditional *increment* on a counter, with a fixed control flag per instance of the counter; see the specification $\mathsf{counter_S}$ in Figure 8. These simplifications eliminate the need to track administrative information about the operation we are trying to perform but do not change the algorithm itself, so adapting our proof of conditional increment to full CCAS is a straightforward exercise.

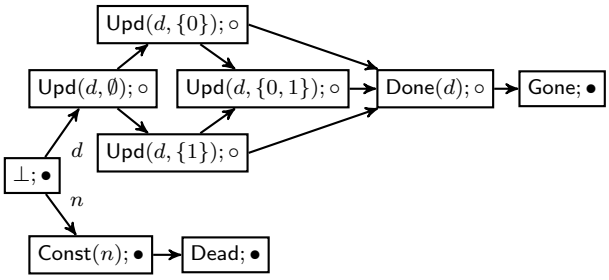
4.1 The protocol

To explain our implementation, $\mathsf{counter_I}$, we begin with its representation and the protocol that governs it. The control flag f is represented using a simple boolean reference; all of the action is in the counter c, which has type $\mathbf{ref}(\mathbf{N} + \mathbf{N})$. A value $\mathbf{inj}_1\ n$ represents an "inactive" counter with logical value n. A value $\mathbf{inj}_2\ n$, in contrast, means that the counter is undergoing a conditional increment, and had the logical value n when the increment began. Because $\mathbf{inj}_2\ n$ records the original value, a concurrent thread attempting another operation on the data structure can help finish the in-progress increment. This helping is actually not so selfless: really, one thread is just "helping" another thread get out of its way.

The question is how to perform a conditional increment without using any locks. Remarkably, the algorithm simply reads the flag f, and then—in a separate step—updates the counter c with a CAS; see the complete function. It is possible, therefore, for one thread performing a conditional increment to read f as **true**, at which point another thread sets f to **false**; the original thread then proceeds with incrementing the counter, even though the control flag is **false**! Proving that $\mathsf{counter_I}$ refines $\mathsf{counter_S}$ despite this blatant race

$\mathsf{counter_S} \triangleq$
  let $c =$ **new** $0, f =$ **new false**, lock $=$ **new false**
  let setFlag$(b) =$ sync(lock) $\{\ f := b\ \}$
  let get$() \quad =$ sync(lock) $\{\ c[1]\ \}$
  let cinc$() \quad =$ sync(lock) $\{\ c[1] := c[1] +$ **if** $f[1]$ **then** 1 **else** $0\ \}$
  in (get, setFlag, cinc)

$\mathsf{counter_I} \triangleq$
  let $c =$ **new inj**$_1\ 0, f =$ **new false**
  let setFlag$(b) = f := b$
  let complete$(x, n) =$ **if** $f[1]$ **then** CAS$(c, x,$ **inj**$_1\ (n+1))$
                                else CAS$(c, x,$ **inj**$_1\ n)$
  let rec get$() =$ **let** $x = c[1]$ **in case** x **of**
    inj$_1\ n \Rightarrow n$
  | **inj**$_2\ n \Rightarrow$ complete(o, n); get$()$
  let rec cinc$() =$ **let** $x = c[1]$ **in case** x **of**
    inj$_1\ n \Rightarrow$ **let** $y =$ **inj**$_2\ n$ **in**
      if CAS(c, x, y) **then** complete(y, n); $()$ **else** cinc$()$
  | **inj**$_2\ n \Rightarrow$ complete(x, n); cinc$()$
  in (get, setFlag, cinc)



$d ::= n, j, K \qquad B \subseteq \{0,1\} \qquad A \triangleq \mathsf{Loc} \qquad S \triangleq \mathsf{Loc} \xrightarrow{\text{fin}} S_0$

$S_0 \triangleq \{\bot, \mathsf{Upd}(d, B), \mathsf{Done}(d), \mathsf{Gone}, \mathsf{Const}(n), \mathsf{Dead}\}$

$$I(s) \triangleq \exists b : \mathbf{B}.\ f_\mathrm{I} \mapsto_\mathrm{I} b * f_\mathrm{S} \mapsto_\mathrm{S} b * \mathsf{lock} \mapsto_\mathrm{S} \textbf{false}$$
$$* \ \exists! \ell_c.\ s(\ell) \in \{\mathsf{Const}(-), \mathsf{Upd}(-,-)\}$$
$$* \begin{cases} \mathsf{linkUpd}(\ell_c, n, j, K, B) & s(\ell_c) = \mathsf{Upd}(n, j, K, B) \\ \mathsf{linkConst}(\ell_c, n) & s(\ell_c) = \mathsf{Const}(n) \end{cases}$$
$$* \ \mathop{\Large *}_{s(\ell)=\mathsf{Done}(n,j,K)} \ell \mapsto_\mathrm{I} \textbf{inj}_2\ n * j \rightarrowtail_\mathrm{S} K[()]$$
$$* \ \mathop{\Large *}_{s(\ell)=\mathsf{Gone}} \ell \mapsto_\mathrm{I} \textbf{inj}_2\ - \quad * \quad \mathop{\Large *}_{s(\ell)=\mathsf{Dead}} \ell \mapsto_\mathrm{I} \textbf{inj}_1\ -$$

$$\mathsf{linkConst}(\ell_c, n) \triangleq c_\mathrm{I} \mapsto_\mathrm{I} \ell_c * \ell_c \mapsto_\mathrm{I} \textbf{inj}_1\ n * c_\mathrm{S} \mapsto_\mathrm{S} n$$

$$\mathsf{linkUpd}(\ell_c, n, j, K, B) \triangleq c_\mathrm{I} \mapsto_\mathrm{I} \ell_c * \ell_c \mapsto_\mathrm{I} \textbf{inj}_2\ n$$
$$* \begin{pmatrix} c_\mathrm{S} \mapsto_\mathrm{S} n * j \rightarrowtail_\mathrm{S} K[\mathsf{cinc}()] \\ \oplus\ c_\mathrm{S} \mapsto_\mathrm{S} n * j \rightarrowtail_\mathrm{S} K[()] & \text{if } 0 \in B \\ \oplus\ c_\mathrm{S} \mapsto_\mathrm{S} (n+1) * j \rightarrowtail_\mathrm{S} K[()] & \text{if } 1 \in B \end{pmatrix}$$

Figure 8. Conditional increment, a simplification of CCAS

condition will require all the features of our model, working in concert.

An initial idea is that when the *physical* value of the counter is **inj**$_2\ n$, its *logical* value is ambiguous: it is either n or $n+1$. This idea will only work if we can associate such logical values with feasible executions of the spec's cinc code, since "logical" value really means the spec's value. The difficulty is in choosing *when* to take spec steps. If we wait to execute the spec code until a successful CAS in complete, we may be too late: as the interleaving above shows, the flag may have changed by then. But we cannot execute the spec when we read the flag, either: the CAS that follows it may fail, in which case some *other* thread must have executed the spec.

The way out of this conundrum is for threads to interact via a speculative protocol, shown in Figure 8. Recall that injections into sum types are heap-allocated, so every value c takes on has

let complete$(x, n) =$
  if $f[1]$ **then**
    CAS$(c, x,$ **inj**$_1\ (n+1))$
  else
    CAS$(c, x,$ **inj**$_1\ n)$

$\langle x \propto \mathsf{Upd}(n, j, K, \emptyset)\rangle$
$\langle x \propto \mathsf{Upd}(n, j, K, \{1\})\rangle$
$\langle x \propto \mathsf{Done}(n, j, K)\rangle$
$\langle x \propto \mathsf{Upd}(n, j, K, \{0\})\rangle$
$\langle x \propto \mathsf{Done}(n, j, K)\rangle$

let rec cinc$() =$
  let $x = c[1]$ **in**
  case x **of**
    inj$_1\ n \Rightarrow$
      let $y =$ **inj**$_2\ n$ **in**
      if CAS(c, x, y) **then**
                $\therefore$
        complete(y, n);
        $()$
                $\therefore$
      else
        cinc$()$
    | **inj**$_2\ n \Rightarrow$
      complete(x, n);
                $\therefore$
      cinc$()$

$\langle j \rightarrowtail_\mathrm{S} K[\mathsf{cinc_S}()] * (\theta, I, \emptyset, \emptyset)\rangle$
$\left\langle j \rightarrowtail_\mathrm{S} K[\mathsf{cinc_S}()] * \left(x \propto \mathsf{Const}(-) \vee x \propto \mathsf{Upd}(-,-)\right)\right\rangle$
$\langle j \rightarrowtail_\mathrm{S} K[\mathsf{cinc_S}()] * x \propto \mathsf{Const}(n)\rangle$
$\langle j \rightarrowtail_\mathrm{S} K[\mathsf{cinc_S}()] * x \propto \mathsf{Const}(n) * y \mapsto \textbf{inj}_2\ n\rangle$
$\langle x \propto \mathsf{Dead}(n) \wedge y \propto_\bullet \mathsf{Upd}(n, j, K, \emptyset)\rangle$
$\langle y \propto_\bullet \mathsf{Upd}(n, j, K, \emptyset)\rangle$
$\langle y \propto_\bullet \mathsf{Done}(n, j, K)\rangle$
$\langle \mathsf{ret.\ ret} = () \wedge j \rightarrowtail_\mathrm{S} K[()] \wedge y \propto \mathsf{Gone}\rangle$
$\langle \mathsf{ret.\ ret} = () \wedge j \rightarrowtail_\mathrm{S} K[()]\rangle$
$\langle j \rightarrowtail_\mathrm{S} K[\mathsf{cinc_S}()] * (\theta, I, \emptyset, \emptyset)\rangle$
$\langle \mathsf{ret.\ ret} = () \wedge j \rightarrowtail_\mathrm{S} K[()]\rangle$
$\langle j \rightarrowtail_\mathrm{S} K[\mathsf{cinc_S}()] * x \propto \mathsf{Upd}(n, -, -, -)\rangle$
$\langle j \rightarrowtail_\mathrm{S} K[\mathsf{cinc_S}()] * x \propto \mathsf{Done}(n, -, -)\rangle$
$\langle j \rightarrowtail_\mathrm{S} K[\mathsf{cinc_S}()] * (\theta, I, \emptyset, \emptyset)\rangle$
$\langle \mathsf{ret.\ ret} = () \wedge j \rightarrowtail_\mathrm{S} K[()]\rangle$

Figure 9. Proof outline for conditional increment

an identity: its location. The protocol gives the life story for every possible location in the heap as a potential value of c, with the usual constraint that all but finitely many locations are in the unborn ($\bot$) state. The first step of the protocol reflects the choice latent in the sum type: either this location is a quiescent **inj**$_1\ n$ (represented initially by $\mathsf{Const}(n)$) or an active increment operation **inj**$_2\ n$ (represented initially by $\mathsf{Upd}(d, \emptyset)$). The *logical descriptor* d gives the old value n of the counter, together with the thread id j and specification evaluation context of the thread attempting the increment. The latter information is necessary because thread j temporarily donates its spec to the protocol, permitting helping threads to execute the spec on its behalf. Following the pattern laid out in Section 2.5, in return for donating its spec, thread j receives a token—call it Attempt—which will later permit it, and only it, to recover its spec. As usual, we depict the token with a bullet.

The life story for a quiescent **inj**$_1\ n$ is quite mundane: either it is the current value pointed to by c, or it is Dead. An active cell **inj**$_2\ n$ leads a much more exciting life. In the first phase of life, $\mathsf{Upd}(d, B)$, the cell records which *branches* $B \subseteq \{0, 1\}$ of the complete code have been entered by a thread. Initially, no thread has executed complete, so the set is empty. If a thread subsequently reads that $f =$ **true** in the first step of executing complete, it moves to the set $\{1\}$, since it is now committed to the branch that adds 1 to the initial value n. Crucially, this step coincides with a *speculative* run of the specification; the un-run spec is also retained, in case some other thread commits to the 0 branch. The branch-accumulation process continues until some thread (perhaps not the original instigator of the increment) actually succeeds in performing its CAS in complete. At that point, the increment is Done, and its **inj**$_2\ n$ cell is effectively dead, but not yet Gone: in the end, the thread that instigated the original increment reclaims its spec, whose execution is guaranteed to be finished.

4.2 The proof

We now formally justify that $\mathsf{counter_I}$ refines $\mathsf{counter_S}$ by giving a concrete interpretation to the protocol and providing a Hoare-style proof outline for complete and cinc. The outline for get is then a straightforward exercise.

To formalize the protocol, we first give the set of states S_0 for an *individual* life story; see Figure 8. The states S for the data structure are then a product of individual STS states indexed by location, with all but finitely many locations required to be in state $\bot$. The set of tokens A for the product STS is just the set of locations, *i.e.,* there is one token per location (and hence per individual life story). The transition relation $\rightsquigarrow$ on the product STS lifts the one for individual life stories: $s \rightsquigarrow s' \triangleq \forall \ell.\ s(\ell) = s'(\ell) \vee s(\ell) \rightsquigarrow s'(\ell)$. If F_0 is the free-token function for an individual STS, we can then define the product STS as follows:

$$\theta \ \triangleq \ (S, A, \rightsquigarrow, \lambda s.\{\ell \mid F_0(s(\ell)) = \{\mathsf{Attempt}\}\})$$

The interpretation I for states of the product STS given in Figure 8 is fairly straightforward. The implementation and specification flag values must always match. There must exist a unique location ℓ_c ("$\exists!\ell_c$") in a "live" state of Const or Upd. This unique live location will be the one currently pointed to by c. In the Upd state, it also owns speculative spec resources according to the branch set B. Finally, Done nodes retain a finished spec, while Dead and Gone nodes are simply garbage $\mathbf{inj}_1\ (-)$ and $\mathbf{inj}_2\ (-)$ nodes, respectively.

To show the refinement $\mathsf{counter_I} \preceq^{\mathcal{E}} \mathsf{counter_S} : \tau$, where $\tau = (\mathbf{1} \to \mathbf{N} \times \mathbf{B} \to \mathbf{1} \times \mathbf{1} \to \mathbf{1})$, it suffices to show the following Hoare triple for every j, K:

$$\langle j \rightarrowtail_{\mathrm{S}} K[\mathsf{counter_S}()]\rangle\ \mathsf{counter_I}\ \left\langle z_{\mathrm{I}}.\ \exists z_{\mathrm{S}}.\ z_{\mathrm{I}} \preceq^{\mathcal{V}} z_{\mathrm{S}} : \tau \wedge j \rightarrowtail_{\mathrm{S}} K[z_{\mathrm{S}}]\right\rangle$$

The execution of $\mathsf{counter_I}$ is short and simple: it allocates the hidden state of the data structure, and then immediately returns three procedures for manipulating that state. In the proof of the triple, after the hidden state is allocated, we construct an island to govern it and add the island to the world (a guarantee extension). The new island is described by the assertion $\exists \ell.\ (\theta, I, [\ell \mapsto \mathsf{Const}(0)], \emptyset)$, which says that it follows the conditional increment protocol (θ and I), is in some rely-future state of $[\ell \mapsto \mathsf{Const}(0)]$ (in which every location other than ℓ is unborn), and currently owns no tokens. Adding this island requires us to show that the initial values of the hidden state in the implementation and specification satisfy the invariant at this state, which they clearly do.

We must then show, in the context of this extended world, that each of the implementation procedures refines the corresponding specification procedure; we give the detailed proof for cinc, *i.e.,*

$$\langle j \rightarrowtail_{\mathrm{S}} K[\mathsf{cinc_S}()] * (\theta, I, \emptyset, \emptyset)\rangle\ \mathsf{cinc_I}()\ \langle \mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_{\mathrm{S}} K[()]\rangle$$

In the precondition, we weaken our knowledge about the island to simply saying that it is in a rely-future state of $\emptyset$ (where *every* location maps to $\bot$), since this is all we need to know.

The locality of the local life stories is manifested in our ability to make isolated, abstract assertions about a particular location governed by the data structure. Because every location is in some rely-future state of $\bot$, we can focus on a location x of interest by asserting that the product STS is in a rely-future state of $[x \mapsto s_0]$, where $s_0 \in S_0$. For readability, we employ the following shorthand for making such local assertions about the island:

$$x \propto s_0 \ \triangleq \ (\theta, I, [x \mapsto s_0], \emptyset) \qquad x \propto_\bullet s_0 \ \triangleq \ (\theta, I, [x \mapsto s_0], \{x\})$$

Thus empowered, we can glean some additional insight about the algorithm: that the complete function satisfies the triple

$$\langle x \propto \mathsf{Upd}(n, j, K, \emptyset)\rangle\ \mathsf{complete}(x, n)\ \langle \mathsf{ret}.\ x \propto \mathsf{Done}(n, j, K)\rangle$$

In reading this triple, it is crucial to remember that assertions are closed under rely moves—so $x \propto \mathsf{Upd}(n, j, K, \emptyset)$ means that the location x *was once* a live, in-progress update. The interesting thing about the triple is that, regardless of the exact initial state of x, on exit we *know* that x is at least Done—and there's no going back.

The proof outline for complete at the top of Figure 9 states that, after reading the value of the flag, the location x is in an appropriately speculative state. To *prove* that fact, we must consider the rely-future states of $\mathsf{Upd}(n, j, K, \emptyset)$, and show that for each such state we can reach (via a guarantee move) a rely-future state of $\mathsf{Upd}(n, j, K, \{1\})$ or $\mathsf{Upd}(n, j, K, \{0\})$, depending on the value read. For example, if the initial state is s_0 and we read that the flag is true, we take a guarantee move to s_0' as follows:

If s_0 is	then s_0' is	If s_0 is	then s_0' is
$\mathsf{Upd}(d, \emptyset)$	$\mathsf{Upd}(d, \{1\})$	$\mathsf{Upd}(d, \{1\})$	$\mathsf{Upd}(d, \{1\})$
$\mathsf{Upd}(d, \{0\})$	$\mathsf{Upd}(d, \{0, 1\})$	$\mathsf{Upd}(d, \{0, 1\})$	$\mathsf{Upd}(d, \{0, 1\})$
$\mathsf{Done}(d)$	$\mathsf{Done}(d)$	Gone	Gone

If the initial state already included the needed speculation (or was Done or Gone), there is nothing to show; otherwise, changing the state requires speculative execution of the spec. We perform a similar case analysis at the CAS step, but there we start with the knowledge that the appropriate speculation has already been performed—which is exactly what we need if the CAS succeeds. If, on the other hand, the CAS fails, it *must* be the case that x is at least Done: if it were still in an Upd state, the CAS would have succeeded.

With complete out of the way, the proof of cinc is relatively easy; see the bottom of Figure 9.[8] When entering the procedure, all that is known is that the island exists, and that the specification is owned. The thread first examines c to see if the counter is quiescent, which is the interesting case. If the subsequent CAS succeeds in installing an active descriptor $\mathbf{inj}_2\ n$, that descriptor is the new live node (in state $\mathsf{Upd}(n, j, K, \emptyset)$)—and the thread, being responsible for this transition, gains ownership of the descriptor's token. The resulting assertion $y \propto_\bullet \mathsf{Upd}(n, j, K, \emptyset)$ is equivalent to

$$y \propto \mathsf{Upd}(n, j, K, \emptyset) * y \propto_\bullet \mathsf{Upd}(n, j, K, \emptyset)$$

which means that we can use $y \propto_\bullet \mathsf{Upd}(n, j, K, \emptyset)$ as a *frame* in an application of the frame rule to the triple for $\mathsf{complete}(y, n)$. This gives us the framed postcondition

$$y \propto \mathsf{Done}(n, j, K) * y \propto_\bullet \mathsf{Upd}(n, j, K, \emptyset)$$

which is equivalent to $y \propto_\bullet \mathsf{Done}(n, j, K)$. Since our thread still owns the token, we know the state is *exactly* $\mathsf{Done}(n, j, K)$, and in the next step (where we return the requisite unit value) we trade the token in return for our spec—which *some* thread has executed.

5. Discussion and related work

We have presented a model for a **high-level language** with concurrency that enables **direct refinement proofs** for sophisticated FCDs, via a notion of **local protocol** that encompasses the fundamental phenomena of **role-playing**, **cooperation**, and **nondeterminism**. In this section, we survey the most closely related work along each of these axes.

High-level language Birkedal *et al.* [5] recently developed the first logical-relations model for a *higher-order* concurrent language similar to the one we consider here. Their aim was to show the soundness of a sophisticated Lucassen-and-Gifford-style [27] type-and-effect system, and in particular to prove the soundness of a Parallelization Theorem for disjoint concurrency expressed by the effect system (when the Bernstein conditions are satisfied). The worlds used in the logical relation capture the all-or-nothing approach to interference implied by the type-and-effect system. As a result, the model has rather limited support for reasoning about FCDs: it can only prove correctness of algorithms that can withstand arbitrary interference.

We are unaware of any other proof methods that handle higher-order languages, shared-state concurrency, *and* local state.

Direct refinement proofs Herlihy and Wing's seminal notion of *linearizability* [21] has long been the gold standard of correctness for FCDs, but as Filipović *et al.* argue [14], what clients of an FCD *really* want is a contextual refinement property. Filipović *et al.* go on

[8] The steps labeled with $\dot{\cdot}$ indicate uses of the rule of consequence.

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 FCDs and their clients [16]. While it is possible to compose this abstraction theorem with a proof of linearizability to prove refinement, there are several advantages to our approach of proving refinement directly. First of all, it allows us to treat refinement as an assertion in our logic, which means that we can *compose* proofs of refinement when reasoning about compound FCDs, and do so while working within a single logic. Second, it allows us to leverage recent work for reasoning about refinement and hidden state, *e.g.,* Dreyer *et al.*'s STS-based logical relations [9]. Third, it allows us to reason about FCDs that use higher-order features, *e.g.,* the universal FCD construction given in [20], which would otherwise require extending the definition of linearizability to the higher-order case. Finally, it allows us to seamlessly combine reasoning about fine-grained concurrency with other kinds of reasoning, *e.g.,* relational parametricity [32].

Turon and Wand developed the first logic for reasoning directly about contextual refinement for FCDs [36]. Their model is based on ideas from rely-guarantee and separation logic and was developed for a simple first-order language, using an extension of Brookes's trace-based denotational semantics [6]. While it is capable of proving refinement for simple FCDs, such as Treiber's stack, it does not easily scale to more sophisticated algorithms.

More recently, Liang *et al.* proposed RGSim [25], a compositional inter-language simulation relation based on rely-guarantee for verifying program transformations in a concurrent setting. Liang *et al.* also use their method to prove that some simple, but realistic, FCDs are simulated by their spec. While the original paper on RGSim did not relate simulation to refinement or linearizability, new (currently unpublished) work has done so [24].

Local protocols O'Hearn *et al.*'s work on *Linearizability with hindsight* [30] clearly articulates the need for local protocols in reasoning about FCDs, and demonstrates how a certain mixture of local and global constraints leads to insightful proofs about lock-free traversals. At the heart of the work is the remarkable *Hindsight Lemma*, which justifies conclusions about reachability in the past based on information in the present. Since O'Hearn *et al.* are focused on providing proofs for a particular class of algorithms, they do not formalize a general notion of protocol, but instead focus on a collection of invariants specific to the traversals they study. We have focused, in contrast, on giving a simple but general account of local protocols that suffices for *temporally-local* reasoning about a range of FCDs. It remains to be seen, however, whether our techniques yield a satisfying temporally-local correctness proof for the kinds of traversals O'Hearn *et al.* study, or whether (as O'Hearn *et al.* argue) these traversals are best understood non-locally.

The notion of protocol most closely related to ours is Dinsdale-Young *et al.*'s *Concurrent abstract predicates* (CAP) [7]. CAP extends separation logic with shared, hidden regions similar to our islands. These regions are governed by a set of *abstract predicates*, which can be used to make localized assertions about the state of the region. In addition, CAP provides a notion of named *actions* which characterize the possible changes to the region. Crucially, actions are treated as a kind of *resource* which can be gained, lost, or split up (in a fractional permissions style), and executing an action can result in a change to the available actions. It is incumbent upon users of the logic to show that their abstract predicates and actions cohere, by showing that every abstract predicate is "self-stable" (remains true after any available action is executed).

While CAP's notion of protocol is very expressive, it is also somewhat "low-level" compared to our STS-based protocols, which would require a somewhat unwieldy encoding to express in CAP. In addition, our protocols make a clear separation between *knowledge* bounding the state of the protocol (treated as a copyable assertion) and *rights* to change the state (treated as a linear resource: tokens), which are mixed in CAP. Another major difference is that CAP exposes the internal protocol of a data structure as part of the specification seen by a client—which means that the spec for a given FCD often depends on how the client is envisioned to use it. Additional specs (and additional correctness proofs) may be necessary for other clients. By contrast, we take a coarse-grained data structure as an all-purpose spec; if clients then want to use that data structure according to some sophisticated internal protocol, they are free to do so. Finally, our protocols support speculation and spec code as a resource, neither of which are supported in CAP.

Role-playing The classic treatment of role-playing in shared-state concurrency is Jones's *rely-guarantee* reasoning [23], in which threads *guarantee* to make only certain updates, so long as they can *rely* on their environment to make only certain (possibly different) updates. More recent work has combined rely-guarantee and separation logic (SAGL [13] and RGSep [38]), in some cases even supporting a frame rule over the rely and guarantee constraints themsevles (LRG [12]). This line of work culminated in Dodds *et al.*'s *deny-guarantee* reasoning [8]—the precursor to CAP—which was designed to facilitate a more dynamic form of rely-guarantee to account for non-well-bracketed thread lifetimes. In the deny-guarantee framework, actions are classified into those that both a thread and its environment can perform, those that neither can perform, and those that only one or the other can perform. The classification of an action is manifested in terms of two-dimensional fractional permissions (the dimensions being "deny" and "guarantee"), which can be split and combined dynamically. Our STSs express dynamic evolution of roles in an arguably more direct and visual way, through tokens.

Cooperation Vafeiadis's thesis [37] set a high-water mark in verification of the most sophisticated FCDs (such as CCAS [18, 15]). Building on his RGSep logic, Vafeiadis established an informal methodology for proving linearizability by employing several kinds of ghost state (including prophecy variables and "one-shot" resources, the latter representing linearization points). By cleverly storing and communicating this ghost state to another thread, one can perform thread-local verification and yet account for cooperation: the other thread "fires" the single shot of the one-shot ghost resource. While this account of cooperation seems intuitively reasonable, it lacks any formal metatheory justifying its use in linearizability or refinement proofs. Our computational resources generalize Vafeiadis's "one-shot" ghost state, since they can (and do) run computations for an arbitrary number of steps, and we have justified their use in refinement proofs—showing, in fact, that the technique of logical relations can be expressed in a "unary" (Hoare logic) style by using these computational resources.

Concurrently with our work, Liang and Feng have extended their RGSim framework to account for cooperation [24]. The new simulation method is parameterized by a "pending thread inference map" Θ, which plays a role somewhat akin to our worlds. For us, worlds impose a relation between the current protocol state, the current implementation heap, and the current, speculative spec resources. By contrast, Θ imposes a relation between the current implementation heap and the current spec thread pool. To recover something like our protocols, one instead introduces *ghost state* into the implementation heap, much as Vafeiadis does; as a result, Θ can be used to do thread-local reasoning about cooperative FCDs. However, there are several important differences from our approach. First, there is no notion of *composition* on thread inference maps, which take the perspective of the global implementation heap and global pool of spec threads. Thus thread inference maps do not work as resources that can be owned, split up, transferred

and recombined. Second, the assertions that are used in pre- and post-conditions cannot talk directly about the thread inference map; they must control it indirectly, via ghost state. Third, the simulation approach does not support speculation or high-level language features like higher-order functions or polymorphism. Finally, it requires encoding protocols via traditional ghost state and rely/guarantee, rather than through standalone, visual protocols.

Groves and Colvin propose [17] a radically different approach for dealing with cooperation, based on Lipton's method of *reduction* [26]. Reduction, in a sense, "undoes" the effects of concurrency by showing that interleaved actions commute with one another—much like linearizability. Groves and Colvin are able to *derive* an elimination stack from its spec by a series of transformations, justifying each by considering possible interleavings and proving, very roughly, that the relevant actions commute. Elmas *et al.* also developed a method for proving linearizability using reduction and abstraction [11], and while they do not study cooperation explicitly, it is likely that their method can cope with it too.

Nondeterminism Forward simulation is well-known to be sensitive to differences in the timing of nondeterminism (also known as the "branching" structure of a transition system) [39]. On the other hand, forward simulation is appealingly *local*, since it considers only one step of a program at a time (as opposed to *e.g.,* trace semantics). To retain temporally-local reasoning but permit differences in nondeterminism (as in the late/early choice example), it suffices to use a combination of forward and *backward* simulation or, equivalently, history and *prophecy* variables [28, 1]. Lynch and Vaandrager showed that there are also *hybrids* of forward and backward simulations, which relate a *single* state in one system to a *set* of states in the other—much like our speculation. Our technique goes further, though, in combining this *temporally*-local form of reasoning with *thread*-local reasoning: hybrid simulations work at the level of complete systems, whereas our threadpool simulations can be composed into larger threadpool simulations. Composability allows us to combine thread-private uses of speculation with shared uses of speculation in protocols; moreover, it is crucial in showing soundness for contextual refinement.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

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

[4] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed Kripke models over recursive worlds. In *POPL*, 2011.

[5] L. Birkedal, F. Sieczkowski, and J. Thamsborg. A concurrent logical relation. In *CSL*, Sept. 2012.

[6] S. Brookes. Full abstraction for a shared variable parallel language. *Information and Computation*, 127(2):145–163, 1996.

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

[8] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

[9] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *ICFP*, 2010.

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

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

[12] X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327. ACM, 2009.

[13] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.

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

[15] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

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

[17] L. Groves and R. Colvin. Trace-based derivation of a scalable lock-free stack algorithm. *Form. Asp. Comput.*, 21(1-2):187–223, 2009.

[18] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *DISC*, 2002.

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

[20] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

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

[22] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

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

[24] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. Manuscript, July 2012.

[25] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.

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

[27] J. Lucassen and D. Gifford. Polymorphic effect systems. In *POPL*, 1988.

[28] N. Lynch and F. Vaandrager. Forward and backward simulations I: untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[29] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998.

[30] P. O'Hearn, N. Rinetzky, M. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC*, 2010.

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

[32] J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *Information Processing*, 1983.

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

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

[35] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency (Technical appendix), 2012. URL: `http://www.ccs.neu.edu/home/turon/relcon`.

[36] A. Turon and M. Wand. A separation logic for refining concurrent objects. In *POPL*, 2011.

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

[38] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

[39] R. J. van Glabbeek. The linear time - branching time spectrum. In *CONCUR*, 1990.