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@cs.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 serialized.

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).

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 serializes 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.

As FCDs are very tricky to reason about directly, they are carefully designed to be contextual refinements of their coarse-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 [21]. 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: Most prior approaches have focused on proving a related property on traces called linearizability [22]. 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
Following recent work by Dreyer et al. in the setting of 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 arguably 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 [20] 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 offer its argument to be pushed; if a thread running the pop operation is very kind of the pop thread to do so, execution of 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 [19, 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 ADT, 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 sharable 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).

\[
\begin{align*}
\tau &::= 1 \mid B \mid \tau + \tau \mid \text{ref}(\tau) \mid \text{ref}(\tau) \mid \alpha \mid \mu.\alpha.\tau \mid \forall a.\tau \mid \tau \rightarrow \tau \\
\sigma &::= 1 \mid B \mid \tau + \tau \mid \text{ref}(\tau) \mid \text{ref}(\tau) \mid \alpha \mid \mu.\alpha.\sigma \\
e &::= \text{true} \mid \text{false} \mid \text{if} \ e \ e_1 \ e_2 \\
&\mid \text{rec} f(x).e \mid \text{new} \tau \ e_1 \ e_2 ::= e \mid \text{CAS}(e_1, e_2, e) \mid \ell \ \text{fork} e \\
&\mid \text{null} \ (e) \mid \text{case}(e, e_1 \Rightarrow e_2, e_3 \Rightarrow e_4) \\
&\mid \text{inj} i \ e \mid \text{case}(e, \text{inj} i \Rightarrow e, \text{inj} j \Rightarrow e) \\
v &::= \text{rec} f(x).e \mid \Delta \ e \mid \text{true} \mid \text{false} \mid \ell \ x \ u ::= (\tau) \mid \text{inj} v \Gamma \::= \cdot \mid \Gamma, x:\tau \\
K &::= \cdot \mid \text{some}(K) \mid \text{case}(K, \text{null} \Rightarrow e, \text{some} \Rightarrow e) \mid K \cdot \ldots
\end{align*}
\]

\[
\begin{align*}
T &\in \text{ThreadPool} \triangleq \aleph_0 \mid N \in \text{Exp} \\
&h \::= :: h, \ell \rightarrow u
\end{align*}
\]

Type rules

\[
\begin{align*}
\Omega &::= e : \tau \\
\Omega &\vdash \text{ref}(\tau) \quad \tau_i = \sigma \\
\Omega &\vdash e_i : \sigma \\
\Omega &\vdash e_n : \sigma \\
\Omega &\vdash e : \tau \\
\Omega &\vdash \text{CAS}(e_i, e_o, e_n) : B \\
\Omega &\vdash \text{fork} e : 1
\end{align*}
\]

Primitive reductions

\[
\begin{align*}
h;\ell \cdot i &\rightarrow h; v_i \quad \text{when } h(\ell) = (\tau) \\
h;\ell \cdot \text{CAS}(h_1, v_o, v_n) &\rightarrow h; [h_1[v_o] = v_n]; \text{true} \quad \text{when } h(\ell) = v_o \\
h;\ell \cdot \text{CAS}(h_1, v_o, v_n) &\rightarrow h; \text{false} \quad \text{when } h(\ell) \neq v_o \\
h;\ell \cdot \text{case}(e_1, \text{null} \Rightarrow e_1, \text{some} \Rightarrow e_2) &\rightarrow h; e_1 \\
h;\ell \cdot \text{case}(e_1, \text{null} \Rightarrow e_1, \text{some} \Rightarrow e_2) &\rightarrow h; e_2[x] \\
h;\ell \cdot \text{null} &\rightarrow h; e_1 \\
h;\ell \cdot \text{some}(x) &\rightarrow h; e_1[x] \\
h;\ell \cdot \text{new} \tau &\rightarrow h; [\ell \rightarrow (\tau)]; \ell \\
h;\ell \cdot \text{inj} i &\rightarrow h; [\ell \rightarrow \text{inj} i]; \ell \\
h;\ell \cdot \text{fork} e &\rightarrow h; [\ell \rightarrow \text{fork} e]; \ell
\end{align*}
\]

Program reduction

\[
\begin{align*}
h;\ell \cdot i &\rightarrow h'; e' \\
h;\ell \cdot \text{fork} e &\rightarrow h; T \cdot [i \rightarrow K[e]] \\
h;\ell \cdot \text{fork} e &\rightarrow h; T \cdot [i \rightarrow K[e]] [j \rightarrow e]
\end{align*}
\]

Figure 1. Our language: \(F^{1st}\) with forking and CAS

2. The main ideas

2.1 The language

Our language is a variant of the polymorphic lambda calculus, extended with tagged sums, general mutable references (higher-order state), equi-recursive types, CAS, and forking. Figure 1 presents the syntax of the language together with excerpts of the dynamic and static semantics. Our terms are not annotated with types, so our syntax for type abstraction is \(\Lambda e.\tau\), whereas for type application \(e:\tau\). Recursive types \(\mu\alpha.\tau\) are required to be productive, meaning that all free occurrences of \(\alpha\) in \(\tau\) must appear under a non-\(\mu\)-type constructor. We support references to tuples \((\text{ref}(\tau))\) as well as option references \((\text{ref}_o(\tau))\), inherited by the terms \(\text{null}\) and \(\text{some}(e)\) which are essentially the untagged union of the unit and reference types. The term \(e[i]\) denotes the \(i\)-th projection from a tuple reference \(e\), and \(e[i] := e'\) denotes assignment to that projection. (If \(e\) is a single-cell reference, then we will often write \(e := e'\) instead of \(e[i] := e'\).) Since CAS should only be able to test and update word-sized values, we restrict CAS to operate on terms with comparable types which we denote using \(\sigma\). We use \(\tau\) to denote values stored in the heap. Tagged sums are allocated on the heap, which justifies the inclusion of sum types in the set of comparable types \(\sigma\). (In examples, we will employ a few other features that extend the language defined here in trivial ways, such as immutable pairs and records.) Figure 1 presents the typing rule for CAS and forking; the remaining rules are standard (see appendix [34]).

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 relation \(\rightarrow\) between program configurations, using evaluation contexts \(K\) to specify a left-to-right evaluation order within each
MSQ: \( \forall \alpha.1 \rightarrow \{ \text{enq : } \alpha \rightarrow 1, \text{deq : } 1 \rightarrow \text{ref}_f(\alpha) \} \)

MSQ \( \triangleq \Lambda. \lambda(). \text{let head = new cons(null, null) in} \{ \)

\( \text{deq = rec try()}, \)

\( \text{let } c = \text{head[1]}, n = \text{getVal}(c) \text{ in case } n[2] \text{ of} \)

null => null \( \{ \ast \text{ queue is empty } \ast \} \)

some(n') => if CAS(head[1], c, n') then n'[1] else try()

\( \text{enq = } \lambda x. \text{let } n = \text{cons}(\text{some}(\text{new } x), \text{null}) \text{ in} \)

\( \text{let rec try(c). case } c[2] \text{ of} \)

null => if CAS(c[2], null, n) then () else try(c)

\( \{ \ast \text{ elided } \ast \} \)

\( \text{try(getVal(head[1]))} \}

CGQ \( \triangleq \Lambda. \lambda(). \text{let head = new null, lock = new false in} \{ \)

\( \text{deq = } \lambda(). \text{sync(lock)} \{ \text{case head[1] of} \)

null => null \| some(n) => head[1] := n[2], some(new(n[1])) \}

\( \text{enq = } \lambda x. \text{sync(lock)} \{ \ldots \} \{ \ast \text{ elided } \ast \} \}

Logically in queue

![Diagram of queue](image)

Reachable

\( I(s) \triangleq \text{head} \to_{L} e_0 \to_{L} e_1 \to_{L} (v_0, t_1) \ast \exists v_s. \text{head}_{t} \rightarrow_{L} v_{S} \)

\( \ast \text{ lock } \to_{L} \ast \text{ false } \ast \text{ link } (v_1, v_{S}, s_L) \ast (s(L') \neq \perp \land L_1 \rightarrow_{L} (v_0, t_0)) \)

\( \text{when } s = [e_0 \rightarrow_{L} \text{Sentinel}(v_0, t_1)] \sqsubseteq_{L} s_L \sqsubseteq_{L} [e_1 \rightarrow_{L} \text{Dead}(v_1, t_1')] \)

\( \text{link(null, null, \emptyset) } \triangleq \text{emp} \)

\( \text{link}(e_0, e_1, [e_0 \rightarrow_{L} \text{Live}(v_1, t_1') ] \sqsubseteq_{L} s) \triangleq \exists v_S, v_{S}', v_{I}. v_{I} \preceq \nu \forall v_S: \alpha \ast \)

\( e_0 \rightarrow_{L} v_0 \to_{L} e_1 \to_{L} v_1 \ast e_1 \to_{L} (e_1', \nu) \ast e_0 \to_{L} (v_0, v_{S}') \ast \text{link}(v_{I}', v_{S}', s) \)

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

thread. Figure 1 presents some of the primitive reduction rules for individual threads, as well as the nondeterministic program reduction rules that enable any thread to progress or fork a new 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].1 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 pro-

1 We use the shorthand cons(e, e') \( \triangleq \text{some(new } (e, e')) \).

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 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

2 The definition of sync is given in the appendix [34].
“unborn” (state ⊥). After birth, nodes go through a progression of life changes. Some changes are manifested physically. The transition from Live(v, null) to Live(v, ℓ), 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 [33], 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:

\[ S_0 \triangleq (⊥) \cup \{ \text{Live}(v, v') \mid v, v' \in \text{Val} \} \cup \{ \text{Sentinel}(v, v') \mid v, v' \in \text{Val} \cup \text{Dead}(v, ℓ) \mid v \in \text{Val}, ℓ \in \text{Loc} \} \]

along with the transition relation \( \rightarrow \) 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 \( S \triangleq \text{Loc} \sqcup S_0 \), where \( \sqcup \) indicates that all but finitely many locations are in the ⊥ (unborn) state in their local STS. The transition relation \( \rightarrow \) for the product STS is just the pointwise lifting of the one for each node’s STS: \( s \rightarrow s' \triangleq \forall ℓ. s(ℓ) \rightarrow s'(ℓ) \). Thus, at the abstract level, the product STS is 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 [32]. The relational nature is apparent in the fact that the invariant makes assertions about both the implementation \( (→_1) \) and specification \( (→_2) \) 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”) [23, 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 \( s_1 \), and a collection of Dead nodes at locations \( ℓ_1 \) (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 \( α \) (\( v_1 \preceq \alpha \) \( v_3 : α \); see Section 3). The invariant also accounts for two representation differences between MSQ and CGQ. First, the node data in the implementation is stored within a ref\((α)\), 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 specification correspond to the linearization points of the implementation.

Finally—and crucially—Dead nodes must have non-null successor pointers whose locations are in a non-live 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 [34].

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. These tokens, which may differ for different STSs, consists of some 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 free tokens that no one thread is permitted to exclusively own. 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 each state.

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.
redFlag $\triangleq \lambda().$ 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 try() else chan := 0 else try() read = $\lambda().$ flag[1] }

blueFlag $\triangleq \lambda().$ let flag = new true, lock = new false in { flip = $\lambda().$ sync(lock) { flag := not flag[1] } read = $\lambda().$ sync(lock) { flag[1] } }

![Figure 3. Red flags versus blue flags](image)

Figure 3. Red flags versus blue flags

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 $i$ 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).

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” [20]—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 chan $\rightarrow 1$, it means the side channel is not currently being used (it is in the Empty state). If chan $\rightarrow 1$, 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 chan $\rightarrow 2$, 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 and/or clear the side channel by setting chan back to 0 and returning to Empty.

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 $\rightarrow 1$), and if so, we accept the offer by setting chan to 2. Then we immediately return, having implicitly committed both flip’s 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 CAS(flag, true, false) and CAS(flag, false, 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 stupidly4) 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 CAS(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 CAS(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 CAS(chan, 0, 1) succeeded, we knew that we had successfully transitioned to the 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 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 another thread from performing chan $\rightarrow 2$ 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, $j \rightarrow_{e} s$, 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

4At 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.
LateChoice should be since we haven’t executed j on its behalf. When its spec code is matched against zero or more steps of spec code—but nondeterminism. The problem, essentially, is that when proving the “conditional CAS” example we consider in Section 4) is dealing with nondeterminism. The problem, essentially, is that when proving that an FCD refines some coarse-grained spec, we want to rea-

One tricky aspect of reasoning about certain FCDs (including the lateChoice component of an island, hence no protocol), and is not in fact an FCD at all, 

The problem is simple: speculate! That is, if you don’t know which speculative set of specification states to whichever one (either j \rightarrow_S K[true] or j \rightarrow_S K[false]) matches ret.

In order to place these Hoare-style proof outlines and sets of specification states on a firm formal footing, we now proceed to present the technical details of our logical-relations model.

3. The formal model

3.1 Resources, protocols, and possible worlds

Fundamentally, protocols govern shared resources η = (h, Σ), which consist of some portion h of the implementation’s heap, and some collection Σ of speculated specification states. Each speculation state \( \varsigma = h; T \) in turn contains a portion h of the specification’s heap, and a portion T of the specification’s code represented as a thread pool. Resources can be combined at every level: heaps and thread pools via the usual disjoint union \( \uplus \), and state sets and “impl-spec” resources via \( \otimes \), which just applies \( \uplus \) pointwise. State sets are only combinable when every combination of their speculation states is defined, making it impossible for a speculated state to “disappear” when state sets are combined.

We assign every disjoint instance of a data structure an island \( (\theta, J, s, A) \) that records the protocol governing its hidden state. The component \( \theta = (s, A, \cdots, F) \) gives the STS for the protocol, where \( S \) is its set of states, A is its set of possible tokens, \( \cdots \) is its transition relation, and F is a function telling which tokens are free at each state. Generally, this STS will be a product construction of small STSs giving local life stories, as we saw with the Michael-Scott queue. In addition to the STS, the island gives a semantic state-dependent invariant J characterizing the concrete resources owned by the island at each state; we will see the syntactic counterpart I in a moment. Together, the components \( \theta, J, \) and s give the shared knowledge of what and where the protocol is. The remaining component of an island, A, records the tokens privately owned by a particular participant in the protocol—generally, the one we are trying to prove correct. The private token set A is needed, of course, to determine which transitions the participant is allowed to take; the judgment \( \theta \vdash (s, A) \leadsto (s', A') \) codifies the law of conservation of tokens (Section 2.4) for a single step.5

A collection of islands \( \omega \) together with a step index k comprise a possible world W. The step index is used to stratify away circularities in the definition of worlds and the logical relation; it and its attendant operators \( \triangleright \) and \( \llcorner \rightarrow \lrcorner \) are completely standard, and so for space reasons we direct the reader to earlier work for a detailed explanation [9, 10]. It is worth mentioning why worlds are circular, however. Essentially, an island invariant may need to assert that certain values are logically related, but the logical relation is indexed by a world; therefore, the invariant must have access to the current world. Since the invariant is also part of the world, the situation is

---

5 We use dot notation like \( W.k \) to project named components from tuples.

6 We use the more readable notation \( s \leadsto_g s' \) in place of \( \theta \leadsto (s, s') \).
Figure 5. Resources, protocols, and possible worlds

Figure 6. Assumptions on worlds and resources
To ensure stability. The speculative ⊕ assertion logic assertions. Implication, however, is explicitly rely-closed against logical connectives; these all follow the usual semantics of separation logic of the specification resources of its operands. The fundamental property of rely-closure: if \( W, \eta \models \rho P \) and \( W \subseteq \text{ref } W' \), then \( W', \eta \models \rho P \). All assertions about the shared state governed by \( W \) are therefore “stable” under interference from other threads. The basic assertions about private resources (e.g., \( I \mapsto v \) or \( i \mapsto e \)) are entirely straightforward, as are most of the basic logical connectives; these all follow the usual semantics of separation logic assertions. Implication, however, is explicitly rely-closed to ensure stability. The speculative \( \odot \) connective simply takes the union of the specification resources of its operands.

There is only one basic assertion about shared resources: the island assertion \( \iota \), inspired by LADR [10], which serves as a syntactic counterpart to the semantic islands in \( W \). In fact, the only difference between the two is the representation of the invariant, which for \( \iota \) is a function \( I \) mapping from states of the STS to assertions. Because island assertions are explicitly rely-closed, they act as rely-lower-bounds on the true state of the island.

The remaining two kinds of knowledge—about refinements and code—are pure, meaning that they do not depend on the currently-owned tokens or on the current private state, although they may depend on the state of the islands in scope. We syntactically distinguish pure assertions \( \varphi \), and write \( U \models^p \varphi \) when the unprivileged (token-free) world \( U \) satisfies the pure assertion \( \varphi \). Purity is crucial for soundness for contextual refinement, because the context might copy a value; if our refinement assertion was impure, this would require us to copy the value’s tokens as well, which is impossible.

Readers familiar with SKLs will recognize the value-refinement assertion \( e_1 \triangleright e_2 : \tau \) as essentially the standard definition of logical approximation for values. Similarly, the expression-refinement relation \( \Delta; \Gamma \vdash e_1 \triangleleft e_2 : \tau \) defines logical approximation for open terms. For function types, we check that the bodies of the functions are related when given related arguments (which, due to 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 actually 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 and no updates, respectively. Hidden references may be governed by arbitrary islands.

Expression refinement bridges the gap between refinement reasoning (which is binary) and Hoare-style reasoning (which is unary). We first explain Hoare-style reasoning.

The Hoare triple \( \langle P \rangle e \langle x. Q \rangle \) says that if we run \( e \) in a world \( U \) that satisfies \( P \) then (1) the expression will obey all the protocols in \( U \) and (2) if it terminates with result \( v \), it will do so in some new world \( W \) and with accumulated private resources \( \eta \) that together satisfy \( Q[v/x] \). In order to check these two properties, we defer to the more general threadpool simulation assertion \( T \odot m \langle x. P \rangle \) which accounts for forking of new threads and acquisition of tokens or private resources. Threadpool simulation also accounts for motion within a protocol: between every step a threadpool takes, its environment may move to any rely-future world, and similarly the threadpool can move to guarantee-future worlds.

That threadpool simulation is, in fact, a simulation is due to its use of the speculative stepping relation \( \Sigma \equiv \Sigma' \), which requires any changes to the specification state to represent feasible execution steps: every new state must be reachable from some old state. When the implementation \( T \) takes a step, the world’s step index is reduced, which makes the definition well-founded. Once the “main thread” \( m \) has terminated, we are given one last chance to take specification steps before checking the postcondition. Since the main thread may have forked others, we continue checking simulation to ensure that the remaining threads are protocol-safe.

An expression \( e_1 \) refines \( e_2 \) if, when given a thread \( j \mapsto K'[e_2] \) as a private computational resource, it finishes its execution only in a state where it still owns thread \( j \), which has produced a related value in context \( K \). During execution, however, ownership of \( j \) can be transferred into and out of shared islands. Semantically, contextual refinement, written \( \Delta; \Gamma \vDash e_1 \triangleleft e_2 : \tau \), holds if for every \( i, j \) and every context \( C \) of type \( N \) with hole of type \( \tau \), we have

\[
\forall n, T_1. \quad \langle i \mapsto C[e_1] \rangle \mapsto^* \langle i \mapsto n \rangle \cup T_1 \implies \exists n, T_2. \quad \langle j \mapsto C[e_2] \rangle \mapsto^* \langle j \mapsto n \rangle \cup T_2
\]

meaning that we observe the termination of the initial (“main”) thread only. We then have:

**Theorem 1 (Soundness).** If \( U \models^p \Delta; \Gamma \vdash e_1 \triangleleft e_2 : \tau \) for all \( U \) then \( \Delta; \Gamma \vDash e_1 \triangleleft e_2 : \tau \).

The proof of this theorem, given in the appendix [34] requires some novel lemmas expressing key framing properties for the threadpool simulation assertion. We also prove the following inference rules for valid, pure assertions (true at every world):

| \( \langle P \rangle e \langle x. Q \rangle \) & \( \exists x. \langle P' \rangle e' \langle y. P'' \rangle \) & \( \langle P \rangle e \langle x. Q \rangle \) & \( \exists x. \langle P' \rangle e' \langle y. P'' \rangle \) |
|---|---|---|---|
| \( P \) & let \( x = e \) in \( e' \langle y. P'' \rangle \) & \( P \) & let \( x = e \) in \( e' \langle y. P'' \rangle \) |
| \( P \) & \( P' \) & \( \langle P' \rangle e \langle x. Q' \rangle \) & \( Q' \Rightarrow Q \) & \( \triangleright P \Rightarrow P \) & \( \triangleright P \Rightarrow P \) |

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

\footnote{We also preserve the environment’s private resources as a frame.}

\footnote{The relation to hybrid simulation is discussed in Section 5.}
**Case study: conditional CAS**

With the details of our model in hand, we are now in a position to tackle, in detail, one of the trickiest FCDs: Harris et al.’s *conditional CAS* [19, 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 \( counter_3 \) in Figure 7. 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 CAS is a straightforward exercise.

### 4.1 The protocol

To explain our implementation, \( counter_3 \), 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 \( \text{ref}(\mathbb{N} + \mathbb{N}) \). A value \( inj_j, n \) represents an “inactive” counter with logical value \( n \). A value \( inj_j, n \), in contrast, means that the counter is undergoing a conditional increment, and had the logical value \( n \) when the increment began. Because \( inj_j, n \) 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 \( f \), 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 \( counter_3 \) refines \( counter_3 \) despite this blatant race 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 \( inj_j, n \), its logical value is potentially ambiguous: it is either \( n \) or \( n + 1 \). However, the validity of such a scheme rests on our ability to associate these values with feasible executions of the spec’s cinc code, since “logical” value here 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 at the point 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 to use speculative computational resources to account for both helping and timing issues. These resources are managed by the protocol shown in Figure 7. Since injections into sum types are heap allocated, each one has an identity: its location. The protocol gives the life story for every possible location in the heap, with the usual constraint that all but finitely many locations are in the unborn (⊥) state. The first step of the protocol reflects the choice latent in the sum type: either this location is a quiescent \( inj_j, n \) (represented initially by Const\((n)\)) or an active increment operation \( inj_j, n \) (represented initially by 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—and 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 \( inj_j, n \) is quite mundane: either it is the current value pointed to by \( c \), or it is Dead. An active cell \( inj_j, n \) leads a much more exciting life. In the first phase of life, 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 \( f = \text{true} \) in the first step of executing complete, it moves to the set \( \{1\} \), since it is now committed to the branch

---

**Figure 7.** Conditional increment, a simplification of CCAS

---

\[
\begin{align*}
\text{counter}_3 \triangleq & \text{let } c = \text{new } 0, f = \text{new } \text{false}, \text{lock} = \text{new } \text{false} \\
\text{let } \text{setFlag}(b) = \text{sync}(\text{lock}) \{ f := b \} \\
\text{let } \text{get()} = \text{sync}(\text{lock}) \{ c[1] \} \\
\text{let } \text{cinc()} = \text{sync}(\text{lock}) \{ c[1] := c[1] + 1 \text{ if } f[1] \text{ then } 1 \text{ else } 0 \} \\
\text{in } \{ \text{get, setFlag, cinc} \}
\end{align*}
\]

\[
\begin{align*}
\text{counter}_3 \triangleq & \text{let } c = \text{new } \text{inj}_j, 0, f = \text{new } \text{false} \\
\text{let } \text{setFlag}(b) = f := b \\
\text{let } \text{complete}(o, x) = \text{if } f[1] \text{ then CAS}(c, o, \text{inj}_j, (x + 1)) \\
\text{else } \text{CAS}(c, o, \text{inj}_j, x) \\
\text{let } \text{rec } \text{get()} = \text{let } o = c[1] \text{ in case } o \text{ of} \\
\text{inj}_j, x \Rightarrow x \Rightarrow \text{inj}_j \text{ x} \Rightarrow \text{complete}(o, x); \text{get()} \\
\text{let } \text{rec } \text{cinc()} = \text{let } o = c[1] \text{ in case } o \text{ of} \\
\text{inj}_j, x \Rightarrow \text{let } n = \text{inj}_j, x \text{ in} \\
\text{if } \text{CAS}(c, o, n) \text{ then } \text{complete}(n, x); () \text{ else } \text{cinc()} \\
\text{in get, setFlag, cinc}
\end{align*}
\]
let complete(o, x) =
  if f(1) then
    CAS(c, o, inj₁(x + 1))
    o ∞ Upd(x, −, −, ∅)
  else
    CAS(c, o, inj₁(x))
    o ∞ Done(x, −, −, ∅)
let rec circ() =
  if o = c[1] then
    case o of
    inj₁ x ⇒
      if CAS(c, o, n) then
        complete(n, x):
        o ∞ Upd(x, −, −, ∅)
      else
        circ()
    inj₂ x ⇒
      if CAS(c, o, n) then
        complete(o, x):
        o ∞ Done(x, −, −, ∅)
      else
        circ()
  else
    circ()

The proof outline for conditional increment

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-acummulation process continues until some thread (perhaps not the original instigator of the increment) actually succeeds in performing its CAS in complete. Once the CAS is complete, the increment is Done, and its inj₂ n cell is effectively dead, but not yet Gone: in the end, the thread that instigated the increment reclaims its spec, whose execution is guaranteed to be finished.

4.2 The proof

We now formally justify that counter₁ refines counter₂ by giving a concrete interpretation to the protocol and providing a Hoare-style proof outline for complete and circ. The outline for get, given in the appendix [34], is then a straightforward exercise.

To formalize the protocol, we first give the set of states S₀ for an individual life story; see Figure 7. The states S for the data structure are then a product of individual STS states indexed by location, with two constraints: all but finitely many locations must be in state ⊥, and there must exist a unique location ℓ (“always live”) in a “live” state of Const or Upd. This unique live location will be the one currently pointed to by c. 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 ⊾ on the product STS is the pointwise lifting of that for the individual life stories s ⊾ s′ ≜ ∀ℓ. s(ℓ) ⊾ s′(ℓ). If F₀ is the free-token function for an individual STS, we can then define the product STS as follows:

\[ \begin{align*}
\theta & \triangleq (S, A, \rightsquigarrow, \lambda s. (\ell \mapsto F₀(s)(\ell) = \text{[Attempted]}))
\end{align*} \]

The interpretation I for states of the product STS given in Figure 7 is fairly straightforward. The implementation and specification flag values must always match. The unique “live” node must be linked to by the implementation’s c reference; if it is in an 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 inj₂ (−) nodes.

To show the refinement counter₁ ≤≤ counter₂; τ, where τ = (1 → N × B → 1 × 1 → 1), it suffices to show the following Hoare triple for every j, K:

\[ (j \rightarrow S K[\text{counter₁}]) \text{ counter₁} \{ x_1 : \exists x_2, x_2 \leq \text{X} \colon \tau \land j \rightarrow S K[x_2] \} \]

The execution of counter₁ 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 governing it and add the island to the world (a guarantee extension). The new island is described by the assertion Bt[θ, I, ℓ \mapsto \text{Const}(0)], which says that it follows the conditional increment protocol (θ and I), in some else-rely-state of [ℓ \mapsto \text{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 circ, i.e.,

\[ (j \rightarrow S K[\text{circ}])(\theta, I, 0, 0) \text{ circ}(\theta, I, 0, 0) \]

In the precondition, we weaken our knowledge about the island to simply saying that it is in a rely-future state of 0 (where every location maps to ⊥), 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 ⊥, we can focus on a location x of interest by asserting that the product STS is in a rely-future state of [x \rightarrow s₀], where s₀ ∈ S₀. For readability, we employ the following shorthand for making such local assertions about the island:

\[ x \times s₀ \triangleq (\theta, I, [x \rightarrow s₀], \emptyset) \]

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

\[ (o \times Upd(x, -) ) \text{ complete}(o, x) (\theta, o \times Done(x, -)) \]

In reading this triple, it is crucial to remember that assertions are closed under rely moves—so o ∞ Upd(x, −, −, 0) means that the location o was once live, in-progress update. The interesting thing about the triple is that, regardless of the exact initial state of o, on exit we know that o is at least Done—and there’s no going back.

The proof outline for complete at the top of Figure 8 states that, after reading the value of the flag, the location o is in an appropriately speculative state. To prove that fact, we must consider the rely-future states of Upd(x, −, −, 0), and show that for each such state we can reach (via a guarantee move) a rely-future state of Upd(x, −, −, {1}) or Upd(x, −, −, 0), depending on the value read. If the initial state already included the needed speculation (or was Done or Gone), there is nothing to show; otherwise, we must speculatively execute 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 o is at least Done: if was still in an Upd state, the CAS would have succeeded.

With complete out of the way, the proof of circ is relatively easy; see the bottom of Figure 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 the active descriptor inj₂ x, that descriptor is the new live node (in state Upd(x, j, K, ∅))—and the thread, being responsible for this transition, gains ownership of the descriptor’s token. The resulting assertion n ×× Upd(x, j, K, ∅) is equivalent to

\[ n \times\text{Upd}(x, j, K, ∅) + n \times\text{Upd}(x, j, K, ∅) \]

1010 The steps labeled with ; indicate uses of the rule of consequence.
which means that we can use \( n \propto_{\ast} \text{Upd}(x, j, K, \emptyset) \) as a frame in an application of the frame rule to the triple for complete \( (n, x) \).

This gives us the framed postcondition,

\[
n \propto \text{Done}(x, - , - ,) + n \propto_{\ast} \text{Upd}(x, j, K, \emptyset)
\]

which is equivalent to \( n \propto_{\ast} \text{Done}(x, j, K) \). Since our thread still owns the token, we know the state is exactly \( \text{Done}(x, j, K) \), 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. The model has only very limited support for reasoning about FCDs—it can only prove correctness of algorithms that can withstand arbitrary interference—essentially because the worlds used in the logical relation only allow for simple invariants.

We are unaware of any other proof methods that handle higher-order languages, shared-state concurrency, and hiding, all at once.

**Direct refinement proofs**

Herlihy and Wing’s seminal notion of linearizability [22] 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 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 [17]. While it is possible to compose this abstraction theorem with a proof of linearizability to prove refinement, there are two significant advantages to our approach of proving refinement directly. First, it allows us to more easily leverage recent work for reasoning about refinement and hidden state, e.g., Dreyer et al.’s STS-based logical relations [9]. Second, it allows us to seamlessly combine reasoning about fine-grained concurrency with reasoning about higher-order code and data abstraction; the latter is particularly well-understood in the refinement setting.

Turon and Wand developed the first logic for reasoning directly about contextual refinement for FCDs [35]. 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 verify some simple, but realistic, FCDs. We suspect that, due to its highly compositional nature, RGSim could be used (or easily extended) to show contextual refinements in the single language setting. Like Turon and Wand’s work, however, it is less clear how to scale it to the more sophisticated FCDs we study here; cooperation, in particular, poses a challenge for composable simulation.

**Local protocols**

O’Hearn et al.’s work on Linearizability with hintsight [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. As we have seen in several examples, the fixed progression of possible abstract states makes it possible to reason about FCDs at a higher level of abstraction. Because CAP’s abstract predicates are exposed to clients, the specification for a given FCD tends to depend on how the client expects to use it—there is sometimes no clear specification that works in all contexts. Our refinement approach, in contrast, is gives a single, clear specification for FCDs, and the protocol used in verifying refinement of that specification is in no way exposed to the FCD’s clients.

**Role-playing**

The classic treatment of role-playing in shared-state concurrency is Jones’s rely-guarantee reasoning [24], 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 [37]). 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 [36] set a high-water mark in verification of the most sophisticated FCDs (such as CCAS [19, 15]). Building on his RG-Sep 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 linearity 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.

Groves and Colvin propose [18] a radically different approach for dealing with cooperation, based on Lipton’s method of \textit{reduction} [26]. Reduction, in a sense, “undoes” the effects of concurrency by showing that interleaved actions commute with one another—much like linearity. 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 \textit{et al.} also developed a method for proving linearity 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) [38]. On the other hand, simulation techniques are appealingly local, since they consider only one step of a program at a time. To retain temporally-local reasoning but permit differences in nondeterminism (as in the late/early choice example), it suffices to use a combination of \textit{forward} and \textit{backward} simulation or, equivalently, history and prophecy variables [28, 1]. Lynch and Vaandrager showed that there are also \textit{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. Their 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, which is crucial in showing soundness for contextual refinement.

Fu \textit{et al.}’s HLRG [16] combines LRG with temporal logic, thereby supporting temporally-nonlocal reasoning. While HLRG incorporates a past-tense logic, it should be straightforward to generalize to the future tense as well, which would provide an alternative approach to reasoning about examples like CCAS.

**References**

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