Modular Reasoning about Separation of Concurrent Data Structures

Kasper Svendsen[1], Lars Birkedal[1], and Matthew Parkinson[2]

[1] IT University of Copenhagen, {kasv,birkedal}@itu.dk
[2] Microsoft Research Cambridge, mattpark@microsoft.com

Abstract. In a concurrent setting, the usage protocol of standard separation logic specifications are not refinable by clients, because standard specifications abstract all information about potential interleavings. This breaks modularity, as libraries cannot be verified in isolation, since the appropriate specification depends on how clients intend to use the library.

In this paper we propose a new logic and a new style of specification for thread-safe concurrent data structures. Our specifications allow clients to refine usage protocols and associate ownership of additional resources with instances of these data structures.

1 Introduction

Why? One of the challenges of specifying the abstract behavior of a library is that the appropriate specification depends on the context in which the library is going to be used. Consider a simple bag library with operations to push and pop elements from the bag. In a sequential setting the standard separation logic specification is:

$$\{\mathsf{bag}_e(\mathsf{x},\mathsf{X})\}\ \mathsf{x.Push(y)}\ \{\mathsf{bag}_e(\mathsf{x},\mathsf{X}\cup\{\mathsf{y}\})\}$$
$$\{\mathsf{bag}_e(\mathsf{x},\mathsf{X})\}\ \mathsf{x.Pop()}\ \{\mathsf{ret}.\ (\mathsf{X}=\emptyset\wedge\mathsf{ret}=\mathsf{null}\wedge\mathsf{bag}_e(\mathsf{x},\mathsf{X}))\ \vee$$
$$(\exists \mathsf{Y}.\ \mathsf{X}=\mathsf{Y}\cup\{\mathsf{ret}\}\wedge\mathsf{bag}_e(\mathsf{x},\mathsf{Y}))\}$$
$$\mathsf{bag}_e(\mathsf{x},\mathsf{X}) * \mathsf{bag}_e(\mathsf{x},\mathsf{Y}) \Rightarrow \bot$$

Here bag_e is an abstract predicate, i.e., implicitly existentially quantified, so that clients cannot depend on its definition [2], x is a reference to a bag object, and X and Y range over multisets of elements. The implication in the third line expresses that the bag_e predicate cannot be duplicated. Hence this specification enforces that clients follow a strict usage protocol, with a single exclusive owner of the bag object. On the other hand, this specification allows the owner of the bag to track the exact contents of the bag. In other words, $\mathsf{bag}_e(\mathsf{x},\mathsf{X})$ asserts full ownership of the bag and that the bag contains exactly the objects in the multiset X.

Now consider a client of the bag library and suppose this client wants to implement a bag of independent tasks scheduled for execution. This client might not care about the exact contents of the bag, only that each task in the bag owns the resources necessary to perform its task. In addition, this client might wish to share the bag to allow multiple users to schedule tasks for execution. Thus this client might prefer the following specification for shared bags:

$$\{\mathsf{bag}_s(\mathsf{x},\mathsf{P}) * \mathsf{P}(\mathsf{y})\}\ \mathsf{x.Push(y)}\ \{\mathsf{bag}_s(\mathsf{x},\mathsf{P})\}$$
$$\{\mathsf{bag}_s(\mathsf{x},\mathsf{P})\}\ \ \mathsf{x.Pop()}\ \ \{\mathsf{ret.}\ \mathsf{bag}_s(\mathsf{x},\mathsf{P}) * (\mathsf{ret} = \mathsf{null}\ \vee\ \mathsf{P}(\mathsf{ret}))\}$$
$$\mathsf{bag}_s(\mathsf{x},\mathsf{P}) \Rightarrow \mathsf{bag}_s(\mathsf{x},\mathsf{P}) * \mathsf{bag}_s(\mathsf{x},\mathsf{P})$$

This specification allows more sharing, but it does not track the exact contents of the bag. Instead, it allows clients to associate additional resources with each element of the bag using the P predicate, and to freely share the bag as expressed by the implication in the third line. Clients thus transfer P(y) to the bag when pushing y, and receive P(ret) from the bag, when pop returns a non-null element.

In a sequential first-order setting without reentrancy, the standard separation logic specification suffices. Using techniques from fictional separation logic [11], clients can refine the standard specification to allow the additional sharing of the shared bag specification. However, in a concurrent setting, it is easy to come up with a non-thread-safe implementation (without synchronization), that satisfies the standard specification (as it enforces a single exclusive owner), but not the shared bag specification. Hence, in a higher-order concurrent setting with reentrancy, this type of refinement is unsound!

What? The key challenge is to provide a logic that enables clients to refine the specifications to their requirements in a concurrent setting. In this paper we propose such a logic, called Higher-Order Concurrent Abstract Predicates (HOCAP), and a new style of specification for thread-safe concurrent data structures.[3] This style of specification allows clients to refine the usage protocol and associate ownership of additional resources with instances of the data structure, in a concurrent higher-order setting.

How? Observe first that while it is not sound to refine specifications to allow *more sharing* in a concurrent setting, it is sound to refine specifications to permit *less sharing*. Thus we will start with a weak specification that allows unrestricted sharing of instances of the data structure, and then let clients refine this specification as needed.

To reason about sharing we partition the state into *regions*, with *protocols* governing how the state in each region is allowed to evolve, following earlier work on concurrent abstract predicates [5]. Our new program logic, HOCAP, also uses *phantom fields* – a logical construct akin to auxiliary variables, that only occur in the logic.

To support abstract refinement of library specifications, we propose to verify the implementation using a region to share the concrete state of the implementation, with a fixed protocol that *relates* the concrete state of the implementation

[3] We consider a concurrent data structure thread-safe if each of its methods has one *or more* synchronization points, where the abstract effects of the method appear to take affect. See Related Work for a discussion of the relation to linearizability.

with an abstract description of the state of the data structure. To refine this specification, clients define a region of their own, with a protocol on the *abstract state* of the data structure. For soundness, these two regions must evolve in lock-step and *synchronize* when the abstract state changes (in synchronization points). We do so by giving each region a half permission to a shared phantom field; synchronization can then be enforced since updating a phantom field requires full permission. Half permissions have previously been used to synchronize local and shared state [14]; here we are using it to synchronize two shared regions.

For the bag example, we introduce a phantom field `cont` that contains the abstract state of the bag: a multiset of references to the elements in the bag. The bag constructor also returns a half permission to the phantom field `cont`:

$$\frac{}{\{\mathsf{emp}\}\mathsf{new\ Bag()}\{\mathsf{ret}.\ \mathsf{bag(ret)} * \mathsf{ret}_{\mathsf{cont}} \xmapsto{1/2} \emptyset\}}$$

Here $\mathsf{ret}_{\mathsf{cont}} \xmapsto{1/2} \emptyset$ asserts partial ownership of the phantom `cont` field. Since the client obtains half the `cont` permission upon calling the constructor, the library cannot update the `cont` field on its own.

The protocol governing the bag x thus relates the concrete state of the bag with its abstract state (the value of the `cont` field):

$$(\exists \mathsf{X}.\ \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{X} \quad * \quad \mathsf{list(x, X)}) \qquad \leadsto \qquad (\exists \mathsf{X}.\ \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{X} \quad * \quad \mathsf{list(x, X)})$$

This protocol permits any atomic update to the region containing the internal state of bag x from a state satisfying the left side of $\leadsto$ to a state satisfying the right side.

To allow the library to update `cont` in synchronization points, we therefore transfer the library's half-permission to the client and require the client to update the phantom field with the abstract effects of the method, and then transfer a half-permission back to the library. When the client updates the phantom field, the client is forced to prove that the abstract effects of the method is permitted by whatever protocols the client may have imposed on the abstract state.

We express the update to the phantom `cont` field using a *view-shift* [4]. Conceptually, a view-shift corresponds to a step in the instrumented semantics that does not change the concrete machine state. View-shifts, written $\mathsf{P} \sqsubseteq \mathsf{Q}$, thus generalize assertion implication by allowing updates to phantom fields (given sufficient permission) and ownership transfer between the local state and shared regions.

The bag push method thus requires the client to provide a view-shift, to update the abstract state from X to $\mathsf{X} \cup \{\mathsf{y}\}$ in the synchronization point:

$$\frac{\forall \mathsf{X}.\ \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{X} * \mathsf{P} \sqsubseteq \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{X} \cup \{\mathsf{y}\} * \mathsf{Q}}{\{\mathsf{bag(x)} * \mathsf{P}\}\mathsf{x.Push(y)}\{\mathsf{bag(x)} * \mathsf{Q}\}}$$

Here, P and Q are universally quantified and thus picked by the client. Hence, the client can use P and Q to perform further updates of the instrumented state in the synchronization point and relate the new abstract state with its local state. We thus refer to P and Q as synchronization pre- and postconditions.

Likewise, the bag pop operation requires two view-shifts; one, in case the bag is empty in the synchronization point, and another, in case the bag is non-empty in the synchronization point:

$$\frac{\begin{array}{c}\mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \emptyset * \mathsf{P} \sqsubseteq \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \emptyset * \mathsf{Q}(\mathsf{null}) \\ \forall \mathsf{X}.\ \forall \mathsf{y}.\ \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{X} \cup \{\mathsf{y}\} * \mathsf{P} \sqsubseteq \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{X} * \mathsf{Q}(\mathsf{y})\end{array}}{\{\mathsf{bag}(\mathsf{x}) * \mathsf{P}\}\mathsf{x.Pop}()\{\mathsf{bag}(\mathsf{x}) * \mathsf{Q}(\mathsf{ret})\}}$$

Finally, the bag predicate is freely duplicable:

$$\mathsf{bag}(\mathsf{x}) \Rightarrow \mathsf{bag}(\mathsf{x}) * \mathsf{bag}(\mathsf{x})$$

Note that since P and Q are universally quantified — our logic is *higher order* — the client could potentially pick instantiations referring to the library's region, thus introducing self-referential region assertions. We can illustrate this problem by instantiating P with an assertion that itself refers to the bag in the specification of `Push`. Since $\mathsf{bag}(\mathsf{x})$ asserts that there exists a shared region that owns half the $\mathsf{x}_{\mathsf{cont}}$ field, it follows that $\mathsf{bag}(\mathsf{x}) * \mathsf{x}_{\mathsf{cont}} \mapsto \_ \Rightarrow \mathsf{false}$. Hence, by instantiating P with $\mathsf{bag}(\mathsf{x}) * \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \_$, we can derive the postcondition false from the specification of `Push`.

To prevent this, we introduce a notion of *region type* and a notion of *support*, as an over-approximation of the types of regions a given assertion refers to. Our formal bag specification (presented in Section 3) thus imposes support restrictions on P and Q to ensure the client does not introduce self-referential region assertions.

Another key challenge we address is higher-order protocols. Higher-order protocols are crucial to allow clients to associate ownership of additional resources with shared data structures. For example, to derive the shared bag specification from the generic specification, we use a second region with a protocol that requires clients to transfer ownership of $\mathsf{P}(\mathsf{x})$, when pushing x into the bag:

$$(\exists \mathsf{X}.\ \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{X} \ * \ \circledast_{\mathsf{y} \in \mathsf{X}} \mathsf{P}(\mathsf{y})) \quad \rightsquigarrow \quad (\exists \mathsf{X}.\ \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{X} \ * \ \circledast_{\mathsf{y} \in \mathsf{X}} \mathsf{P}(\mathsf{y}))$$

Again, P is a predicate variable and could be instantiated to refer to the state and protocol of this and other regions – making the above protocol a higher-order protocol. We also use region types to break a circularity introduced by higher-order protocols. In particular, instead of assigning protocols to individual regions, we assign parameterized protocols to region types. This allows us to reason about higher-order protocols that refer to the region types – and thus, implicitly, the protocol – of other regions. We show that this well-behaved subset of higher-order protocols, called *state-independent* protocols, suffices for sophisticated libraries, such as the Joins library [16].

To summarize, our new logic and specification methodology allows clients to refine the usage protocol of the bag. It also allows clients to transfer ownership of resources to the bag, by transferring them to a client region synchronized with the abstract state of the bag.

Related work. Jacobs and Piessens introduced the idea of parameterizing the specification of concurrent methods with ghost code, to be executed in synchronization points [10]. Here we build on their idea, using a much stronger logic based on CAP [5], to address the main problem with their approach.

Instead of regions with protocols, Jacobs and Piessens use ghost objects – data structures built from ghost variables – with handles that represent partial information about the data structure and permissions to modify it. While these handles provide support for reasoning about the state of shared ghost objects, they lack the ability to associate ownership of additional state with ghost objects. Instead, Jacobs and Piessens use the lock invariant of the lock protecting the concurrent data structure to associate ownership of additional state.

However, this approach is problematic without proper storable locks. In particular, Jacobs and Piessens logic and model of storable locks only supports lock labels parameterized over simple types (i.e., not assertions). This forces the *client* to create the synchronization primitive, so that the *client* can pick a lock invariant containing both the state of the concurrent data structure and any additional resources the client may wish to associated with the data structure. This breaks abstraction, by exposing internal implementation details to the client (the synchronization primitive used) and it requires the client to reprove the shared bag specification every time it is needed. Hence, Jacobs and Piessens cannot derive the shared bag specification. We solve this problem using higher-order protocols.

CAP was designed to verify concurrent data structures [5]. However, the original specifications and proofs are non-modular in the sense that implementations have been verified against unrefinable specifications with fixed usage protocols.

Recently, Dodds et. al. introduced a higher-order variant of CAP to give a generic specification for a library for deterministic parallelism [6]. While their proofs make explicit use of nested region assertions and higher-order protocols, the authors failed to recognize the semantic difficulties these features introduce. Consequently, their reasoning is unsound. In particular, their higher-order representation predicates are not stable. See Appendix C for concrete counter-examples.

Another approach for achieving modular reasoning is to prove concurrent implementations to be contextual refinements of coarse-grained counterparts – thus taking the coarse-grained counterparts as specifications. Previous efforts for proving such contextual refinements have mostly focused on indirect proofs through a linearizability property on traces of concurrent libraries [9, 7]. So far, this approach lacks support for transfer of ownership of resources between client and library. More recently, there has been work on proving such contextual refinements directly, using logical relations [21]. Unless combined with a program logic, both of these approaches restrict all reasoning to statements about contextual refinement or contextual equivalence. As our approach demonstrates, if a Hoare-style specification is what we are ultimately interested in, then contextual refinement is unnecessary; what we really want is a generic specification that is refinable by clients.

Conceptually, linearizability aims to provide a fiction of atomicity to clients of concurrent libraries. Our approach does not. Instead, we aim to allow clients to reason about changes of the abstract state in synchronization points *inside* concurrent libraries. To illustrate the distinction, consider an extension of the bag library with a `Push2(x, y)` method that takes two elements and pushes them one at a time (i.e., with the implementation `Push(x); Push(y)`). This method is not linearizable, as it has two synchronization points. However, it still has a natural specification expressed in terms of two view-shifts, one for each synchronization point:

$$\frac{\begin{array}{c}\forall \mathsf{X}.\ \mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} \mathsf{X} * \mathsf{P} \sqsubseteq \mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} \mathsf{X} \cup \{\mathsf{y}\} * \mathsf{Q} \\ \forall \mathsf{X}.\ \mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} \mathsf{X} * \mathsf{Q} \sqsubseteq \mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} \mathsf{X} \cup \{\mathsf{z}\} * \mathsf{R}\end{array}}{\{\mathsf{bag}(\mathsf{x}) * \mathsf{P}\}\mathsf{x.Push2(y,z)}\{\mathsf{bag}(\mathsf{x}) * \mathsf{R}\}}$$

From this specification, a client can derive a natural shared bag specification:

$$\{\mathsf{bag}_s(\mathsf{x}, \mathsf{P}) * \mathsf{P}(\mathsf{y}) * \mathsf{P}(\mathsf{z})\}\mathsf{x.Push2(y,\ z)}\{\mathsf{bag}_s(\mathsf{x}, \mathsf{P})\}$$

Contributions. We propose a new style of specification for thread-safe concurrent data structures. Using protocol synchronization, this style of specification allows clients to refine the usage protocol of concurrent data structures. Moreover, using nested region assertions and state-independent higher-order protocols, our specification style allows clients to associate additional resources with the data structure.

Technically, we realize the ideas by developing HOCAP, a higher-order separation logic for a subset of C$^\sharp$ featuring named delegates and fork concurrency. The logic allows two or more protocols to be synchronized and evolve in lock-step. In addition, we support nested region assertions, state-independent higher-order protocols, and guarded recursive assertions. We present a step-indexed model of the logic and use it to prove the logic sound. We emphasize that unlike earlier versions of CAP, our logic includes sufficient proof rules for carrying out all proofs (including stability proofs) of examples *in the logic*, i.e., without passing to the semantics.

Lastly, we demonstrate the power and utility of the logic by verifying a library for executing tasks in parallel, based on Doug Lea's Fork/Join framework [12]. We have also used the logic to specify and verify the Joins library [16] and clients thereof, which will be described in a separate paper.

2 The logic

Our logic is a general program logic for a subset of C$^\sharp$, featuring delegates referring to named methods[4] and an atomic compare-and-swap statement. New

[4] Anonymous delegates in C$^\sharp$ may capture the l-values of free variables and hence the semantics and logic for anonymous methods is non-trivial, see our earlier paper [19].

threads are allocated via a fork statement that forks a delegate. Each thread has a private stack, but all threads share a common heap. We use an interleaving semantics.

The specification logic is an intuitionistic higher-order logic over a simply typed term language, and the assertion logic an intutionistic higher-order separation logic over the same simply typed term language. Types are closed under the usual type constructors, $\rightarrow$, $\times$, and $+$. Basic types include the type of assertions, Prop, the type of specifications, Spec, the type of C$^\sharp$ values, Val, and the type of fractional permissions, Perm.

2.1 Concurrent Abstract Predicates

Recall that the basic idea behind CAP is to provide an abstraction of possible interference from concurrently executing threads, by partitioning the state into regions, with protocols governing how the state in each region is allowed to evolve. Requiring all assertions to be *stable* – i.e., closed under protocols – and proving all specifications with respect to arbitrary stable frames, then achieves thread-local reasoning about shared mutable state.

Following earlier work on CAP [5], we use a shared region assertion, written $\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},\mathsf{a}}$, which asserts that r is a region and that the resources in region r satisfy the assertion P. Unlike earlier versions, the region assertion is also annotated with a region type t and a protocol argument a, since we assign parameterized protocols to region types instead of regions, as mentioned above. Region assertions are freely duplicable and thus satisfy,

$$\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},\mathsf{a}} \Leftrightarrow \boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},\mathsf{a}} * \boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},\mathsf{a}} \tag{1}$$

Protocols consist of *named actions* and updates to a shared region require *ownership* of a named action justifying the update. Protocols are specified using protocol assertions, written $\mathsf{protocol}(\mathsf{t}, \mathsf{I})$. Here t is a region type and I is a parametric protocol. We use the following notation for a parametric protocol I with parameter a and named actions α_1, ..., α_n:

$$\mathsf{I}(\mathsf{a}) = (\alpha_1 : (\varDelta_1).\ \mathsf{P}_1 \rightsquigarrow \mathsf{Q}_1; \cdots ; \alpha_n : (\varDelta_n).\ \mathsf{P}_n \rightsquigarrow \mathsf{Q}_n)$$

Here $\varDelta_i$ is a context of logical variables relating the action precondition P_i with the action postcondition Q_i. The action α_i thus allows updates from states satisfying P_i to states satisfying Q_i. We use $\mathsf{I}(\mathsf{a})[\alpha_i]$ to refer to the definition of the α_i action in protocol I applied to argument a. Hence, $\mathsf{I}(\mathsf{a})[\alpha_i] = (\varDelta_i).\ \mathsf{P}_i \rightsquigarrow \mathsf{Q}_i$. We use $\boxed{\mathsf{P}}_{\mathsf{I}}^{\mathsf{r},\mathsf{t},\mathsf{a}}$ as shorthand for $\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},\mathsf{a}} * \mathsf{protocol}(\mathsf{t}, \mathsf{I})$.

We can distinguish different client roles in protocols through ownership of named actions. An action assertion $[\alpha]_\pi^{\mathsf{r}}$ asserts fractional ownership of the named action α on region r with fraction π. Fractions are used to allow multiple

Those semantic issues are orthogonal to what we discuss in the present paper and hence we omit anonymous delegates here.

clients to use the same action. We can split or reassemble action assertions using the following property,

$$[\alpha]^{\mathsf{r}}_{\mathsf{p}+\mathsf{q}} \Leftrightarrow [\alpha]^{\mathsf{r}}_{\mathsf{p}} * [\alpha]^{\mathsf{r}}_{\mathsf{q}} \tag{2}$$

where $\mathsf{p}, \mathsf{q}, \mathsf{p}+\mathsf{q}$ are terms of type Perm – permissions in $(0, 1]$.

An assertion p is *stable* if it is closed under interference from the environment. In the absence of self-referential region assertions and higher-order protocols, the region assertion, $\boxed{\mathsf{P}}^{\mathsf{r,t,a}}_{\mathsf{I}}$ is stable if P is closed under all $\mathsf{I}(\mathsf{a})$ actions:[5]

$$\forall \tilde{\mathsf{y}}.\ \mathsf{valid}(\mathsf{P} \wedge \mathsf{P}_i(\tilde{\mathsf{y}}) \Rightarrow \bot) \vee \mathsf{valid}(\mathsf{Q}_i(\tilde{y}) \Rightarrow \mathsf{P})$$

for all i, where $\mathsf{I}(\mathsf{a})[\alpha_i] = (\tilde{\mathsf{x}}).\ \mathsf{P}_i(\tilde{\mathsf{x}}) \rightsquigarrow \mathsf{Q}_i(\tilde{\mathsf{x}})$.

Example. To illustrate reasoning about sharing, consider a counter with read and increment methods. Since the count can only be increased, this counter satisfies the specification of a monotonic counter [15]:

$$\begin{array}{lll} \{\mathsf{counter}(\mathsf{x}, \mathsf{n})\} & \mathsf{x.Increment}() & \{\mathsf{counter}(\mathsf{x}, \mathsf{n}+1)\} \\ \{\mathsf{counter}(\mathsf{x}, \mathsf{n})\} & \mathsf{x.Read}() & \{\mathsf{ret.\ counter}(\mathsf{x}, \mathsf{ret}) * \mathsf{n} \leq \mathsf{ret}\} \end{array}$$

$$\mathsf{counter}(\mathsf{x}, \mathsf{n}) \Rightarrow \mathsf{counter}(\mathsf{x}, \mathsf{n}) * \mathsf{counter}(\mathsf{x}, \mathsf{n})$$

Here $\mathsf{counter}(\mathsf{x}, \mathsf{n})$ asserts that n is a lower-bound on the current count. Hence we expect that this predicate can be freely duplicated, as expressed by the third line above.

To verify a counter implementation against this specification, we place the current count in a shared region, with a protocol that allows the current count to be increased. Assertions about lower bounds are thus invariant under the protocol. If the counter implementation maintains the current count in field count, then we can specify the counter protocol as follows:

$$\mathsf{counter}(\mathsf{x}, \mathsf{n}) \stackrel{\text{def}}{=} \exists \mathsf{r}, \pi.\ [\textsc{incr}]^{\mathsf{r}}_{\pi} * \boxed{\exists \mathsf{m}.\ \mathsf{n} \leq \mathsf{m} * \mathsf{x.count} \mapsto \mathsf{m}}^{\mathsf{r,Counter,x}}_{\mathsf{I}}$$

where I is a parametric protocol with parameter x and a single action $\textsc{incr}$, that allows the count field of x to be increased:

$$\mathsf{I}(\mathsf{x}) = (\textsc{incr} : (\mathsf{m}, \mathsf{k} : \mathsf{N}).\ \mathsf{x.count} \mapsto \mathsf{m} * \mathsf{m} \leq \mathsf{k} \rightsquigarrow \mathsf{x.count} \mapsto \mathsf{k})$$

Here we have used a fixed region type $\mathsf{Counter}$ for the counter region r. Since fractional permissions can always be split (2), and region assertions always duplicated (1), it follows that $\mathsf{counter}(\mathsf{x}, \mathsf{n}) \Rightarrow \mathsf{counter}(\mathsf{x}, \mathsf{n}) * \mathsf{counter}(\mathsf{x}, \mathsf{n})$, as required by the specification. Since the shared region assertion in $\mathsf{counter}(\mathsf{x}, \mathsf{n})$ contains no self-referential region assertions or higher-order protocols, to prove it stable, it suffices to show that,

$$\begin{array}{l} \forall \mathsf{m}, \mathsf{k}.\ \mathsf{valid}((\exists \mathsf{m} : \mathsf{N}.\ \mathsf{n} \leq \mathsf{m} * \mathsf{x.count} \mapsto \mathsf{m}) \wedge (\mathsf{x.count} \mapsto \mathsf{m} * \mathsf{m} \leq \mathsf{k}) \Rightarrow \bot) \vee \\ \qquad \mathsf{valid}(\mathsf{x.count} \mapsto \mathsf{k} \Rightarrow (\exists \mathsf{m} : \mathsf{N}.\ \mathsf{n} \leq \mathsf{m} * \mathsf{x.count} \mapsto \mathsf{m})) \end{array}$$

[5] This is a formula in the specification logic; P and Q are assertions and for an assertion P, $\mathsf{valid}(\mathsf{P})$ is the specification that expresses that P is valid in the assertion logic.

This follows easily by case analysis on $\mathsf{n} \leq \mathsf{k}$. Lastly, to verify the implementation of $\mathsf{Increment}$ and Read, we have to prove they satisfy the protocol, namely that they do not decrease the current count. This is easy.

2.2 Higher-order Concurrent Abstract Predicates

As the above example illustrates, we can use CAP to reason about a shared counter by imposing a protocol on the shared count field. Since this is a protocol on a primitive resource (the count field), first-order CAP suffices. To reason about examples, such as the shared bag, which associates ownership of general resources – through the P predicate – with a shared bag, we need Higher-Order CAP. In particular, to define the bag_s predicate requires region and protocol assertions containing the predicate variable P.

To support modular reasoning about region and protocol assertions containing predicate and assertion variables, ideally, we want to treat predicate and assertion variables as black boxes. For instance, consider the assertion,

$$\mathsf{Q} \stackrel{\text{def}}{=} \boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},-} * \mathsf{protocol}(\mathsf{t},\mathsf{I}) \tag{3}$$

where I is the parametric protocol $\mathsf{I}(-) = (\tau : \mathsf{P} \rightsquigarrow \mathsf{P})$ expressed in terms of the assertion variable P. Treating P as a black box, Q is clearly stable if P is stable, as Q asserts that P holds of the resources in region r, which is clearly closed under the protocol I. However, in general P could itself be instantiated with region and protocol assertions, introducing the possibility of *self-referential region assertions* and turning I into a *higher-order protocol.* This makes reasoning significantly more challenging. In particular, some self-referential region assertions do not admit modular stability proofs: it is possible to instantiate P with stable assertions for which Q is not stable. Furthermore, higher-order protocols introduce a circularity in the definition of the model.

Self-referential region assertions. To see how self-referential region assertions can break the modularity of stability proofs, consider assertion P below:

$$\mathsf{P} \stackrel{\text{def}}{=} \mathsf{x} \mapsto 0 * \boxed{\mathsf{y} \mapsto 0}^{\mathsf{r}',\mathsf{t}',-} * \mathsf{protocol}(\mathsf{t}',\mathsf{J}),$$

where J is the protocol with a single α action that allows the y variable to be changed from 0 to 1, provided region r owns variable x and x is zero:

$$\mathsf{J}(-) = \left(\alpha : \boxed{\mathsf{x} \mapsto 0}^{\mathsf{r},\mathsf{t},-} * \mathsf{y} \mapsto 0 \rightsquigarrow \boxed{\mathsf{x} \mapsto 0}^{\mathsf{r},\mathsf{t},-} * \mathsf{y} \mapsto 1\right)$$

Then P is stable, because P asserts full ownership of the x variable, ensuring that the environment cannot perform the α action, as x cannot also be owned by region r. However, the region assertion Q defined above is not stable when instantiated with this P, as $\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},-}$ asserts that region r *does* own x, thus allowing the environment to perform the α action. As this example illustrates, some self-referential region assertions thus do not admit modular stability proofs. A similar problem occurs when reasoning about atomic updates to shared regions.

Support. To ensure modular reasoning about stability and atomic updates to shared regions, we require clients to explicitly prove that their instantiations of predicate variables do not introduce self-referential region assertions. To facilitate these proofs, we introduce a notion of support, which gives an over-approximation of the types of regions a given assertion refers to.

An assertion P is supported by a set of region types A, if P is invariant under arbitrary changes to the state and protocol of any region of a region type not in A. To support modular reasoning about hierarchies of concurrent libraries, instead of reasoning directly in terms of sets of regions, we introduce a partial order on region types and reason in terms of upwards-closed sets of region types. More formally, we introduce a new type, RType, of region types with a partial order $\leq$: RType $\times$ RType $\to$ Spec, with a bottom element $\bot$: RType and finite meets. We say that an assertion P is dependent on region type t if it is supported by the set of region types greater than or equal to t. We introduce two new specification assertions, $\mathsf{dep}, \mathsf{indep}$: RType $\times$ Prop $\to$ Spec for asserting that an assertion is dependent and independent of a given region type, respectively. Figure 3 in Appendix A contains a set of natural inference rules for dep and indep, for instance, one expressing that if P is dependent on region type t_1, then $\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t}_2,\mathsf{a}}$ is dependent on the greatest lower bound, of t_1 and t_2.

Whenever we reason about region assertions, $\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},\mathsf{a}}$ we thus require that P is independent of the region type t. This excludes self-referential region assertions through protocols (such as in (3)), and through nested region assertions (such as $\boxed{\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},\mathsf{a}}}^{\mathsf{r},\mathsf{t},\mathsf{a}}$).

Stability. General higher-order protocols would introduce a circularity in the definition of the model. We break this circularity by exploiting the indirection of region types – i.e., that we assign protocols to region types instead of individual regions. This allows us to support protocols with assertions about the region types of regions, but without assertions about the protocols assigned to those region types. Technically, we enforce this restriction by ignoring protocol assertions in action pre- and postconditions when interpreting protocols. The parameterized higher-order protocol I,

$$\mathsf{I}(\mathsf{x}) = (\mathsf{x} \mapsto 0 * \mathsf{protocol}(\mathsf{t}, \mathsf{J}) \leadsto \mathsf{x} \mapsto 1 * \mathsf{protocol}(\mathsf{t}, \mathsf{J}))$$

is thus interpreted as $\mathsf{I}(\mathsf{x}) = (\mathsf{x} \mapsto 0 \leadsto \mathsf{x} \mapsto 1)$. The interpretation simply ignores the $\mathsf{protocol}(\mathsf{t}, \mathsf{J})$ assertion (See definition of *act* in the technical report [20]).

In the absence of self-referential region assertions, a region assertion $\boxed{\mathsf{P}}_{\mathsf{I}}^{\mathsf{r},\mathsf{t},\mathsf{a}}$ is stable under the α action, if P is closed under the action pre- and postcondition of the α action of $\mathsf{I}(\mathsf{a})$ and I is a first-order protocol. If I is a higher-order protocol, then the assertion $\boxed{\mathsf{P}}_{\mathsf{I}}^{\mathsf{r},\mathsf{t},\mathsf{a}}$ is stable under the α action, if P is closed under the action pre- and postcondition of the α action of $\mathsf{I}(\mathsf{a})$ and P is also *protocol-pure*.

We thus have the following proof rule for stability:

$$\frac{\begin{array}{c}\mathsf{I}(\mathsf{a})[\alpha] = (\tilde{\mathsf{x}}).\mathsf{I}_p(\tilde{\mathsf{x}}) \rightsquigarrow \mathsf{I}_q(\tilde{\mathsf{x}}) \qquad \forall \tilde{\mathsf{x}}.\ \mathsf{valid}(\mathsf{P} \wedge \mathsf{I}_p(\tilde{\mathsf{x}}) \Rightarrow \bot) \vee \mathsf{valid}(\mathsf{I}_q(\tilde{\mathsf{x}}) \Rightarrow \mathsf{P}) \\ \mathsf{indep}_\mathsf{t}(\mathsf{P}) \qquad \mathsf{indep}_\mathsf{t}(\mathsf{Q}) \qquad \mathsf{stable}(\mathsf{P} * \mathsf{Q}) \qquad \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}) \qquad \mathsf{pure}_{\mathsf{state}}(\mathsf{Q})\end{array}}{\mathsf{stable}^{\mathsf{r}}_{\alpha}\left(\boxed{\mathsf{P}}^{\mathsf{r,t,a}}_{\mathsf{I}} * \mathsf{Q}\right)}\ \mathrm{SA}$$

Here $\mathsf{pure}_{\mathsf{protocol}}$ and $\mathsf{pure}_{\mathsf{state}}$ are propositions in the specification logic; $\mathsf{pure}_{\mathsf{protocol}}(\mathsf{P})$ expresses that P is invariant under any changes to protocols and $\mathsf{pure}_{\mathsf{state}}(\mathsf{P})$ expresses that P is invariant under any change to the local or shared state. The SA proof rule thus allows us to prove stability of region assertions, by first "pulling out" any protocol assertions, Q, from the region assertion. We say that an assertion is *expressible using state-independent protocols* if the protocol assertions can be "pulled out" in this sense. Formally,

$$\mathsf{sip} \stackrel{\text{def}}{=} \lambda\mathsf{P} : \mathsf{Prop}.\ \exists \mathsf{Q}, \mathsf{R} : \mathsf{Prop}.\ \mathsf{valid}(\mathsf{P} \Leftrightarrow \mathsf{Q} * \mathsf{R}) \wedge \mathsf{pure}_{\mathsf{protocol}}(\mathsf{Q}) \wedge \mathsf{pure}_{\mathsf{state}}(\mathsf{R})$$

In particular, if $\mathsf{P} \Leftrightarrow \mathsf{Q} * \mathsf{R}$ and $\mathsf{pure}_{\mathsf{state}}(\mathsf{R})$, then $\boxed{\mathsf{P}}^{\mathsf{r,t,a}}_{\mathsf{I}} \Leftrightarrow \boxed{\mathsf{Q}}^{\mathsf{r,t,a}}_{\mathsf{I}} * \mathsf{R}$. Thus, if $\mathsf{sip}(\mathsf{P})$, then $\boxed{\mathsf{P}}^{\mathsf{r,t,a}}_{\mathsf{I}}$ can be rewritten to a form that satisfies the $\mathsf{pure}_{\mathsf{protocol}}$ premise of the SA rule. Expressibility using state-independent protocols is closed under conjunction and separating conjunction, but in general not under disjunction or existential quantification. To achieve closure under existential quantification, $\exists \mathsf{x} : \mathsf{X}.\ \mathsf{P}(\mathsf{x})$, we have to impose a stronger restriction on the predicate family P. Namely, P has to be uniformly expressible using state-independent protocols:

$$\begin{aligned}\mathsf{usip}_{\mathsf{X}} \stackrel{\text{def}}{=} \lambda\mathsf{P} : \mathsf{X} \to \mathsf{Prop}.\ \exists \mathsf{R} : \mathsf{Prop}.\ \exists \mathsf{Q} : \mathsf{X} \to \mathsf{Prop}.\ \mathsf{pure}_{\mathsf{state}}(\mathsf{R}) \wedge \\ \forall \mathsf{x} \in \mathsf{X}.\ (\mathsf{P}(\mathsf{x}) \Leftrightarrow \mathsf{Q}(\mathsf{x}) * \mathsf{R}) \wedge \mathsf{pure}_{\mathsf{protocol}}(\mathsf{Q}(x))\end{aligned}$$

Then we have that $\mathsf{usip}_{\mathsf{X}}(\mathsf{P}) \Rightarrow \mathsf{sip}(\exists \mathsf{x} \in \mathsf{X}.\ \mathsf{P}(\mathsf{x}))$.

2.3 View-shifts.

Phantom state. Proofs in Hoare logic often employ auxiliary variables [13], as an abstraction of the history of execution and state. To support this style of reasoning, without changing the formal operational semantics, we instrument our abstract semantics with phantom fields.

We thus extend our logic with a phantom points-to assertion, written $\mathsf{x}_\mathsf{f} \overset{\mathsf{p}}{\mapsto} \mathsf{v}$, which asserts partial ownership, with fraction p, of the phantom field f on object x, and that the current value of the phantom field is v.

Phantom fields live in the instrumented state and are thus updated through view-shifts. Updating a phantom field requires full ownership of the field ($\mathsf{x}_\mathsf{f} \overset{1}{\mapsto} \mathsf{v}_1 \sqsubseteq_\bot \mathsf{x}_\mathsf{f} \overset{1}{\mapsto} \mathsf{v}_2$).[6] A fractional phantom field permission can be split and reassembled arbitrarily. As a partial fraction only confers read-only ownership, two partial fractional assertions must agree on the current value of a given phantom

[6] The view-shift is annotated with the $\bot$ region type; we explain the reason for such annotations on view-shifts in the following.

field ($\mathsf{x_f} \overset{\mathsf{p_1}}{\mapsto} \mathsf{v_1} * \mathsf{x_f} \overset{\mathsf{p_2}}{\mapsto} \mathsf{v_2} \Rightarrow \mathsf{v_1} = \mathsf{v_2}$). To create a phantom field f we require that the field does not already exist, so that we can take full ownership of the field. We thus require all phantom fields of an object o to be created simultaneously when o is first constructed (in the proof rule for constructors, see the technical report [20]).

Simultaneous updates. To support synchronization of two regions by splitting ownership of a common phantom field, we need to update the value of the phantom field in both regions *simultaneously*. Previous versions of CAP have only supported sequences of *independent* updates to *single* regions. To support synchronization of protocols we thus extend CAP with support for simultaneous updates of *multiple* regions.

We have chosen a semantics that requires that updates of regions have the same action granularity (you cannot have one simultaneous update of two regions, where the update of one region is justified by one action, and the update of the other region is justified by two actions). This is a choice; it simplifies stability proofs, but it means that we must explicitly track the regions that may have been updated by a view-shift. (See Section B in Appendix for examples illustrating this choice.) We thus index the view-shift relation with a region type t. The indexed view-shift relation, $\sqsubseteq_{\mathsf{t}}$, thus describes a *single* update that, in addition to updating the local state, may update *multiple* shared regions with region types not greater than or equal to t, where each update must be justified by a *single* action. The indexed view-shift relation is thus *not* transitive.

Figure 1 contains a selection of proof rules for view-shifts. The two main rules, VSNOpen and VSOpen, are used to open a region, to allow access to the resources in that shared region. Both rules allow us to open a region and perform a nested view-shift on the contents of that region. This is how we reason about simultaneous updates to multiple regions in the logic. Rule VSNOpen allows the nested view-shift to modify further regions, while VSOpen does not (note the use of region type $\bot$ on the nested view shift in VSOpen). Both rules require a proof the update is possible –

$$\mathsf{P_1} * \mathsf{P_2} \sqsubseteq_{\mathsf{t_1} \sqcap \mathsf{t_2}} \mathsf{Q_1} * \mathsf{Q_2} \qquad \text{and} \qquad \mathsf{P_1} * \mathsf{P_2} \sqsubseteq_{\bot} \mathsf{Q_1} * \mathsf{Q_2},$$

respectively – and a proof that the update is allowed by the protocol, denoted

$$\boxed{\mathsf{P_1}}_{\mathsf{I}}^{\mathsf{r,t_1,a}} * \mathsf{P_2} \rightsquigarrow^{\mathsf{r,t_2}} \boxed{\mathsf{Q_1}}_{\mathsf{I}}^{\mathsf{r,t_1,a}} * \mathsf{Q_2}$$

and explained below.

Since actions owned by shared regions cannot be used to perform updates to shared regions, the VSNOpen rule further requires that $\mathsf{P_1}$ does not assert ownership of any local action permissions ($\mathsf{pure_{perm}}(\mathsf{P_1})$). This ensures that no local action permissions from $\mathsf{P_1}$ were used to justify any actions performed in the nested view-shift. Since VSOpen does not allow the nested view-shift to update any regions, this restriction is unnecessary for the VSOpen rule.

Update allowed. The update allowed relation, $\mathsf{P} \rightsquigarrow^{\mathsf{r,t}} \mathsf{Q}$, asserts that the update described by P and Q to region r is justified by an action owned by P.

$$\frac{\begin{array}{c}\mathsf{pure}_{\mathsf{perm}}(\mathsf{P}_1) \qquad \mathsf{indep}_{\mathsf{t}_1 \sqcap \mathsf{t}_2}(\mathsf{P}_1, \mathsf{P}_2, \mathsf{Q}_1, \mathsf{Q}_2) \qquad \mathsf{t}_2 \not\leq \mathsf{t}_1 \\ \boxed{\mathsf{P}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\mathsf{I}} * \mathsf{P}_2 \rightsquigarrow^{\mathsf{r},\mathsf{t}_2} \boxed{\mathsf{Q}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\mathsf{I}} * \mathsf{Q}_2 \qquad \mathsf{P}_1 * \mathsf{P}_2 \sqsubseteq_{\mathsf{t}_1 \sqcap \mathsf{t}_2} \mathsf{Q}_1 * \mathsf{Q}_2\end{array}}{\boxed{\mathsf{P}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\mathsf{I}} * \mathsf{P}_2 \sqsubseteq_{\mathsf{t}_2} \boxed{\mathsf{Q}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\mathsf{I}} * \mathsf{Q}_2} \text{ VSNOpen}$$

$$\frac{\begin{array}{c}\mathsf{indep}_{\mathsf{t}_1 \sqcap \mathsf{t}_2}(\mathsf{P}_1, \mathsf{P}_2, \mathsf{Q}_1, \mathsf{Q}_2) \qquad \mathsf{t}_2 \not\leq \mathsf{t}_1 \\ \boxed{\mathsf{P}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\mathsf{I}} * \mathsf{P}_2 \rightsquigarrow^{\mathsf{r},\mathsf{t}_2} \boxed{\mathsf{Q}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\mathsf{I}} * \mathsf{Q}_2 \qquad \mathsf{P}_1 * \mathsf{P}_2 \sqsubseteq_{\bot} \mathsf{Q}_1 * \mathsf{Q}_2\end{array}}{\boxed{\mathsf{P}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\mathsf{I}} * \mathsf{P}_2 \sqsubseteq_{\mathsf{t}_2} \boxed{\mathsf{Q}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\mathsf{I}} * \mathsf{Q}_2} \text{ VSOpen}$$

$$\frac{\mathsf{P} \sqsubseteq_{\mathsf{t}} \mathsf{Q} \qquad \mathsf{stable}(\mathsf{R})}{\mathsf{P} * \mathsf{R} \sqsubseteq_{\mathsf{t}} \mathsf{Q} * \mathsf{R}} \text{ VSFrame} \qquad\qquad \frac{\mathsf{P} \sqsubseteq_{\mathsf{t}_1} \mathsf{Q} \qquad \mathsf{t}_1 \leq \mathsf{t}_2}{\mathsf{P} \sqsubseteq_{\mathsf{t}_2} \mathsf{Q}} \text{ VSWeaken}$$

Fig. 1. Selected view-shift proof rules

Thus the basic proof rule for the update allowed relation is:

$$\frac{\mathsf{indep}_{\mathsf{t}_2}(\mathsf{P}(\tilde{\mathsf{v}}), \mathsf{Q}(\tilde{\mathsf{v}})) \qquad \mathsf{t}_2 \not\leq \mathsf{t}_1 \qquad \mathsf{I}(\mathsf{a})[\alpha] = (\tilde{\mathsf{x}}).\ \mathsf{P}(\tilde{\mathsf{x}}) \rightsquigarrow \mathsf{Q}(\tilde{\mathsf{x}})}{\boxed{\mathsf{P}(\tilde{\mathsf{v}})}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\mathsf{I}} * [\alpha]^{\mathsf{r}}_{\pi} \rightsquigarrow^{\mathsf{r},\mathsf{t}_2} \boxed{\mathsf{Q}(\tilde{\mathsf{v}})}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\mathsf{I}} * [\alpha]^{\mathsf{r}}_{\pi}} \text{ UAAct}$$

Since the update allowed relation simply asserts that any update described by P and Q is allowed, it satisfies a slightly non-standard rule of consequence, that allows strengthening of both the pre- and postcondition. From this non-standard rule-of-consequence, it follows that the update allowed relation satisfies a frame rule that allows arbitrary changes to the context:

$$\frac{\mathsf{P} \Rightarrow \mathsf{P}' \qquad \mathsf{P}' \rightsquigarrow^{\mathsf{r},\mathsf{t}} \mathsf{Q}' \qquad \mathsf{Q} \Rightarrow \mathsf{Q}'}{\mathsf{P} \rightsquigarrow^{\mathsf{r},\mathsf{t}} \mathsf{Q}} \text{ UAConseq} \qquad\qquad \frac{\mathsf{P} \rightsquigarrow^{\mathsf{s},\mathsf{t}} \mathsf{Q}}{\mathsf{P} * \mathsf{R}_1 \rightsquigarrow^{\mathsf{s},\mathsf{t}} \mathsf{Q} * \mathsf{R}_2} \text{ UAF}$$

3 Concurrent Bag

We now return to the concurrent bag from the introduction, and show how to formalize the informal specification from the introduction. Next, we show how to derive the two bag specifications from the introduction, using protocol synchronization, nested region assertions, and higher-order protocols.

Specification. In the introduction we proposed a refineable bag specification with phantom variables to force protocol synchronization and with view-shifts to synchronize client and library in synchronization points. In the formal specification we restrict the synchronization pre- and postconditions, P and Q, using region types, to ensure that the client's instantiation does not introduce self-referential region assertions. Upon creation of new bag instances, the client picks a region type t for that bag instance and the client is then required to prove that all its synchronization pre- and postconditions are independent of region type t. The formal refinable bag specification is:

$$\frac{}{\{\mathsf{emp}\}\mathsf{new\ Bag}()\{\mathsf{ret.\ bag}(\mathsf{t},\mathsf{ret}) * \mathsf{ret}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \emptyset\}}$$

$$\frac{\begin{array}{c}\mathsf{stable}(\mathsf{P}) \quad \mathsf{stable}(\mathsf{Q}) \quad \mathsf{indep}_{\mathsf{t}}(\mathsf{P}) \quad \mathsf{indep}_{\mathsf{t}}(\mathsf{Q}) \\ \forall \mathsf{x}.\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \emptyset * \mathsf{P}(\mathsf{x}) \sqsubseteq_{\mathsf{t}} \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \emptyset * \mathsf{Q}(\mathsf{x},\mathsf{null}) \\ \forall \mathsf{X}.\ \forall \mathsf{x},\mathsf{y}.\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} \cup \{\mathsf{y}\} * \mathsf{P}(\mathsf{x}) \sqsubseteq_{\mathsf{t}} \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{Q}(\mathsf{x},\mathsf{y})\end{array}}{\{\mathsf{bag}(\mathsf{t},\mathsf{x}) * \mathsf{P}(\mathsf{x})\}\mathsf{x.Pop}()\{\mathsf{ret.\ bag}(\mathsf{t},\mathsf{x}) * \mathsf{Q}(\mathsf{x},\mathsf{ret})\}}$$

$$\frac{\begin{array}{c}\mathsf{stable}(\mathsf{P}) \quad \mathsf{stable}(\mathsf{Q}) \quad \mathsf{indep}_{\mathsf{t}}(\mathsf{P}) \quad \mathsf{indep}_{\mathsf{t}}(\mathsf{Q}) \\ \forall \mathsf{X}.\ \forall \mathsf{x},\mathsf{y}.\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{P}(\mathsf{x},\mathsf{y}) \sqsubseteq_{\mathsf{t}} \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} \cup \{\mathsf{y}\} * \mathsf{Q}(\mathsf{x},\mathsf{y})\end{array}}{\{\mathsf{bag}(\mathsf{t},\mathsf{x}) * \mathsf{P}(\mathsf{x},\mathsf{y})\}\mathsf{x.Push}(\mathsf{y})\{\mathsf{bag}(\mathsf{t},\mathsf{x}) * \mathsf{Q}(\mathsf{x},\mathsf{y})\}}$$

$$\frac{}{\mathsf{bag}(\mathsf{t},\mathsf{x}) \Leftrightarrow \mathsf{bag}(\mathsf{t},\mathsf{x}) * \mathsf{bag}(\mathsf{t},\mathsf{x})} \qquad \frac{}{\mathsf{dep}_{\mathsf{t}}(\mathsf{bag}(\mathsf{t},\mathsf{x}))}$$

The $\mathsf{indep}_{\mathsf{t}}$ assumptions on the synchronization pre- and postconditions ensure that P and Q do not introduce self-referential region assertions. Furthermore, the index on the view-shifts, $\sqsubseteq_{\mathsf{t}}$, ensures that the granularity of actions match between the library and any client protocols.

See Appendix 3.1 for a proof outline of a fine-grained bag implementation against this refineable bag specification.

Exclusive owner. We now show how to derive the standard specification with a single exclusive owner. This specification is very simple to derive; we simply let the exclusive owner of the bag keep the 1/2 permission of the phantom field containing the abstract state of the bag: $\mathsf{bag}_e(\mathsf{t},\mathsf{x},\mathsf{X}) \overset{\text{def}}{=} \mathsf{bag}(\mathsf{t},\mathsf{x}) * \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X}$.

Shared bag. The derivation of the shared bag specification is more interesting, as it uses both protocol synchronization and higher-order protocols. We begin by formalizing the shared bag specification in our logic:

$$\frac{\mathsf{dep}_{\mathsf{r}}(\mathsf{P})}{\mathsf{dep}_{\mathsf{r}\sqcap\mathsf{t}}(\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P}))} \qquad \frac{\mathsf{stable}(\mathsf{P}) \quad \mathsf{indep}_{\mathsf{t}}(\mathsf{P}) \quad \mathsf{usip}_{\mathrm{Val}}(\mathsf{P})}{\{\mathsf{emp}\}\mathsf{new\ Bag}()\{\mathsf{ret.\ bag}_s(\mathsf{t},\mathsf{ret},\mathsf{P})\}}$$

$$\frac{}{\{\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P}) * \mathsf{P}(\mathsf{y})\}\mathsf{x.Push}(\mathsf{y})\{\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P})\}}$$

$$\frac{}{\{\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P})\}\mathsf{x.Pop}()\{\mathsf{ret.\ bag}_s(\mathsf{t},\mathsf{x},\mathsf{P}) * (\mathsf{ret} = \mathsf{null} \ \vee \ \mathsf{P}(\mathsf{ret}))\}}$$

$$\frac{}{\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P}) \Leftrightarrow \mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P}) * \mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P})}$$

This corresponds to the specification from the introduction, except with restrictions on predicate P to ensure it is expressible using state-independent protocols and does not introduce self-referential protocol or region assertions.

With these restrictions on P we can now derive the shared bag specification from our generic specification. The idea is to introduce a new region containing the state associated with each element currently in the bag:

$$\begin{aligned}\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P}) &\stackrel{\text{def}}{=} \exists \mathsf{r} : \mathrm{RId}.\ \exists \pi : \mathrm{Perm}.\ \exists \mathsf{t}_1, \mathsf{t}_2 : \mathrm{RType}.\\ &\qquad \mathsf{t} \leq \mathsf{t}_1 \land \mathsf{t} \leq \mathsf{t}_2 \land \mathsf{t}_1 \not\leq \mathsf{t}_2 \land \mathsf{t}_2 \not\leq \mathsf{t}_1 \land \mathsf{indep}_{\mathsf{t}}(\mathsf{P}) \land \mathsf{usip}(\mathsf{P}) \land\\ &\qquad \mathsf{bag}(\mathsf{t}_1, \mathsf{x}) * \boxed{\mathsf{q}(\mathsf{x},\mathsf{P})}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * [\textsc{Upd}]^{\mathsf{r}}_{\pi}\\ \mathsf{q}(\mathsf{x},\mathsf{P}) &\stackrel{\text{def}}{=} \exists \mathsf{X} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} \mathsf{X} * \circledast_{\mathsf{y}\in\mathsf{X}} \mathsf{P}(\mathsf{y})\\ \mathsf{I}(\mathsf{P})(\mathsf{x}) &\stackrel{\text{def}}{=} (\textsc{Upd} : \mathsf{q}(\mathsf{x},\mathsf{P}) \rightsquigarrow \mathsf{q}(\mathsf{x},\mathsf{P}))\end{aligned}$$

The parametric protocol $\mathsf{I}(\mathsf{P})$ allows the bag to be changed arbitrarily, provided the region still contains the state associated with each element currently in the bag. From the assumption that each $\mathsf{P}(\mathsf{x})$ is stable and that $\mathsf{usip}_{\mathrm{Val}}(\mathsf{P})$ it follows that $\mathsf{q}(\mathsf{x},\mathsf{P})$ is stable and $\mathsf{sip}(\mathsf{q}(\mathsf{x},\mathsf{P}))$. Hence, there exists $\mathsf{R},\mathsf{S} : \mathsf{Prop}$ such that $\mathsf{q}(\mathsf{x},\mathsf{P}) \Leftrightarrow \mathsf{R} * \mathsf{S}$, $\mathsf{pure}_{\mathsf{protocol}}(\mathsf{S})$ and $\mathsf{pure}_{\mathsf{state}}(\mathsf{R})$. Thus, $\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P})$ is equivalent to the following assertion:

$$\exists r, \pi, \mathsf{t}_1, \mathsf{t}_2.\ \mathsf{t} \leq \mathsf{t}_1 \land \mathsf{t} \leq \mathsf{t}_2 \land \mathsf{t}_1 \not\leq \mathsf{t}_2 \land \mathsf{t}_2 \not\leq \mathsf{t}_1 \land \mathsf{bag}(\mathsf{t}_1,\mathsf{x}) * \boxed{\mathsf{S}}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * \mathsf{R} * [\textsc{Upd}]^{\mathsf{r}}_{\pi}$$

Hence, to prove $\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P})$ stable, it suffices to prove stability of $\boxed{\mathsf{S}}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * \mathsf{R}$. Applying rule SA, it thus suffices to prove,

$$\mathsf{valid}(\mathsf{q}(\mathsf{x},\mathsf{P}) \land \mathsf{S} \Rightarrow \bot) \lor \mathsf{valid}(\mathsf{q}(\mathsf{x},\mathsf{P}) \Rightarrow \mathsf{S})$$

and the right disjunct follows easily from the assumption that $\mathsf{q}(\mathsf{x},\mathsf{P}) \Leftrightarrow \mathsf{R} * \mathsf{S}$.

To derive the shared bag specification for `push`, we thus have to transfer the resources associated with the element being pushed, $\mathsf{P}(\mathsf{y})$, to the client region containing the element resources. We thus instantiate P and Q in the generic bag specification with $\mathsf{P}(\mathsf{y}) * \boxed{\mathsf{q}(\mathsf{x},\mathsf{P})}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * [\textsc{Upd}]^{\mathsf{r}}_{\pi}$ and $\boxed{\mathsf{q}(\mathsf{x},\mathsf{P})}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * [\textsc{Upd}]^{\mathsf{r}}_{\pi}$, respectively.

We thus have to provide a view-shift to synchronize the abstract state of the library protocol with our client protocol r:

$$\begin{aligned}\forall \mathsf{X} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} \mathsf{X} * \mathsf{P}(\mathsf{y}) * \boxed{\mathsf{q}(\mathsf{x},\mathsf{P})}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * [\textsc{Upd}]^{\mathsf{r}}_{\pi} &\sqsubseteq_{\mathsf{t}_1}\\ \mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} (\mathsf{X} \cup \{\mathsf{y}\}) * \boxed{\mathsf{q}(\mathsf{x},\mathsf{P})}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * [\textsc{Upd}]^{\mathsf{r}}_{\pi}&\end{aligned}$$

Since $\mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} \mathsf{X} * \mathsf{P}(\mathsf{y}) * [\textsc{Upd}]^{\mathsf{r}}_{\pi}$ and $\mathsf{q}(\mathsf{x},\mathsf{P})$ are all independent of region type t, by rule VSOpen it suffices to prove that the change to region r is allowed and possible. The update is easily shown to be allowed by the Upd action, using the UAAct rule and update action frame rule (UAF). To show the possibility of the view shift it suffices to prove that:

$$\begin{aligned}\mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} \mathsf{X} * \mathsf{P}(\mathsf{y}) * \exists \mathsf{Z} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} \mathsf{Z} * \circledast_{\mathsf{z}\in\mathsf{Z}} \mathsf{P}(\mathsf{z}) * [\textsc{Upd}]^{\mathsf{r}}_{\pi} &\sqsubseteq_{\bot}\\ \mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} (\mathsf{X} \cup \{\mathsf{y}\}) * \exists \mathsf{Z} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{x}_{\mathsf{cont}} \xrightarrow{1/2} \mathsf{Z} * \circledast_{\mathsf{z}\in\mathsf{Z}} \mathsf{P}(\mathsf{z}) * [\textsc{Upd}]^{\mathsf{r}}_{\pi}&\end{aligned}$$

which follows easily, as $\mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{X} * \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{Z} \Rightarrow \mathsf{X} = \mathsf{Z}$.

Note that to provide a view-shift to synchronize the abstract state of the library protocol with the client protocol, we were essentially forced to update the phantom field cont in the client region, which in turn forced us to transfer ownership of $\mathsf{P(y)}$ to the client region.

3.1 Proof of Bag specification

In the previous section we showed how to derive the exclusive owner and shared bag specification from the refineable bag specification. In this section we sketch how to prove that a concurrent bag implementation satisfies the refinable specification. In particular, we sketch a proof of a non-locking concurrent bag, implemented using compare-and-swap.

Representation predicates. As mentioned in the introduction, the idea is to share the *concrete state* of the concurrent bag using a shared region. To allow the client to refine the specification, we store the *abstract state* of the concurrent bag in a phantom field and let the client keep a half-permission to the phantom field. The library protocol thus has to relate the concrete state of the concurrent bag with its abstract state (i.e., a multiset of elements).

In our implementation, the bag is represented as a singly-linked list. An abstract state given by a multiset of elements X is thus related to a concrete state containing a singly-linked list with the elements of X. We can express this relation between the concrete and abstract state as follows:

$$\mathsf{q(x)} \overset{\text{def}}{=} \exists \mathsf{h} : \text{Val}.\ \exists \mathsf{l} : \textit{seq}\ \text{Val}.\ \mathsf{x.head} \mapsto \mathsf{h} * \mathsf{lst}_r(\mathsf{h}, \mathsf{l}) * \mathsf{x}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{mem(l)}$$

where lst_r is the following list-representation predicate:

$$\begin{aligned} \mathsf{lst}_r(\mathsf{x}, \varepsilon) &\overset{\text{def}}{=} \mathsf{x} = \mathsf{null} \\ \mathsf{lst}_r(\mathsf{x}, \mathsf{v} :: \mathsf{l}) &\overset{\text{def}}{=} \exists \mathsf{y} : \text{Val}.\ \mathsf{x.value} \overset{-}{\mapsto} \mathsf{v} * \mathsf{x.next} \overset{-}{\mapsto} \mathsf{y} * \mathsf{lst}_r(\mathsf{y}, \mathsf{l}) \end{aligned}$$

and $\mathsf{mem} : \textit{seq}\ \text{Val} \to \mathcal{P}_m(\text{Val})$ returns the multiset of elements of the given sequence. The bag predicate thus asserts that there exists a shared region containing the concrete and abstract state of the bag, which is currently related by q, and a protocol that enforces that the concrete and abstract state is always related by q:

$$\begin{aligned} \mathsf{bag(t, x)} \overset{\text{def}}{=}\ & \exists \mathsf{r} : \text{RId}.\ \exists \pi \in \text{Perm}.\ \exists \mathsf{s} : \text{RType}. \\ & \mathsf{t} \leq \mathsf{s} \wedge \boxed{\mathsf{q(x)}}^{\mathsf{r,s,x}} * \mathsf{protocol(s, I)} * [\textsc{Upd}]^{\mathsf{r}}_{\pi} \end{aligned}$$

where I is the following parametric protocol:

$$\mathsf{I(x)} = (\textsc{Upd} : \mathsf{q(x)} \rightsquigarrow \mathsf{q(x)})$$

Note that the lst_r representation predicate only asserts read-only ownership of the underlying singly-linked list[7]. By only asserting read-only ownership of the underlying singly-linked list, we ensure that once an element has been added to the list, its next-field never changes; this is used in the correctness proof of `Pop`. The bag predicate is trivially stable using the SA rule, and freely duplicable.

Proof outline.

```
using Interlocked;

internal class Node⟨A⟩ {
  internal Node⟨A⟩ next;
  internal A value;

  public Node(A value) {
$\{\mathbf{this}.\mathsf{next} \mapsto \mathsf{null} * \mathbf{this}.\mathsf{value} \mapsto \mathsf{null}\}$
    this.value = value;
$\{\mathbf{this}.\mathsf{next} \mapsto \mathsf{null} * \mathbf{this}.\mathsf{value} \mapsto \mathsf{value}\}$
  }
}

public class Bag⟨A⟩ where A : class {
  internal Node⟨A⟩ head;

  public Bag() {
$\{\mathbf{this}.\mathsf{head} \mapsto \mathsf{null} * \mathbf{this}_{\mathsf{cont}} \mapsto \emptyset\}$
$\{\exists \mathsf{l}, \mathsf{x}.\ \mathbf{this}.\mathsf{head} \mapsto \mathsf{x} * \mathbf{this}_{\mathsf{cont}} \xmapsto{1/2} \mathsf{mem}(\mathsf{l}) * \mathsf{lst}_r(\mathsf{x}, \mathsf{l}) * \mathbf{this}_{\mathsf{cont}} \xmapsto{1/2} \emptyset\}$
$\{\mathsf{bag}(\mathsf{t}, \mathbf{this}) * \mathbf{this}_{\mathsf{cont}} \xmapsto{1/2} \emptyset\}$
  }
```

In the bag constructor, we introduce a phantom field cont, a region type s, and a region with region type s, initialized with ownership of the head field and a half-permission to the cont field.

```
  public void Push(A x) {
    Node⟨A⟩ nHead; Node⟨A⟩ oHead; Node⟨A⟩ tmp;
$\{\mathsf{bag}(\mathsf{t}, \mathbf{this}) * \mathsf{P}\}$
$\{\mathsf{t} \leq \mathsf{s} * \boxed{\mathsf{q}(\mathbf{this})}^{\mathsf{r},\mathsf{s},\mathbf{this}}_{\mathsf{I}} * [\textsc{Upd}]^{\mathsf{r}}_{\pi} * \mathsf{P}\}$
    nHead = new Node⟨A⟩(x);
$\{\mathsf{t} \leq \mathsf{s} * \boxed{\mathsf{q}(\mathbf{this})}^{\mathsf{r},\mathsf{s},\mathbf{this}}_{\mathsf{I}} * [\textsc{Upd}]^{\mathsf{r}}_{\pi} * \mathsf{P} * \mathsf{nHead}.\mathsf{next} \mapsto \_ * \mathsf{nHead}.\mathsf{value} \mapsto \mathsf{x}\}$
    do {
      oHead = this.head;
      nHead.next = oHead;
```

[7] We use $\mathsf{x.f} \overset{-}{\mapsto} \mathsf{v}$ as shorthand for $\exists \mathsf{p} \in \mathrm{Perm}.\ \mathsf{x.f} \overset{\mathsf{p}}{\mapsto} \mathsf{v}$.

```
$\{t \le s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * P * nHead.next \mapsto oHead * nHead.value \mapsto x\}$
      tmp = CompareExchange⟨Node⟨A⟩⟩(ref head, nHead, oHead);
$\{t \le s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * ((tmp = oHead * Q) \vee$
    $(tmp \ne oHead * P * nHead.next \mapsto oHead * nHead.value \mapsto x))\}$
    } while (tmp != oHead);
$\{t \le s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * Q\}$
$\{bag(t, \mathbf{this}) * Q\}$
  }
```

The Push method allocates a new node and inserts it at the front of the list, using compare-and-swap to atomically update the head field. The synchronization point of this method, if it terminates, occurs when the compare-and-swap succeeds. At this point, we transfer ownership of the newly allocated node to the shared region, and use the client-provided view-shift to update the phantom cont field (thus synchronizing with any protocols the client may have imposed on the abstract state). To verify the atomic compare-and-swap, we use a proof rule corresponding to VSNOpen, for “opening” the shared r region to perform a nested *atomic update* (rule OpenA in the accompanying technical report).

```
  public A Pop() {
    Node⟨A⟩ oHead; Node⟨A⟩ nHead; Node⟨A⟩ tmp;
    A res; bool done;
$\{bag(t, \mathbf{this}) * P\}$
    done = false;
$\{bag(t, \mathbf{this}) * ((done = false * P) \vee (done = true * Q(res)))\}$
    while(!done) {
$\{bag(t, \mathbf{this}) * done = false * P\}$
$\{t \le s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * done = false * P\}$
      oHead = this.head;
$\{t \le s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * done = false$
    $* ((oHead = null * Q(null)) \vee (\exists l.\ oHead \ne null * lst_{r}(oHead, l) * P))\}$
      if (oHead == null) {
$\{t \le s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * done = false * Q(null)\}$
        res = null;
        done = true;
$\{bag(t, \mathbf{this}) * done = true * Q(res)\}$
      } else {
$\{\exists l.\ t \le s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * done = false * oHead \ne null * lst_{r}(oHead, l) * P\}$
$\{\exists l.\ t \le s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * done = false$
    $* oHead.next \overset{-}{\mapsto} y * oHead.value \overset{-}{\mapsto} v * lst_{r}(y, l) * P\}$
        nHead = oHead.next;
$\{\exists l.\ t \leq s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * done = false$
    $* oHead.next \overset{-}{\mapsto} nHead * oHead.value \overset{-}{\mapsto} v * lst_{r}(nHead, l) * P\}$
        tmp = CompareExchange⟨Node⟨A⟩⟩(ref head, nHead, oHead);
$\{t \leq s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * done = false$
    $* ((tmp = oHead * Q(v) * oHead.value \overset{-}{\mapsto} v) \vee (tmp \neq oHead * P))\}$
        if (tmp == oHead) {
$\{t \leq s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * done = false * Q(v) * oHead.value \overset{-}{\mapsto} v\}$
          res = oHead.value;
          done = true;
$\{bag(t, \mathbf{this}) * done = true * Q(res)\}$
        } else {
$\{t \leq s * \boxed{q(\mathbf{this})}^{r,s,\mathbf{this}}_{I} * [\textsc{Upd}]^{r}_{\pi} * done = false * P\}$
$\{bag(t, \mathbf{this}) * done = false * P\}$
        }
      }
    }
$\{bag(t, \mathbf{this}) * Q(res)\}$
    return res;
$\{ret.\ bag(t, \mathbf{this}) * Q(ret)\}$
  }
}
```

4 Concurrent Runner

The concurrent runner is a small library for parallelizing divide-and-conquer algorithms, inspired by Doug Lea's Fork/Join framework [12]. Clients interact with the library by registering a delegate for parallelization. The library then provides methods for scheduling executions of the registered delegate using a set of worker threads. In general, this delegate could have side-effects. The registered delegate is also allowed to schedule itself for execution, leading to recursion through the store through the library. The concurrent runner thus makes for an interesting specification challenge, as it requires a specification that supports higher-order code with effects and recursion through the store through the library. It also makes for an interesting verification challenge, as the library is implemented using shared mutable state. In particular, the implementation uses a shared bag to share tasks scheduled for execution between the worker threads. To demonstrate that our approach scales, we have verified the concurrent runner. The proof uses the shared bag specification derived in the previous section.

The API for the library is given in Figure 2. The library consists of a runner class for parallelizing executions of a delegate. The constructor takes as argument a delegate and the number of worker threads to create. Calling `Fork` on a concurrent runner schedules the registered delegate for execution with the given argument. `Fork` further returns a task representing the scheduled computation. Calling `Join` on the returned task causes the caller to wait until the given task has completed. While the caller is waiting, it turns into a worker thread, helping to execute tasks from the underlying pool. Once the given task has completed, `Join` returns the return-value of the terminated delegate.

```
public class Runner⟨A,B⟩
  public Runner(Func⟨Runner⟨A,B⟩,A,B⟩ body, int n)
  public Task⟨A,B⟩ Fork(A a)

public class Task⟨A,B⟩
  public B Join()
```

Fig. 2. Concurrent Runner API

Note that the runner constructor takes as argument a delegate, which itself takes as argument a runner. This is to allow the delegate to schedule parallel executions of itself on smaller sub-problems. In particular, when a scheduled task with argument `a` on a runner `r` is chosen for execution, the delegate is called with runner `r` and argument `a`. The delegate can thus schedule itself for execution by calling `Fork` on its runner argument. We can thus implement a concurrent version of a naive fibonacci divide-and-conquer computation as follows:

```
public int parFib(Runner⟨int,int⟩ f, int a) {
  if (a < 25) return seqFib(a);
  else {
    Task⟨int, int⟩ t1 = f.Fork(a − 1);
    Task⟨int, int⟩ t2 = f.Fork(a − 2);
    return t1.Join() + t2.Join();
  }
}

public int fib(int n, int a) {
  return (new Runner⟨int, int⟩(parFib, n)).Fork(a).Join();
}
```

Here `fib` creates a new concurrent runner with `n` worker threads and schedules `parFib` for execution with argument `a`. `parFib` itself computes the fibonacci number of its argument `a`. When `a` is below a certain threshold, it uses a sequential fibonacci implementation, `seqFib`, to avoid the overhead of scheduling a task. Otherwise, it schedules itself recursively to compute the desired result.

The reason we have chosen this API, which requires the client to choose a fixed delegate upon constructing the runner, instead of letting `Fork` take the delegate as argument, is that this API better illustrates the challenges of verifying the Joins library.

Specification. To verify the concurrent runner implementation, we first need a specification. We want a specification that supports concurrent execution of effectful delegates. The idea behind the concurrent runner specification is to let the client pick a specification for the delegate, and require that the client proves the delegate satisfies the given specification. To allow the delegate to schedule new executions of itself for execution, we explicitly allow the delegate to assume it is called with a concurrent runner with a delegate that satisfies the chosen specification. The delegate specification simply consists of a precondition P, indexed by the argument, and a postcondition Q, indexed by the argument and return value. The precondition describes the resources the delegate needs to perform its task, while the postcondition describes the resources the delegate transfers back to the owner of the task. We can express this formally as follows:

$$\frac{\mathsf{indep_t(P)} \qquad \mathsf{indep_t(Q)} \qquad \mathsf{stable(P)} \qquad \mathsf{stable(Q)} \qquad \mathsf{usip(P)} \qquad \mathsf{usip(Q)}}{\begin{array}{l}\left\{\begin{array}{l}\mathsf{b} \mapsto (\mathsf{r},\mathsf{a}).\ \{\mathsf{runner(t,r,P,Q)} * \mathsf{P(a)}\} \\ \qquad\qquad \{\mathsf{ret.\ runner(t,r,P,Q)} * \mathsf{Q(a,ret)}\}\end{array}\right\} \\ \qquad \mathsf{new\ Runner(b,n)} \\ \{\mathsf{ret.\ runner(t,ret,P,Q)}\}\end{array}}$$

Upon creation of a new concurrent runner the client thus chooses a delegate precondition P and postcondition Q. The client is then required to prove that the supplied delegate satisfies the chosen specification. This is expressed using a *nested Hoare triple* [17], which asserts that `b` refers to a delegate satisfying the given specification. In the case of the concurrent runner, the delegate is allowed to use the resources described by its precondition, $\mathsf{P(a)}$, and is required to transfer back the resources described by its postcondition, $\mathsf{Q(a,ret)}$. The $\mathsf{runner(t,r,P,Q)}$ assertion further allows the delegate to assume it is called with a concurrent runner `r` with a delegate that satisfies the specification chosen by the client.

To schedule the registered delegate for execution with argument a, the client thus has to transfer the state needed by the delegate, $\mathsf{P(a)}$. When scheduling a delegate, the client is given exclusive ownership of the returned task.

$$\frac{}{\{\mathsf{runner(t,x,P,Q)} * \mathsf{P(arg)}\}\mathsf{x.Fork(arg)}\{\mathsf{ret.\ task(ret,x,P,Q,arg)}\}}$$

Ownership of a task gives the client the right to join the task, and take ownership of the state returned by the scheduled computation.

$$\frac{}{\{\mathsf{runner(t,y,P,Q)} * \mathsf{task(x,y,P,Q,a)}\}\mathsf{x.Join()}\{\mathsf{ret.\ Q(a,ret)}\}}$$

While tasks have exclusive owners, the concurrent runner itself can be freely shared, to allow multiple threads to schedule tasks concurrently.

$$\frac{}{\mathsf{runner(t,x,P,Q)} \Leftrightarrow \mathsf{runner(t,x,P,Q)} * \mathsf{runner(t,x,P,Q)}}$$

With this specification it is easy to show that `parFib` implements fibonacci.

Representation predicates. The concurrent runner library is implemented using shared mutable state. In particular, each concurrent runner maintains a bag of scheduled tasks, which is shared between each worker thread. Upon creation of the concurrent runner, each worker thread enters an infinite loop, in which they keep popping and running tasks from the bag. Tasks can be in one of two states, pending and executed. They start in the pending state and transition to the executed state once they have been popped and executed.

To verify the implementation, we use the shared bag specification to share the bag of tasks between each worker thread. We let each task own the resources needed by the delegate to perform the given task (i.e., the resources described by the delegate precondition P). We can thus define the runner representation predicate as follows:

$$\begin{aligned}
&\mathsf{runner}(\mathsf{t},\mathsf{x},\mathsf{P},\mathsf{Q}) \overset{\text{def}}{=} \exists \mathsf{y},\mathsf{z} : \mathrm{Val}.\ \exists \mathsf{t}_1,\mathsf{t}_2 : \mathrm{RType}.\\
&\quad \mathsf{t} \leq \mathsf{t}_1 \wedge \mathsf{t} \leq \mathsf{t}_2 \wedge \mathsf{t}_1 \not\leq \mathsf{t}_2 \wedge \mathsf{t}_2 \not\leq \mathsf{t}_1 \wedge \mathsf{usip}(\mathsf{P}) \wedge \mathsf{usip}(\mathsf{Q})\\
&\quad \wedge\ \mathsf{stable}(\mathsf{P}) \wedge \mathsf{stable}(\mathsf{Q}) \wedge \mathsf{indep}_\mathsf{t}(\mathsf{P}) \wedge \mathsf{indep}_\mathsf{t}(\mathsf{Q})\\
&\quad *\ \mathsf{x.bag} \overset{-}{\mapsto} \mathsf{y} * \mathsf{x.body} \overset{-}{\mapsto} \mathsf{z} * \mathsf{protocol}(\mathsf{t}_2, \mathsf{I}(\mathsf{Q}))\\
&\quad *\ \mathsf{bag}_s(\mathsf{t}_1, \mathsf{y}, \lambda \mathsf{y} : \mathrm{Val}.\ \mathsf{isTask}(\mathsf{t}_2, \mathsf{y}, \mathsf{x}, \mathsf{P}, \mathsf{Q}))\\
&\quad *\ \triangleright \begin{pmatrix} \mathsf{z} \mapsto (\mathsf{r},\mathsf{a}).\ \{\mathsf{runner}(\mathsf{t},\mathsf{r},\mathsf{P},\mathsf{Q}) * \mathsf{P}(\mathsf{a})\} \\ \qquad\qquad \{\mathsf{ret}.\ \mathsf{runner}(\mathsf{t},\mathsf{r},\mathsf{P},\mathsf{Q}) * \mathsf{Q}(\mathsf{a},\mathsf{ret})\} \end{pmatrix}
\end{aligned}$$

where $\mathsf{I}(\mathsf{Q})$ is the parametric protocol defined after the isTask predicate below. Note that, since we explicitly allow the delegate to assume it is called with a concurrent runner satisfying the chosen specification, the runner representation predicate needs to refer to itself to specify the registered delegate. Hence, we define runner by guarded recursion, as signaled by the use of the so-called later operator, $\triangleright$.

The $\mathsf{isTask}(\mathsf{t},\mathsf{y},\mathsf{x},\mathsf{P},\mathsf{Q})$ representation predicate asserts that y refers to a task in the pending state, and asserts ownership of the resources required to perform the given task. Thus, when a worker thread pops a task from the shared bag, it takes ownership of the resources needed to execute the delegate. Upon termination of the delegate, the resources produced by the delegate must be transferred back to the owner of the task, upon joining the task. To achieve this, we associate a shared region with each task, that owns the field containing the state of the task. This allows us to impose a protocol on the task state, that forces the worker thread to transfer the resources produced by the delegate to this shared region, when changing the state of the task from pending to executed. Lastly, to ensure only the owner of the task can take ownership of the resources, we give the owner of the task exclusive ownership of the action required to do so. We thus define isTask as follows:

$$\begin{aligned}
&\mathsf{isTask}(\mathsf{t},\mathsf{x},\mathsf{y},\mathsf{P},\mathsf{Q}) \overset{\text{def}}{=} \exists \mathsf{r} : \mathrm{RId}.\ \exists \mathsf{a} : \mathrm{Val}.\\
&\quad \mathsf{x.runner} \overset{-}{\mapsto} \mathsf{y} * \mathsf{x.arg} \overset{-}{\mapsto} \mathsf{a} * \mathsf{P}(\mathsf{a}) * \mathsf{protocol}(\mathsf{t}, \mathsf{I}(\mathsf{Q}))\\
&\quad *\ \boxed{\mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \_}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a})} * [\textsc{Exec}]^{\mathsf{r}}_1 * [\textsc{SetRes}]^{\mathsf{r}}_1
\end{aligned}$$

where $\mathsf{I}(\mathsf{Q})$ is the parametric protocol,

$$\mathsf{I}(\mathsf{Q})(\mathsf{x},\mathsf{a}) = \left(\begin{array}{rl} \textsc{SetRes}: & \mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \mathsf{null} \rightsquigarrow \\ & \quad \exists \mathsf{r}:\mathrm{Val}.\ \mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \mathsf{r} \\ \textsc{Exec}: & \exists \mathsf{r}:\mathrm{Val}.\ \mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \mathsf{r} \rightsquigarrow \\ & \quad \exists \mathsf{r}:\mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} * \mathsf{Q}(\mathsf{a},\mathsf{r}) \\ \textsc{Join}: & \exists \mathsf{r}:\mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} * \mathsf{Q}(\mathsf{a},\mathsf{r}) \rightsquigarrow \\ & \quad \exists \mathsf{r}:\mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} \\ \tau_1: & \exists \mathsf{r}:\mathrm{Val}.\ \mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \mathsf{r} \rightsquigarrow \\ & \quad \exists \mathsf{r}:\mathrm{Val}.\ \mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \mathsf{r} \\ \tau_2: & \exists \mathsf{r}:\mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} * \mathsf{Q}(\mathsf{a},\mathsf{r}) \rightsquigarrow \\ & \quad \exists \mathsf{r}:\mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} * \mathsf{Q}(\mathsf{a},\mathsf{r}) \end{array}\right)$$

The owner of the isTask assertion thus owns $\mathsf{P}(\mathsf{a})$ and the full Exec action, to change the state of the task from pending (0) to executed (1), by transferring $\mathsf{Q}(\mathsf{a},\mathsf{r})$ to the shared region. Conversely, the owner of the task asserts full ownership of the Join action, allowing only the owner to grab $\mathsf{Q}(\mathsf{a},\mathsf{r})$, once the task has been executed:

$$\begin{array}{l} \mathsf{task}(\mathsf{x},\mathsf{y},\mathsf{P},\mathsf{Q},\mathsf{a}) \overset{\text{def}}{=} \exists \mathsf{r}:\mathrm{RId}.\ \exists \mathsf{t}:\mathrm{RType}. \\ \quad \mathsf{x.runner} \overset{-}{\mapsto} \mathsf{y} * \mathsf{x.arg} \overset{-}{\mapsto} \mathsf{a} * \mathsf{protocol}(\mathsf{t},\mathsf{I}(\mathsf{Q})) * [\textsc{Join}]_1^{\mathsf{r}} * [\tau_1]_1^{\mathsf{r}} * [\tau_2]_1^{\mathsf{r}} \\ \quad * \boxed{\begin{array}{l} (\mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \_) \vee \\ (\exists \mathsf{r}:\mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} * \mathsf{Q}(\mathsf{a},\mathsf{r})) \end{array}}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a})} \end{array}$$

To ensure that the isTask predicate is expressible using state-independent protocols, we use a parametric task protocol $\mathsf{I}(\mathsf{Q})$ and existentially quantify over the task region type t_2 in the runner predicate instead of the isTask predicate.

Stability. The isTask predicate is trivially stable under the Exec action, as it asserts full ownership of Exec. Its region assertion is also trivially expressible using state-independent protocols. It is thus also stable under the Join action, by rule SA, as $(\mathsf{x.state} \mapsto 0 * ...) \wedge (\mathsf{x.state} \mapsto 1 * ...) \Rightarrow \bot$. As isTask makes no assertions about the value of the res field, it is also easily shown to be stable under the SetRes, τ_1 and τ_2 actions, using rule SA.

Furthermore, the predicate $\lambda \mathsf{y}:\mathrm{Val}.\ \mathsf{isTask}(\mathsf{t},\mathsf{x},\mathsf{y},\mathsf{P},\mathsf{Q})$ is uniformly expressible using state-independent protocols, for any fixed t, y, P and Q, as P is uniformly expressible using state-independent protocols. Stability of runner thus follows easily from the stability of bag_s.

Lastly, task is trivially stable under the Join action, and by rewriting task using the $\mathsf{usip}(\mathsf{Q})$ assumption, it is easily proven to be stable under the Exec, SetRes, τ_1, and τ_2 actions, using the SA rule.

Proof outline. In this section we sketch the proof of concurrent runner implementation, in the form of a proof outline. To keep the size of the outline manageable, we will not write out the assumptions about stability, independence or expressibility using state-independent protocols on P and Q; nor will we write out the assumptions about the ordering on region types t, t_1 and t_2, when working with the runner predicate. Define S_{del} as follows:

$$\mathsf{S}_{del}(\mathsf{t}, \mathsf{b}, \mathsf{P}, \mathsf{Q}) \overset{\text{def}}{=} \triangleright \begin{pmatrix} \mathsf{b} \mapsto (\mathsf{r}, \mathsf{a}).\ \{\mathsf{runner}(\mathsf{t}, \mathsf{r}, \mathsf{P}, \mathsf{Q}) * \mathsf{P}(\mathsf{a})\} \\ \qquad\qquad \{\mathsf{ret}.\ \mathsf{runner}(\mathsf{t}, \mathsf{r}, \mathsf{P}, \mathsf{Q}) * \mathsf{Q}(\mathsf{a}, \mathsf{ret})\} \end{pmatrix}$$

```
public class Runner⟨A,B⟩ {
  internal readonly Func⟨Runner⟨A,B⟩,A,B⟩ body;
  internal readonly Bag⟨Task⟨A,B⟩⟩ bag;

  public Runner(Func⟨Runner⟨A,B⟩, A, B⟩ body, int n) {
    int i;
{$\mathsf{body} \mapsto (\mathsf{r}, \mathsf{a}).\ \{\mathsf{runner}(\mathsf{t}, \mathsf{r}, \mathsf{P}, \mathsf{Q}) * \mathsf{P}(\mathsf{a})\}\{\mathsf{ret}.\ \mathsf{runner}(\mathsf{t}, \mathsf{r}, \mathsf{P}, \mathsf{Q}) * \mathsf{Q}(\mathsf{a}, \mathsf{ret})\}$
    $* \textbf{this}.\mathsf{bag} \mapsto \mathsf{null} * \textbf{this}.\mathsf{body} \mapsto \mathsf{null}$}
    this.bag = new Bag⟨Task⟨A,B⟩⟩();
    this.body = body;
{$\mathsf{S}_{del}(\mathsf{t}, \mathsf{body}, \mathsf{P}, \mathsf{Q}) * \textbf{this}.\mathsf{bag} \bar{\mapsto} \mathsf{b} * \textbf{this}.\mathsf{body} \bar{\mapsto} \mathsf{body}$
    $* \mathsf{bag}_s(\mathsf{t}_1, \mathsf{b}, \lambda \mathsf{y} : \mathrm{Val}.\ \mathsf{isTask}(\mathsf{t}_2, \mathsf{y}, \mathsf{x}, \mathsf{P}, \mathsf{Q}))$}
{$\mathsf{S}_{del}(\mathsf{t}, \mathsf{body}, \mathsf{P}, \mathsf{Q}) * \textbf{this}.\mathsf{bag} \bar{\mapsto} \mathsf{b} * \textbf{this}.\mathsf{body} \bar{\mapsto} \mathsf{body}$
    $* \mathsf{bag}_s(\mathsf{t}_1, \mathsf{b}, \lambda \mathsf{y} : \mathrm{Val}.\ \mathsf{isTask}(\mathsf{t}_2, \mathsf{y}, \mathsf{x}, \mathsf{P}, \mathsf{Q})) * \mathsf{protocol}(\mathsf{t}_2, \mathsf{I}(\mathsf{Q}))$}
{$\mathsf{runner}(\mathsf{t}, \mathsf{ret}, \mathsf{P}, \mathsf{Q})$}
    i = 0;
    while (i < n) {
{$\mathsf{runner}(\mathsf{t}, \mathsf{ret}, \mathsf{P}, \mathsf{Q})$}
{$\mathsf{runner}(\mathsf{t}, \mathsf{ret}, \mathsf{P}, \mathsf{Q}) * \mathsf{runner}(\mathsf{t}, \mathsf{ret}, \mathsf{P}, \mathsf{Q})$}
      new Thread(runTasks).Start();
      i++;
{$\mathsf{runner}(\mathsf{t}, \mathsf{ret}, \mathsf{P}, \mathsf{Q})$}
    }
{$\mathsf{runner}(\mathsf{t}, \mathsf{ret}, \mathsf{P}, \mathsf{Q})$}
  }
```

The proof of the runner constructor is fairly straightforward; since assertions are downwards-closed in the step-index, we have $\mathsf{P} \Rightarrow \triangleright\mathsf{P}$, allowing us to "forget a step". The body and bag fields are never modified are their assignment in the constructor. We can thus freely share read-only access to these fields. Furthermore, nested Hoare triples are freely duplicable, allowing us to duplicate the runner assertion and transfer a runner assertion to each new worker thread. Note that the runner constructor also allocates a new task region type t_2 with the parametric protocol $\mathsf{I}(\mathsf{Q})$.

```
  public Task⟨A,B⟩ Fork(A arg) {
```

```
$\{\mathsf{runner}(t, \mathbf{this}, P, Q) * P(arg)\}$
$\{S_{del}(t, body, P, Q) * \mathbf{this}.bag \mapsto b * \mathbf{this}.body \mapsto body$
    $* \mathsf{bag}_s(t_1, b, \lambda y : \mathrm{Val.}\ \mathsf{isTask}(t_2, y, \mathbf{this}, P, Q))$
    $* \mathsf{protocol}(t_2, I(Q)) * \mathsf{protocol}(t_2, I(Q))\}$
    Task⟨A,B⟩ task = new Task⟨A, B⟩(this, arg);
$\{S_{del}(t, body, P, Q) * \mathbf{this}.bag \mapsto b * \mathbf{this}.body \mapsto body$
    $* \mathsf{bag}_s(t_1, b, \lambda y : \mathrm{Val.}\ \mathsf{isTask}(t_2, y, \mathbf{this}, P, Q)) * \mathsf{protocol}(t_2, I(Q))$
    $* \mathsf{isTask}(t_2, task, \mathbf{this}, P, Q) * \mathsf{task}(task, \mathbf{this}, P, Q, arg)\}$
    tasks.Push(task);
$\{S_{del}(t, body, P, Q) * \mathbf{this}.bag \mapsto b * \mathbf{this}.body \mapsto body$
    $* \mathsf{bag}_s(t_1, b, \lambda y : \mathrm{Val.}\ \mathsf{isTask}(t_2, y, \mathbf{this}, P, Q))$
    $* \mathsf{protocol}(t_2, I(Q)) * \mathsf{task}(task, \mathbf{this}, P, Q, arg)\}$
    return task;
$\{ret.\ \mathsf{task}(ret, \mathbf{this}, P, Q, arg)\}$
  }
```

The `Fork` method constructs a new task, resulting in an isTask assertion – asserting the task is concurrently pending and the resources and permission needed to execute it – and a task assertion – asserting ownership of the task. The fork method transfers the isTask assertion to the shared bag by calling `Push` and returns the task assertion to the caller.

```
  internal void runTasks() {
$\{\mathsf{runner}(t, \mathbf{this}, P, Q)\}$
    while(true) {
$\{\mathsf{runner}(t, \mathbf{this}, P, Q)\}$
      runTask();
$\{\mathsf{runner}(t, \mathbf{this}, P, Q)\}$
    }
$\{\mathsf{runner}(t, \mathbf{this}, P, Q)\}$
  }

  internal void runTask() {
$\{\mathsf{runner}(s, \mathbf{this}, P, Q)\}$
    Task⟨A,B⟩ t = tasks.Pop();
$\{s \leq s_1 * \mathsf{runner}(s, \mathbf{this}, P, Q) * (t = \mathrm{null} \vee \mathsf{isTask}(s_1, t, \mathbf{this}, P, Q))\}$
    if (t != null) {
$\{s \leq s_1 * \mathsf{runner}(s, \mathbf{this}, P, Q) * \mathsf{isTask}(s_1, t, \mathbf{this}, P, Q)\}$
      t.Run();
$\{\mathsf{runner}(s, \mathbf{this}, P, Q)\}$
    }
$\{\mathsf{runner}(s, \mathbf{this}, P, Q)\}$
  }
}
```

The `runTasks` and `runTask` methods are internal methods used by the worker threads to pop and execute tasks. Their proofs are fairly obvious.

```
public class Task⟨A, B⟩ {
  internal readonly Runner⟨A,B⟩ runner;
  internal readonly A arg;
  internal int state = 0;
  internal B res;

  internal Task(Runner<A,B> runner, A arg) {
$\{\mathsf{P}(\mathsf{arg}) * \mathsf{protocol}(\mathsf{t}, \mathsf{I}(\mathsf{Q})) * \mathbf{this}.\mathsf{runner} \mapsto \mathsf{null}$
    $* \mathbf{this}.\mathsf{arg} \mapsto \mathsf{null} * \mathbf{this}.\mathsf{state} = 0 * \mathbf{this}.\mathsf{res} \mapsto \mathsf{null}\}$
    this.runner = runner;
    this.arg = arg;
$\{\mathsf{P}(\mathsf{arg}) * \mathsf{protocol}(\mathsf{t}, \mathsf{I}(\mathsf{Q})) * \mathbf{this}.\mathsf{runner} \mapsto \mathsf{runner}$
    $* \mathbf{this}.\mathsf{arg} \mapsto \mathsf{arg} * \mathbf{this}.\mathsf{state} = 0 * \mathbf{this}.\mathsf{res} \mapsto \mathsf{null}\}$
$\{\mathsf{isTask}(\mathsf{t}, \mathbf{this}, \mathsf{runner}, \mathsf{P}, \mathsf{Q}) * \mathsf{task}(\mathbf{this}, \mathsf{runner}, \mathsf{P}, \mathsf{Q}, \mathsf{arg})\}$
  }
```

In the task constructor, we allocate a new region and transfer ownership of the status and res field to this new region. We explicitly *do not* allocate a new region type, but use the existing task region type t, allocated by the runner constructor.

```
  internal void Run() {
    Func⟨Runner⟨A,B⟩,A,B⟩ body;
    Runner⟨A,B⟩ runner;
    A arg; B tmp;
$\{\mathsf{t} \le \mathsf{t}_2 * \mathsf{runner}(\mathsf{t}, \mathsf{x}, \mathsf{P}, \mathsf{Q}) * \mathsf{isTask}(\mathsf{t}_2, \mathbf{this}, \mathsf{x}, \mathsf{P}, \mathsf{Q})\}$
    runner = this.runner;
    body = this.body;
    arg = this.arg;
$\{\mathsf{t} \le \mathsf{t}_2 * \mathsf{runner}(\mathsf{t}, \mathsf{x}, \mathsf{P}, \mathsf{Q}) * \mathsf{P}(\mathsf{arg}) * [\textsc{SetRes}]^{\mathsf{r}}_{1} * [\textsc{Exec}]^{\mathsf{r}}_{1}$
    $* \boxed{\mathbf{this}.\mathsf{state} \mapsto 0 * \mathbf{this}.\mathsf{res} \mapsto \mathsf{null}}^{\mathsf{r}, \mathsf{t}_2, (\mathbf{this}, \mathsf{arg})} * \mathsf{protocol}(\mathsf{t}_2, \mathsf{I}(\mathsf{Q}))\}$
$\{\mathsf{t} \le \mathsf{t}_2 * \mathsf{runner}(\mathsf{t}, \mathsf{x}, \mathsf{P}, \mathsf{Q}) * \mathsf{S}_{del}(\mathsf{t}, \mathsf{body}, \mathsf{P}, \mathsf{Q}) * \mathsf{P}(\mathsf{arg}) * [\textsc{SetRes}]^{\mathsf{r}}_{1} * [\textsc{Exec}]^{\mathsf{r}}_{1}$
    $* \boxed{\mathbf{this}.\mathsf{state} \mapsto 0 * \mathbf{this}.\mathsf{res} \mapsto \mathsf{null}}^{\mathsf{r}, \mathsf{t}_2, (\mathbf{this}, \mathsf{arg})} * \mathsf{protocol}(\mathsf{t}_2, \mathsf{I}(\mathsf{Q}))\}$
    tmp = body(runner, arg);
$\{\mathsf{t} \le \mathsf{t}_2 * \mathsf{runner}(\mathsf{t}, \mathsf{x}, \mathsf{P}, \mathsf{Q}) * \mathsf{S}_{del}(\mathsf{t}, \mathsf{body}, \mathsf{P}, \mathsf{Q}) * \mathsf{Q}(\mathsf{arg}, \mathsf{tmp}) * [\textsc{SetRes}]^{\mathsf{r}}_{1} * [\textsc{Exec}]^{\mathsf{r}}_{1}$
    $* \boxed{\mathbf{this}.\mathsf{state} \mapsto 0 * \mathbf{this}.\mathsf{res} \mapsto \mathsf{null}}^{\mathsf{r}, \mathsf{t}_2, (\mathbf{this}, \mathsf{arg})} * \mathsf{protocol}(\mathsf{t}_2, \mathsf{I}(\mathsf{Q}))\}$
    this.res = tmp;
$\{\mathsf{t} \le \mathsf{t}_2 * \mathsf{runner}(\mathsf{t}, \mathsf{x}, \mathsf{P}, \mathsf{Q}) * \mathsf{S}_{del}(\mathsf{t}, \mathsf{body}, \mathsf{P}, \mathsf{Q}) * \mathsf{Q}(\mathsf{arg}, \mathsf{tmp}) * [\textsc{SetRes}]^{\mathsf{r}}_{1} * [\textsc{Exec}]^{\mathsf{r}}_{1}$
    $* \boxed{\mathbf{this}.\mathsf{state} \mapsto 0 * \mathbf{this}.\mathsf{res} \mapsto \mathsf{tmp}}^{\mathsf{r}, \mathsf{t}_2, (\mathbf{this}, \mathsf{arg})} * \mathsf{protocol}(\mathsf{t}_2, \mathsf{I}(\mathsf{Q}))\}$
    state = 1;
$\{\mathsf{runner}(\mathsf{t}, \mathsf{x}, \mathsf{P}, \mathsf{Q})\}$
  }
```

To run a task, we first duplicate the nested Hoare triple assertion about the runner delegate, $\mathsf{S}_{del}(...)$. From the isTask assertion, we know the task is pending, we own the resources necessary to execute the delegate and permission to change its status afterwards. Each of the atomic updates, **this**.res = tmp and status = 1, requires a proof that the update satisfies the protocol. See the technical report for proof rules and examples of how we prove such atomic updates.

```
  public B Join() {
    Runner⟨A,B⟩ runner; B tmp;
{$\mathsf{runner}(t, y, P, Q) * \mathsf{task}(\mathbf{this}, y, P, Q, a)$}
    runner = this.runner;
{$\mathsf{runner}(t, \mathsf{runner}, P, Q) * \mathsf{task}(\mathbf{this}, \mathsf{runner}, P, Q, a)$}
    while (state != 1) {
{$\mathsf{runner}(t, \mathsf{runner}, P, Q) * \mathsf{task}(\mathbf{this}, \mathsf{runner}, P, Q, a)$}
      runner.runTask();
{$\mathsf{runner}(t, \mathsf{runner}, P, Q) * \mathsf{task}(\mathbf{this}, \mathsf{runner}, P, Q, a)$}
    }
{$\boxed{\exists r : \mathrm{Val}.\ \mathbf{this}.\mathsf{state} \mapsto 1 * \mathbf{this}.\mathsf{res} \mapsto r * Q(a, r)}^{r, t_2, (\mathbf{this}, a)}$
   $* [\textsc{Join}]^r_1 * [\tau_1]^r_1 * [\tau_2]^r_1 * \mathsf{protocol}(t_2, I(Q))$}
    tmp = this.res;
{$Q(a, \mathsf{tmp})$}
    return tmp;
{$\mathsf{ret}.\ Q(a, \mathsf{ret})$}
  }
}
```

Finally, the `Join` method uses the τ action to continually test whether the task has executed. If this test succeeds, from the definition of the task() predicate, the client knows the shared region contains the resources produced by the task. Using the Join action, the client can thus transfer these resources from the shared region to its local state.

5 Semantics

In this section we sketch the model and the interpretation of our logic. Due to lack of space, we focus on parts presented in Section 2. The full model, interpretation and accompanying soundness proof can be found in the technical report [20].

The presentation of the model is strongly inspired by the Views framework presentation [4]. The model is an instance of the Views framework extended with step-indexing to model guarded recursion, and thread local state to model dynamic allocation of threads.

The basic structure of the model is defined below. Assertions are modeled as step-indexed predicates on instrumented states ($\mathcal{M}$). Instrumented states consist of three components, a local state, a shared state and an action model. The local state specifies the current local resources. The shared state is further partitioned into regions and each region consists of a local state, a region type and a protocol parameter. The action model maps region types to parameterized protocols, which are functions from a tuple containing a protocol argument, a region identifier and an action identifier to an action. Lastly, actions are modeled as certain step-indexed relations on shared states. In particular, actions are *not* relations on shared states *and* action models, and thus do not support general higher-order protocols. Actions do however support state-independent protocols, through the region type indirection.

$$\mathrm{LState} \stackrel{\mathrm{def}}{=} \mathrm{Heap} \times \mathrm{PHeap} \times \mathrm{Cap} \qquad \mathrm{SState} \stackrel{\mathrm{def}}{=} \mathrm{RId} \rightharpoonup (\mathrm{LState} \times \mathrm{RType} \times \mathrm{Val})$$

$$\mathcal{M} \stackrel{\mathrm{def}}{=} \mathrm{LState} \times \mathrm{SState} \times \mathrm{AMod} \qquad \mathrm{AMod} \stackrel{\mathrm{def}}{=} \mathrm{RType} \rightharpoonup ((\mathrm{Val} \times \mathrm{RId} \times \mathrm{AId}) \to \mathrm{Act})$$

$$\mathrm{Cap} \stackrel{\mathrm{def}}{=} \{f \in \mathrm{RId} \times \mathrm{AId} \to [0,1] \mid \exists R \subseteq_{fin} \mathrm{RId}.\ \forall r \in \mathrm{RId} \setminus R.\ \forall \alpha \in \mathrm{AId}.\ f(r,\alpha) = 0\}$$

$$\begin{aligned}
\mathrm{Act} \stackrel{\mathrm{def}}{=} \{R \in \mathcal{P}(\mathsf{N} \times \mathrm{SState} \times \mathrm{SState}) \mid &\\
\forall (i, s_1, s_2) \in R.\ \forall j \leq i.\ \forall r \in \mathrm{RId} \setminus dom(s_2).\ \forall n \in \mathrm{RType}.\ \forall l, l' \in \mathrm{LState}.&\\
s_1 \leq s_2 \wedge (j, s_1, s_2) \in \mathrm{R} \wedge (j, s_1, s_2[r \mapsto (l', n)]) \in R \wedge&\\
(j, s_1[r \mapsto (l, n)], s_2[r \mapsto (l', n)]) \in R\}&\\
\mathsf{Prop} \stackrel{\mathrm{def}}{=} \{U \in \mathcal{P}(\mathsf{N} \times \mathcal{M}) \mid \forall (i, m_1) \in U.\ \forall j \leq i.\ \forall m_2 \in \mathcal{M}.&\\
(m_1 =_j m_2 \vee m_1 \leq m_2) \Rightarrow (j, m_2) \in U\}&\\
\mathrm{Spec} \stackrel{\mathrm{def}}{=} \{U \in \mathcal{P}(\mathsf{N}) \mid \forall i \in U.\ \forall j \leq i.\ j \in U\}&
\end{aligned}$$

The semantics of both the assertion logic and specification logic is step-indexed. The specification logic is step-indexed to allow reasoning about mutual recursion. The assertion logic is step-indexed to support nested triples (which embed specifications in the assertion logic) [18] and guarded recursive predicates [1, 3]. Specifications are thus modeled as downwards closed subsets of numbers, and assertions are modeled as step-indexed predicates on instrumented states, that are downwards closed in the step-index and upwards closed in $\mathcal{M}$. The upwards closure in $\mathcal{M}$ ensures that assertions are closed under allocation of new regions and protocols (the ordering $\leq$ on $\mathcal{M}$ is defined as expected). To define guarded recursive functions and predicates, the types of our logic are modeled as sets with a step-indexed equivalence relation, $=_i$, and terms and predicates are modeled as non-expansive functions. However, as this part of the model is mostly orthogonal to CAP, we will elide the details, which can be found in the technical report [20].

Comparison with previous models of CAP. The original model of (first-order) CAP [5] employed a syntactic treatment of actions to break a circularity in the definition of worlds. Our model follows the previous model of higher-order CAP (without higher-order protocols) [6] in treating actions semantically. However, to support higher-order protocols we introduce a new indirection, in the form of region types. Actions are thus relations on shared states, which include the region types of allocated regions. Actions can thus implicitly refer to the protocol on regions through the region type indirection. While previous work has only considered CAP for a *first-order programming language*, our HOCAP is for a *higher-order programming language*. We thus step-index both the specification and assertion logic, instead of just the specification logic.

Model operations. Separating conjunction is interpreted as the lifting of the partial commutative $\bullet_{\mathcal{M}}$ function to Prop (point-wise in the step-index). The $\bullet_{\mathcal{M}}$

function expresses how to compose two instrumented states. Two instrumented states are combinable if they agree on the shared state and action model, by combining their local states, using $\bullet_{\text{LState}}$. Local states are combined using the standard combination function, $\bullet_{\uplus}$, on disjoint partial functions, on the heap and phantom heap component, and by point-wise summing up the action permissions.

While assertions are modeled as step-indexed predicates on instrumented states, which include phantom fields, protocols, and regions, the operational semantics operates on concrete states, which are simply heaps. The main soundness theorem (Theorem 1) expresses that any step in the concrete semantics has a corresponding step in the instrumented semantics. This is expressed in terms of an erasure function, $\lfloor - \rfloor \in \mathcal{M} \rightharpoonup \text{Heap}$, that erases the instrumentation from an instrumented state. The erasure of an instrumented state is simply the combination of the local state and all shared regions.

$$\lceil (l, s) \rceil \overset{\text{def}}{=} l \bullet_{\text{LState}} \prod_{r \in dom(s)} s(r).l$$

$$\lfloor (l, s, \varsigma) \rfloor \overset{\text{def}}{=} \begin{cases} h & \text{if } (h, ph, c) = \lceil (l, s) \rceil \text{ and } \pi_1(dom(ph)) \subseteq objs(h) \\ \text{undef} & \text{otherwise} \end{cases}$$

Interference. The interference relation $R_i^A \subseteq \mathcal{M} \times \mathcal{M}$ describes possible interference from the environment. It is defined as the reflexive, transitive closure of the single-action interference relation, $\hat{R}_i^A$ (defined below), that describes possible environment interference using at most one action on each region. Defining R_i^A as the reflexive, transitive closure of $\hat{R}_i^A$ forces a common action granularity on updates to multiple regions with protocols referring to each other(see Appendix B). In addition to the step-index $i \in \mathsf{N}$, the single-action interference relation is also indexed by a set $A \in \mathcal{P}(\text{RType})$ of region types of those regions that are allowed to change and that actions justifying those changes are allowed to depend on.

$$\begin{aligned} &(l_1, s_1, \varsigma_1)\ \hat{R}_i^A\ (l_2, s_2, \varsigma_2) \quad \text{iff} \quad l_1 = l_2 \wedge s_1 \leq s_2 \wedge \varsigma_1 \leq \varsigma_2 \wedge \lceil (l_1, s_1) \rceil \text{ defined} \wedge \\ &\quad (\forall r \in dom(s_1).\ s_1(r) = s_2(r) \vee (\exists \alpha.\ s_1(r).t \in A \wedge \\ &\quad\quad (\lceil (l_1, s_1) \rceil .c)(r, \alpha) < 1 \wedge (i, s_1|_A, s_2|_A) \in \varsigma_1(s_1(r).t)(s_1(r).a, r, \alpha))) \end{aligned}$$

$$s|_A \overset{\text{def}}{=} \lambda r \in \text{RId}. \begin{cases} s(r) & \text{if } r \in dom(s) \text{ and } s(r).t \in A \\ \text{undef} & \text{otherwise} \end{cases}$$

In particular, the $\hat{R}_i^A$ relation expresses that the environment is not allowed to change the local state ($l_1 = l_2$), but it is allowed to allocate new regions and protocols ($s_1 \leq s_2$ and $\varsigma_1 \leq \varsigma_2$). Furthermore, the environment is allowed to update the resources of any region r with a region type in A ($s_1(r).t \in A$), provided the update is justified by an action α that is partially owned by the environment ($\lceil (l_1, s_1) \rceil (r, \alpha) < 1$).

An assertion is stable if it is closed under interference to all region types:

$$stable(p) \overset{\text{def}}{=} \{ i \in \mathsf{N} \mid \forall j \leq i.\ \forall (m_1, m_2) \in R_j^{\text{RType}}.\ (j, m_1) \in p \Rightarrow (j, m_2) \in p \}$$

Previous models of CAP have only permitted multiple independent updates, whereas our model supports multiple dependent updates. Previous models thus lack the A-index that we use to enforce a common action granularity on updates to multiple dependent regions.

View-shifts. View-shifts describe a step in the instrumented semantics that correspond to a no-op in the concrete semantics. To perform a view-shift from p to q we thus have to prove that for every concrete state c in the erasure of some instrumented state $m \in p$ there exists an instrumented state $m' \in q$ such that c is in the erasure of m'.

$$p \sqsubseteq_t q \overset{\text{def}}{=} \{i \in \mathsf{N} \mid \forall m \in \mathcal{M}.\ \forall j \in \mathsf{N}.\ 0 \leq j \leq i \ \Rightarrow \\ \lfloor p * \{(j,m)\} \rfloor_j \subseteq \lfloor q * \{(j,m') \mid m\ \hat{R}_j^{\{t' \mid t \not\leq t'\}}\ m'\} \rfloor_j\}$$

To allow framing on view-shifts (rule VSFrame in Section 2.3) we bake in framing under certain stable frames. The frames in question depend on the region index $t \in \text{RType}$. In particular, $\sqsubseteq_t$ permits a single simultaneous update of multiple regions with region types not greater than or equal to t, each justified by a single action. Hence, we require that $\sqsubseteq_t$ is closed under arbitrary frames that are stable under a single simultaneous update of multiple regions with region types not greater than or equal to t, each justified by a single action, i.e., $\hat{R}^{\{t' \mid t \not\leq t'\}}$.

Support. In Section 2.2 we introduced specification logic assertions indep and dep, to internalize a notion of region type support in the logic, to allow explicit proofs of the absence of self-referential region assertions. Their meaning is defined in terms of the following supp assertion, which asserts that p is supported by the set of region types $A \in \mathcal{P}(\text{RType})$. Formally, $\mathsf{supp}_A(p)$ asserts that p is closed under arbitrary shared states that agree on all regions of type A ($s|_A = s'|_A$) and arbitrary action models that are A equivalent ($\varsigma \equiv_A \varsigma'$).

$$\mathsf{supp}_A(p) \overset{\text{def}}{=} \{i \in \mathsf{N} \mid \forall j \leq i.\ \forall (j,(l,s,\varsigma)) \in p.\ \forall s'.\ \forall \varsigma'. \\ s|_A = s'|_A \wedge \varsigma \equiv_A \varsigma' \Rightarrow (j,(l,s',\varsigma')) \in p\}$$

Intuitively, two action models are considered A-equivalent if they agree on the regions of types in A (but they are allowed to differ on regions of types not in A). An assertion p is then dependent on region type $t \in \text{RType}$ if p is supported by the set of region types greater than or equal to t, and independent if it is supported by the set of region types not greater than or equal to t:

$$\mathsf{dep}_t(p) \overset{\text{def}}{=} \mathsf{supp}_{\{t' \mid t \leq t'\}}(p) \qquad\qquad \mathsf{indep}_t(p) \overset{\text{def}}{=} \mathsf{supp}_{\{t' \mid t \not\leq t'\}}(p)$$

Purity. To reason about state-independent protocols and nested view-shifts we have introduced several types of purity; namely, state, protocol and permission purity. Since our assertion logic is intuitionistic, we interpret purity as closure under arbitrary changes to the state, protocols, and permissions, respectively. For instance, $\mathit{pure}_{\mathit{prot}}(p) \overset{\text{def}}{=} \{i \in \mathsf{N} \mid \forall j \leq i.\ \forall (j,(l,s,\varsigma)) \in p.\ \forall \varsigma'.\ (j,(l,s,\varsigma')) \in p\}$.

Soundness. The main soundness theorem expresses that for any derivable Hoare triple, $\{p\}\bar{\mathsf{c}}\{q\}$, if $\bar{\mathsf{c}}$ is executed with a local stack s as thread t, with a global heap h that is in the erasure of some instrumented state in $p(s)$, then, if t (and any threads t may have forked) terminates, then the terminal heap h' is in the erasure of some instrumented state in $q(s')$, where s' is the terminal stack of t.

Theorem 1. *If $\Gamma \vdash (\Delta).\{\mathsf{P}\}\bar{\mathsf{c}}\{\mathsf{Q}\}$ then for all $\vartheta \in [\![\Gamma]\!]$, thread identifiers $t \in TId$, stacks $s \in [\![\Delta]\!]$, and heaps $h \in \lfloor [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, s) \rfloor$, if*

$$(h, \{(t, s, \bar{\mathsf{c}})\}) \to (h', \{(t, s', skip)\} \uplus T')$$

and T' is irreducible then $h' \in \lfloor [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, s') \rfloor$.

6 Conclusion and Future Work

We have proposed a new style of specification for thread-safe data structures that allows the client to refine the specification with a usage protocol, in a concurrent setting. We have shown how to apply it to the bag and concurrent runner example. To realize this style of specification we have presented a new higher-order separation logic with Concurrent Abstract Predicates, that supports state-independent higher-order protocols and synchronization of multiple regions. We have also used the logic to specify and verify Joins, a sophisticated library implemented using higher-order code and shared mutable state.

We have demonstrated that our logic and style of specification scales to implementations of fine-grained concurrent data structures without helping [8]. Future work includes investigating concurrent data structures that use helping.

References

1. A. Appel, P.-A. Melliès, C. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proc. of POPL*, 2007.
2. B. Biering, L. Birkedal, and N. Torp-Smith. BI-Hyperdoctrines, Higher-order Separation Logic, and Abstraction. *ACM TOPLAS*, 2007.
3. L. Birkedal, R. Møgelberg, J. Schwinghammer, and K. Støvring. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *Proc. of LICS*, 2011.
4. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of POPL*, 2013.
5. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of ECOOP*, 2010.
6. M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. In *Proceedings of POPL*, pages 259–270, 2011.
7. I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *Proceedings of ESOP 2009*, pages 252–266, 2009.
8. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

9. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12:463–492, 1990.
10. B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of POPL*, pages 271–282, 2011.
11. J. B. Jensen and L. Birkedal. Fictional Separation Logic. In *Proceedings of ESOP*, pages 377–396, 2012.
12. D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43. ACM, 2000.
13. S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs.* PhD thesis, Cornell, 1975.
14. M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. *SIGPLAN Not.*, 42(1), 2007.
15. A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *Proceedings of TLDI*, pages 73–86, 2011.
16. C. V. Russo. The Joins Concurrency Library. In *Proceedings of PADL*, pages 260–274, 2007.
17. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *Proceedings of CSL*, Apr. 2009.
18. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare Triples and Frame Rules for Higher-Order Store. *LMCS*, 7(3:21), 2011.
19. K. Svendsen, L. Birkedal, and M. Parkinson. Verifying Generics and Delegates. In *Proceedings of ECOOP*, pages 175–199, 2010.
20. K. Svendsen, L. Birkedal, and M. Parkinson. Higher-order Concurrent Abstract Predicates. Technical report, IT University of Copenhagen, 2012. Available at `http://www.itu.dk/people/kasv/hocap-tr.pdf`.
21. A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical Relations for Fine-Grained Concurrency. In *Proceedings of POPL*, 2013.

A Proof rules

$$\frac{}{\mathsf{dep}_{\mathsf{t}}(\bot)} \qquad \frac{}{\mathsf{dep}_{\mathsf{t}}(\top)} \qquad \frac{\mathsf{t}_1 \leq \mathsf{t}_2 \quad \mathsf{dep}_{\mathsf{t}_2}(\mathsf{P})}{\mathsf{dep}_{\mathsf{t}_1}(\mathsf{P})} \qquad \frac{op \in \{\vee, \wedge, *, \Rightarrow\} \quad \mathsf{dep}_{\mathsf{t}}(\mathsf{P}) \quad \mathsf{dep}_{\mathsf{t}}(\mathsf{Q})}{\mathsf{dep}_{\mathsf{t}}(\mathsf{P}\ op\ \mathsf{Q})}$$

$$\frac{\mathsf{dep}_{\mathsf{t}_1}(\mathsf{P})}{\mathsf{dep}_{\mathsf{t}_1 \sqcap \mathsf{t}_2}(\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t}_2,\mathsf{a}})} \qquad \frac{\mathsf{dep}_{\mathsf{t}_1}(\mathsf{I})}{\mathsf{dep}_{\mathsf{t}_1 \sqcap \mathsf{t}_2}(\mathsf{protocol}(\mathsf{t}_2, \mathsf{I}))} \qquad \frac{}{\mathsf{dep}_{\mathsf{t}}([\alpha]^{\mathsf{r}}_{\pi})}$$

$$\frac{\mathsf{dep}_{\mathsf{t}}(\mathsf{P}) \quad Q \in \{\exists, \forall\}}{\mathsf{dep}_{\mathsf{t}}(Q\mathsf{x}.\ \mathsf{P}(\mathsf{x}))} \qquad \frac{\mathsf{t}_1 \not\leq \mathsf{t}_2 \quad \mathsf{dep}_{\mathsf{t}_1}(\mathsf{P}) \quad \mathsf{t}_2 \not\leq \mathsf{t}_1}{\mathsf{indep}_{\mathsf{t}_2}(\mathsf{P})} \qquad \frac{\mathsf{t}_1 \not\leq \mathsf{t}_2 \quad \mathsf{indep}_{\mathsf{t}_1}(\mathsf{P})}{\mathsf{indep}_{\mathsf{t}_1}(\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t}_2,\mathsf{a}})}$$

Fig. 3. Proof rules for dependence and independence. To simplify the presentation we also use indep and dep for the point-wise lifting of indep and dep to predicates.

B On the granularity of Actions

As mentioned in Section 2.3, to support synchronization of protocols we extend CAP with simultaneous updates of multiple regions. This raises questions about the granularity of actions. For instance, it seems natural that the following view-shift should hold, by sequencing the α action followed by the β action:

$$\boxed{\mathsf{x_f} \mapsto 0}^{\mathsf{r},\mathsf{t},-}_{I} * [\alpha]^{\mathsf{r}}_1 * [\beta]^{\mathsf{r}}_1 \sqsubseteq \boxed{\mathsf{x_f} \mapsto 2}^{\mathsf{r},\mathsf{t},-}_{I} * [\alpha]^{\mathsf{r}}_1 * [\beta]^{\mathsf{r}}_1$$

$$\mathsf{I}(-) = \begin{pmatrix} \alpha : \mathsf{x_f} \mapsto 0 \rightsquigarrow \mathsf{x_f} \mapsto 1 \\ \beta : \mathsf{x_f} \mapsto 1 \rightsquigarrow \mathsf{x_f} \mapsto 2 \end{pmatrix}$$

However, it is not clear whether the following view-shift should hold,

$$\begin{aligned} &\boxed{\mathsf{x_f} \mapsto 0}^{\mathsf{r},\mathsf{t},-}_{\mathsf{I}} * \boxed{\mathsf{x_g} \mapsto 0}^{\mathsf{r}',\mathsf{t}',-}_{\mathsf{J}} * [\alpha]^{\mathsf{r}}_1 * [\beta]^{\mathsf{r}}_1 * [\eta]^{\mathsf{r}'}_1 \\ &\sqsubseteq \boxed{\mathsf{x_f} \mapsto 2}^{\mathsf{r},\mathsf{t},-}_{\mathsf{I}} * \boxed{\mathsf{x_g} \mapsto 2}^{\mathsf{r}',\mathsf{t}',-}_{\mathsf{J}} * [\alpha]^{\mathsf{r}}_1 * [\beta]^{\mathsf{r}}_1 * [\eta]^{\mathsf{r}'}_1 \end{aligned} \tag{4}$$

$$\mathsf{J}(-) = \left(\eta : \mathsf{x_g} \mapsto 0 * \boxed{\mathsf{x_f} \mapsto 0}^{\mathsf{r},\mathsf{t},-} \rightsquigarrow \mathsf{x_g} \mapsto 2 * \boxed{\mathsf{x_f} \mapsto 2}^{\mathsf{r},\mathsf{t},-}\right)$$

as it requires a *simultaneous* update of two regions, r and r', using two actions on r and one action on r'. We have chosen a semantics that does not allow the

$$\frac{}{\mathsf{pure}_{\mathsf{protocol}}(\bot)} \qquad \frac{}{\mathsf{pure}_{\mathsf{protocol}}(\top)}$$

$$\frac{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}) \qquad \mathsf{pure}_{\mathsf{protocol}}(\mathsf{Q}) \qquad op \in \{\vee, \wedge, *, \Rightarrow\}}{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}\ op\ \mathsf{Q})}$$

$$\frac{\forall \mathsf{x} : \mathsf{X}.\ \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}(\mathsf{x}))}{\mathsf{pure}_{\mathsf{protocol}}(\exists \mathsf{x} : \mathsf{X}.\ \mathsf{P}(\mathsf{x}))} \qquad \frac{\forall \mathsf{x} : \mathsf{X}.\ \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}(\mathsf{x}))}{\mathsf{pure}_{\mathsf{protocol}}(\forall \mathsf{x} : \mathsf{X}.\ \mathsf{P}(\mathsf{x}))}$$

$$\frac{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{P})}{\mathsf{pure}_{\mathsf{protocol}}(\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},\mathsf{a}})} \qquad \frac{}{\mathsf{pure}_{\mathsf{protocol}}([\alpha]^{\mathsf{r}}_{\pi})} \qquad \frac{}{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{x} \mapsto (\Delta).\{\mathsf{P}\}\{\mathsf{Q}\})}$$

$$\frac{}{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{x.f} \overset{\pi}{\mapsto} \mathsf{v})} \qquad \frac{}{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{x}_{\mathsf{f}} \overset{\pi}{\mapsto} \mathsf{v})}$$

Fig. 4. $\mathsf{pure}_{\mathsf{protocol}}$ proof rules

$$\frac{}{\mathsf{pure}_{\mathsf{state}}(\bot)} \qquad \frac{}{\mathsf{pure}_{\mathsf{state}}(\top)}$$

$$\frac{\mathsf{pure}_{\mathsf{state}}(\mathsf{P}) \qquad \mathsf{pure}_{\mathsf{state}}(\mathsf{Q}) \qquad op \in \{\vee, \wedge, *, \Rightarrow\}}{\mathsf{pure}_{\mathsf{state}}(\mathsf{P}\ op\ \mathsf{Q})}$$

$$\frac{\forall \mathsf{x} : \mathsf{X}.\ \mathsf{pure}_{\mathsf{state}}(\mathsf{P}(\mathsf{x}))}{\mathsf{pure}_{\mathsf{state}}(\exists \mathsf{x} : \mathsf{X}.\ \mathsf{P}(\mathsf{x}))} \qquad \frac{\forall \mathsf{x} : \mathsf{X}.\ \mathsf{pure}_{\mathsf{state}}(\mathsf{P}(\mathsf{x}))}{\mathsf{pure}_{\mathsf{state}}(\forall \mathsf{x} : \mathsf{X}.\ \mathsf{P}(\mathsf{x}))}$$

$$\frac{}{\mathsf{pure}_{\mathsf{state}}(\mathsf{protocol}(\mathsf{t}, \mathsf{I}))}$$

Fig. 5. $\mathsf{pure}_{\mathsf{state}}$ proof rules granularity of actions to differ when updating multiple regions. Thus, in our logic, (4) does not hold. This is a choice; it simplifies stability proofs, but it means that we must explicitly track which regions that may have been updated by a view-shift. We thus index the view-shift relation with a region type t. The indexed view-shift relation, $\sqsubseteq_{\mathsf{t}}$, thus describes a *single* update that, in addition to updating the local state, may update *multiple* shared regions with region types not greater than or equal to t, where each update must be justified by a *single* action. The indexed view-shift relation is thus *not* transitive.

C Modular Reasoning for Deterministic Parallelism

As mentioned in the introduction, Dodds et. al., recently proposed a higher-order variant of Concurrent Abstract Predicates [6] in their paper on "Modular Reasoning for Deterministic Parallelism". In their paper, the authors define a model for a higher-order variant of CAP and give proof rules for the specification logic, but not the CAP parts. Stability and atomic updates are left as semantic proof obligations in the model. The authors use this logic to verify a library for deterministic parallelism. The proof explicitly uses higher-order protocols and higher-order region assertions, without any restrictions to ensure the absence of self-referential region or protocol assertions or that the higher-order protocols are expressible using state-independent protocols. Instead of proving semantic proofs of stability and atomic updates in their model, the authors give informal proofs. This style of informal proof is unsound in the model proposed in [6] in the presence of some self-referential region or protocol assertions and some state-dependent higher-order protocols.

To illustrate one stability counterexample in the presence of state-dependent protocols, define P as follows:

$$\mathsf{P} \stackrel{\text{def}}{=} (\mathsf{x} \mapsto 0 * (\boxed{\mathsf{y} \mapsto 0}^{\mathsf{r}}_{\mathsf{I}} \vee \boxed{\mathsf{y} \mapsto 0}^{\mathsf{r}}_{\mathsf{J}})) \vee (\mathsf{x} \mapsto 1 * \boxed{\mathsf{y} \mapsto 0}^{\mathsf{r}}_{\mathsf{J}})$$

where I and J are protocols with a single α action:

$$\mathsf{I} \stackrel{\text{def}}{=} (\alpha : \mathsf{y} \mapsto 1 \rightsquigarrow \mathsf{y} \mapsto 2) \qquad\qquad \mathsf{J} \stackrel{\text{def}}{=} (\alpha : \mathsf{y} \mapsto 1 \rightsquigarrow \mathsf{y} \mapsto 3)$$

Then P is stable, because both α actions require that $\mathsf{y} \mapsto 1$, which is impossible when P holds. Note that this P is not expressible using state-independent protocols, as the protocol assertions cannot be "pulled outside" the disjunctions. This P does not support modular stability proofs, due to the interpretation of protocols, which ignores protocol assertions in protocols. In particular, define Q as follows, where K is the protocol with a single α action:

$$\mathsf{Q} \stackrel{\text{def}}{=} \boxed{\mathsf{emp} \vee \mathsf{P}}^{\mathsf{r}'}_{\mathsf{K}} \qquad\qquad \mathsf{K} \stackrel{\text{def}}{=} (\alpha : \mathsf{emp} \rightsquigarrow \mathsf{P})$$

Then Q is not stable in the model proposed in [6], as the interpretation of K ignores any assertions P makes about protocols on r. Thus for Q to be stable, it would have to be closed under the action $\mathsf{emp} \rightsquigarrow \mathsf{x} \mapsto 1$, which it is not, if the protocol on region r is I.

In the presence of state-dependent higher-order protocols it is thus unsound to treat propositional variables (referring to stable assertions) in protocols as black boxes. This renders the informal stability argument for the box and fut representation predicates defined in Figure 8 of [6] unsound.