Impredicative Concurrent Abstract Predicates

Kasper Svendsen and Lars Birkedal

Aarhus University, {ksvendsen,birkedal}@cs.au.dk

Abstract. We present impredicative concurrent abstract predicates – iCAP – a program logic for modular reasoning about concurrent, higher-order, reentrant, imperative code. Building on earlier work, iCAP uses protocols to reason about shared mutable state. A key novel feature of iCAP is the ability to define impredicative protocols; protocols that are parameterized on arbitrary predicates, including predicates that themselves refer to protocols. We demonstrate the utility of impredicative protocols through a series of examples, including the specification and verification, *in the logic*, of a spin-lock, a reentrant event loop, and a concurrent bag implemented using cooperation, against *modular* specifications.

1 Introduction

It is well-known that modular specification and verification of concurrent higher-order imperative programs is very challenging. Recently good progress has been made on reasoning about subsets of these language features. For instance, concurrent abstract predicates [8] has proved useful for reasoning about shared mutable data structures in a concurrent setting and state transition systems [10] have proved useful for reasoning about higher-order functions and shared mutable data structures and, very recently, also concurrency [23].

Internal and external sharing. The logics referred to above extend rely-guarantee versions of separation logic [24, 11] with protocols governing access to shared mutable state. These logics are sufficiently expressive to verify implementations of abstract data structures that use sharing *internally* against abstract specifications that hide this *internal sharing*. However, in practice programmers often also use shared mutable data structures to facilitate *external* sharing – the sharing of a mutable data structure through another shared mutable data structure. A lock is the canonical example of a data structure used to facilitate external sharing. In higher-order separation logic we can easily express *specifications* that support reasoning about external sharing, by parameterizing our specifications with assertions that describe the external resources shared through the data structure. However, without imposing severe predicativity restrictions (as in our earlier [21]), verifying implementations against such higher-order specifications *in the logic* is currently impossible!

To illustrate, consider a simple lock. We can specify a lock in higher-order separation logic by parameterizing our lock specification with a resource invariant R that describes the resources protected by the lock (the *externally* shared

resources):

$$\begin{array}{rl} \{\mathsf{R}\} & \texttt{new Lock()} \ \{\mathsf{isLock}(\mathsf{R},\mathsf{ret})\} \\ \{\mathsf{isLock}(\mathsf{R},\mathsf{x})\} & \texttt{x.Acquire()} \ \{\mathsf{locked}(\mathsf{R},\mathsf{x}) * \mathsf{R}\} \\ \{\mathsf{locked}(\mathsf{R},\mathsf{x}) * \mathsf{R}\} & \texttt{x.Release()} \ \{\mathsf{isLock}(\mathsf{R},\mathsf{x})\} \end{array}$$

$$\mathsf{isLock}(\mathsf{R},\mathsf{x}) \Leftrightarrow \mathsf{isLock}(\mathsf{R},\mathsf{x}) * \mathsf{isLock}(\mathsf{R},\mathsf{x})$$

Here isLock and locked are abstract predicates; $\mathsf{isLock}(\mathsf{R},\mathsf{x})$ expresses that x is a lock protecting the resource invariant R and $\mathsf{locked}(\mathsf{R},\mathsf{x})$ expresses that the lock x is indeed locked. Acquiring the lock grants ownership of R, while releasing the lock requires the client to relinquish ownership of R. Since the resource invariant R is universally quantified, this is a very strong specification; in particular, the client is free to instantiate R with any assertion, *including assertions about other shared resources or even the lock itself*. In Section 2.2 we will see that resource invariants that refer to the lock itself are useful for reasoning about reentrancy.

There has been some previous work on logics for languages with *built-in* locks [12, 14]. In [14] the built-in locks were shown to satisfy a similar higher-order specification, as part of the logic's soundness proof. However, to reason about libraries in general (not just built-in locks), we, of course, need to able to verify that implementations satisfy such specifications in the logic.

Our first contribution is a new program logic, *impredicative Concurrent Abstract Predicates* or *iCAP*, that is sufficiently expressive to support modular reasoning about both internal *and* external sharing. It is the first logic that can verify implementations of synchronization primitives, such as locks, against such higher-order specifications *in the logic*.

Layered and recursive abstractions. One of the main objectives of iCAP is to support modular reasoning about *libraries* consisting of concurrent, higher-order, reentrant, imperative code. In iCAP we have focused on two types of modularity that are both important in programming practice. The first type is simply the ability to build layers of abstractions; for instance, we want to be able to reason about a hashtable library implemented using a linked list library through an abstract linked list specification and hide this internal use of the linked list abstraction in the abstract hashtable specification. The second type of modularity is more challenging, namely the ability to build recursive abstractions; for instance, we want to be able to verify *reentrant libraries* against an abstract specification that allows clients to register callbacks that can themselves use the abstract library specification to reason about calls into the library.

To illustrate the problem of modular reasoning about reentrant libraries, consider an *event loop library* satisfying the interface given below.[1]

```
public delegate void handler();

interface IEventLoop {
  void loop();
```

[1] The first line declares a delegate type (a type-safe function pointer type) by the name handler for delegates that do not take any arguments and do not return anything.

```
  void signal();
  void when(handler f);
}
```

This library allows clients to emit an event using the `signal` method, to register an event handler using the `when` method and start the event loop using the `loop` method. Crucially, this library explicitly allows event handlers to emit events and thereby schedule themselves for execution once again! We have simplified this example to focus on the main difficulty introduced by reentrancy – namely that clients can *tie Landin's knot through the library* – however, this pattern is *ubiquitous* in the real world. For instance, event-driven code is ubiquitous in GUI applications and high-performance network applications as a way of implementing asynchronous I/O [1, 18].

To support *modular* reasoning about such examples and in particular the event loop, we need a logic that is sufficiently expressive to (1) define an abstract event loop specification that allows clients to register callbacks that emit events *and reason about these callbacks using the same abstract specification* and (2) allow implementors to verify an implementation of the event loop against the abstract event loop specification.

In Section 2.2 we explain how to verify an implementation against such an abstract specification, by defining the memory footprint of the event loop implementation recursively. In iCAP we achieve this using *guarded recursion*. One of the implementations we consider uses the lock module and thus, as we shall see, the recursively defined predicate involves the abstract isLock predicate. To the best of our knowledge, iCAP is the first program logic that supports such modular reasoning about layered and recursive abstractions, and a key technical contribution of our work is the model used to show the soundness of this expressive logic, in particular, the ability to define predicates recursively *across abstraction boundaries*.

Fine-grained concurrency. Fine-grained concurrent ADTs allow multiple threads to interleave memory operations on the underlying data representation, with the goal of reducing critical sections as much as possible, often down to basic compare-and-swap operations. Sophisticated fine-grained concurrent ADTs also employ *cooperation* [23] among threads and in particular the technique of *helping*, where one thread may help another complete its operation [13].

Conceptually, separation logic achieves modular reasoning about shared mutable state through the notion of resources that describe *information* about some part of the state and assert certain *rights* to modify this part of the state. To support *modular* reasoning about ADTs we need abstract specifications that allow *clients* to define high-level ADT resources with a notion of rights expressed in terms of the operations provided by the ADT rather than rights to modify the underlying data representation. Previous CAP-based techniques for reasoning about fine-grained concurrent data structures have either not scaled to handle implementations that employed helping [21], or have verified implementations with helping against *non-modular* specifications [8] that imposed a fixed notion of rights – choosen by the module *implementor* – on clients.

In iCAP, using impredicative protocols, we can verify fine-grained concurrent ADTs implemented with helping against *modular* specifications that allow *clients* to define high-level ADT resources. In Section 2.3 we present an example to show how this is done.

Details and proofs can be found in the accompanying appendix and technical report available at:

`https://bitbucket.org/logsem/public/wiki/icap`

1.1 Summary of contributions

In summary, our contributions include the design of a new sound program logic, iCAP, for *modular* reasoning about concurrent, higher-order, reentrant, imperative programs. In particular, iCAP supports modular reasoning about internal and external sharing, layered and recursive abstractions, and fine-grained cooperative implementations of concurrent data structures.

iCAP's expressiveness derives from the fact that it is a *higher-order* logic supporting guarded recursion and impredicative protocols. The presence of these features means that soundness of iCAP is non-trivial. Thus a key technical contribution of our work is our soundness proof of iCAP, which uses a novel model construction that we explain in Section 3.

2 Examples

2.1 Internal and external sharing – a lock

Following concurrent separation logic and its descendants [16, 12, 14], assertions in iCAP describe resources that may potentially be shared between several threads according to some protocol. A key feature of iCAP is that it supports full higher-order quantification over assertions in specifications. This means that we can give the following general abstract specification for a lock, which we explained informally in the introduction.

$$
\begin{aligned}
&\exists \mathsf{isLock}, \mathsf{locked} : \mathsf{Prop} \times \mathsf{Val} \to \mathsf{Prop}.\ \forall \mathsf{R} : \mathsf{Prop}.\ \mathsf{stable}(\mathsf{R}) \ \Rightarrow \\
&\qquad \{\mathsf{R}\}\,\mathbf{new}\ \mathsf{Lock}(-)\,\{\mathsf{ret}.\ \mathsf{isLock}(\mathsf{R}, \mathsf{ret})\} \\
&\quad \wedge\ \{\mathsf{isLock}(\mathsf{R}, \mathsf{x})\}\mathsf{x}.\mathsf{Acquire}(-)\,\{\mathsf{locked}(\mathsf{R}, \mathsf{x}) * \mathsf{R}\} \\
&\quad \wedge\ \{\mathsf{locked}(\mathsf{R}, \mathsf{x}) * \mathsf{R}\}\mathsf{x}.\mathsf{Release}(-)\,\{\mathsf{ret}.\ \mathsf{emp}\} \\
&\quad \wedge\ \mathsf{valid}(\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{isLock}(\mathsf{R}, \mathsf{x}) \Leftrightarrow \mathsf{isLock}(\mathsf{R}, \mathsf{x}) * \mathsf{isLock}(\mathsf{R}, \mathsf{x})) \\
&\quad \wedge\ \forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathsf{isLock}(\mathsf{R}, \mathsf{x})) \wedge \mathsf{stable}(\mathsf{locked}(\mathsf{R}, \mathsf{x}))
\end{aligned}
$$

In this formal specification Prop is the type of iCAP assertions, which includes assertions about shared resources. The specification thus explicitly requires the resource invariant R to be stable (invariant under any changes to the state the environment is permitted to make). The existentially quantified predicates

isLock and locked are used to support modular reasoning about internal sharing, whereas the universally quantified predicate R is used to support modular reasoning about external sharing.

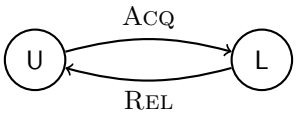
This specification asserts the existence of a lock representation predicate that is *parametric* in the resource invariant R. This allows us to use the lock representation predicate itself when defining resource invariants and thus to define recursive resource invariants. Had we simply asserted that for each resource invariant R, there exists a *non-parametric* lock representation predicate $\mathsf{isLock}_{\mathsf{R}}$, this would *not* be possible! As we will see in Section 2.2, the above *third-order* lock specification and the ability to define recursive resource invariants is critical for reasoning about the reentrancy of a multi-threaded event loop.

To verify an implementation against this specification we have to provide concrete instantiations for the parametric isLock and locked predicates and prove that the implementation satisfies the specification with these concrete instantiations. Since this specification explicitly allows clients to define recursive resource invariants, most of the difficulty in verifying an implementation boils down to defining the parametric isLock and locked predicates. In iCAP this is trivial using impredicative protocols.

A spin-lock. The implementation we have in mind is a simple spin-lock; it maintains a single boolean field, locked, which is true if and only if the lock is currently held. When the lock is unlocked, the lock owns the resource invariant R. Once the lock has been locked, only the *exclusive* owner of the locked resource is allowed to unlock the lock! We can express this protocol formally using iCAP.

iCAP extends separation logic with shared regions. Resources in shared regions are — as the name implies — shared between all clients. Upon allocation of a new shared region, we can pick a protocol of our choice, describing what resources the shared region must own. A protocol consists of a labelled transition system, labelled with *action identifiers*, and an assertion for each abstract state in the transition system that describes the resources the shared region must own in the given state. The transitions then specify how the abstract states of the region are allowed to evolve and the labels how different clients are allowed to evolve the states. In particular, for a client to change the state from s_1 to s_2, the client must own permissions to labels along a path from s_1 to s_2.

In the case of the spin-lock, for each instance of the spin-lock, we introduce a new shared region that governs that spin-lock and the resources protected by that lock. The labelled transition system that governs a spin-lock is very simple: it contains two abstract states — locked (L) and unlocked (U) — and the two obvious transitions:



The next step is to define an assertion for each abstract state describing the resources owned by the spin-lock region (conceptually, the lock) in the given state. This is mostly straightforward. When the lock is locked, the lock owns the

locked field, which contains true. However, when the lock is unlocked, the lock owns both the locked field, which contains false, and the resource invariant R:

$$\mathsf{I}(\mathsf{n},\mathsf{R},\mathsf{x})(\mathsf{L}) \stackrel{\text{def}}{=} \mathsf{x.locked} \mapsto \mathsf{true} \qquad \mathsf{I}(\mathsf{n},\mathsf{R},\mathsf{x})(\mathsf{U}) \stackrel{\text{def}}{=} \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Rel}]^{\mathsf{n}}_1$$

To capture that only the owner of the locked resource is allowed to unlock the lock, we also let the lock take full ownership of the Rel transition for the given region, when the lock is unlocked. Formally this is expressed using an action assertion, $[\alpha]^{\mathsf{n}}_{\pi}$. Here π is a fraction between 0 and 1, and $[\alpha]^{\mathsf{n}}_{\pi}$ asserts π-ownership of the α action on region n. When the client locks the lock, the client can thus take ownership of both the resource invariant R and the exclusive (1) permission to transition the shared region back into the unlocked state. Note that this spin-lock protocol is parameterized over an arbitrary resource invariant R provided by the client and is thus an *impredicative protocol*.

With these ingredients we are now ready to instantiate isLock. The isLock predicate asserts that there exists a shared region governed by the above spin-lock protocol, that is either in the unlocked or locked state; and furthermore, it asserts *non-exclusive* permission to the Acq action. Formally, this is expressed as follows (where T_{lock} refers to the labelled transition system above):

$$\mathsf{isLock}(\mathsf{R},\mathsf{x}) \stackrel{\text{def}}{=} \exists \mathsf{n} : \mathsf{RId}.\ [\textsc{Acq}]^{\mathsf{n}}_{\_} * \mathsf{rintr}(\mathsf{I}(\mathsf{n},\mathsf{R},\mathsf{x}),\mathsf{n}) * \mathsf{region}(\{\mathsf{L},\mathsf{U}\}, T_{lock}, \mathsf{n})$$

The region assertion, $\mathsf{region}(X, T, n)$, asserts that there exists a region with region identifier n, whose labelled transition system is T and that the current abstract state of region n is a member of the set X. The labelled transition system T is represented as a function from action identifiers to relations on abstract states; see the appendix for details. To specify the spin-lock protocol we also need to specify what resources the spin-lock must own in the different abstract states. This is expressed using the region interpretation assertion, $\mathsf{rintr}(\mathsf{I},\mathsf{n})$, which takes as argument a predicate I, indexed by the abstract states of the given region. We use $[\alpha]^{\mathsf{n}}_{\_}$ as shorthand for $\exists\pi.\ [\alpha]^{\mathsf{n}}_{\pi}$ to express non-exclusive ownership of an action α.

This illustrates the use of iCAP's impredicative protocols for defining the concrete instantion of isLock for a spin-lock. Now that isLock has been defined, the actual verification of the spin-lock implementation with this concrete instantiation follows the structure of the original CAP proof of a spin-lock [8]. One crucial difference is that iCAP features enough proof rules to carry out the proof *in the logic*, including stability proofs. In the appendix, we show in detail how to verify the spin-lock implementation with this concrete instantiation using the formal iCAP proof system.

Compared to earlier work on concurrent abstract predicates [8, 9, 21], in iCAP we simplify the description of protocols, by describing them using state transition systems. This presentation is inspired by earlier work by Dreyer et. al. on protocols for reasoning about local state in higher-order programs [10]. We believe this description of protocols is a useful conceptual simplification compared to the original CAP presentation [8], in particular since protocols can now easily be drawn. However, we stress that this presentation also simplifies stability

proofs in the program logic, since they only have to refer to abstract states in the transition system. The iCAP stability obligations for the verification of the spin-lock are thus significantly easier to prove that the corresponding stability obligations from the original spin-lock CAP proof.

2.2 Layered and recursive abstractions – an event loop

The previous example illustrated how iCAP's impredicative protocols allow modular reasoning about internal and external sharing. In this section we illustrate how higher-order logic and guarded recursion allow modular reasoning about layered and recursive abstractions.

A single-threaded event loop. Recall the *reentrant* event loop library from the introduction. We can express an abstract event loop specification for single-threaded event loops that explicitly allows clients to reason about event handlers using the same event loop specification as follows:

$$
\begin{aligned}
&\exists \mathsf{eloop} : \mathsf{Val} \to \mathsf{Prop}.\\
&\qquad \{\mathsf{emp}\}\textbf{new}\ \mathsf{EventLoop}(\_)\,\{\mathsf{ret}.\ \mathsf{eloop}(\mathsf{ret})\}\\
&\quad \wedge\ \{\mathsf{eloop}(\mathsf{x})\}\mathsf{x.loop}(\_)\,\{\mathsf{eloop}(\mathsf{x})\}\\
&\quad \wedge\ \{\mathsf{eloop}(\mathsf{x})\}\mathsf{x.signal}(\_)\,\{\mathsf{eloop}(\mathsf{x})\}\\
&\quad \wedge\ \{\mathsf{eloop}(\mathsf{x}) * \mathsf{f} \mapsto \{\mathsf{eloop}(\mathsf{x})\}\{\mathsf{eloop}(\mathsf{x})\}\}\mathsf{x.when}(\mathsf{f})\,\{\mathsf{eloop}(\mathsf{x})\}
\end{aligned}
$$

This specification asserts the existence of an abstract event loop resource, eloop, which is created by creating a new event loop instance and preserved by all event loop methods. The when method for registering event handlers requires that the given event handler satisfies the nested Hoare triple

$$\mathsf{f} \mapsto \{\mathsf{eloop}(\mathsf{x})\}\{\mathsf{eloop}(\mathsf{x})\},$$

thereby explicitly allowing event handlers to use the abstract eloop resource to emit events. In a sense this is a very *weak* specification, in that it only allows us to reason about memory safety of our clients. However, in the presence of reentrancy, verifying an implementation against this simplified specification is highly non-trivial and beyond almost all current program logics.[2]

To define a concrete eloop resource, imagine a concrete implementation that maintains a set of pending events and a set of registered handlers, as sketched below.

```
class EventLoop : IEventLoop {
  private Set<event> signals;
  private Set<handler> handlers;
  ...
}
```

[2] The exception being our own HOCAP [21], which, however, had other severe restrictions compared to iCAP.

To allow the event loop to call registered event handlers, the eloop resource must assert that the registered event handlers satisfy some specification. To allow event handlers to emit events, this specification must itself refer to the eloop resource. In iCAP we can express this recursion by guarding the recursive occurence of the eloop resource and defining eloop by *guarded recursion* (note the use of the "later" ($\triangleright$) connective, which serves as a guard):

$$\begin{aligned}\mathsf{eloop} = \mathsf{fix}(\,\lambda\mathsf{eloop} : \mathsf{Val} \to \mathsf{Prop}.\ \lambda\mathsf{x} : \mathsf{Val}.\ \exists\mathsf{y},\mathsf{z} : \mathsf{Val}.\ \exists A, B : \mathcal{P}_{fin}(\mathsf{Val}).\\ \mathsf{x.signals} \mapsto \mathsf{y} * \mathsf{x.handlers} \mapsto \mathsf{z} * \mathsf{set}(\mathsf{y}, A) * \mathsf{set}(\mathsf{z}, B) *\\ \forall b \in B.\ \triangleright\, b \mapsto \{\mathsf{eloop}(\mathsf{x})\}\{\mathsf{eloop}(\mathsf{x})\})\end{aligned}$$

This event loop resource asserts exclusive ownership of the signals field, the handlers field, the set of pending events, the set of registered handlers, and that all registered handlers satisfy the specification $\mathsf{f} \mapsto \{\mathsf{eloop}(\mathsf{x})\}\{\mathsf{eloop}(\mathsf{x})\}$, one step later.

In our operational semantics each atomic statement takes one step to execute and executing a method or delegate call executes one atomic statement before the body of the method or delegate is executed. Hence, to verify a call to a method or delegate, it suffices to know the specification of the method or delegate body, one step later. We can thus verify calls from the event loop to the registered event handlers using the guarded eloop resource defined above.

Note that eloop is *not* definable by induction, as the recursive argument is not applied to a structurally smaller argument, nor by Tarski's fixed-point theorem, as nested Hoare triples are contravariant in the pre-condition and covariant in the postcondition.

As with the lock example, the interesting part of the verification of a reentrant event loop is the definition of the event loop resource. Once the event loop resource has been defined, the verification is routine. The real challenge is defining a logic and accompanying model that supports such recursive resource definitions!

A multi-threaded event loop. The single-threaded event loop example illustrated the use of guarded recursion for reasoning about recursive abstractions. To make the example even more challenging and truly illustrate the power of impredicative protocols, let us now consider a *multi-threaded* reentrant event loop library. The abstract event loop specification remains the same, expect with the added axiom that the abstract event loop resource is freely duplicable (thus allowing any number of clients to use the event loop concurrently):

$$\mathsf{valid}(\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{eloop}(\mathsf{x}) \Leftrightarrow \mathsf{eloop}(\mathsf{x}) * \mathsf{eloop}(\mathsf{x}))$$

As for the implementation, imagine a lock-based implementation that extends the previous implementation with a lock that protects the set of pending events and the set of registered event handlers. Conceptually, we thus have a library that allows clients to tie Landin's knot *through a reference protected by a lock*. To verify the single-threaded implementation, we needed to refer to eloop to specify the registered handlers when defining eloop. Likewise, now eloop must

assert the existence of a lock that protects the registered event handlers that are again specified in terms of eloop (note the use of isLock):

$$\begin{aligned}\mathsf{eloop} = \mathsf{fix}(\ & \lambda \mathsf{eloop} : \mathsf{Val} \to \mathsf{Prop}.\ \lambda \mathsf{x} : \mathsf{Val}.\ \exists \mathsf{l} : \mathsf{Val}.\ \mathsf{x.lock} \mapsto\!\!\!\!\!\!- \ \mathsf{l} * \\ & \mathsf{isLock}(\mathsf{l}, \exists \mathsf{y}, \mathsf{z} : \mathsf{Val}.\ \exists A, B : \mathcal{P}_{fin}(\mathsf{Val}). \\ & \qquad \mathsf{x.signals} \mapsto \mathsf{y} * \mathsf{x.handlers} \mapsto \mathsf{z} * \mathsf{set}(\mathsf{y}, A) * \mathsf{set}(\mathsf{z}, B) * \\ & \qquad \forall b \in B.\ \triangleright\, b \mapsto \{\mathsf{eloop}(\mathsf{x})\}\{\mathsf{eloop}(\mathsf{x})\}))\end{aligned}$$

This definition is extremely interesting! First of all, it illustrates the true power of the third-order lock specification to define *recursive* resource invariants *that refer back to the lock itself*. This is only possible because the abstract lock specification asserts the existence of a *parameterized* lock representation predicate; thus allowing us to *define the resource invariant in terms of the lock itself* (the resource invariant we use for the lock is the argument given to isLock, which refers to eloop, which again refers to isLock).

This example also illustrates the ability of iCAP to combine layered and recursive abstractions; in this example we are reasoning about the recursive event loop abstraction in terms of the lock abstraction defined in Section 2.1. In particular, the eloop representation predicate is defined in terms of an *abstract* isLock representation predicate. To ensure that eloop is well-defined we thus have to prove guardedness across an abstraction boundary (i.e., that the recursive occurence of eloop inside the abstract isLock assertion is guarded). This is automatically enforced in iCAP (!), and thus iCAP supports modular reasoning about guardedness. Semantically, this is enforced in the interpretation of the iCAP function space, which intuitively does not consist of all set-theoretic functions, but only those functions that are suitably non-expansive. Note that these intracies in the model are abstracted away by the iCAP logic and the proof in iCAP of the well-definedness of the eloop predicate above is completely trivial and just follows from the fact that the recursive occurrence of eloop is under a $\triangleright$ guard.

These two event loop examples illustrate how we can reason about recursive abstractions in iCAP and also exemplify the power of impredicative protocols. For presentation purposes we considered the core part of a simple example — we emphasize that this style of reasoning also scales to full functional verification of complicated examples such as the joins library [19], which combines layered and recursive abstractions with internal and external sharing in a higher-order, concurrent, reentrant, imperative library. We have previously verified a lock-based joins implementation in HOCAP against an abstract joins specification with an explicit predicative stratification [20]. In iCAP, using impredicative protocols, we can verify the joins implementation against a much *simpler* and more *expressive* joins specification. Furthermore, in HOCAP we could not verify a fine-grained implementation of the joins library; in iCAP this is now possible using the techniques explained in the following.

2.3 Fine-grained concurrency – a concurrent bag

In this section we illustrate how iCAP supports *modular* reasoning about advanced concurrent ADTs by verifying a fine-grained implementation of a concurrent bag, implemented using helping, against a *modular* ADT specification.

We start by recalling our specification pattern from HOCAP [21] for expressing modular ADT specifications that allow clients to define a high-level ADT resources with a notion of rights that matches the client's intended use. Next, we sketch how to verify a fine-grained implementation with helping of a concurrent bag against an abstract bag specification expressed using this specification pattern. See the associated technical report for the full proof.

A modular ADT specification. Recall that in a sequential setting, one typically specifies data structure operations by relating an abstraction of the initial and terminal state of the operation through an abstract representation predicate. For instance, we might specify a Push method for an unordered bag as follows:

$$\{\mathsf{bag}(\mathsf{x}, A)\}\ \texttt{x.Push(y)}\ \{\mathsf{bag}(\mathsf{x}, A \cup \{\mathsf{y}\})\}$$

This says that if, initially, the bag contains the elements in the multiset A, then, upon termination, the bag contains the elements in A and y. Crucially, this specification relates the abstract initial and terminal effects of the *entire* Push method.

In a concurrent setting we can reason modularly about implementations that satisfy that for each intermediate state in its execution, there exists some abstract state describing the concrete state and the method contains zero or more atomic instructions that modify the abstract state. Following our earlier work [21], the idea now is to allow clients to reason about the abstract initial and terminal state for each of these *atomic instructions*, rather than the abstract initial and terminal state of the *entire method*. By allowing clients to reason about the atomic instructions that modify the abstract state, clients can define their own high-level ADT resources with a notion of rights expressed in terms of the abstract state.

Technically, we achieve this using a *phantom field*, shared between the concurrent ADT and any clients, that stores the current abstract state of an instance. Phantom fields play a similar role as ghost/auxiliary variables [17], in that they are fields used only for specification purposes. We use $\mathsf{x}_\mathsf{f} \overset{\pi}{\mapsto} v$ to assert fractional ownership of phantom field f on object x with fraction π. By splitting ownership of the phantom field we ensure that the concurrent ADT and any clients agree on the current abstract state. To allow clients to reason about the atomic instructions that modify the abstract state (and thus the phantom field), we further parameterize the specification of each method with a *view shift*. View shifts describe updates to the instrumented state that do not affect the concrete state. View shifts can thus be used to update phantom fields, allocate new regions and change the abstract state of a region, potentially transferring ownership of some resource in the process. We use $P \sqsubseteq Q$ to express that P can be view shifted to Q. See the appendix for proof rules relating to phantom fields and view shifts.

A modular bag specification. We present part of the bag specification in Figure 1; we now explain it. We only include an operation to create the bag and a push operation, the specification for a pop method is similar and omitted.

$$\exists \mathsf{bag} : \mathsf{RId} \times \mathsf{Val} \to \mathsf{Prop}.$$

$$\{\mathsf{emp}\}\ \mathbf{new}\ \mathsf{Bag}(-)\ \{\mathsf{ret}.\ \exists \mathsf{n} : \mathsf{RId}.\ \mathsf{bag}(\mathsf{n}, \mathsf{ret}) * \mathsf{ret}_{\mathsf{cont}} \overset{1/2}{\mapsto} \emptyset\}$$

$$\forall \mathsf{P}, \mathsf{Q} : \mathsf{Val} \times \mathsf{Val} \to \mathsf{Prop}.\ \forall \mathsf{n} : \mathsf{RId}.$$
$$(\forall \mathsf{X} : \mathcal{P}_m(\mathsf{Val}).\ \forall \mathsf{x}, \mathsf{y} : \mathsf{Val}.$$
$$\mathsf{x}_{\mathsf{cont}} \overset{1/2}{\mapsto} \mathsf{X} * \mathsf{P}(\mathsf{x}, \mathsf{y}) \sqsubseteq^{RId \setminus \{\mathsf{n}\}} \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\mapsto} (\mathsf{X} \cup \{\mathsf{y}\}) * \mathsf{Q}(\mathsf{x}, \mathsf{y})) \Rightarrow$$
$$\{\mathsf{bag}(\mathsf{n}, \mathsf{x}) * \mathsf{P}(\mathsf{x}, \mathsf{y})\}\ \mathsf{x}.\mathsf{Push}(\mathsf{y})\ \{\mathsf{bag}(\mathsf{n}, \mathsf{x}) * \mathsf{Q}(\mathsf{x}, \mathsf{y})\}$$

Fig. 1. Part of a modular bag specification

In the case of the Push method, assuming it only contains a single atomic instruction that "commits" the push, we can express this formally by relating the effects of the Push method with an arbitrary "push" view shift provided by the client, see Lines 3–6 in Figure 1. The view shift expresses what should happen at the client side when the abstract state of the push operation takes place. The assertion $\mathsf{x}_{\mathsf{cont}} \overset{1/2}{\mapsto} \mathsf{X}$ asserts half-ownership of the phantom field cont, which contains the current abstract state of the bag. By letting the data structure own half and clients share the other half (clients get the other half by calling the `new` method), clients can impose a protocol on the abstract state that matches their intended notion of rights through their half of the phantom field. Since updating the phantom field requires both halves, this forces clients to prove that the abstract effects of any call to the Push method satisfies any protocols clients may have imposed.

We call the view shift in the premise of the above rule a "push" view shift because it requires the client to update the initial abstract state from X to $\mathsf{X} \cup \{\mathsf{y}\}$, for any abstract state X. Conceptually, this view shift is thus an atomic "push" method at the instrumented level and the push specification expresses that the Push method simulates any such "push" view shift provided by the client. The universally quantified predicates P and Q allow the client to relate its local state with the abstract initial and terminal state of the atomic instruction that "commits" the push. We refer to P and Q as *synchronization pre- and postconditions*.

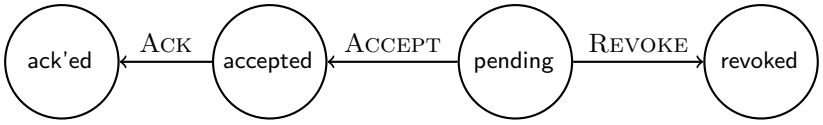
In [21] we had to impose severe restrictions on P and Q due to the lack of impredicative protocols, but with iCAP, there are no restrictions on P and Q, resulting in *much simpler* and *more expressive* refinable specifications.

Finally we comment on the superscript on the view shifts and the region identifier n argument to the bag predicate, $\mathsf{bag}(\mathsf{n}, \mathsf{x})$. In iCAP, when reasoning about an atomic instruction, we can "open" a shared region and move the shared resources into our local state for the duration of the atomic instruction, provided we obey the protocol imposed by the region. Clearly, it is only sound to "open" each region once for each atomic instruction (opening a region twice results in

two local copies of the shared region's resources).[3] Since the "push" view shifts provided by the client are used during the atomic instruction that "commits" the push, we have to ensure that the client does not "open" the module's region with its view shifts. We thus parameterize the bag predicate with a region identifier n to reveal that the bag module may use region n. As discussed in the appendix, this allows us to express, qua the superscript on the view shift, that the view shift provided by the client should not open the region n.

Now we have explained how to give a modular, refinable, specification to a concurrent data structure. We now sketch how iCAP can be used to verify that a sophisticated fine-grained concurrent implementation using cooperation actually meets the modular bag specification. iCAP is the first program logic that supports verification of such sophisticated implementations against such modular specifications (in particular, in our earlier work [21] we could not deal with implementations using cooperation).

To reduce contention on the main data structure used to implement the bag, a thread seeking to push an element (the "pusher") may offer the push operation to other threads, using a side-channel. If a thread seeking to pop (the "popper") then comes along, it may notice and accept the push-offer, without touching the main data structure at all. By accepting the push-offer the popper also completes the operation of the pusher, and in that sense it has *helped* the pusher. The heart of the verification is the protocol used for handling offers. In our case, that protocol can be described using the following labelled transition system, denoted $\mathsf{T}_{\mathit{offer}}$:



Intuitively, pending means that an offer has been made and it is waiting for somebody to accept it, accepted means that the offer has been accepted, ack'ed means that we have acknowledged that somebody has accepted the offer, and revoked is used for the case where we revoke the offer (since no one accepted it and now we will re-attempt to push).

The interpretation of the states of $\mathsf{T}_{\mathit{offer}}$ are as follows:

$$\begin{aligned}
&\mathsf{I}_{\mathit{offer}}(\mathsf{n},\mathsf{P},\mathsf{Q},\mathsf{b},\mathsf{x},\mathsf{y})(\mathsf{pending}) \stackrel{\text{def}}{=} \mathsf{x.state} \mapsto 0 * \mathsf{P}(\mathsf{b},\mathsf{y}) * \\
&\quad \mathit{spec}(\forall \mathsf{X} : \mathcal{P}_m(\mathrm{Val}).\ \forall \mathsf{x},\mathsf{y} : \mathrm{Val}. \\
&\qquad \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\mapsto} \mathsf{X} * \mathsf{P}(\mathsf{x},\mathsf{y}) \sqsubseteq^{\mathit{RId}\setminus\{\mathsf{n}\}} \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\mapsto} (\mathsf{X} \cup \{\mathsf{y}\}) * \mathsf{Q}(\mathsf{x},\mathsf{y}))
\end{aligned}$$

$$\mathsf{I}_{\mathit{offer}}(\mathsf{n},\mathsf{P},\mathsf{Q},\mathsf{b},\mathsf{x},\mathsf{y})(\mathsf{accepted}) \stackrel{\text{def}}{=} \mathsf{x.state} \mapsto 1 * \mathsf{Q}(\mathsf{b},\mathsf{y})$$

$$\mathsf{I}_{\mathit{offer}}(\mathsf{n},\mathsf{P},\mathsf{Q},\mathsf{b},\mathsf{x},\mathsf{y})(\mathsf{revoked}) \stackrel{\text{def}}{=} \mathsf{x.state} \mapsto 2$$

$$\mathsf{I}_{\mathit{offer}}(\mathsf{n},\mathsf{P},\mathsf{Q},\mathsf{b},\mathsf{x},\mathsf{y})(\mathsf{ack'ed}) \stackrel{\text{def}}{=} \mathsf{x.state} \mapsto 1$$

[3] See the appendix for a concrete counterexample.

Here b refers to the bag, x refers to the offer, y is the value on offer, and P and Q are the pusher's synchronization pre- and postconditions.

The interpretations of the states contain information about the value of program variable state, which is used by the implementation to keep track of which state the offer is in. The point to notice, however, is that the $\mathsf{pending}$ state contains both the pusher's synchronization precondition, $\mathsf{P}(\mathsf{b},\mathsf{y})$, and the pusher's "push" view shift, and the $\mathsf{accepted}$ state contains the pusher's synchronization postcondition, $\mathsf{Q}(\mathsf{b},\mathsf{y})$. To accept an offer (transition the abstract state from $\mathsf{pending}$ to $\mathsf{accepted}$) the popper thus has to perform the pusher's "push" view shift. Conceptually, the offer protocol "transfers" the pusher's view shift to the popper.

Note how the combination of view shifts and impredicative protocols together allows us to prove that the fine-grained implementation with helping meets the modular bag specification.

See the accompanying technical report for the full proof.

3 Model

In this section we present a model of iCAP. Soundness of iCAP is non-trivial. Indeed, in our earlier work on HOCAP [21], we discovered that a recent proposal for a higher-order variant of concurrent abstract predicates [9] was unsound. This led us to consider only predicative protocols in [21], which simplified the construction of a sound model, but also resulted in much *weaker* and more *complicated* specifications and proofs. Here instead, we follow ideas from models of impredicative type systems with higher-order store, e.g. [2, 4], and define our model of iCAP using a *guarded-recursively defined space of protocols*. We define our model in the type theory and logic of the topos of trees [4]. This has the advantage that most of the model construction is done as if we were working with ordinary sets, except for those places where we need to guard some recursive definitions for well-definedness. More importantly, it makes it straightforward to define a higher-order logic, since the recursively defined space of protocols is now simply a type in the type theory of the topos of trees, which already includes function and powerset types! The resulting program logic includes the later operator from the ambient type theory. As we have already seen, we use this later operator to define guarded-recursive assertions and protocols. It can also be used to define guarded-recursive specifications. We emphasize that readers need not be familiar with [4] in order to understand the present paper.

Topos of Trees. The internal language of the topos of trees is an intuitionistic higher-order logic over a simply-typed term language extended with subset types and guarded-recursive types. This internal language features a new type former $\blacktriangleright$, pronounced later, for defining guarded-recursive types, and a new logical connective $\triangleright$, also pronounced later, for defining guarded-recursive predicates. (For readers who are familiar with the use of theorem provers, such as Coq, for formalizing models of logics or programming languages, it may be helpful to think of the type theory of the topos of trees as playing the rôle of the

Coq type theory.) The point of using the guarded type theory and logic is that it makes it easy to define the space of protocols, which needs to be recursively defined. Crucially, this results in a type in the guarded type theory, and thus, since that type theory includes higher function types, we can then easily define the interpretation of the function spaces in iCAP.[4] The guarded type theory also includes types of the form ΔX, for any ordinary set X. Such types ΔX are referred to as constant sets. We define the non-recursive part of the model in the category of sets and use these as constant sets to construct the recursive part of the model in the topos of trees.

The presentation of the iCAP model is inspired by the Views framework [7] and models of impredicative type systems, e.g., [5], with higher-order store.

The Views framework provides a general way of relating a *concrete semantics* with an *instrumented semantics* and constructing a separation logic over the instrumented semantics. In our case the concrete semantics is a subset of C# with an interleaving semantics. The instrumented semantics extends the concrete C# states with phantom fields, shared regions, and protocols and enforces that clients respect the protocols governing shared resources.

The model of iCAP is defined in Figure 2, and is defined over countably infinite and disjoint set of action identifiers, *AId*, state identifiers, *SId*, object identifiers, *OId*, closure identifiers *CId*, region identifiers *RId*, class names *CN*, field names *FN*, and method names *MN*.

Instrumented states ($m \in \mathcal{M}$) are tuples consisting of three components: a local state, a shared state and an action model. We use $m.l, m.s$ and $m.a$ to refer to the first, second and third component of m. The local state ($l \in LS$) consists of a partial C# heap, a partial phantom heap and a capability map. The partial C# heap and phantom heaps specify the current value and permissions to heap cells and phantom fields, respectively. The capability map specifies action permissions on shared regions. In particular, it records the fractional permission the client owns on each region and action identifier. The shared state ($s \in SS$) specifies the current abstract state of each allocated region and the labelled transition system governing the given region. We use $s(r).s$ and $s(r).p$ to refer to the state and labelled transition system of region r of a shared state s. The set of abstract states, AS, consists of pairs of local and shared state. Finally, the action model ($\varsigma \in AMod$) specifies the interpretation of the abstract states of each allocated region. Since the interpretation of each abstract state is given by a general assertion, which is itself a subset of instrumented states, a naive definition of $AMod$ in set theory is not well-defined. Instead, we let *RIntr* (the type of interpretations of abstract states for a single region) denote a solution to the following guarded-recursive equation

$$RIntr \cong \blacktriangleright((\Delta SId \times (\Delta RId \rightharpoonup_{fin} RIntr)) \to_{mon} \mathcal{P}^{\uparrow}(\Delta AS))$$

[4] If we had worked in the category of sets and used step-indexing directly, we would have had to define an appropriate notion of function space between the resulting indexed space of protocols and itself, and that would essentially amount to unrolling the definition from the topos of trees model.

Semantic domains in the category of Sets

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

$$Heap \stackrel{\text{def}}{=} (OId \times FN \rightharpoonup_{fin} Val) \times (OId \rightharpoonup_{fin} CN) \times (CId \rightharpoonup_{fin} OId \times MN)$$

$$PHeap \stackrel{\text{def}}{=} \{(pc, ph) \in (OId \times FN \to [0,1]) \times (OId \times FN \rightharpoonup_{fin} Val) \mid \forall o \in OId.\ \forall f \in FN.\ pc(o,f) = 0 \Rightarrow (o,f) \notin dom(ph)\}$$

$$l \in LS \stackrel{\text{def}}{=} PHeap \times Heap \times Cap \qquad LTS \stackrel{\text{def}}{=} AId \to \mathcal{P}(SId \times SId)$$

$$s \in SS \stackrel{\text{def}}{=} RId \rightharpoonup_{fin} (SId \times LTS) \qquad AS \stackrel{\text{def}}{=} LS \times SS$$

Semantic domains in the topos of trees

$$RIntr \cong \blacktriangleright((\Delta SId \times (\Delta RId \rightharpoonup_{fin} RIntr)) \to_{mon} \mathcal{P}^{\uparrow}(\Delta AS)) \qquad s \in Spec \stackrel{\text{def}}{=} \Omega$$

$$\varsigma \in AMod \stackrel{\text{def}}{=} \Delta RId \rightharpoonup_{fin} RIntr \qquad m \in \mathcal{M} \stackrel{\text{def}}{=} \Delta LS \times \Delta SS \times AMod \qquad p \in Prop \stackrel{\text{def}}{=} \mathcal{P}^{\uparrow}(\mathcal{M})$$

where the ordering on $\mathcal{M}$ is R_{RId} and the ordering on ΔAS is

$$m_1 \leq m_2 \quad \text{iff} \quad m_1.l \leq m_2.l \wedge m_1.s \leq m_2.s \wedge m_1.a \leq m_2.a$$

Interference relation

$$\begin{aligned} acts(l,r) &\stackrel{\text{def}}{=} \{\alpha \mid \pi_3(l)(r,\alpha) < 1\} \\ upds(l,r,p) &\stackrel{\text{def}}{=} \{(s_1,s_2) \mid \exists \alpha \in acts(l,r).\ (s_1,s_2) \in p(\alpha)\} \\ R_A &\stackrel{\text{def}}{=} \{((l_1,s_1),(l_2,s_2)) \mid l_1 \leq l_2 \wedge \forall r \in dom(s_1).\ ((r \in A \wedge (s_1(r).s, s_2(r).s) \in \overline{upds(l_1, r, s_1(r).p)}) \\ &\qquad \vee s_1(r).s = s_2(r).s) \wedge s_1(r).p = s_2(r).p\} \end{aligned}$$

where $\overline{(-)}$ denotes the transitive, reflexive closure.

Orderings

$$\begin{aligned} l_1 \leq l_2 \quad &\text{iff} \quad \exists l_3.\ l_2 = l_1 \bullet_{LS} l_3 \\ s_1 \leq s_2 \quad &\text{iff} \quad \forall r \in dom(s_1).\ r \in dom(s_2) \wedge s_1(r) = s_2(r) \\ \varsigma_1 \leq \varsigma_2 \quad &\text{iff} \quad \forall r \in dom(\varsigma_1).\ r \in dom(\varsigma_2) \wedge \varsigma_1(r) = \varsigma_2(r) \end{aligned}$$

Composition

$$x \bullet_= y \stackrel{\text{def}}{=} x \quad \text{if } x = y \qquad f \bullet_+ g \stackrel{\text{def}}{=} \lambda x.\ f(x) + g(x) \quad \text{if } \forall x.\ f(x) + g(x) \leq 1$$

$$f \bullet_\cup g \stackrel{\text{def}}{=} f \cup g \quad \text{if } dom(f) \cap dom(g) = \emptyset$$

$$f \bullet_? g \stackrel{\text{def}}{=} f \cup g \quad \text{if } \forall x \in dom(f) \cap dom(g).\ f(x) = g(x)$$

$$\bullet_{Heap} \stackrel{\text{def}}{=} \bullet_\cup \times \bullet_= \times \bullet_= \qquad \bullet_{PHeap} \stackrel{\text{def}}{=} \bullet_+ \times \bullet_?$$

$$\bullet_{LS} \stackrel{\text{def}}{=} \bullet_{PHeap} \times \bullet_{Heap} \times \bullet_+ \qquad \bullet_{\mathcal{M}} \stackrel{\text{def}}{=} \bullet_{LS} \times \bullet_= \times \bullet_=$$

Erasure

$$\lfloor (s,\varsigma) \rfloor_r \stackrel{\text{def}}{=} \{l \in LS \mid (l,s) \in app(\varsigma(r))(s(r).s, \varsigma)\}$$

$$\lfloor (l,s,\varsigma) \rfloor_A \stackrel{\text{def}}{=} \{h \in Heap \mid \exists l', sr : dom(s) \cap A \to LS.\ h = l'.h \wedge l' = l \bullet \Pi_{r \in dom(s) \cap A} sr(r) \wedge \forall r \in dom(s) \cap A.\ sr(r) \in \lfloor (s,\varsigma) \rfloor_r\}$$

Fig. 2. Model of iCAP.

Here ΔSId is the constant set of state identiers, ΔRId is the constant set of region identifiers, ΔAS is the constant set of abstract states, and $\mathcal{P}^{\uparrow}(\Delta AS)$ consistst of the upwards-closed subsets of ΔAS with respect to the ordering shown in Figure 2. Note the use of the $\blacktriangleright$ operator, which acts as a guard, and ensures that $RIntr$ is well-defined (unique up to isomorphism). Using $RIntr$ we can then define the type of action models as the type of finite functions from region identifiers to region interpretations: $AMod \stackrel{\text{def}}{=} \Delta RId \rightharpoonup_{fin} RIntr$. From the above isomorphism we can define the following abstraction and application functions to fold and unfold elements of $RIntr$:

$$lam : (\Delta SId \times AMod \to_{mon} \mathcal{P}^{\uparrow}(\Delta AS)) \to RIntr$$
$$app : RIntr \to (\Delta SId \times AMod \to_{mon} \mathcal{P}^{\uparrow}(\Delta AS))$$

Crucially, because of the guard, if we unfold a folded element x, we get back the element x, one step later: $app \circ lam = \triangleright$, where $\triangleright$ refers to the pointwise lifting of $\triangleright$ to function spaces.

Assertions in iCAP are modeled as upwards-closed subsets of instrumented states (see the definition of $Prop$ in Figure 2), where the upwards-closure expresses that assertions should be closed under allocation of new regions and extensions of the local state. Assertions in the specification logic are simply modeled as assertions in the topos of trees. The function types of iCAP are simply modeled using the function space in the guarded type theory! (We emphasize again that this is one of the advantages of using the topos of trees as the ambient theory in which to define the model of iCAP; if we had worked in ordinary sets, then iCAP types could not simply be interpreted as sets,[5] they would have to be indexed families of sets, and then the iCAP function space would also have to be appropriate families of functions satisfying certain naturality conditions.

Interference relation & stability. The interference relation, R_A, specifies how the environment is allowed to modify the *abstract state* of shared regions. The interference relation is indexed by a set of region identifiers, A, of regions that are allowed to change. R_{RId} thus allows the environment to change the abstract state using any path in the labelled transition system governing the region, along actions not exclusively owned by the client. R_A is defined in Figure 2 in terms of two functions, $acts$ and $upds$. The $acts$ function specifies the actions not exclusively owned by the client and the $upds$ function specifies the set of transitions labelled with actions not exclusively owned by the client.

Unlike previous models of CAP, this interference relation is expressed entirely in terms of *abstract states* and is completely independent of the interpretation of these abstract states. This is why stability in iCAP is also expressed at the abstract level and why it is much simpler than previous versions of CAP. An assertion is A-stable if it is closed under R_A:

$$stable_A(p) \stackrel{\text{def}}{=} (R_A \times id_{AMod})(p) \subseteq p$$

[5] Why? Because then we could not guarantee the existence of guarded recursive predicates involving higher-order functions (such as the eloop predicate from Section 2.2).

where $R(p) = \{m' \in \mathcal{M} \mid \exists m \in p.\ (m, m') \in R\}$. Intuitively, an assertion is A-stable if it is closed under interference from the environment on regions in A. An assertion is thus stable if it is *RId*-stable.

Erasure. The relation between the instrumented semantics and the concrete semantics is expressed through an erasure function, $\lfloor - \rfloor_A$, that maps instrumented states to sets of concrete states. Like the interference relation, the erasure is indexed by a set of region identifiers, A, of regions to erase. The erasure works by picking a concrete state l_r for each allocated region $r \in A$ that satisfies the interpretation of the current abstract state of the given region, and composing all these states with the current local state. The erasure is defined in terms of a single-region erasure, $\lfloor - \rfloor_r$, that defines the set of concrete states satisfying the interpretation of the current abstract state of region r. Note that this is expressed in terms of the application function, *app*, introduced earlier for unfolding a region interpretation.

View shifts describe changes at the instrumented level that preserves the state at the concrete level. An A-view shift $p \sqsubseteq^A q$ describes a view shift that is only allowed to modify regions in A. We can express this formally (and build-in framing) by requiring the view shift to preserve all A-stable frames r:

$$p \sqsubseteq^A q \stackrel{\text{def}}{=} \forall r \in \mathit{Prop}.\ \mathit{stable}_A(r) \Rightarrow \lfloor p * r \rfloor_A \subseteq \lfloor q * r \rfloor_A$$

The operational semantics of the underlying programming language is defined in terms of a labelled thread pool evaluation relation, $\xrightarrow{a}$, and an action semantics, $[\![-]\!]$. The labelled thread pool evaluation relation, $\xrightarrow{a}$, defines the local effects (i.e., stack effects) of executing a single thread for one step of execution, while the action semantics defines the global effects (i.e., heap effects) of executing an atomic action. Atomic satisfaction expresses what it means for an atomic action a to satisfy a given Hoare specification:

$$\begin{aligned} a \text{ sat}^A \ \{p\}\,\{q\} \stackrel{\text{def}}{=} \forall r \in \mathit{Prop}.\ \forall m \in \mathcal{M}.\ \forall h, h' \in \mathit{Heap}.\\ m \in p * \triangleright r \wedge h \in \lfloor m \rfloor_A \wedge h' \in [\![a]\!](h) \wedge \mathit{stable}_A(r)\\ \Rightarrow\ \exists m' \in \mathcal{M}.\ \triangleright (m' \in q * r \wedge h' \in \lfloor m' \rfloor_A) \end{aligned}$$

This is the case, if, executing a from any initial concrete state h in the erasure of p there exists an abstract state in q that erases to the terminal concrete state $h' \in [\![a]\!](h)$, and preserves $\triangleright r$, for all stable frames r. Intuitively, the $\triangleright$ operator expresses that executing an atomic action corresponds to one step of execution in the operational semantics.

Safety, $\mathit{safe}(s, p, q)$, extends satisfaction from atomic actions to statements s. Intuitively, it expresses that every step of s at the concrete level has a corresponding step at the abstract level. Formally, *safe* is defined using guarded recursion to establish the connection between steps in the underlying operational semantics and steps in the topos of trees. See the accompanying technical report for the formal definition.

Interpretation. Most of the interpretation of iCAP is fairly straightforward and reduces directly to the topos of trees. For instance, conjunction in iCAP is

interpreted using conjunction in the topos of trees: $p \wedge q \stackrel{\text{def}}{=} \{m \in \mathcal{M} \mid m \in p \wedge m \in q\}$. The most interesting case is the interpretation of the region interpretation assertion, $\mathsf{rintr}(-)$:

$$\begin{aligned}\mathsf{rintr}(I, r) \stackrel{\text{def}}{=} \{(l, s, \varsigma) \in \mathcal{M} \mid r \in dom(\varsigma) \wedge \forall x \in \Delta(SId).\\ \forall \varsigma' \geq \varsigma.\ app(\varsigma(r))(x, \varsigma') = \triangleright(\lambda(l, s).\ I(x)(l, s, \varsigma'))\}\end{aligned}$$

Readers familiar with models of ML references may understand this region interpretation assertion by analogy to the ref type constructor of ML, which can be modelled by a similar equation [2, 4].[6] The reference type in ML describes a simple invariant for a single location, which expresses that the values stored at that location are always of the given type. With iCAP we can describe invariants given by a protocol and covering a region of memory (varying according to the protocol).

4 Logic

In the accompanying appendix we introduce a formal proof system for iCAP and in the accompanying technical report we present the entire proof system. We stress that the logic contains sufficient proof rules for proving all the examples sketched in this paper, *including all stability proofs and all proofs about atomic instructions*!

In the accompanying technical report we prove that iCAP is sound with respect to the model described in the previous section. As a corollary of this soundness theorem it follows that if $\Gamma \mid - \vdash (\Delta).\{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\}$, then

$$\forall \vartheta \in [\![\Gamma]\!].\ safe(\mathsf{s}, [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta), [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta)).$$

5 Discussion

We have presented iCAP, the first program logic for modular reasoning about higher-order concurrent imperative programs that supports full impredicative quantification over general predicates, including predicates describing protocols over shared regions of memory. We have presented examples illustrating how iCAP supports modular reasoning about internal and external sharing, layered and recursive abstractions, and fine-grained concurrent ADTs implemented using helping, entirely in the logic.

We have discussed related work on program logics along the way. As an alternative to program logics, there has also been several recent advances on using relational models for reasoning about concurrent programs. In particular, Liang et. al. [15] presented a simulation relation based on rely-guarantee to verify program transformations for a first-order concurrent imperative language; Birkedal

[6] Think of ς as the world in models of references; then the equation says that, for all future worlds, the interpretation of the region recorded in the world agrees with the interpretation given by I.

et. al. [5] presented a logical relations model for verifying effect-based program transformations for a higher-order concurrent imperative language, and Turon et. al. [23, 22] extended [5] with an extension of the protocols of Dreyer et. al. [10] to allow for relational refinement proofs of sophisticated fine-grained concurrent algorithms, including cooperation. To reason about cooperation, the model and logic of Turon et. al. [23, 22] uses specification code (i.e., an expression of the programming language) as a transferrable resource. This is similar to how view shifts are transferred here to reason about cooperation; the difference is that here we do not use code (since we are not proving refinement), but allow for transfer of more abstract specifications given by view shifts. The model in [23] is defined using step-indexing and involves an indexed definition of what essentially amounts to a recursively defined space of protocols, similar in spirit to the one we are using in this paper. However, the model in [23] does not support impredicative protocols, technically since island predicates (corresponing to region predicates) in *loc. cit.* have a restriction on how they can be parameterized. It is probably possible to lift this restriction, but one would still need a richer notion of model in order to model impredicative higher-order logic, essentially since constant sets would no longer suffice. As explained earlier, we use the type theory of the topos of trees as our metatheory for that purpose.

In this paper we have focused on the foundational issue of establishing soundness of a new very expressive logic for reasoning about higher-order concurrent imperative programs. Future work includes implementing a tool for interactive verification of programs using iCAP. We plan to do so in Coq, following the approaches of the Bedrock [6] and Charge! [3] tools, which have been successful in using Coq tactics to automate large parts of formal reasoning.

Acknowledgements

This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

References

1. Node.js. http://www.nodejs.org.
2. A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A Very Modal Model of a Modern, Major, General Type System. In *Proceedings of POPL*, 2007.
3. J. Bengtson, J. Jensen, and L. Birkedal. Charge! a framework for higher-order separation logic in Coq. In *Proceedings of ITP*, 2012.
4. L. Birkedal, R. Møgelberg, J. Schwinghammer, and K. Støvring. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *Proceedings of LICS*, 2011.
5. L. Birkedal, F. Sieczkowski, and J. Thamsborg. A Concurrent Logical Relation. In *Proceedings of CSL*, 2012.
6. A. Chlipala. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. In *Proceedings of PLDI*, 2011.

7. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of POPL*, 2013.
8. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of ECOOP*, 2010.
9. M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. In *Proceedings of POPL*, pages 259–270, 2011.
10. D. Dreyer, G. Neis, and L. Birkedal. The Impact of Higher-Order State and Control Effects on Local Relational Reasoning. In *Proceedings of ICFP*, 2010.
11. X. Feng, R. Ferreira, and Z. Shao. On the Relationship between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *Proceedings of ESOP*, 2007.
12. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local Reasoning for Storable Locks and Threads. In *Proceedings of APLAS*, pages 19–37, 2007.
13. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
14. A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. In *Proceedings of ESOP*, pages 353–367, 2008.
15. H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.
16. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
17. S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell, 1975.
18. N. Provos and N. Mathewson. libevent – an event notification library. http://www.monkey.org/~provos/libevent.
19. C. V. Russo. The Joins Concurrency Library. In *Proceedings of PADL*, 2007.
20. K. Svendsen, L. Birkedal, and M. Parkinson. Joins: a Case Study in Modular Specification of a Concurrent Reentrant Higher-order Library. In *Proceedings of ECOOP*, 2013.
21. K. Svendsen, L. Birkedal, and M. Parkinson. Modular Reasoning about Separation of Concurrent Data Structures. In *Proceedings of ESOP*, 2013.
22. A. Turon, D. Dreyer, and L. Birkedal. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *Proceedings of ICFP*, 2013.
23. A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical Relations for Fine-Grained Concurrency. In *Proceedings of POPL*, 2013.
24. V. Vafeiadis and M. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *Proceedings of CONCUR*, 2007.