Joins: A Case Study in Modular Specification of a Concurrent Reentrant Higher-order Library

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. We present a case study of formal specification for the C♯ joins library, an advanced concurrent library implemented using both shared mutable state and higher-order methods. The library is specified and verified in HOCAP, a higher-order separation logic extended with a higher-order variant of concurrent abstract predicates.

1 Introduction

It is well-known that modular specification and verification of concurrent higher-order imperative programs is very challenging. In the last decade good progress has been made on reasoning about subsets of these programming language features. For example, higher-order separation logic with nested triples has proved useful for modular specification and verification of higher-order imperative programs that use state with little sharing, e.g., [22, 16, 15]. Nested triples support specification of higher-order methods and higher-order quantification allows library specifications to abstract over the internal state maintained by the library and the state effects of function arguments.

Likewise, concurrent abstract predicates [7] has proved useful for reasoning about shared mutable data structures in a concurrent setting. Concurrent abstract predicates (CAP) extends separation logic with protocols governing access to shared mutable state. Thus CAP supports modular specification of shared mutable data structures that abstract over the *internal sharing*, e.g., [5]. However, CAP does not support modular reasoning about *external sharing* – the sharing of *other* mutable data structures through a shared mutable data structure. For instance, CAP does not support modular reasoning about locks[3] – the canonical example of a shared mutable data structure used to facilitate external sharing.

We have recently proposed HOCAP [25], a new program logic which combines higher-order separation logic with concurrent abstract predicates and extends concurrent abstract predicates with *state-independent higher-order protocols*. To reason about external sharing through a data structure, we parameterise the specification of the data structure with assertions that clients can instantiate to describe the resources they wish to share through the data structure. Higher-order protocols allow us to impose protocols on these external resources when

[3] See Section 6 for a discussion of this issue.

reasoning about the implementation of the data structure. State-independent higher-order protocols allow us to reason about non-circular external sharing patterns.

HOCAP is thus intended as a general purpose program logic for modular specification and verification of concurrent higher-order imperative programs with support for modular reasoning about *both* internal and external sharing. We have previously verified simple examples in HOCAP. In this paper we report on an extensive case study of a sophisticated and realistic library that combines all these challenges in one, to test whether HOCAP can in fact be used to give an abstract formal specification.

In particular, we explore how to give a modular specification to a concurrent library that features internal sharing and is used to facilitate external sharing. Clients interact with the library using reentrant callbacks. The specification should thus abstract over the internal state while allowing abstract reasoning about external sharing through the library and reentrant calls back into the library. Furthermore, the specification should of course be strong enough to reason about clients, and weak enough to allow the implementation of the library to be verified against the specification.

Our case study of choice is the C$^\sharp$ joins library [20]. The joins library, which is based on the join calculus [8, 9], provides a declarative concurrency model based on message passing. Declarative message patterns are used to specify synchronisation conditions and function arguments are used to specify synchronisation actions. Synchronisation actions might themselves cause new messages to be sent, leading to reentrant callbacks. The joins concurrency model is useful for defining new synchronisation primitives – i.e., to facilitate external sharing. Finally, the library itself is implemented using internal state.

In this paper we present a formal specification of a subset of the C$^\sharp$ joins library in HOCAP. The specification is expressed in terms of the high-level join primitives exposed by the library and hides all internal state from clients. Moreover, we test the specification of the joins library by verifying a number of synchronisation primitives for which there are already accepted specifications in the literature. For example, we verify that a reader-writer lock implemented using joins can be proved to satisfy the standard separation logic specification for a reader-write lock. We have chosen to focus on synchronisation primitives because synchronisation primitives are specifically designed to facilitate external sharing.

In addition to its role as a case study of a higher-order reentrant concurrent library the specification of the joins library is itself interesting. The main idea behind the specification is to allow clients of the joins library to impose ownership transfer protocols at the level of the join primitives exposed by the library. As illustrated with several examples, this leads to natural and short proofs of synchronisation primitives implemented using the joins library.

We have also verified a simple lock-based implementation of the joins library. However, in this paper we focus on the joins specification and the use thereof, since the main point is to investigate how HOCAP can be used to give abstract specifications for concurrent higher-order imperative libraries. We refer

the interested reader to the accompanying technical report for details about the verification of the joins implementation [24].

Outline. The remainder of the paper is organised as follows. In Section 2 we give an extensive introduction to the joins library using a series of examples to explain each feature of the library. Along the way, we sketch how one can reason informally, in separation-logic style, about the correctness of the applications. In Section 3 we summarise the necessary bits of HOCAP. This leads us to Section 4, where we introduce the formal specification of the joins library. In Section 5 we revisit a couple of the example applications and show how the informal proof sketches from Section 2 can be turned into formal proofs using the formal specification from Section 4. Finally, we evaluate and discuss the case study in Section 6.

2 Introducing joins

The joins concurrency model is based on the concept of messages, which are used both for synchronisation and communication between threads. Conceptually, a join instance consists of a single message pool and a number of *channels* for adding messages to this pool. Channels come in two varieties, *synchronous* and *asynchronous*. Sending a message via a synchronous channel adds the message to the message pool and blocks the sender until the message has been received. Asynchronous channels simply add messages to the message pool, without blocking the sender.

The power of the joins calculus stems from how messages are received. One declares a set of *chords*, each consisting of a *pattern* (a condition on the message pool) and a continuation. When a pattern matches a set of messages in the message pool, the chord may *fire*, causing the continuation to execute. Crucially, once a chord fires the messages that matched the pattern are removed from the message pool *atomically*, making them unavailable for future matches. Upon termination of the continuation, the clients that added the removed messages via synchronous channels are woken up and allowed to continue. We say that a message has been *received* when it has been matched by a chord and the chord continuation has terminated.

In the rest of this section we introduce the C# joins library, one feature at a time. Each new feature is introduced with a joins example of a synchronisation primitive implemented using this feature. For each example, we sketch an informal proof of the synchronisation primitive in separation logic. The examples thus serve both to introduce the joins library and motivate the main ideas behind our formal specification of the joins library.

We take as a starting point Russo's joins library for C$^\sharp$ [20], with a slightly simplified API. In particular, we have omitted value-carrying channels, as value-carrying channels do not add any conceptual difficulties.

2.1 Synchronous channels

Sending a message via a synchronous channel causes the sender to block until the message has been received. To illustrate, we consider the example of a 2-barrier – an asymmetric barrier restricted to two clients.

Implementation. One can implement a 2-barrier as a joins instance with two synchronous channels – one for each client to signal its arrival. Clients should block at the barrier until both clients have signalled their arrival. This can be achieved with a single chord with a pattern that allows it to fire when there is a pending message on both channels (i.e., when both clients have arrived). The C♯ code for a 2-barrier is given in Figure 1.

The TwoBarrier constructor creates a join instance, j, and two synchronous channels, ch1 and ch2, attached to the underlying message pool of this join instance. Next, the constructor creates a pattern p that matches any pair of messages in the message pool consisting of a ch1 message (i.e., a message added via the ch1 channel) and a ch2 message. Lastly, it registers this pattern as a chord without a continuation. Hence, this chord may fire when there is a pending message on both channels and when it fires, it atomically removes and receives these two messages from the message pool. Each Arrive method signals the client's arrival by sending a message on the corresponding channel using the Call method.

```
class TwoBarrier {
  private SyncChannel ch1;
  private SyncChannel ch2;

  public TwoBarrier() {
    Join j = new Join();
    ch1 = new SyncChannel(j);
    ch2 = new SyncChannel(j);
    Pattern p = j.When(ch1).And(ch2);
    p.Do();
  }

  public void Arrive1() { ch1.Call(); }
  public void Arrive2() { ch2.Call(); }
}
```

Fig. 1. Joins 2-barrier implementation.

All the examples we consider in this article follow the same structure as the above example: the constructor creates a join instance with accompanying channels and registers a number of chords. After this initialisation phase, the chords and channels stay fixed and interaction with the joins instance is limited to the sending of messages.

We now sketch a proof of this 2-barrier implementation using separation logic. Recall that separation logic assertions, say P and Q, describe and assert ownership of resources and that $P * Q$ holds if P and Q describe (conceptually) disjoint resources. The logic will be introduced in greater detail in Section 4 when we get to the formal specification and formal reasoning.

Desired specification. From the point of view of resources, a barrier allows clients to exchange resources. We call these resources external as they are typically external to the barrier data structure itself. On arrival at the barrier each client may transfer ownership of some resource to the barrier, which is then redistributed atomically once both clients have arrived. For the purpose of this

introduction we will make the simplifying assumption that each client transfers the same resource to the barrier on each arrival and that these resources are redistributed in the same way at each round of synchronisation. In Section 5.2 we consider a general specification without these simplifying assumptions.

Under these assumptions we can specify the barrier in terms of two predicates, B_i^{in} and B_i^{out}, where B_i^{in} describes the resources client i transfers to the barrier upon arrival, and B_i^{out} describes the resources client i expects to receive back from the barrier upon leaving. These predicates thus describe the external resources clients intend to share through the barrier. Since a barrier can only redistribute resources (i.e., it cannot create resources out of thin air), the combined resources transferred to the barrier must imply the combined resources transferred back from the barrier: $B_1^{\mathsf{in}} * B_2^{\mathsf{in}} \Rightarrow B_1^{\mathsf{out}} * B_2^{\mathsf{out}}$.

The *client* of the barrier is thus free to pick any B_i^{in} and B_i^{out} predicates satisfying the above redistribution property. We can now express the expected specification of a 2-barrier b in terms of these abstract predicates:

$$\{B_1^{\mathsf{in}}\}\ \mathsf{b.Arrive1()}\ \{B_1^{\mathsf{out}}\} \qquad\qquad \{B_2^{\mathsf{in}}\}\ \mathsf{b.Arrive2()}\ \{B_2^{\mathsf{out}}\}$$

That is, for client 1 to arrive at the barrier (i.e., to call $\mathsf{Arrive1}$), it has to provide the resource described by B_1^{in}, and if the call to $\mathsf{Arrive1}$ terminates (i.e., client 1 has left the barrier), it will have received the resource described by B_1^{out}.

Proof sketch. The main idea behind our specification of the joins library is to allow clients to impose an ownership transfer protocol on messages. An ownership transfer protocol consists of a *channel precondition* and a *channel postcondition* for each channel. The channel precondition describes the resources the sender is required to transfer to the recipient when sending a message on the channel. The channel postcondition describes the resources the recipient is required to transfer to the sender upon receiving the message.

In the 2-barrier example, sending a message on a channel corresponds to signalling one's arrival at the barrier. The channel preconditions of the barrier thus describe the resources clients are required to transfer to the barrier upon their arrival. Hence, we take each channel precondition to be the corresponding B^{in} predicate: $P_{\mathsf{ch1}} = B_1^{\mathsf{in}}$ and $P_{\mathsf{ch2}} = B_2^{\mathsf{in}}$. Throughout this section we use the notation P_{ch} to refer to the channel precondition of channel ch and Q_{ch} to refer to the channel postcondition.

The barrier implementation features a single chord that matches and receives both arrival messages, once both clients have arrived. The channel postconditions of the barrier thus describes the resources the barrier is required to transfer back to the clients, once both clients have arrived. We thus take each channel postcondition to be the corresponding B^{out} predicate: $Q_{\mathsf{ch1}} = B_1^{\mathsf{out}}$ and $Q_{\mathsf{ch2}} = B_2^{\mathsf{out}}$.

One can thus think of the channel pre- and postconditions as specifications for channels. Since the channel postcondition describes the resources transferred back to the sender *once* its message has been received, one should think of it as a partial correctness specification. In particular, without any chords to receive messages on a given channel we can pick any channel postcondition, as no message sent on that channel will ever be received. Conversely, whenever

we add a new chord we have to prove that it satisfies the chosen ownership transfer protocol. For chords without continuations, this reduces to proving that the preconditions of the channels that match the chord's pattern imply the postconditions of these channels.

The 2-barrier consists of a single chord that matches any pair of messages consisting of a ch1 message and a ch2 message. Correctness thus reduces to proving $P_{\mathsf{ch1}} * P_{\mathsf{ch2}} \Rightarrow Q_{\mathsf{ch1}} * Q_{\mathsf{ch2}}$, which follows from the assumed redistribution property.

2.2 Asynchronous channels

The previous example illustrated the use of synchronous channels that block the sender until its message has been received. The joins library also supports *asynchronous* channels, allowing messages to be sent without blocking the sender. A lock is a simple example that illustrates the use of both asynchronous and synchronous channels. Acquiring a lock must wait for the previous thread using the lock to finish: it is synchronous. However, releasing a lock need not wait for the next thread to attempt to acquire it: it is asynchronous.

Implementation. We can implement a lock using the joins library as follows:

We use two channels acq and rel to represent the two actions one can perform on a lock. The join instance has a single chord with a pattern that matches any pair of messages consisting of an acq message and a rel message. Thus, to acquire the lock, a thread sends a message on the acq channel; the call will block until the chord fires, which can only happen if there is a rel message in the message pool. The lock is thus unlocked if and only if there is a pending rel message in the message pool. The release can happen asynchronously; it does not have to wait for the next thread to attempt to acquire the lock.

The lock is initially unlocked by calling rel.

```
class Lock {
  private SyncChannel acq;
  private AsyncChannel rel;

  public Lock() {
    Join j = new Join();
    acq = new SyncChannel(j);
    rel = new AsyncChannel(j);
    j.When(acq).And(rel).Do();
    rel.Call();
  }

  public void Acquire() { acq.Call(); }
  public void Release() { rel.Call(); }
}
```

Fig. 2. A Joins implementation of a lock.

Desired specification. Locks are used to ensure exclusive access to some shared resource. We can specify a lock in separation logic in terms of an abstract resource predicate R (picked by the client of the lock) as follows:

$$\{R\}\ \mathsf{new\ Lock()}\ \{\mathrm{emp}\} \qquad \{\mathrm{emp}\}\ \mathsf{l.Acquire()}\ \{R\} \qquad \{R\}\ \mathsf{l.Release()}\ \{\mathrm{emp}\}$$

When the lock is unlocked the resource described by R is owned by the lock. Upon acquiring the lock, the client takes ownership of R, until it releases the lock again. Since the lock is initially unlocked, creating a new lock requires ownership of R to be transferred to the lock. This is the standard separation logic

specification for a lock [17, 10, 11]. Here R thus describes the external resources shared through the lock.

Proof sketch. Informally, we can understand the rel message as moving the resource protected by the lock from the thread to the join instance, and the acq message as doing the converse. This can be stated more formally using channel pre- and postconditions as follows: $P_{\mathsf{acq}} = \text{emp}$, $Q_{\mathsf{acq}} = R$, $P_{\mathsf{rel}} = R$, and $Q_{\mathsf{rel}} = \text{emp}$.

Recall that channel postconditions describe the resources the recipient is required to transfer to the sender upon receiving the message. Since the sender of a message on an asynchronous channel has no way of knowing if its message has been received, channel postconditions do not make sense for asynchronous channels. We thus require channel postconditions for asynchronous channels to be empty, emp.

As before, to prove that the acq and rel chord satisfies the channel postconditions, we have to show that the combined channel preconditions imply the combined channel postconditions: $P_{\mathsf{acq}} * P_{\mathsf{rel}} \Rightarrow Q_{\mathsf{acq}} * Q_{\mathsf{rel}}$. This follows directly from the fact that $*$ is commutative.

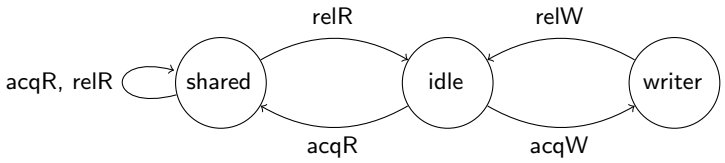
2.3 Continuations

So far, every chord we have considered has simply matched and removed messages from the message pool. In general, a chord can have a continuation that is executed when the chord fires, before any blocked synchronous senders are allowed to continue.

Continuations can, for instance, be used to automatically send a message on a certain channel when a chord fires. Thus they can be used to encode a state machine. Moreover, one can also ensure that a state of the state machine is correlated with the history or state of the synchronisation primitive that one is implementing. To illustrate, we extend the lock from the previous example into a biased reader/writer lock.[4]

A reader/writer lock [4] generalises a lock by introducing read-only permissions. This allows multiple readers to access a shared resource concurrently. To determine whether a read or write access request should be granted, three states suffice: (idle) no readers or writers, (writer) exactly one writer, or (shared) one or more readers. In the idle state there are no readers or writers, so it is safe to grant both read and write access. In the shared state, as one client has already been granted read access, it is only safe to grant read access. We can express this as a state machine as follows:

[4] Biased here means that this reader/writer implementation may starve the writer thread. It is possible to extend this implementation into an unbiased reader/writer lock by introducing an additional asynchronous channel to distinguish between whether or not there are any pending writers when a reader request has been granted.



Here acqR and acqW refers to the acquire read and acquire write operation, and relR and relW refers to the release read and release write operation.

Implementation. The idea is to encode this state machine using three asynchronous channels, idle, shared, and writer, with the invariant that there is at most one pending asynchronous message in the message pool at any given time. This gives a direct encoding of the three states in the above state machine, and adds a fourth intermediate state (when there is no pending message on any of the three asynchronous channels). The intermediate state is necessary for the implementation, as it does not transition atomically between the states of the above state machine. The joins implementation is given below.

```
class RWLock {
  private SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel idle, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join j = new Join();
    // ... initialise channels ...

    j.When(acqR).And(idle).Do(AcqR);
    j.When(acqR).And(shared).Do(AcqR);
    j.When(relR).And(shared).Do(RelR);
    j.When(acqW).And(idle).Do(writer.Call);
    j.When(relW).And(writer).Do(idle.Call);

    idle.Call();
  }

  private void AcqR() {
    readers++;
    shared.Call();
  }

  private void RelR() {
    if (--readers == 0)
      idle.Call();
    else
      shared.Call();
  }

  public void AcquireR() { acqR.Call(); }
  public void AcquireW() { acqW.Call(); }
  public void ReleaseR() { relR.Call(); }
  public void ReleaseW() { relW.Call(); }
}
```

We use three asynchronous channels to encode the current state in the above state machine and thus to determine whether a read or write access can be granted. In addition, we use the readers field to count the actual number of readers, to determine which state to transition to when releasing a reader. Note that the continuation given to Do is a named C♯ delegate, and that in all five cases, the given continuation sends a message on an asynchronous channel. These calls are *reentrant* calls back into the joins library, making these continuations *reentrant callbacks*.

Note further that all five chords consume exactly one asynchronous message and sends exactly one asynchronous message. Between consuming and sending the asynchronous message, there are no pending asynchronous messages and the reader/writer is in the previously mentioned fourth state. Hence, between

consuming and sending an asynchronous message, no other chord can fire and the currently executing continuation has exclusive access to the internal state of the reader/writer lock (i.e., the readers field).

Desired specification. The standard separation logic specification for a reader/writer lock is expressed using counting permissions [2]. Counting permissions allow a full write permission to be split into any number of read permissions, counting the total number of read permissions, to allow them to be joined up to a full write permission later. The standard specification is given below in terms of an abstract resource predicate for writing to the resource R_{write} and an abstract resource predicate for reading the resource R_{read}.

$$\begin{array}{rll} \{R_{write}\} & \textbf{new RWLock()} & \{\text{emp}\} \\ \{\text{emp}\} & \text{l.AcquireR()} & \{R_{read}\} \\ \{\text{emp}\} & \text{l.AcquireW()} & \{R_{write}\} \\ \{R_{read}\} & \text{l.ReleaseR()} & \{\text{emp}\} \\ \{R_{write}\} & \text{l.ReleaseW()} & \{\text{emp}\} \end{array} \tag{1}$$

To avoid introducing counting permissions directly, we specify the reader/write lock in terms of an additional family of abstract resource predicates $R(n)$, indexed by $n \in \mathbb{N}$, satisfying that $R(0)$ is the full write permission R_{write}, and $R(n)$ is the permission left after splitting off n read permissions. Thus R should satisfy, $\forall n \in \mathbb{N}.\ R(n) \Leftrightarrow R_{read} * R(n+1)$ and $R(0) \Leftrightarrow R_{write}$. Note that a client of the reader/writer lock is free to pick any R_{write}, R_{read} and R that satisfies these two properties.

Proof sketch. The three asynchronous channels encode the current state of the reader/writer lock. The channel preconditions of the three asynchronous channels thus describe the resources owned by the reader/writer lock in the idle, shared and writer state, respectively. In the idle state (no readers or writers), the reader/writer lock owns the readers field and the full write permission, and the readers field contains 0. In the shared state (one or more readers), the reader/writer lock owns the readers field and the remaining permission after splitting off n read permissions and the readers field contains n. Lastly, in the writer state (exactly one writer), the writer owns the full resource and the reader/writer lock only owns the readers field.

$$P_{\mathsf{idle}} = \mathsf{readers} \mapsto 0 * R(0) \qquad P_{\mathsf{writer}} = \mathsf{readers} \mapsto 0$$
$$P_{\mathsf{shared}} = \exists n \in \mathbb{N}.\ n > 0 * \mathsf{readers} \mapsto n * R(n)$$

Since idle, shared, and writer are asynchronous, their channel postconditions must be empty (as explained earlier).

For the synchronous channels we can read off their channel pre- and postconditions directly from the desired specification (1):

$$P_{\mathsf{acqR}} = \text{emp} \qquad Q_{\mathsf{acqR}} = R_{read} \qquad P_{\mathsf{acqW}} = \text{emp} \qquad Q_{\mathsf{acqW}} = R(0)$$
$$P_{\mathsf{relR}} = R_{read} \qquad Q_{\mathsf{relR}} = \text{emp} \qquad P_{\mathsf{relW}} = R(0) \qquad Q_{\mathsf{relW}} = \text{emp}$$

To register a chord without a continuation we had to show that the combined channel preconditions *implied* the combined channel postconditions. What about

the present case with a proper continuation? Since the continuation runs before the release of any blocked synchronous callers, we have to show that the continuation *transforms* the combined channel preconditions to the combined channel postconditions. For the reader/writer lock we thus have to show the proof obligations on the left in Figure 3. These proof obligations are all completely standard and mostly trivial separation logic proofs. For instance, the proof of the first obligation is given on the right in Figure 3. Note that in this proof we use the

$$
\begin{array}{ccc}
\{P_{\mathsf{acqR}} * P_{\mathsf{idle}}\} & \mathsf{AcqR()} & \{Q_{\mathsf{acqR}} * Q_{\mathsf{idle}}\} \\
\{P_{\mathsf{acqR}} * P_{\mathsf{shared}}\} & \mathsf{AcqR()} & \{Q_{\mathsf{acqR}} * Q_{\mathsf{shared}}\} \\
\{P_{\mathsf{relR}} * P_{\mathsf{shared}}\} & \mathsf{RelR()} & \{Q_{\mathsf{relR}} * Q_{\mathsf{shared}}\} \\
\{P_{\mathsf{acqW}} * P_{\mathsf{idle}}\} & \mathsf{writer.Call()} & \{Q_{\mathsf{acqW}} * Q_{\mathsf{idle}}\} \\
\{P_{\mathsf{relW}} * P_{\mathsf{writer}}\} & \mathsf{idle.Call()} & \{Q_{\mathsf{relW}} * Q_{\mathsf{writer}}\}
\end{array}
$$

$\{P_{acqR} * P_{idle}\}$
$\{readers \mapsto 0 * R(0)\}$
  readers++;
$\{readers \mapsto 1 * R(0)\}$
$\{readers \mapsto 1 * R_{\mathsf{read}} * R(1)\}$
  shared.Call();
$\{R_{\mathsf{read}}\}$
$\{Q_{acqR} * Q_{idle}\}$

Fig. 3. Left: Proof obligations for the reader/writer lock chords. Right: A proof sketch for the first proof obligation of the reader/writer lock.

channel pre- and postcondition of the shared channel. These proofs thus have a similar character to partial correctness proofs of a recursive method, where one is allowed to assume the specification of a method while proving that its body satisfies the assumed specification. Here we assume the shared channel obeys the chosen ownership transfer protocol while proving that the first chord obeys the chosen protocol.

2.4 Nonlinear patterns

The public interface of the 2-barrier in Section 2.1 is slightly non-standard, as it has two distinct arrival methods. A more standard barrier interface would provide a common Arrive method, for both clients. The joins library also supports an implementation of a barrier with such an interface, through the use of *nonlinear patterns*. Nonlinear patterns match multiple messages from the same channel.

Implementation. We can thus implement a more standard 2-barrier as a joins instance with a single synchronous arrival channel and a single chord with a nonlinear pattern that matches two messages on the arrival channel. Clearly this generalises to an n-barrier, which can be implemented as follows.

```
class Barrier {
  private SyncChannel arrive;

  public Barrier(int n) {
    Join j = new Join(); arrive = new SyncChannel(j); Pattern p = j.When(arrive);
    for(int i = 1; i < n; i++) { p = p.And(arrive); };
    p.Do();
  }

  public void Arrive() { arrive.Call(); }
```

}

This code registers a single chord with a pattern that matches n messages on the synchronous arrive channel.

Desired specification. As before, assume predicates B_i^{in} and B_i^{out} (picked by the client), where B_i^{in} describes the resources client i transfers to barrier upon arrival and B_i^{out} describes the resources client i expects to receive back from the barrier upon leaving. These predicates should satisfy the following redistribution property, $\circledast_{i\in\{1,\ldots,n\}} B_i^{\mathsf{in}} \Rightarrow \circledast_{i\in\{1,\ldots,n\}} B_i^{\mathsf{out}}$, to allow the barrier to redistribute the combined resources, once every client has arrived.

From the informal description of B_i^{in} and B_i^{out} one might thus expect an n-barrier b to satisfy the following specification:

$$\{B_i^{\mathsf{in}}\}\ \mathsf{b.Arrive()}\ \{B_i^{\mathsf{out}}\}$$

That is, if a client transfers B_i^{in} to the barrier upon arrival, it should receive back B_i^{out} from the barrier upon leaving. However, this specification is not quite right. In particular, what prevents client i from impersonating client j when it arrives at the barrier? To apply the redistribution property to the combined resources transferred to the barrier we need to ensure that when client i arrives at the barrier, it actually transfers B_i^{in} to the barrier, even if it also happens to own B_j^{in}. Hence, while the barrier *implementation* no longer distinguishes between clients, we still need a way to distinguish clients *logically*. We can achieve this by introducing a client identity predicate, $\mathsf{id}(i)$ to assert that the owner is client i. By making this predicate non-duplicable, we can enforce that clients cannot impersonate each other.

We can now express a correct barrier specification in terms of this id predicate as follows:

$$\{\mathrm{emp}\}\ \textbf{new}\ \mathsf{Barrier(n)}\ \{\circledast_{i\in\{1,\ldots,\mathsf{n}\}}\mathsf{id}(i)\} \qquad \{B_i^{\mathsf{in}} * \mathsf{id}(i)\}\ \mathsf{b.Arrive()}\ \{B_i^{\mathsf{out}} * \mathsf{id}(i)\}$$

Upon creation of a new n-barrier we get back n id assertions. These are then distributed to each client to witness their identity when they arrive at the barrier.

Proof sketch. Our proof sketch of the 2-barrier in Section 2.1 exploited that the implementation used distinct channels to signal the arrival of each client, which allowed us to pick different channel pre- and postconditions for each client. Since the above implementation uses a single arrival channel we have to pick a common channel pre- and postcondition that works for every client. We can achieve this using a *logical argument* to relate the channel precondition and the channel postcondition. In this case we index the channel pre- and postcondition with the client identifier i: $P_{\mathsf{arrive}}(i) = B_i^{\mathsf{in}} * \mathsf{id}(i)$ and $Q_{\mathsf{arrive}}(i) = B_i^{\mathsf{out}} * \mathsf{id}(i)$.

For the id predicate to witness the identity of clients, it must be non-duplicable. That is, it must satisfy, $\mathsf{id}(i) * \mathsf{id}(j) \Rightarrow i \neq j$. To define the id predicate such that it satisfies the above property, we need to introduce a bit more of our logic. We return to this example in Section 5.2.

3 Logic

The program logic is a higher-order separation logic [1] with support for reasoning about concurrency, shared mutable data structures [7, 6], and recursive delegates [22]. We use this one program logic to reason about both *clients* of the joins library, and an *implementation* of the joins library.

Our program logic is a general purpose logic for reasoning about higher-order concurrent C♯ programs. We have presented the logic in a separate paper [25]. The full logic and its soundness proof is included in the accompanying technical report [23] of that paper. For the present paper we limit our attention to those features necessary to verify our client examples. To this end, it suffices to consider a minor extension of higher-order separation logic with fractional permissions, phantom/auxiliary state and nested triples [21].

Higher-order separation logic. Every specification in Section 2 was expressed in terms of abstract resource predicates, such as the lock invariant R. This is easily and directly expressible in a higher-order logic, by quantification over predicates [1, 19].

Our assertion logic is an intuitionistic higher-order separation logic over a simply typed term language. The set of types is closed under function space and products, and includes the type of propositions, Prop, the type of specifications, Spec, and the type of mathematical values, Val. The Val type includes all C♯ values and strings, and is closed under formation of pairs, such that mathematical sequences and other mathematical objects can be conveniently represented.[5]

Fractional permissions. The notion of ownership in standard separation logic is very limited, supporting only two extremes: exclusive ownership and no ownership. To formalise the examples from the previous section we need a middle ground of read-only permissions, which can be freely duplicated. Fractional permissions [3] provide a popular solution to this problem, by annotating the points-to predicate with a fraction $p \in (0, 1]$, written $x.f \overset{p}{\mapsto} v$. Full permission corresponds to $p = 1$ and grants exclusive access to the field f. Permissions can be split and combined arbitrarily ($x.f \overset{p}{\mapsto} v * x.f \overset{q}{\mapsto} v \Leftrightarrow x.f \overset{p+q}{\mapsto} v$). Any fraction less than 1 grants partial read-only access to the field f. We write $x.f \mapsto v$ as shorthand for $x.f \overset{1}{\mapsto} v$ and $x.f \overset{-}{\mapsto} v$ as shorthand for $\exists p \in (0, 1].\ x.f \overset{p}{\mapsto} v$.

Phantom state. Auxiliary variables [18] are commonly used as an abstraction of the history of execution and state in Hoare logics. Normally, one declares a subset of program variables as auxiliary variables that can be updated using standard variable assignments, but are not allowed to affect the flow of execution. To support this style of reasoning, we extend separation logic with phantom state. Like standard auxiliary variables, phantom state allows us to record an abstraction of the history of execution, but unlike standard auxiliary variables phantom state is purely a logical construct (i.e., the operational semantics of the programming language is not altered to accommodate phantom state and

[5] We use a single universe Val for the universe of mathematical values to avoid also having to quantify over types in the logic. We omit explicit encodings of pairs and write $(v_1, ..., v_n)$ for tuples coded as elements of Val.

phantom state is not updated through programming level assignments). When combined logical arguments, phantom state allows us to logically distinguish and relate multiple messages on the same channel, as needed for the n-barrier example.

Phantom state extends objects with a logical notion of phantom fields and an accompanying phantom points-to predicate, written $x_f \overset{p}{\mapsto} v$, to make assertions about the value and ownership of these phantom fields. To support read-only phantom fields, we further enrich the notion of ownership with fractional permissions. Thus $x_f \overset{p}{\mapsto} v$ asserts the ownership of phantom field f of object x, with fractional permission p, and that this phantom field currently contains the value v. Like the normal points-to predicate, phantom points-to satisfies $x_f \overset{p_1}{\mapsto} v_1 * x_f \overset{p_2}{\mapsto} v_2 \Rightarrow v_1 = v_2$ as phantom fields contain a single fixed value at any given point in time.

Phantom fields are updated using a *view shift*. The notion of a view shift comes from the Views framework for compositional reasoning about concurrency [6], and generalises assertion implication. A view shift from assertion p to assertion q is written $p \sqsubseteq q$. Views shifts can be applied to pre- and postconditions using the following generalised rule of consequence:

$$\frac{p \sqsubseteq p' \qquad \{p'\}c\{q'\} \qquad q' \sqsubseteq q}{\{p\}c\{q\}}$$

Given full ownership (fractional permission 1) of a phantom field f, one can perform a logical update of the field ($x_f \overset{1}{\mapsto} v_1 \sqsubseteq x_f \overset{1}{\mapsto} v_2$). 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.

Figure 4 contains a selection of inference rules from our program logic, related to view shifts and phantom state.

Nested triples. To reason about delegates we use nested triples [21]. We write $x \mapsto \{P\}\{Q\}$ to assert that x refers to a delegate satisfying the given specification.

$$\frac{p \Rightarrow q}{p \sqsubseteq q} \qquad \frac{p \sqsubseteq q}{p * r \sqsubseteq q * r} \qquad \frac{p \sqsubseteq p' \qquad \{p'\}c\{q'\} \qquad q' \sqsubseteq q}{\{p\}c\{q\}}$$

$$\frac{}{x_f \overset{1}{\mapsto} v_1 \sqsubseteq x_f \overset{1}{\mapsto} v_2} \qquad \frac{}{x_f \overset{p_1}{\mapsto} v_1 * x_f \overset{p_2}{\mapsto} v_2 \Rightarrow v_1 = v_2} \qquad \frac{}{x_f \overset{p}{\mapsto} v * x_f \overset{q}{\mapsto} v \Leftrightarrow x_f \overset{p+q}{\mapsto} v}$$

Fig. 4. Selected program logic inference rules

Reasoning about the implementation. Fractional permissions introduce a more lenient ownership discipline that allows for read-only sharing. To verify the *implementation* of the joins library, we need even more general forms of sharing. To reason about general sharing patterns we base our logic on concurrent abstract predicates [7].

Conceptually, concurrent abstract predicates (CAP) partitions the heap into a set of regions that each come with a protocol governing how the state in that region may evolve. This allows *stable* assertions – assertions that are closed under changes permitted by the protocol – to be freely duplicated and shared. To ensure soundness, the logic requires that all pre- and postconditions in the specification logic are stable. We thus introduce a new type, SProp, of stable assertions.

Concurrent abstract predicates with first-order protocols (i.e., protocols that only refer to the state of their own region) suffice for reasoning about sharing of *primitive resources* such as individual heap cells. To reason about sharing of *shared resources* requires *higher-order protocols* that can relate the state of multiple regions. In general, to reason about sharing of shared resources requires reasoning about circular sharing patterns. HOCAP extends concurrent abstract predicates with a limited form of higher-order protocols – called state-independent higher-order protocols – and introduce partial orders to explicitly rule out these circular sharing patterns.

Since we are using the same program logic to reason about join clients and the underlying join implementation, join clients could themselves use CAP to describe shared resources when picking the channel pre- and postconditions. This could potentially introduce circular sharing patterns. To simplify the presentation and focus on the main ideas behind our specification of the joins library we have chosen to present a specification that *does not* allow clients to use CAP in their channel pre- and postconditions. This allows us to give a simple specification without any proof obligations about the absence of circular sharing patterns. In the accompanying technical report, we define a stronger joins specification that *does* allow clients to use CAP, but requires clients to prove the absence of circular sharing patterns. In the technical report we verify the joins implementation against this stronger specification. See Section 6 for further discussion.

To prevent joins clients from using CAP, we introduce a new type, LProp, of local propositions. Every predicate expressible in the language of higher-order separation logic extended with phantom state and nested triples is of type LProp, provided all higher-order quantifications quantify over LProp rather than Prop. However, LProp is not closed under region and action assertions for reasoning about shared mutable data structures using CAP. All assertions of type LProp are trivially stable and LProp is thus a subtype of SProp. We thus require all channel pre- and postconditions to be of type LProp. This ensures that clients do not introduce circular sharing patterns.

For details about the logic see our HOCAP paper and accompanying technical report [25, 23].

4 Joins specification

In this section we present our formal specification for the joins library.

The full specification of the joins library is presented in Figure 5. To simplify the specification and exposition of the joins library, we require all channels and chords be registered before clients start sending messages.[6] Formally, we introduce three phases:

ch: This phase allows new channels to be registered.
pat: This phase allows new chords to be registered.
call: This phase allows messages to be sent.

A newly created join instance starts in the ch phase. Once all channels have been registered, it transitions to the pat phase. Once all chords have been registered, it transitions to the call phase. In the call phase, the only way to interact with the join instance is by sending messages on its channels.

The specification is expressed in terms of a number of abstract representation predicates. We use three join representation predicates, $\mathsf{join}_{\mathsf{ch}}, \mathsf{join}_{\mathsf{pat}}$ and $\mathsf{join}_{\mathsf{call}}$ – one for each phase – which will be explained below. In addition, we use two representation predicates for channels and patterns:

$\mathsf{ch}(c, j)$: This predicate asserts that c refers to a channel registered with join instance j.
$\mathsf{pat}(p, j, X)$: This predicate asserts that p refers to a pattern on join instance j that matches the multi-set of channels X.

These representation predicates are all existentially quantified in the specification; clients thus reasons abstractly in terms of these predicates.

Channel initialisation phase. In the first phase we use the join representation predicate: $\mathsf{join}_{\mathsf{ch}}(A, S, j)$. This predicate asserts that j refers to a join instance with asynchronous channels A and synchronous channels S.

The join constructor (Join) returns a new join instance in the ch phase with no registered channels.

The two rules for creating and registering new channels (Sync and Async) take as argument a join instance j in the ch phase and return a reference to a new channel. In both cases, we get back a ch assertion, ch(r,j), that asserts that this newly created channel is registered with join instance j. In addition, both postconditions explicitly assert that this newly created channel is distinct from all previously registered channels, $\mathsf{r} \notin A \cup S$. As the channel predicate is duplicable ($\mathsf{ch}(c, j) \Leftrightarrow \mathsf{ch}(c, j) * \mathsf{ch}(c, j)$), to allow multiple clients to use the same channel, we have to state this explicitly.

Chord initialisation phase. In the second phase we use the join representation predicate: $\mathsf{join}_{\mathsf{pat}}(P, Q, j)$. This predicate asserts that j refers to a join instance with channel preconditions P and channel postconditions Q. Here P and Q are functions that assign channel pre- and postconditions to each channel. To relate the pre- and postcondition of a channel (as needed, e.g., in the n-barrier example to distinguish clients), we index each channel pre- and postcondition

[6] This restriction rules out reasoning about self-modifying synchronisation primitives. We are not aware of any examples of self-modifying join clients.

with a logical argument of type Val.[7] Formally P and Q are thus functions of type $P, Q : \text{Val} \times \tau_{\text{chan}} \to \text{LProp}$ where τ_{chan} is the type of channel references.[8]

Once sufficient channels have been registered, the join instance can transition into the chord initialisation phase using a view shift:

$$\frac{\forall c, a.\ c \in A \Rightarrow Q(a, c) = \text{emp}}{\mathsf{join}_{\mathsf{ch}}(A, S, j) \sqsubseteq \mathsf{join}_{\mathsf{pat}}(P, Q, j)}$$

This forces all channel pre- and postconditions to be fixed before any chords can be registered. This rule explicitly requires that the channel postconditions of asynchronous channel are empty, emp, as explained in Section 2.2.

Rules When and And create a new singleton pattern, and add new channels to an existing pattern, respectively. Note that a pattern matches a multi-set of channels and the set-union in And is thus multi-set union.

The rules for Do are more interesting. Rule Do1 deals with patterns without a continuation. Recall from our informal proof sketches that to add a new chord without a continuation we showed that the combined channel preconditions of the chord pattern *implied* the combined channel postconditions. Our specification generalises this to require that the combined channel preconditions can be *view shifted* to the combined channel postconditions. This generalisation allows us to perform logical updates of phantom state when the chord fires. We will see why this is useful in Section 5.2.

Furthermore, since our channel pre- and postconditions are now indexed by a logical argument, we have to prove that we can perform this view shift for any logical arguments (we have a logical argument for each channel). Formally,

$$\forall Y \in \mathcal{P}_m(\text{Val} \times \tau_{\text{chan}}).\ \pi_{\mathsf{ch}}(Y) = X \Rightarrow \circledast_{y \in Y} P(y) \sqsubseteq \circledast_{y \in Y} Q(y)$$

where $\mathcal{P}_m(-)$ denotes the finite power multi-set operator and π_{ch} is the power set lifting of π_2. Y thus associates a logical argument with each channel. To register a chord that matches channels x and y this thus reduces to two universally quantified logical arguments, say a and b:

$$\forall a, b \in \text{Val}.\ P(a, x) * P(b, y) \sqsubseteq Q(a, x) * Q(b, y)$$

The rule for Do with a continuation (Do2) is very similar, but instead of requiring a view-shift, it takes a delegate b that transforms the combined preconditions into the combined postconditions. Crucially, the delegate is given access to the join instance in the message phase. This enables it to send messages, as used in the reader/writer lock example (Section 2.3).

Message phase. The final phase allows messages to be send. We use a third abstract predicate, $\mathsf{join}_{\mathsf{call}}(P, Q, j)$, with the same parameters as the previous abstract predicate $\mathsf{join}_{\mathsf{pat}}(P, Q, j)$. Once all chords have been registered, the join instance can transition into the third phase using a view shift: $\mathsf{join}_{\mathsf{pat}}(P, Q, j) \sqsubseteq \mathsf{join}_{\mathsf{call}}(P, Q, j)$.

[7] As Val is closed under pairs this allows us to encode an arbitrary number of logical arguments of type Val.

[8] Formally, τ_{chan} is simply a synonym for Val, introduced to improve the exposition.

The only operation in the third phase is to send messages using Call. The rule for sending a message is very similar to the standard method call rule: we provide the precondition $P(a, c)$ and get back the postcondition $Q(a, c)$. Here a is the logical argument, which the client is free to pick.

Both the $\mathsf{join}_{\mathsf{call}}$ and $\mathsf{ch}(-)$ predicate is freely duplicable, to allow multiple clients to send messages on the same channel:

$$\mathsf{ch}(c, j) \Leftrightarrow \mathsf{ch}(c, j) * \mathsf{ch}(c, j)$$
$$\mathsf{join}_{\mathsf{call}}(P, Q, j) \Leftrightarrow \mathsf{join}_{\mathsf{call}}(P, Q, j) * \mathsf{join}_{\mathsf{call}}(P, Q, j)$$

Reasoning about joins. We have verified a simple lock-based implementation of the joins library (see the accompanying technical report for details). We have thus given concrete definitions of the abstract predicates pat, ch, $\mathsf{join}_{\mathsf{ch}}$, $\mathsf{join}_{\mathsf{pat}}$, $\mathsf{join}_{\mathsf{call}}$ and proved that the implementation satisfies a generalisation of the joins specification in Figure 5.

5 Reasoning with joins

In this section we revisit the lock and the n-barrier example, and sketch their formal correctness proofs in terms of our formal specification of the joins library. The lock example is intended to illustrate the joins specification in general, and has thus been written out in full. The n-barrier example is intended to illustrate the use of logical arguments and phantom state.

5.1 Lock

We begin by formalising the previous informal lock specification. As mentioned in Section 3, to avoid reasoning about sharing of shared mutable data structures through themselves, we require all channel pre- and postconditions to be local assertions – i.e., assertions of type LProp. Since the channel pre- and postconditions are defined in terms of the lock resource invariant, the lock resource invariant must be a local assertion. The formal specification of the lock is thus:

$$\begin{array}{l}\forall R : \mathrm{LProp}.\ \exists\, \mathsf{lock} : \mathrm{Val} \to \mathrm{SProp}.\\ \qquad \begin{array}{rl} \{R\}\ \mathbf{new}\ \mathsf{Lock}() & \{\mathsf{r}.\ \mathsf{lock}(\mathsf{r})\} \\ \{\mathsf{lock}(\mathsf{l})\}\ \ \mathsf{l.Acquire}() & \{\mathsf{lock}(\mathsf{l}) * R\} \\ \{\mathsf{lock}(\mathsf{l}) * R\}\ \ \mathsf{l.Release}() & \{\mathsf{lock}(\mathsf{l})\} \end{array}\\ \qquad \wedge\ \ \forall x : \mathrm{Val}.\ \mathsf{lock}(x) \Leftrightarrow \mathsf{lock}(x) * \mathsf{lock}(x)\end{array}$$

This specification introduces an explicit lock representation predicate, lock, which is freely duplicable.

We now formalise the proof sketch of the joins-based lock implementation from Section 2. Hence, for any predicate R, we have to define a concrete lock predicate and show that the above specifications for the lock operations hold for the concrete lock predicate.

Channel initialisation phase

$$\frac{}{\{\mathrm{emp}\}\,\textbf{new}\ \mathsf{Join}()\,\{\mathsf{r}.\ \mathsf{join}_{\mathsf{ch}}(\emptyset,\emptyset,\mathsf{r})\}}\ \textsc{Join}$$

$$\frac{}{\{\mathsf{join}_{\mathsf{ch}}(A,S,\mathsf{j})\}\,\textbf{new}\ \mathsf{SyncChannel}(\mathsf{j})\,\{\mathsf{r}.\ \mathsf{join}_{\mathsf{ch}}(A,S\cup\{\mathsf{r}\},\mathsf{j})*\mathsf{ch}(\mathsf{r},\mathsf{j})*\mathsf{r}\notin A\cup S\}}\ \textsc{Sync}$$

$$\frac{}{\{\mathsf{join}_{\mathsf{ch}}(A,S,\mathsf{j})\}\,\textbf{new}\ \mathsf{AsyncChannel}(\mathsf{j})\,\{\mathsf{r}.\ \mathsf{join}_{\mathsf{ch}}(A\cup\{\mathsf{r}\},S,\mathsf{j})*\mathsf{ch}(\mathsf{r},\mathsf{j})*\mathsf{r}\notin A\cup S\}}\ \textsc{Async}$$

Chord initialisation phase

$$\frac{}{\left\{\mathsf{join}_{\mathsf{pat}}(P,Q,\mathsf{j})*\mathsf{ch}(\mathsf{c},\mathsf{j})\right\}\mathsf{j}.\mathsf{When}(\mathsf{c})\left\{\mathsf{r}.\ \mathsf{join}_{\mathsf{pat}}(P,Q,\mathsf{j})*\mathsf{pat}(\mathsf{r},\mathsf{j},\{\mathsf{c}\})\right\}}\ \textsc{When}$$

$$\frac{}{\left\{\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\mathsf{pat}(\mathsf{p},j,X)*\mathsf{ch}(\mathsf{c},j)\right\}\mathsf{p}.\mathsf{And}(\mathsf{c})\left\{\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\mathsf{pat}(\mathsf{p},j,X\cup\{\mathsf{c}\})\right\}}\ \textsc{And}$$

$$\frac{\forall Y\in\mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y)=X\Rightarrow\circledast_{y\in Y}P(y)\sqsubseteq\circledast_{y\in Y}Q(y)}{\left\{\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\mathsf{pat}(\mathsf{p},j,X)\right\}\mathsf{p}.\mathsf{Do}()\left\{\mathsf{join}_{\mathsf{pat}}(P,Q,j)\right\}}\ \textsc{Do1}$$

$$\frac{}{\left\{\begin{array}{l}\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\mathsf{pat}(\mathsf{p},j,X)*\circledast_{z\in Z}\mathsf{ch}(z,j)\ *\\ \quad\forall Y\in\mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y)=X\Rightarrow\\ \qquad b\mapsto\{\mathsf{join}_{\mathsf{call}}(P,Q,j)*\circledast_{z\in Z}\mathsf{ch}(z,j)*\circledast_{y\in Y}P(y)\}\ \{\circledast_{y\in Y}Q(y)\}\end{array}\right\}\mathsf{p}.\mathsf{Do}(\mathsf{b})\left\{\mathsf{join}_{\mathsf{pat}}(P,Q,j)\right\}}\ \textsc{Do2}$$

Message phase

$$\frac{}{\{\mathsf{join}_{\mathsf{call}}(P,Q,j)*\mathsf{ch}(\mathsf{c},j)*P(a,\mathsf{c})\}\,\mathsf{c}.\mathsf{Call}()\,\{\mathsf{join}_{\mathsf{call}}(P,Q,j)*Q(a,\mathsf{c})\}}\ \textsc{Call}$$

Phase transitions

$$\frac{\forall c,a.\ c\in A\Rightarrow Q(a,c)=\mathrm{emp}}{\mathsf{join}_{\mathsf{ch}}(A,S,j)\sqsubseteq\mathsf{join}_{\mathsf{pat}}(P,Q,j)}\qquad\frac{}{\mathsf{join}_{\mathsf{pat}}(P,Q,j)\sqsubseteq\mathsf{join}_{\mathsf{call}}(P,Q,j)}$$

$$\mathsf{ch}(c,j)\Leftrightarrow\mathsf{ch}(c,j)*\mathsf{ch}(c,j)\qquad\mathsf{join}_{\mathsf{call}}(P,Q,j)\Leftrightarrow\mathsf{join}_{\mathsf{call}}(P,Q,j)*\mathsf{join}_{\mathsf{call}}(P,Q,j)$$

Abstract predicates

$$\mathsf{pat}:\tau_{\mathrm{pat}}\times\tau_{\mathrm{join}}\times\mathcal{P}_m(\tau_{\mathrm{chan}})\to\mathrm{SProp}\qquad\mathsf{ch}:\tau_{\mathrm{chan}}\times\tau_{\mathrm{join}}\to\mathrm{SProp}$$
$$\mathsf{join}_{\mathsf{ch}}:\mathcal{P}_m(\tau_{\mathrm{chan}})\times\mathcal{P}_m(\tau_{\mathrm{chan}})\to\mathrm{SProp}\qquad\mathsf{join}_{\mathsf{pat}},\mathsf{join}_{\mathsf{call}}:(\mathcal{E}\to\mathrm{LProp})\times(\mathcal{E}\to\mathrm{LProp})\times\mathrm{Val}\to\mathrm{SProp}$$

Here $\mathcal{P}_m(X)$ denotes the set of finite multi-subsets of X and

$$\tau_{\mathrm{join}}=\tau_{\mathrm{chan}}=\tau_{\mathrm{pat}}\stackrel{\mathrm{def}}{=}\mathrm{Val}\quad\mathcal{E}\stackrel{\mathrm{def}}{=}\mathrm{Val}\times\tau_{\mathrm{chan}}\quad\pi_{\mathsf{ch}}(X)\stackrel{\mathrm{def}}{=}\{\pi_2(x)\mid x\in X\}:\mathcal{P}_m(\mathcal{E})\to\mathcal{P}_m(\tau_{\mathrm{chan}})$$

Fig. 5. Specification of the joins library.

The channel pre- and postconditions do not change relative to the informal proof. For any pair of channels c_a and c_r we define the channel pre- and postcondition, $P(c_a, c_r), Q(c_a, c_r) : \mathcal{E} \to \text{LProp}$, as follows:

$$P(c_a, c_r)(a, c) = \begin{cases} \text{emp} & \text{if } c = c_a \\ R & \text{if } c = c_r \\ \bot & \text{otherwise} \end{cases} \qquad Q(c_a, c_r)(a, c) = \begin{cases} R & \text{if } c = c_a \\ \text{emp} & \text{if } c = c_r \\ \bot & \text{otherwise} \end{cases}$$

In the proof, c_a will be instantiated with the acquire channel and c_r with the release channel. Note that the logical argument a is simply ignored.

The lock predicate then asserts that there exists some join instance and that fields acq and rel refer to channels with the above channel pre- and postcondition.

$$\begin{aligned} \mathsf{lock}(x) = \exists a, r, j : \text{Val}.\ & a \neq r \wedge x.\mathsf{acq} \mapsto a * x.\mathsf{rel} \mapsto r \\ & * \ \mathsf{ch}(a, j) * \mathsf{ch}(r, j) * \mathsf{join}_{\mathsf{call}}(P(a, r), Q(a, r), j) \end{aligned}$$

We explicitly require that c_a and c_r are distinct to ensure that the above definition of P and Q by case analysis on the second argument is well-defined. The lock predicate only asserts partial ownership of fields acq and rel, to allow the lock predicate to be freely duplicated.

Below is a full proof outline for the lock constructor.

```
public Lock() {
  Join j; Pattern p;
{this.acq ↦ null * this.rel ↦ null * R}
  j = new Join();
{this.acq ↦ null * this.rel ↦ null * R * join_ch(∅, ∅, j)}
  acq = new SyncChannel(j);
  rel = new AsyncChannel(j);
{R * this.acq ↦ a * this.rel ↦ r * join_ch({r}, {a}, j) * a ≠ r * ch(a, j) * ch(r, j)}
{R * this.acq ↦ a * this.rel ↦ r * join_pat(P(a, r), Q(a, r), j) * a ≠ r * ch(a, j) * ch(r, j)}
  p = j.When(acq).And(rel);
{R * this.acq ↦ a * this.rel ↦ r * a ≠ r * join_pat(P(a, r), Q(a, r), j)
    * ch(a, j) * ch(r, j) * pat(p, j, {a, r})}
  p.Do();
{R * this.acq ↦ a * this.rel ↦ r * join_pat(P(a, r), Q(a, r), j) * a ≠ r * ch(a, j) * ch(r, j)}
{R * this.acq ↦ a * this.rel ↦ r * join_call(P(a, r), Q(a, r), j) * a ≠ r * ch(a, j) * ch(r, j)}
  rel.Call();
{this.acq ↦ a * this.rel ↦ r * join_call(P(a, r), Q(a, r), j) * a ≠ r * ch(a, j) * ch(r, j)}
{lock(this)}
}
```

The call to Do further requires that we prove:

$$\forall Y \in \mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y) = \{a, r\} \Rightarrow \circledast_{y \in Y} P(a, r)(y) \sqsubseteq \circledast_{y \in Y} Q(a, r)(y)$$

which follows easily from the commutativity of $*$.

The full proof outline for $\mathsf{Acquire}$ is given below. The proof for $\mathsf{Release}$ is similar.

```
public void Acquire() {
  SyncChannel c;
```

$\{\mathsf{lock(this)}\}$
$\{\mathsf{this.acq} \mapsto a * \mathsf{this.rel} \mapsto r * \mathsf{join}_{\mathsf{call}}(P(a,r), Q(a,r), \mathsf{j}) * a \neq r * \mathsf{ch}(a, \mathsf{j}) * \mathsf{ch}(r, \mathsf{j})\}$
  $\mathsf{c} = \mathbf{this}.\mathsf{acq};$
$\{\mathsf{this.acq} \mapsto \mathsf{c} * \mathsf{this.rel} \mapsto r * \mathsf{join}_{\mathsf{call}}(P(\mathsf{c},r), Q(\mathsf{c},r), \mathsf{j}) * \mathsf{c} \neq r * \mathsf{ch}(\mathsf{c}, \mathsf{j}) * \mathsf{ch}(r, \mathsf{j})\}$
  $\mathsf{c.Call}();$
$\{\mathsf{this.acq} \mapsto \mathsf{c} * \mathsf{this.rel} \mapsto r * \mathsf{join}_{\mathsf{call}}(P(\mathsf{c},r), Q(\mathsf{c},r), \mathsf{j}) * \mathsf{c} \neq r$
    $* \mathsf{ch}(\mathsf{c}, \mathsf{j}) * \mathsf{ch}(r, \mathsf{j}) * Q(\mathsf{c},r)(0, \mathsf{c})\}$
$\{\mathsf{lock(this)} * R\}$
}

When we call the acq channel we have to pick a logical argument a. Since the channel pre- and postcondition ignores the a, we can pick anything. In the above proof we arbitrarily picked 0, hence the $Q(\mathsf{c}, r)(0, \mathsf{c})$ in the postcondition.

5.2 n-barrier

In this section we formalise a proof of the n-barrier from Section 2.4. This example illustrates how logical arguments combined with phantom state allows us to logically distinguish messages on a single channel. The example also illustrates the use of a non-trivial view-shift to update a phantom field upon firing of a chord.

Desired specification. In Section 2.4 we gave an informal specification of an n-barrier, under the assumption that clients transferred the same resources to the barrier at every round of synchronisation, and that the barrier redistributed these resources in the same way at every round of synchronisation. As these assumptions are unrealistic, we start by generalising the specification.

The simplified n-barrier specification was expressed in terms of two assertions B_i^{in} and B_i^{out} that described the resources client i transferred to and from the barrier at every round of synchronisation. Here, instead, we take B^{in} and B^{out} to be predicates indexed by a client identifier i and the current round of synchronisation m. The general n-barrier specification is given in Figure 6.

$$
\begin{array}{l}
\forall n \in \mathbb{N}.\ \forall B^{\mathsf{in}}, B^{\mathsf{out}} : \{1, ..., n\} \times \mathbb{N} \to \mathrm{LProp}. \\
\quad (\forall m \in \mathbb{N}.\ \circledast_{i \in \{1,...,n\}} B_i^{\mathsf{in}}(m) \sqsubseteq \circledast_{i \in \{1,...,n\}} B_i^{\mathsf{out}}(m))) \ \Rightarrow \\
\qquad \exists \mathsf{barrier} : \mathrm{Val} \to \mathrm{SProp}.\ \exists \mathsf{client} : \mathrm{Val} \times \{1, ..., n\} \times \mathbb{N} \to \mathrm{SProp}. \\
\qquad\quad \{\mathsf{n} = n\}\, \mathbf{new}\ \mathsf{Barrier(n)}\, \{\mathsf{ret}.\ \mathsf{barrier(ret)} * \circledast_{i \in \{1,...,n\}} \mathsf{client(ret}, i, 0)\} \\
\qquad\quad \wedge\ \forall i \in \{1, ..., n\}.\ \forall m \in \mathbb{N}. \\
\qquad\qquad \{\mathsf{barrier(b)} * \mathsf{client(b}, i, m) * B_i^{\mathsf{in}}(m)\} \\
\qquad\qquad\quad \mathsf{b.Arrive()} \\
\qquad\qquad \{\mathsf{barrier(b)} * \mathsf{client(b}, i, m+1) * B_i^{\mathsf{out}}(m)\} \\
\qquad\quad \wedge\ \forall x : \mathrm{Val}.\ \mathsf{barrier}(x) \Leftrightarrow \mathsf{barrier}(x) * \mathsf{barrier}(x)
\end{array}
$$

Fig. 6. General n-barrier specification. This specification requires that the number of clients, n, is known statically. This simplifies the exposition. We can also specify and verify a specification without this assumption.

Here barrier is the barrier representation predicate, which can be freely duplicated. The client predicate plays two roles: namely, (1) to witness the identity of each barrier client (like the id predicate from Section 2.4), and (2) to ensure that every client of the barrier agrees on the round of synchronisation, m, whenever they arrive at the barrier. These two properties are necessary to ensure that we can redistribute the combined resources when every client has arrived at the barrier. When one creates a new n-barrier, one thus receives n client predicates – one for each client – each with 0 as the current round of synchronisation. The current round of synchronisation is incremented by one at each arrival at the barrier.

Predicate definitions. We start by giving concrete definitions for the abstract barrier and client predicate. Hence, assume $n \in \mathbb{N}$ clients and abstract predicates $B^{\mathsf{in}}, B^{\mathsf{out}} : \{1, ..., n\} \times \mathbb{N} \to \text{LProp}$ satisfying,

$$\forall m \in \mathbb{N}.\ \circledast_{i \in \{1,\ldots,n\}} B_i^{\mathsf{in}}(m) \sqsubseteq \circledast_{i \in \{1,\ldots,n\}} B_i^{\mathsf{out}}(m) \tag{2}$$

Since the n-barrier only has a single channel, we need to pick a single channel pre- and postcondition that works for every client, for every round of synchronisation. We thus take the logical argument for the arrival channel to be a pair consisting of a client identifier i and the current synchronisation round m. From the specification above, when client i arrives for synchronisation round m it transfers $B_i^{\mathsf{in}}(m)$ to the barrier and expects to receive back $B_i^{\mathsf{out}}(m)$. In addition, the client gives up its client predicate and gets back a new one, with the same logical client identifier i and an incremented synchronisation round, $m + 1$. For any barrier b and channel c_1 we thus define the channel pre- and postcondition $P(b, c_1), Q(b, c_1) : \mathcal{E} \to \text{LProp}$ as follows:

$$P(b, c_1)((i, m), c) = \begin{cases} \mathsf{client}(b, i, m) * B_i^{\mathsf{in}}(m) & \text{if } c = c_1 \\ \bot & \text{otherwise} \end{cases}$$

$$Q(b, c_1)((i, m), c) = \begin{cases} \mathsf{client}(b, i, m+1) * B_i^{\mathsf{out}}(m) & \text{if } c = c_1 \\ \bot & \text{otherwise} \end{cases}$$

Here the (i, m) is the logical argument consisting of the logical client identifier i and synchronisation round m. In the proof, c_1 will be instantiated with the arrival channel.

Above, we defined the channel pre- and postcondition in terms of an abstract client predicate, which we have not defined yet. We thus need to define client. This is the main technical challenge of the proof. So, to motivate its definition, we start by considering what properties the client predicate should satisfy. Recall that we use the client predicate to (1) witness the identity of clients, and to (2) ensure that clients agree on the current round of synchronization when they arrive at the barrier.

To witness the identity of clients, disjoint client predicates must refer to distinct clients, as expressed by property (3) below. To ensure that clients agree on the current round of synchronisation, the client predicate should also satisfy (4). Lastly, to update the current round of synchronisation when every client

has arrived at the barrier, the client predicate should satisfy (5).

$$\forall b, i, j, m.\ \mathsf{client}(b, i, m) * \mathsf{client}(b, j, m) \Rightarrow i \neq j \tag{3}$$

$$\forall b, i, j, m, k.\ \mathsf{client}(b, i, m) * \mathsf{client}(b, j, k) \Rightarrow m = k \tag{4}$$

$$\forall b, m.\ \circledast_{i \in \{1,\ldots,n\}} \mathsf{client}(b, i, m) \sqsubseteq \circledast_{i \in \{1,\ldots,n\}} \mathsf{client}(b, i, m+1) \tag{5}$$

Note that (5) is consistent with (4), since we update all n client predicates simultaneously.

We can satisfy (4) and (5) by introducing a phantom field to keep track of the current round of synchronisation. By giving each client $1/n$-th permission of this phantom field, we ensure that every client agrees on the current round of synchronisation, (4). Furthermore, given all n client predicates, these fractions combine to the full permission, allowing the phantom field to be updated arbitrarily, and thus in particular, to be incremented; thus satisfying (5). We can satisfy (3) by associating each client identifier i with a non-duplicable resource $\bullet_i$ in the logic, and requiring ownership of $\bullet_i$ in the client predicate. We thus define client as follows, $\mathsf{client}(b, i, m) = b_{\mathsf{round}} \overset{1/n}{\longmapsto} m * \bullet_i^b$, where $\bullet_i^b$ is defined as follows: $\bullet_i^b = \exists v : \mathrm{Val}.\ b_i \mapsto v$.

The barrier predicate is now trivial to define:

$$\mathsf{barrier}(b) = \exists j, c : \mathrm{Val}.\ b.\mathsf{arrive} \rightharpoonup c * \mathsf{join}_{\mathsf{call}}(P(b, c), Q(b, c), j) * \mathsf{ch}(c, j)$$

It simply asserts that arrive refers to a channel on a join instance with the channel pre- and postcondition we defined above.

Proof. Now that we have defined a client predicate satisfying (3), (4), and (5), we can proceed with the verification of the n-barrier. The main proof obligation is proving that the barrier chord satisfies the postconditions of the channels it matches. Since the barrier chord matches n arrival messages, by rule Do1 we thus have to prove that:

$$\forall Y \in \mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y) = \{c_1^n\} \Rightarrow \circledast_{y \in Y} P(b, c_1)(y) \sqsubseteq \circledast_{y \in Y} Q(b, c_1)(y)$$

To simplify the exposition, we consider the case for $n = 2$. The proof for $n > 2$ follows the same structure. For $n = 2$ the above proof obligation reduces to:

$$\begin{aligned} &\forall i_1, i_2, m_1, m_2. \\ &\quad \mathsf{client}(b, i_1, m_1) * B_{i_1}^{\mathsf{in}}(m_1) * \mathsf{client}(b, i_2, m_2) * B_{i_2}^{\mathsf{in}}(m_2) \ \sqsubseteq \\ &\qquad \mathsf{client}(b, i_1, m_1 + 1) * B_{i_1}^{\mathsf{out}}(m_1) * \mathsf{client}(b, i_2, m_2 + 1) * B_{i_2}^{\mathsf{out}}(m_2) \end{aligned} \tag{6}$$

At this point we cannot directly apply the user-supplied redistribution property, (2), as it requires that $m_1 = m_2$ and $i_1 \neq i_2$. First, we need to use properties (3) and (4) to constrain what logical arguments clients could have choose when they send their arrival messages. By property (4) it follows that $m_1 = m_2$. Furthermore, from property (3) it follows that i_1 and i_2 are distinct. Since $i_1, i_2 \in \{1, 2\}$, (6) thus reduces to:

$$\begin{aligned} &\forall m.\ \mathsf{client}(b, 1, m) * B_1^{\mathsf{in}}(m) * \mathsf{client}(b, 2, m) * B_2^{\mathsf{in}}(m) \ \sqsubseteq \\ &\qquad \mathsf{client}(b, 1, m + 1) * B_1^{\mathsf{out}}(m) * \mathsf{client}(b, 2, m + 1) * B_2^{\mathsf{out}}(m) \end{aligned} \tag{7}$$

Using the redistribution property, (2), and (5) it follows that,

$$\forall m.\ B_1^{\mathsf{in}}(m) * B_2^{\mathsf{in}}(m) \sqsubseteq B_1^{\mathsf{out}}(m) * B_2^{\mathsf{out}}(m)$$
$$\forall m.\ \mathsf{client}(b,1,m) * \mathsf{client}(b,2,m) \sqsubseteq \mathsf{client}(b,1,m+1) * \mathsf{client}(b,2,m+1)$$

Combining these two we thus get (7). We have thus proven (6). Note that here we implicitly used the ability to perform a view shift when a chord fires, to increment the value of the phantom field round.

The verification of the constructor and Arrive method is now straightforward.

In summary, using logical arguments and phantom state we can thus show that the generalised n-barrier from Section 2.4 satisfies the generalised barrier specification. While the proof is more technically challenging than any of the previous examples, it is still a high-level proof about *barrier* concepts. Informally, we proved that clients agree on the current synchronisation round and that clients identify themselves correctly; both natural proof obligations for a barrier.

6 Discussion

We first relate our specification of joins and the clients thereof to earlier work and then evaluate what we have learned about HOCAP from this case study.

In terms of reasoning about external sharing, O'Hearn's original concurrent separation logic supports reasoning about shared variable concurrency using critical regions [17]. This was subsequently extended to a language with locking primitives by Hobor et al. [11] and Gotsman et al. [10], and to a language with barrier primitives by Hobor et al. [12]. In all four cases, the underlying synchronisation primitives were taken as language primitives and their soundness was proven meta-theoretically.

Concurrent abstract predicates by Dinsdale-Young et al. [7] extends standard separation logic with support for reasoning about shared mutable state by imposing protocols on shared resources. Dinsdale-Young et al. used this logic to verify a spin-lock implemented using compare-and-swap. The spin-lock was verified against a non-standard lock specification *without* built-in support for reasoning about external sharing. Hence, to reason about external sharing, *clients* would have to define a protocol of their own, relating ownership of the shared resources with the state of the lock. This type of reasoning is *not* modular, as it requires the specification of concurrent libraries to expose *internal* implementation details of synchronisation primitives, to allow clients to define a protocol governing the external sharing.

Jacobs and Piessens recently extended their VeriFast tool with support for fine-grained concurrency [14] and verified a lock-based barrier implementation [13] inspired by [11]. They verify the implementation against a specification without built-in support for reasoning about external sharing. Compared to our barrier specification, their specification is thus fairly low-level, requiring *clients* of the barrier to use auxiliary variables to encode who has arrived and what resources they have transferred to the barrier.

The goal of this case study was to test whether HOCAP supports modular reasoning about concurrent higher-order imperative libraries. To this end, we have proposed an abstract specification of the C♯ joins library, expressed in terms of high-level join primitives. We have demonstrated that this abstract specification suffices for formal reasoning about a series of classic synchronisation primitives, which allow for external sharing. Compared to previous work on verifying synchronisation primitives using separation logic, our specifications are stronger and our proofs are considerably simpler. Thus, from this perspective, our case study supports the thesis that HOCAP is useful for modular reasoning about concurrent higher-order imperative libraries. However, as explained in Section 3, the joins specification presented in this paper is restricted to local pre- and postconditions for channels, which means that synchronization primitives implemented using joins can only have local assertions as resource invariants. Recall, e.g., the lock specification in Section 5.1, where the resource invariant ranges over LProp, which means that clients of the lock cannot use CAP when picking a resource invariant for the lock. In the technical report [24] we have presented a stronger specification of joins, which does allow clients to use CAP for such resource invariants, but that is at the expense of complicating the specification, to avoid circular sharing patterns. Thus future work includes finding stronger models of HOCAP that support simple specifications and circular sharing patterns.

Acknowledgements

We would like to thank Mike Dodds, Bart Jacobs, Jonas Braband Jensen, Hannes Mehnert, Claudio Russo, and Aaron Turon for helpful discussions and feedback.

References

1. B. Biering, L. Birkedal, and N. Torp-Smith. BI-Hyperdoctrines, Higher-order Separation Logic, and Abstraction. *ACM TOPLAS*, 2007.
2. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, pages 259–270, 2005.
3. J. Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, pages 55–72, 2003.
4. P.-J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with "Readers" and "Writers". *Commun. ACM*, 14(10):667–668, 1971.
5. P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, and M. Wheelhouse. A simple abstraction for complex concurrent indexes. *SIGPLAN Not.*, 46(10):845–864, Oct. 2011.
6. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of POPL*, 2013.
7. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of ECOOP*, 2010.
8. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL*, 1996.

9. C. Fournet and G. Gonthier. The Join Calculus: A Language for Distributed Mobile Programming. In *Proceedings of APPSEM*, pages 268–332, 2000.
10. 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.
11. A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. In *Proceedings of ESOP*, pages 353–367, 2008.
12. A. Hobor and C. Gherghina. Barriers in concurrent separation logic. In *Proceedings of ESOP*, pages 276–296, 2011.
13. B. Jacobs. Verified general barriers implementation. http://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/barrier.c.html.
14. B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of POPL*, pages 271–282, 2011.
15. N. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2012.
16. N. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying Event-Driven Programs using Ramified Frame Properties. In *Proceedings of TLDI*, 2010.
17. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
18. S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell, 1975.
19. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, pages 247–258, 2005.
20. C. V. Russo. The Joins Concurrency Library. In *Proceedings of PADL*, pages 260–274, 2007.
21. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare Triples and Frame Rules for Higher-Order Store. *LMCS*, 7(3:21), 2011.
22. K. Svendsen, L. Birkedal, and M. Parkinson. Verifying Generics and Delegates. In *Proceedings of ECOOP*, pages 175–199, 2010.
23. K. Svendsen, L. Birkedal, and M. Parkinson. Higher-order Concurrent Abstract Predicates. Technical report, IT University of Copenhagen, 2012. Available at `www.itu.dk/people/kasv/hocap-tr.pdf`.
24. K. Svendsen, L. Birkedal, and M. Parkinson. Verification of the Joins Library in Higher-order Separation Logic. Technical report, IT University of Copenhagen, 2012. Available at `www.itu.dk/people/kasv/joins-tr.pdf`.
25. K. Svendsen, L. Birkedal, and M. Parkinson. Modular Reasoning about Separation for Concurrent Data Structures. In *Proceedings of ESOP*, 2013.