Verification of the Joins Library in Higher-Order Separation Logic

Kasper Svendsen
IT University of Copenhagen
kasv@itu.dk

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

Matthew Parkinson
Microsoft Research, Cambridge
mattpark@microsoft.com

September 8, 2012

Contents

1 Introduction 2

2 Implementation 2

3 Specification 5

4 Proof 6
- 4.1 The Message class . . . 6
- 4.2 The MessagePool class . . . 9
- 4.3 The Pattern class . . . 11
- 4.4 The AsyncChannel, SyncChannel, Chord and Join classes . . . 16
- 4.5 The Join class . . . 22
- 4.6 Auxiliary classes . . . 24

References 27

1 Introduction

In this technical report we define a formal specification for the joins library and verify a naive lock-based implementation against this specification. We assume the reader is familiar with the joins library and has read the accompanying article [1]. We take as our program logic the higher-order version of concurrent abstract predicates described in [?] and formally defined in [3]. We assume the reader is familiar with this logic.

The joins library implementation makes for an interesting verification challenge, as it combines state, sharing, concurrency and recursion through the store, in a realistic and reasonably small library. We use our higher-order variant of Concurrent Abstract Predicates to reason about sharing and concurrency and we use representation predicates defined by guarded recursion to reason about guarded recursion.

2 Implementation

We start by sketching our naive lock-based implementation. Figure 1 defines the static structure of the implementation. The implementation consists of seven main classes:

- Join: Join objects represent join instances. Each join object consists a list of registered chords and a list of registered channels. Conceptually, we think of a join instance as having a single common message pool. However, in the implementation, each channel maintains its own pool of messages. Each join object contains a lock to synchronize access to all its chords and channels. Once a join object has been initialized (that is, once all chords and channels have been registered) all its state except the channel message pools is fixed. Assuming non self-modifying join instances, the lock is thus only necessary to synchronize access to message pools.

- Pattern: Pattern objects represent conditions on the message pool. The type of conditions supported by this version of the joins library can be represented as a list of channels – representing the condition that matches a distinct message for each channel in the list.

- Chord: Chord objects represent chords, which are simply implemented as a pair consisting of a Pattern and an Action.

- MessagePool, Message: The abstract MessagePool class implements a message pool. Each message pool consists of a list of pending messages. Each message consists of an integer status field indicating whether the message has been received. This status field is thus 0 (not received) for all pending messages. When a message is matched by a pattern it is removed from its message pool and once the chord continuation has terminated its status field set to 1 (received).

- AsyncChannel, SyncChannel: AsyncChannel and SyncChannel objects represent channels. They both inherit from the abstract MessagePool class.

The meat of the implementation is the Call methods for sending a message on a channel. Both Call methods start by acquiring the join lock. Once the lock has been acquired, in each case they add a new pending message to the message pool of the given channel (using the AddMessage method). In the asynchronous case, this is all that happens; Call simply releases the lock and returns. However, in the synchronous case, the method enters a busy-loop, constantly checking whether its message has been received and firing any chords ready to fire (using the CheckChords method). To support reentrant callbacks, the CheckChords method releases and reacquires the join lock before executing chord continuations.

The CheckChords method iterates through the list of chords, checking if each chord is ready to fire (using the Matches method). Whenever it encounters a chord ready to fire, it (1) removes the messages that matched the chord pattern from their message pools, (2) releases the join lock (to allow continuations to send messages on the same join instance without dead-locking), (3) calls the continuation, (4) reacquires the join lock and (5) updates the status of all the matched messages to 1 (received).

The Matches method optimistically tries to match and remove a pending message from each channel of the given pattern (using the Pop method). If it succeeds, it simply returns a list of the messages it matched. If it fails, it adds back the messages already removed to their respective message pools (using the Push method), and returns `null`.

The implementation makes use of three auxiliary classes: Lock, List and Pair. The Lock class implements a spin-lock. The List class implements a singly-linked list. It supports methods Push, Pop and Count, to modify and query lists. In addition, it supports higher-order methods ForEach and Map, to iterate over lists. Lastly, the Pair class implements a pair.

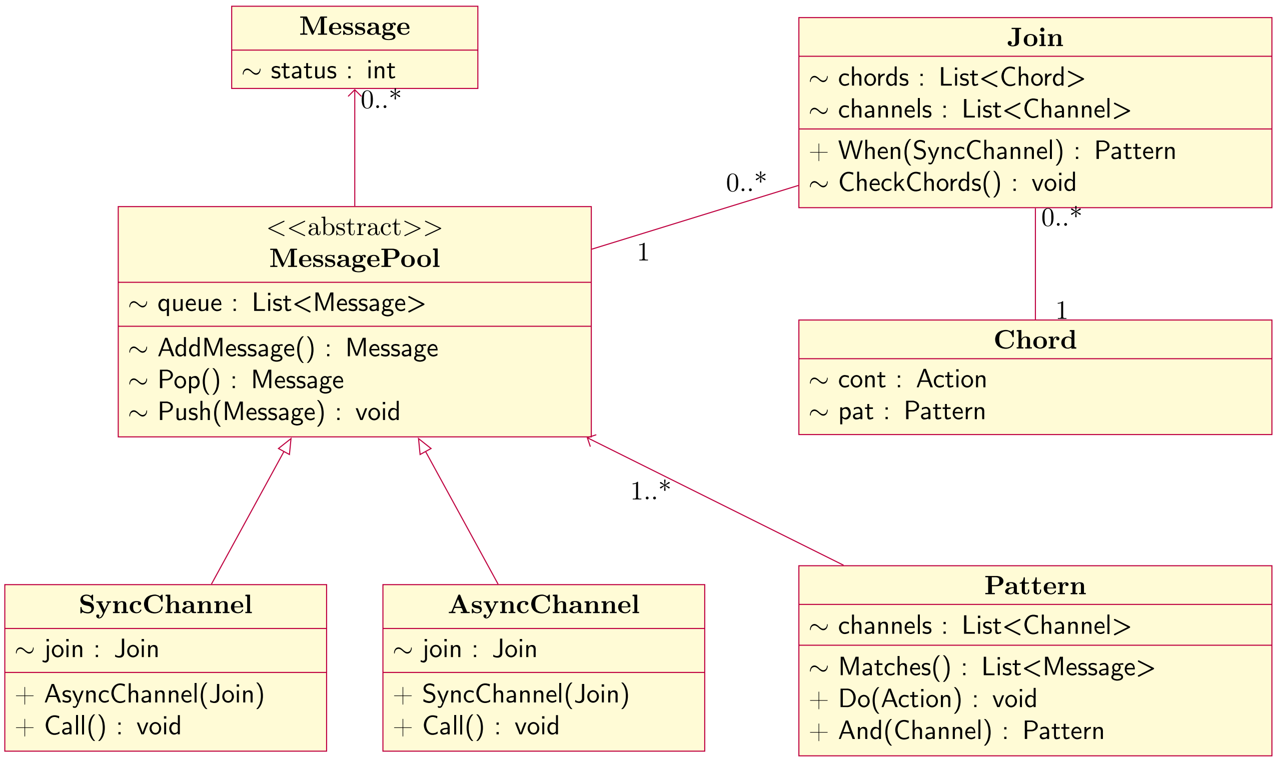

Figure 1: Class diagram

3 Specification

The specification of the joins library is given below. This is the same specification as the specification that appears in the accompanying article [1], only expressed as a proposition in the specification logic, to make all quantifications explicit. The reader is referred to the accompanying article for an explanation of the specification.

$$
\begin{array}{l}
\exists \mathsf{join}_{\mathsf{init\text{-}ch}} : \mathrm{RType} \times \mathcal{P}_m(\mathrm{Val}) \times \mathcal{P}_m(\mathrm{Val}) \times \mathrm{Val} \to \mathrm{Prop}.\\
\exists \mathsf{join}_{\mathsf{init\text{-}path}}, \mathsf{join}_{\mathsf{call}} : \mathrm{RType} \times (\mathrm{Val} \times \mathrm{Val} \to \mathrm{Prop}) \times (\mathrm{Val} \times \mathrm{Val} \to \mathrm{Prop}) \times \mathrm{Val} \to \mathrm{Prop}.\\
\exists \mathsf{chan} : \mathrm{Val} \times \mathrm{Val} \to \mathrm{Prop}.\ \exists \mathsf{pattern} : \mathrm{Val} \times \mathrm{Val} \times \mathcal{P}_m(\mathrm{Val}) \to \mathrm{Prop}.\\
\quad \forall \mathsf{t} : \mathrm{RType}.\ \forall \mathsf{P}, \mathsf{Q} : \mathrm{Val} \times \mathrm{Val} \to \mathrm{Prop}.\ \forall \mathsf{C}_A, \mathsf{C}_S, \mathsf{X} : \mathcal{P}_m(\mathrm{Val}).\ \forall \mathsf{a}, \mathsf{c}, \mathsf{j} : \mathrm{Val}.\\
\quad\quad \mathsf{usip}(\mathsf{P},\mathsf{Q}) \wedge \mathsf{indep}_\mathsf{t}(\mathsf{P}) \wedge \mathsf{indep}_\mathsf{t}(\mathsf{Q}) \wedge \mathsf{stable}(\mathsf{P}) \wedge \mathsf{stable}(\mathsf{Q}) \Rightarrow\\
\quad\quad\quad \mathsf{Join} : (\text{-}).\ \{\mathrm{emp}\}\ \{\mathsf{ret}.\ \mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \emptyset, \emptyset, \mathsf{ret})\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{SyncChannel} : (\mathsf{j}).\ \{\mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \mathsf{C}_A, \mathsf{C}_S,\ \mathsf{j})\}\\
\qquad\qquad \{\mathsf{ret}.\ \mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \mathsf{C}_A, \mathsf{C}_S \cup \{\mathsf{ret}\},\ \mathsf{j}) * \mathsf{chan}(\mathsf{ret},\ \mathsf{j}) * \mathsf{ret} \notin \mathsf{C}_A \cup \mathsf{C}_S\}\\
\mathsf{AsyncChannel} : (\mathsf{j}).\ \{\mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \mathsf{C}_A, \mathsf{C}_S,\ \mathsf{j})\}\\
\qquad\qquad \{\mathsf{ret}.\ \mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \mathsf{C}_A \cup \{\mathsf{ret}\}, \mathsf{C}_S,\ \mathsf{j}) * \mathsf{chan}(\mathsf{ret},\ \mathsf{j}) * \mathsf{ret} \notin \mathsf{C}_A \cup \mathsf{C}_S\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Join.When} : (\mathsf{c}).\ \big\{\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathtt{this}) * \mathsf{chan}(\mathsf{c}, \mathsf{j})\big\}\\
\qquad\qquad \big\{\mathsf{ret}.\ \mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathtt{this}) * \mathsf{pattern}(\mathsf{ret}, \mathsf{j}, \{\ \mathsf{c}\})\big\}\\
\mathsf{Pattern.And} : (\mathsf{c}).\ \big\{\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{pattern}(\mathtt{this}, \mathsf{j}, \mathsf{X}) * \mathsf{chan}(\mathsf{c}, \mathsf{j})\big\}\\
\qquad\qquad \big\{\mathsf{ret}.\ \mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{pattern}(\mathsf{ret}, \mathsf{j}, \mathsf{X} \cup \{\mathsf{c}\})\big\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Pattern.Do} : (\mathsf{act}).\ \left\{\begin{array}{l}
\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{pattern}(\mathtt{this}, \mathsf{j}, \mathsf{X}) *\\
\quad \forall \mathsf{Y} : \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \pi_{\mathsf{chan}}(\mathsf{Y}) = \mathsf{X} \Rightarrow\\
\quad\quad act \mapsto \{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{P}(\mathsf{y})\}\\
\quad\quad\quad \{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{Q}(\mathsf{y})\}
\end{array}\right\}\\
\qquad\qquad \big\{\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j})\big\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{AsyncChannel.Call,SyncChannel.Call} : (\text{-}).\ \{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{chan}(\mathtt{this}, \mathsf{j}) * \mathsf{P}(\mathsf{a}, \mathtt{this})\}\\
\qquad\qquad \{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{chan}(\mathtt{this}, \mathsf{j}) * \mathsf{Q}(\mathsf{a}, \mathtt{this})\}
\end{array}
$$

$$
\begin{array}{l}
\boldsymbol{valid}\ (\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) \Leftrightarrow \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}))\\
\boldsymbol{valid}\ (\mathsf{chan}(\mathsf{c}, \mathsf{j}) \Leftrightarrow \mathsf{chan}(\mathsf{c}, \mathsf{j}) * \mathsf{chan}(\mathsf{c}, \mathsf{j}))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{stable}(\mathsf{join}_{\mathsf{init\text{-}ch}}) \wedge \mathsf{stable}(\mathsf{join}_{\mathsf{init\text{-}pat}}) \wedge \mathsf{stable}(\mathsf{join}_{\mathsf{call}}) \wedge \mathsf{stable}(\mathsf{chan}) \wedge \mathsf{stable}(\mathsf{pattern})\\
(\forall \mathsf{a} : \mathrm{Val}.\ \forall \mathsf{c} \in \mathsf{C}_A.\ \mathsf{Q}(\mathsf{a}, \mathsf{c}) = \mathrm{emp}) \Rightarrow \mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \mathsf{C}_A, \mathsf{C}_S, \mathsf{j}) \sqsubseteq \mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j})\\
\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) \sqsubseteq \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j})
\end{array}
$$

4 Proof

In the previous sections we introduced our naive lock-based implementation of the joins library and formalized our high-level joins specification as a proposition in our specification logic. In this section we prove that our naive lock-based implementation satisfies this specification. As the full implementation consists of approximately 150 lines of mini C# code, to make the proof manageable, the proof is written in the form of proof outlines (that is, code with inline assertions about the current state at that point of execution).

We thus have to instantiate each of the existentially quantified predicates from the specification and prove that every method satisfies its specification with these instantiations. In addition, the implementation contains several internal methods that also needs to be specified and verified. For the Message, MessagePool, and Pattern classes it is possible to define their representation predicates independently of the rest of the joins library. However, for the remaining classes this is not the case due to the possibility of reentrant continuations. We will thus start by verifying the Message, MessagePool and Pattern classes.

The proofs of the List class and Pair class are completely standard, and have thus been omitted. Their specifications are given in Section 4.6. The Lock specification and proof follows by a slightly generalization of the spin-lock example in [3]. The Lock specification is also given in Section 4.6.

4.1 The Message class

Message objects are very simple; they consist of a single integer field, status, which indicates whether the message has been received (status $= 0$) or not (status $= 1$). However, conceptually, message objects are significantly more complicated: When a client sends a synchronous message it (1) creates a new message, (2) transfers ownership of the channel pre-condition to the message, (3) enters a busy-loop waiting for someone to receive the message, before (4) transferring back ownership of the channel post-condition to the sender. During the busy-loop, (3), other threads are allowed to match the message and receive the message, transferring ownership of the channel pre-condition from the message to the continuation and ownership of the channel post-condition from the continuation back to the message.

Conceptually, messages can thus be in one of four states, `pending`, `removed`, `received`, and `released`. Each message starts out in the `pending` state. When a message is matched by a pattern it transitions to the `removed` state. When the continuation of the chord that matched the message terminates, it transitions to the `received` state. Lastly, when the client that inserted the message in the first place notices that the message has been received, and exits its busy-loop, the message transitions to the `released` state. In the pending state the message owns its channel pre-condition and the `status` field is 0. In the `removed` state the `status` field is 0, but it no longer owns its channel precondition. In the `received` state the `status` field is 1 and it owns its channel postcondition. Lastly, in the `released` state the `status` field is 1, but it no longer owns its channel post-condition. Crucially, while every thread is allowed to transition messages from the `pending` state to the `removed` and `received` state, only the client that inserted the message is allowed to transition the message to the `released` state.

We can thus specify the $\mathsf{Message}$ constructor in terms of three message representation predicates, $\mathsf{msgprotocol}(-)$, $\mathsf{pending}(-)$ and $\mathsf{msg}(-)$, as follows:

$$\begin{aligned}&\mathsf{new\ Message} : (\text{-}).\, \{\mathsf{P}(\mathsf{a},\mathsf{c}) * \mathsf{msgprotocol}(\mathsf{t},\mathsf{P},\mathsf{Q})\}\\ &\qquad\qquad\{\mathsf{ret}.\ \mathsf{pending}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{ret},\mathsf{c}) * \mathsf{msg}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{ret},\mathsf{c})\}\end{aligned}$$

Here $\mathsf{msgprotocol}(\mathsf{t},\mathsf{P},\mathsf{Q})$ asserts that the protocol on region type t is a message protocol. The $\mathsf{pending}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{ret},\mathsf{c})$ predicate asserts that ret refers to a message on channel c that is currently in the `pending` state *and* asserts permission to transition the message to the `removed` and `received` state *and* asserts ownership of the channel pre-condition. Likewise, $\mathsf{msg}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{ret},\mathsf{c})$ asserts that ret refers to a message on channel c that is currently in the `pending`, `removed` or `received` state *and* asserts permission to transition the message from the `received` state to the `released` state.

These predicates should thus satisfy,

$$\begin{aligned}\mathsf{pending}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{x},\mathsf{c}) &\sqsubseteq \exists \mathsf{a} : \mathrm{Val}.\ \mathsf{P}(\mathsf{a},\mathsf{c}) * \mathsf{removed}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{x},\mathsf{c})\\ \mathsf{Q}(\mathsf{a},\mathsf{c}) * \mathsf{removed}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{x},\mathsf{c}) &\sqsubseteq \mathsf{received}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{x},\mathsf{c})\\ \mathsf{msg}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{x},\mathsf{c}) * \mathsf{received}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{x},\mathsf{c}) &\sqsubseteq \mathsf{Q}(\mathsf{a},\mathsf{c})\end{aligned}$$

Here $\mathsf{removed}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{x},\mathsf{c})$ asserts that x refers to a message on channel c in the `removed` state and asserts permission to transition it to the `received` state. Likewise, $\mathsf{received}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{x},\mathsf{c})$ simply asserts that x refers to a message on channel c in the `received` state.

When adding a new message to the message pool, the client that created the message thus keeps the $\mathsf{msg}(-)$ assertion, and transfer ownership of the $\mathsf{pending}(-)$ assertion to the message pool.

Predicate definitions

We can express this sharing and ownership pattern in CAP, using a shared region to transfer the channel post-condition from the message recipient to the sender of the message. We need two actions, INSERT, for transferring ownership of the channel post-condition to the shared region, and REMOVE, for transferring ownership of the channel post-condition out of the shared region. Given channel pre-conditions P and channel post-conditions Q the protocol on message x on channel c with logical argument a is:

$$\mathsf{I}(\mathsf{Q})(\mathsf{x},\mathsf{c},\mathsf{a}) \stackrel{\mathrm{def}}{=} \left(\begin{array}{r}\textsc{Insert} : \mathsf{x.status} \mapsto 0 \rightsquigarrow \mathsf{x.status} \mapsto 1 * \mathsf{Q}(\mathsf{a},\mathsf{c})\\ \textsc{Remove} : \mathsf{x.status} \mapsto 1 * \mathsf{Q}(\mathsf{a},\mathsf{c}) \rightsquigarrow \mathsf{x.status} \mapsto 1\\ \tau_1 : \mathsf{x.status} \mapsto 0 \rightsquigarrow \mathsf{x.status} \mapsto 0\\ \tau_2 : \mathsf{x.status} \mapsto 1 * \mathsf{Q}(\mathsf{a},\mathsf{c}) \rightsquigarrow \mathsf{x.status} \mapsto 1 * \mathsf{Q}(\mathsf{a},\mathsf{c})\end{array}\right)$$

Here $\mathsf{I}(\mathsf{Q})$ is thus a parametric protocol with parameters $(\mathsf{x}, \mathsf{c}, \mathsf{a})$. We can now define the message representation predicates as follows:

$$\begin{aligned}
\mathsf{msgprotocol}(\mathsf{t}, \mathsf{P}, \mathsf{Q}) &\stackrel{\text{def}}{=} \mathit{protocol}(\mathsf{t}, \mathsf{I}(\mathsf{Q})) \\
\mathsf{pending}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x}, \mathsf{c}) &\stackrel{\text{def}}{=} \exists \mathsf{r} : \text{RId}.\ \exists \pi \in \text{Perm}.\ \exists \mathsf{a} : \text{Val}. \\
&\qquad \mathsf{x}_{larg} \mapsto \mathsf{a} * \mathsf{P}(\mathsf{a}, \mathsf{c}) * [\text{INSERT}]^{\mathsf{r}}_{1} * [\tau_1]^{\mathsf{r}}_{\pi} * [\tau_2]^{\mathsf{r}}_{\pi} \\
&\qquad * \boxed{\mathsf{x.status} \mapsto 0}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a},\mathsf{c})}_{\mathsf{I}(\mathsf{Q})} \\
\mathsf{removed}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{x}, \mathsf{c}) &\stackrel{\text{def}}{=} \exists \mathsf{r} : \text{RId}.\ \exists \pi \in \text{Perm}. \\
&\qquad \mathsf{x}_{larg} \mapsto \mathsf{a} * [\text{INSERT}]^{\mathsf{r}}_{1} * [\tau_1]^{\mathsf{r}}_{\pi} * [\tau_2]^{\mathsf{r}}_{\pi} \\
&\qquad * \boxed{\mathsf{x.status} \mapsto 0}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a},\mathsf{c})}_{\mathsf{I}(\mathsf{Q})} \\
\mathsf{received}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x}, \mathsf{c}) &\stackrel{\text{def}}{=} \exists \mathsf{r} : \text{RId}.\ \exists \pi \in \text{Perm}.\ \exists \mathsf{a} : \text{Val}. \\
&\qquad \mathsf{x}_{larg} \mapsto \mathsf{a} * [\text{INSERT}]^{\mathsf{r}}_{1} * [\tau_1]^{\mathsf{r}}_{\pi} * [\tau_2]^{\mathsf{r}}_{\pi} \\
&\qquad * \boxed{\mathsf{x.status} \mapsto 1 * (\mathsf{Q}(\mathsf{a}, \mathsf{c}) \vee \text{emp})}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a},\mathsf{c})}_{\mathsf{I}(\mathsf{Q})} \\
\mathsf{msg}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{x}, \mathsf{c}) &\stackrel{\text{def}}{=} \exists \mathsf{r} : \text{RId}.\ \exists \pi \in \text{Perm}. \\
&\qquad \mathsf{x}_{larg} \mapsto \mathsf{a} * [\text{REMOVE}]^{\mathsf{r}}_{1} * [\tau_1]^{\mathsf{r}}_{\pi} * [\tau_2]^{\mathsf{r}}_{\pi} \\
&\qquad * \boxed{\mathsf{x.status} \mapsto 0 \vee (\mathsf{x.status} \mapsto 1 * \mathsf{Q}(\mathsf{a}, \mathsf{c}))}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a},\mathsf{c})}_{\mathsf{I}(\mathsf{Q})}
\end{aligned}$$

Stability

By assumption, all channel pre- and post-conditions are uniformly expressible using state-independent protocols:

$$\begin{aligned}
&\exists \mathsf{R} : \text{Prop}.\ \exists \mathsf{S}_P, \mathsf{S}_Q : \text{Val} \times \text{Val} \to \text{Prop}.\ \mathsf{nostate}(R)\ \wedge \\
&\quad (\forall \mathsf{x} : \text{Val} \times \text{Val}.\ (\mathsf{P}(\mathsf{x}) \Leftrightarrow \mathsf{S}_P(\mathsf{x}) * \mathsf{R}) \wedge \mathsf{noprotocol}(\mathsf{S}_P(\mathsf{x})))\ \wedge \\
&\quad (\forall \mathsf{x} : \text{Val} \times \text{Val}.\ (\mathsf{Q}(\mathsf{x}) \Leftrightarrow \mathsf{S}_Q(\mathsf{x}) * \mathsf{R}) \wedge \mathsf{noprotocol}(\mathsf{S}_Q(\mathsf{x})))
\end{aligned}$$

Assuming that channel pre- and post-conditions are independent of the message region type t, by rule STABLEA we can thus prove stability of the message representation predicates by proving that the boxed assertions are closed under actions potentially owned by the environment. For instance, $\mathsf{pending}(-)$ asserts the status field is 0 which is stable as it also asserts full ownership of the only action that allows the status field to change, namely, the INSERT action. Formally, we prove that,

$$\begin{aligned}
&\forall \mathsf{x}, \mathsf{c}, \mathsf{a} : \text{Val}.\ \forall \mathsf{t} : \text{RType}.\ (\forall \mathsf{a}, \mathsf{c} : \text{Val}.\ \mathsf{indep}_{\mathsf{t}}(\mathsf{P}(\mathsf{a}, \mathsf{c})) \wedge \mathsf{indep}_{\mathsf{t}}(\mathsf{Q}(\mathsf{a}, \mathsf{c})))\ \Rightarrow \\
&\quad \mathsf{stable}(\mathsf{pending}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x}, \mathsf{c})) \wedge \mathsf{stable}(\mathsf{removed}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{x}, \mathsf{c}))\ \wedge \\
&\quad \mathsf{stable}(\mathsf{received}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x}, \mathsf{c})) \wedge \mathsf{stable}(\mathsf{msg}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{x}, \mathsf{c}))
\end{aligned}$$

Proof outline

The proof of the constructor is now fairly straightforward: we allocate a new phantom field, *larg*, set the status field to 0, and setup a new region with region type t and region arguments $(\mathsf{x}, \mathsf{a}, \mathsf{c})$.

```
public class Message {
  public int status;

  public Message() {
      {this_larg ↦ a ∗ this.status ↦ _ ∗ P(a, c) ∗ protocol(t, I(Q))}
    this.status = 0;
      {this_larg ↦ a ∗ this.status ↦ 0 ∗ P(a, c) ∗ protocol(t, I(Q))}
      {∃r ∈ RId. this_larg ↦ a ∗ [this.status ↦ 0]^{r,t,(x,a,c)}_{I(Q)}
         ∗ [INSERT]^r_1 ∗ [REMOVE]^r_1 ∗ [τ1]^r_1 ∗ [τ2]^r_1 ∗ P(a, c)}
      {pending(t, P, Q, this, c) ∗ msg(t, P, Q, a, this, c)}
  }
}
```

$$\{\mathtt{this}_{larg} \mapsto \mathsf{a} * \mathtt{this}.\mathsf{status} \mapsto \_ * \mathsf{P}(\mathsf{a},\mathsf{c}) * \mathit{protocol}(\mathsf{t}, \mathsf{I}(\mathsf{Q}))\}$$

$$\{\mathtt{this}_{larg} \mapsto \mathsf{a} * \mathtt{this}.\mathsf{status} \mapsto 0 * \mathsf{P}(\mathsf{a},\mathsf{c}) * \mathit{protocol}(\mathsf{t}, \mathsf{I}(\mathsf{Q}))\}$$

$$\{\exists \mathsf{r} \in \mathrm{RId}.\ \mathtt{this}_{larg} \mapsto \mathsf{a} * \boxed{\mathtt{this}.\mathsf{status} \mapsto 0}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a},\mathsf{c})}_{\mathsf{I}(\mathsf{Q})} * [\textsc{Insert}]^{\mathsf{r}}_1 * [\textsc{Remove}]^{\mathsf{r}}_1 * [\tau_1]^{\mathsf{r}}_1 * [\tau_2]^{\mathsf{r}}_1 * \mathsf{P}(\mathsf{a},\mathsf{c})\}$$

$$\{\mathsf{pending}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathtt{this},\mathsf{c}) * \mathsf{msg}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathtt{this},\mathsf{c})\}$$

4.2 The MessagePool class

The abstract MessagePool class implements a message pool. Each method pool maintains a list of pending messages. Message pools supports three methods:

- AddMessage: The AddMessage method creates a new message, adds it to the message pool and returns a reference to the new message.
- Pop: The Pop method optimistically tries to pop a pending message from the message pool, returning `null` if there are no pending messages.
- Push: The Push method adds a given message to the message pool.

Formally, the MessagePool class satisfies the following specification.

$$\mathsf{MessagePool.AddMessage} : (\text{-}).\ \{\mathsf{pool}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathtt{this}) * \mathsf{P}(\mathsf{a},\mathtt{this})\}$$
$$\{\mathsf{ret}.\ \mathsf{pool}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathtt{this}) * \mathsf{msg}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{ret},\mathtt{this})\}$$

$$\mathsf{MessagePool.Pop} : (\text{-}).\ \{\mathsf{pool}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathtt{this})\}$$
$$\{\mathsf{ret}.\ \mathsf{pool}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathtt{this}) * (\mathsf{ret} = \mathtt{null} \lor \mathsf{pending}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{ret},\mathtt{this})\}$$

$$\mathsf{MessagePool.Push} : (\mathsf{m}).\ \{\mathsf{pool}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathtt{this}) * \mathsf{pending}(\mathsf{t},\mathsf{P},\mathsf{Q},\ \mathsf{m},\mathtt{this})\}$$
$$\{\mathsf{pool}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathtt{this})\}$$

Here the $\mathsf{pool}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{x})$ predicate asserts ownership of the message pool x. The AddMessage method thus returns the $\mathsf{msg}(-)$ assertion, representing the message sender handle, but keeps the $\mathsf{pending}(-)$ assertion, representing the message receiver handle.

Predicate definitions

The $\mathsf{pool}(-)$ predicate simply asserts that the **queue** field refers a list and each element of that list is a message in the `pending` state.

$$\begin{aligned}\mathsf{pool}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{c}) \overset{\text{def}}{=}\ & \exists \mathsf{x} : \mathrm{Val}.\ \exists \mathsf{M} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{c.queue} \mapsto \mathsf{x} * \mathsf{lst}(\mathsf{x},\mathsf{M}) \\ & * \ \mathit{protocol}(\mathsf{t}, \mathsf{I}(\mathsf{Q})) * \circledast_{\mathsf{m}\in\mathsf{M}} \mathsf{pending}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{m},\mathsf{c})\end{aligned}$$

Proof outline

```
public abstract class MessagePool {
  public List<Message> queue = new List<Message>();

  public Message AddMessage() {
    List<Message> q; Message m;
      $\{\mathsf{pool}(t,P,Q,\mathtt{this}) * P(a,\mathtt{this})\}$
    m = new Message();
      $\{\mathsf{pool}(t,P,Q,\mathtt{this}) * \mathsf{pending}(t,P,Q,m,\mathtt{this}) * \mathsf{msg}(t,P,Q,a,m,\mathtt{this})\}$
    q = this.queue;
      $\{\mathtt{this}.\mathsf{queue} \mapsto q * \mathsf{lst}(q,M) * \circledast_{m\in M}\mathsf{pending}(t,P,Q,m,\mathtt{this})$
         $* \ \mathsf{pending}(t,P,Q,m,\mathtt{this}) * \mathsf{msg}(t,P,Q,a,m,\mathtt{this})\}$
    q.Push(m);
      $\{\mathtt{this}.\mathsf{queue} \mapsto q * \mathsf{lst}(q,\{m\}\cup M) * \circledast_{m\in M}\mathsf{pending}(t,P,Q,m,\mathtt{this})$
         $* \ \mathsf{pending}(t,P,Q,m,\mathtt{this}) * \mathsf{msg}(t,P,Q,a,m,\mathtt{this})\}$
      $\{\mathtt{this}.\mathsf{queue} \mapsto q * \mathsf{lst}(q,\{m\}\cup M) * \circledast_{m\in\{m\}\cup M}\mathsf{pending}(t,P,Q,m,\mathtt{this})$
         $* \ \mathsf{msg}(t,P,Q,a,m,\mathtt{this})\}$
      $\{\mathsf{pool}(t,P,Q,\mathtt{this}) * \mathsf{msg}(t,P,Q,a,m,\mathtt{this})\}$
    return m;
      $\{\mathsf{ret}.\ \mathsf{pool}(t,P,Q,\mathtt{this}) * \mathsf{msg}(t,P,Q,a,\mathsf{ret},\mathtt{this})\}$
  }

  public Message Pop() {
    List<Message> q; Message m;
      $\{\mathsf{pool}(t,P,Q,\mathtt{this})\}$
    q = this.queue;
      $\{\mathtt{this}.\mathsf{queue} \mapsto q * \mathsf{lst}(q,M) * \mathit{protocol}(t,\mathsf{I}(Q)) * \circledast_{m\in M}\mathsf{pending}(t,P,Q,m,\mathtt{this})\}$
        $\{\mathsf{lst}(q,M) * \circledast_{m\in M}\mathsf{pending}(t,P,Q,m,\mathtt{this})\}$
    if (q.Count > 0) {
        $\{\mathsf{lst}(q,M) * \circledast_{m\in M}\mathsf{pending}(t,P,Q,m,\mathtt{this}) * M \neq \emptyset\}$
      m = q.Pop();
        $\{\mathsf{lst}(q,M') * \circledast_{m\in\{m\}\cup M'}\mathsf{pending}(t,P,Q,m,\mathtt{this}) * M = M'\cup\{m\}\}$
        $\{\exists M.\ \mathsf{lst}(q,M) * \circledast_{m\in M}\mathsf{pending}(t,P,Q,m,\mathtt{this}) * \mathsf{pending}(t,P,Q,m,\mathtt{this})\}$
    } else {
        $\{\mathsf{lst}(q,M) * \circledast_{m\in M}\mathsf{pending}(t,P,Q,m,\mathtt{this})\}$
      m = null;
        {$\mathsf{lst}(q, M) * \circledast_{m \in M} \mathsf{pending}(t, P, Q, m, \mathtt{this}) * m = \mathtt{null}$}
    }
        {$\exists M.\ \mathsf{lst}(q, M) * \circledast_{m \in M} \mathsf{pending}(t, P, Q, m, \mathtt{this}) * (m = \mathtt{null} \vee \mathsf{pending}(t, P, Q, m, \mathtt{this})$}
      {$\exists M.\ \mathtt{this}.\mathsf{queue} \mapsto q * \mathsf{lst}(q, M) * \mathit{protocol}(t, I(Q)) * \circledast_{m \in M} \mathsf{pending}(t, P, Q, m, \mathtt{this})$
        $* (m = \mathtt{null} \vee \mathsf{pending}(t, P, Q, m, \mathtt{this})$}
      {$\mathsf{pool}(t, P, Q, \mathtt{this}) * (m = \mathtt{null} \vee \mathsf{pending}(t, P, Q, m, \mathtt{this})$}
    return m;
      {$\mathsf{ret}.\ \mathsf{pool}(t, P, Q, \mathtt{this}) * (\mathsf{ret} = \mathtt{null} \vee \mathsf{pending}(t, P, Q, \mathsf{ret}, \mathtt{this})$}
  }

  public void Push(Message m) {
    List<Message> q;
      {$\mathsf{pool}(t, P, Q, \mathtt{this}) * \mathsf{pending}(t, P, Q, m, \mathtt{this})$}
    q = this.queue;
      {$\mathtt{this}.\mathsf{queue} \mapsto q * \mathsf{lst}(q, M) * \mathit{protocol}(t, I(Q))$
        $* \mathsf{pending}(t, P, Q, m, \mathtt{this}) * \circledast_{m \in M} \mathsf{pending}(t, P, Q, m, \mathtt{this})$}
      {$\mathtt{this}.\mathsf{queue} \mapsto q * \mathsf{lst}(q, M) * \mathit{protocol}(t, I(Q))$
        $* \circledast_{m \in \{m\} \cup M} \mathsf{pending}(t, P, Q, m, \mathtt{this})$}
        q.Push(m);
      {$\mathtt{this}.\mathsf{queue} \mapsto q * \mathsf{lst}(q, \{m\} \cup M) * \mathit{protocol}(t, I(Q))$
        $* \circledast_{m \in \{m\} \cup M} \mathsf{pending}(t, P, Q, m, \mathtt{this})$}
      {$\mathsf{pool}(t, P, Q, \mathtt{this})$}
  }
}
```

4.3 The Pattern class

Pattern objects represent conditions on the message pool. This version of the joins library supports conditions of the form:

match a distinct message from each channel x from the multiset of channels X

Conditions are thus implemented as a list of channels representing the multiset X.

In addition to its public methods, the Pattern class has an internal method, Matches to determine whether the pattern matches the current message pool. If it does, Matches returns a list of messages that matches the pattern; otherwise, it returns null. Its specification is as follows:

Pattern.Matches : (-)

$$\{\mathsf{pattern}_{\mathsf{internal}}(\mathtt{this}, j, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)\}$$

$$\left\{\begin{array}{l}\mathsf{ret}.\ \exists Y : \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \mathsf{pattern}_{\mathsf{internal}}(\mathtt{this}, j, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x) \\ \qquad (\mathsf{ret} = \mathtt{null} \vee (\mathsf{lst}(\mathsf{ret}, \pi_2(Y)) * \circledast_{(c,m) \in Y} \mathsf{pending}(t, P, Q, m, c) * \pi_1(Y) = X))\end{array}\right\}$$

Here $\mathsf{pattern}_{\mathsf{internal}}(p, j, X)$ asserts that p refers to a condition on join instance j that matches a distinct message from each channel of the multiset of channels X.

Predicate definitions

The $\mathsf{pattern}(p, j, X)$ predicate asserts full ownership of the underlying the underlying list of channels, whereas $\mathsf{pattern}_{\mathsf{internal}}(p, j, X)$ only asserts read-only ownership. Both predicates assert that all the channels of the pattern belong to the correct join instance j using the chan(-) predicate:

$$\mathsf{pattern}(p, j, X) \stackrel{\text{def}}{=} \exists x : \text{Val}.\ p.join \mapsto j * p.channels \mapsto x * \mathsf{lst}(x, X) * \circledast_{c \in X} \mathsf{chan}(c, j)$$

$$\mathsf{pattern}_{\mathsf{internal}}(p, j, X) \stackrel{\text{def}}{=} \exists x : \text{Val}.\ p.join \mapsto_r j * p.channels \mapsto_r x * \mathsf{lst}_r(x, X) * \circledast_{c \in X} \mathsf{chan}(c, j)$$

The chan(c,j) predicate asserts that c refers to a channel registered with join instance j. For a channel to be registered with a join instance j, its join field should contain a reference to j *and* the channel should be in the channel list of join instance j. Since we require that the chan(-) predicate be duplicable, it cannot assert full ownership of the channel list of join instance j. Furthermore, since we use the chan(-) predicate in the join initialization phases (while the channel list is still growing), chan(-) cannot assert read-only ownership of the underlying channel list either. Instead, we extend the join instance j with a phantom field *chans* containing a finite multiset of channels, with a protocol that restricts modifications to addition of new channels:

$$\mathsf{chan}(c, j) \stackrel{\text{def}}{=} \exists t \in \text{RType}.\ j_{reg_c} \mapsto_r t * c.join \mapsto_r j * c \in^t j_{chans}$$

Here $c \in^t j_{chans}$ asserts that the channel c is a member of the finite multiset that *chans* contains:

$$c \in^t j_{chans} \stackrel{\text{def}}{=} \exists r : \text{RId}.\ \exists X : \mathcal{P}_m(\text{Val}).\ \boxed{j_{chans} \mapsto X}_I^{r, t\cdot\text{“3”}, j} * c \in X$$

where I is the parametric protocol:

$$I(j) = (\textsc{Add} : (Y : \mathcal{P}_m(\text{Val}), x : \text{Val}) : j_{chans} \mapsto Y \rightsquigarrow j_{chans} \mapsto Y \cup \{x\})$$

Clearly $c \in^t j_{chans}$ is stable as the protocol only allows new channels to be added and the $\mathsf{chan}(-)$ predicate is freely duplicable:

$$\mathsf{chan}(c, j) \Leftrightarrow \mathsf{chan}(c, j) * \mathsf{chan}(c, j)$$

Proof outline

Morally, Matches is implemented as follows,

```
internal List<Message> Matches() {
  List<Pair<Message, Channel>> consumed = new List<Pair<Message, Channel>>();
  bool rollback;

  channels.ForEach(ch => {
    if(!rollback) {
      Message msg = ch.Pop();
      if (msg != null)
        consumed.Push(new Pair<Message, Channel>(msg, ch));
      else
        rollback = true;
    }
  });

  if(rollback) {
    consumed.ForEach(p => {
      p.snd.Push(p.fst);
    });
    return null;
  } else {
    return consumed.Map(List.pi2<Message, Channel>);
  }
}
```

using the higher-order List.ForEach method to optimistically try to Pop a message from each channel of the pattern. However, as mini C# lacks anonymous delegates (to avoid the difficulties of reasoning about variable capture [2]) we introduce an explicit inner class, Inner, with fields to replace the captured variables consumed and rollback.

```
public class Inner {
  public List<Pair<Channel, Message>> consumed;
  public bool rollback;

  public Inner() {
      $\{\circledast_{x \in X} \mathsf{pool}(t, P, Q, x)\}$
    consumed = new List<Pair<Channel, Message>>();
    rollback = false;
      $\{\mathtt{this}.\mathsf{consumed} \mapsto x * \mathsf{lst}_{pair}(x, \emptyset) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)\}$
      $\{\mathsf{inner}(t, P, Q, \mathtt{this}, X, \emptyset)\}$
  }

  public void Pop(Channel ch) {
    List<Pair<Channel, Message>> con;
    Pair<Channel, Message> p;
    Message msg;
      $\{\mathsf{inner}(t, P, Q, \mathtt{this}, X, Y) * ch \in X\}$
    con = this.consumed;
      $\{\mathtt{this}.consumed \mapsto con * \mathsf{lst}_{pair}(con, Z) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x) * \circledast_{(c,m) \in Z} \mathsf{pending}(t, P, Q, m, c)$
         $* \mathtt{this}.rollback \mapsto r * (r = true \vee \pi_1(Z) = Y) * ch \in X\}$
        $\{\mathsf{lst}_{pair}(con, Z) * \mathsf{pool}(t, P, Q, ch) * \circledast_{(c,m) \in Z} \mathsf{pending}(t, P, Q, m, c)$
```

```
          $* \texttt{this}.rollback \mapsto r * (r = true \vee \pi_1(Z) = Y)\}$
    if (!rollback) {
        $\{\mathsf{lst}_{pair}(con, Z) * \mathsf{pool}(t, P, Q, ch) * \circledast_{(c,m)\in Z}\mathsf{pending}(t, P, Q, m, c)$
            $* \texttt{this}.rollback \mapsto false * \pi_1(Z) = Y\}$
      msg = ch.Pop();
        $\{\mathsf{lst}_{pair}(con, Z) * \mathsf{pool}(t, P, Q, ch) * \circledast_{(c,m)\in Z}\mathsf{pending}(t, P, Q, m, c)$
            $* \texttt{this}.rollback \mapsto false * \pi_1(Z) = Y$
            $* (msg = \texttt{null} \vee \mathsf{pending}(t, P, Q, msg, ch))\}$
      if (msg != null) {
          $\{\mathsf{lst}_{pair}(con, Z) * \mathsf{pool}(t, P, Q, ch) * \circledast_{(c,m)\in Z}\mathsf{pending}(t, P, Q, m, c)$
              $* \texttt{this}.rollback \mapsto false * \pi_1(Z) = Y * \mathsf{pending}(t, P, Q, msg, ch))\}$
        con.Push(new Pair<Channel, Message>(ch, msg));
          $\{\mathsf{lst}_{pair}(con, \{(ch, msg)\} \cup Z) * \mathsf{pool}(t, P, Q, ch) * \circledast_{(c,m)\in\{(ch,msg)\cup Z}\mathsf{pending}(t, P, Q, m, c)$
              $* \texttt{this}.rollback \mapsto false * \pi_1(Z) = Y)\}$
      } else {
          $\{\mathsf{lst}_{pair}(con, Z) * \mathsf{pool}(t, P, Q, ch) * \circledast_{(c,m)\in Z}\mathsf{pending}(t, P, Q, m, c)$
              $* \texttt{this}.rollback \mapsto false * \pi_1(Z) = Y * msg = \texttt{null})\}$
        rollback = true;
          $\{\mathsf{lst}_{pair}(con, Z) * \mathsf{pool}(t, P, Q, ch) * \circledast_{(c,m)\in Z}\mathsf{pending}(t, P, Q, m, c)$
              $* \texttt{this}.rollback \mapsto true * \pi_1(Z) = Y)\}$
      }
    }
        $\{\exists Z.\ \mathsf{lst}_{pair}(con, Z) * \mathsf{pool}(t, P, Q, ch) * \circledast_{(c,m)\in Z}\mathsf{pending}(t, P, Q, m, c)$
            $* \texttt{this}.rollback \mapsto r * (r = true \vee \pi_1(Z) = \{ch\} \cup Y)\}$
      $\{\exists Z.\ \texttt{this}.consumed \mapsto con * \mathsf{lst}_{pair}(con, Z) * \circledast_{x\in X}\mathsf{pool}(t, P, Q, x) * \circledast_{(c,m)\in Z}\mathsf{pending}(t, P, Q, m, c)$
          $* \texttt{this}.rollback \mapsto r * (r = true \vee \pi_1(Z) = \{ch\} \cup Y)\}$
      $\{\mathsf{inner}(t, P, Q, \texttt{this}, X, \{ch\} \cup Y)\}$
  }
}
```

where

$$\begin{aligned}
&\mathsf{inner}(t, P, Q, x, X, Y) = \\
&\quad \exists Z : \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \exists v : \mathrm{Val}. \\
&\qquad x.consumed \mapsto y * \mathsf{lst}_{pair}(y, Z) * \circledast_{x\in X}\mathsf{pool}(t, P, Q, x) \\
&\qquad \circledast_{(c,m)\in Z}\,\mathsf{pending}(t, P, Q, m, c) * x.rollback \mapsto v * (b = true \vee \pi_1(Z) = Y)
\end{aligned}$$

```
public class Pattern {
  internal List<Channel> channels = new List<Channel>();
  internal Join join;

  internal List<Message> Matches() {
    Inner inner; Message msg; List<Message> msgs; List<Channel> chs;
    $\{\mathsf{pattern}_{\mathsf{internal}}(\mathtt{this}, j, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)\}$
  msgs = null;
  chs = this.channel;
    $\{\mathtt{this}.join \mapsto j * \mathtt{this}.channels \mapsto chs * \mathsf{lst}_r(chs, X) \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)\}$
      $\{\mathsf{lst}_r(chs, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)\}$
  inner = new Inner();
      $\{\mathsf{lst}_r(chs, X) * \mathsf{inner}(t, P, Q, inner, X, \emptyset)$
          $* \forall Y.\ inner.Pop \mapsto \begin{matrix}(\mathsf{ch}).\ \{\mathsf{inner}(t, P, Q, inner, X, Y) * ch \in X\} \\ \{\mathsf{inner}(t, P, Q, inner, X, \{ch\} \cup Y)\}\end{matrix}\}$
  chs.ForEach(inner.Pop);
      $\{\mathsf{lst}_r(chs, X) * \mathsf{inner}(t, P, Q, inner, X, X)\}$
      $\{\mathsf{lst}_r(chs, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)$
           $* inner.consumed \mapsto y * \mathsf{lst}_{pair}(y, Y) * \circledast_{(c,m) \in Y} \mathsf{pending}(t, P, Q, m, c)$
           $* inner.rollback \mapsto r * (r = true \vee \pi_1(Y) = X)\}$
  if(inner.rollback) {
      $\{\mathsf{lst}_r(chs, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)$
           $* inner.consumed \mapsto y * \mathsf{lst}_{pair}(y, Y) * \circledast_{(c,m) \in Y} \mathsf{pending}(t, P, Q, m, c)$
           $* inner.rollback \mapsto r\}$
    inner.consumed.ForEach(Push);
      $\{\mathsf{lst}_r(chs, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)$
           $* inner.consumed \mapsto y * \mathsf{lst}_{pair}(y, Y) * inner.rollback \mapsto r\}$
      $\{\mathsf{lst}_r(chs, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x) * msgs = null\}$
  } else {
      $\{\mathsf{lst}_r(chs, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)$
           $* inner.consumed \mapsto y * \mathsf{lst}_{pair}(y, Y) * \circledast_{(c,m) \in Y} \mathsf{pending}(t, P, Q, m, c)$
           $* inner.rollback \mapsto r * \pi_1(Y) = X\}$
    msgs = inner.consumed.Map(pi2<Channel, Message>);
      $\{\mathsf{lst}_r(chs, X) * \mathsf{lst}(msgs, \pi_2(Y)) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)$
           $* inner.consumed \mapsto y * \mathsf{lst}_{pair}(y, Y) * \circledast_{(c,m) \in Y} \mathsf{pending}(t, P, Q, m, c)$
           $* inner.rollback \mapsto r * \pi_1(Y) = X\}$
  }
      $\{\mathsf{lst}_r(chs, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)$
           $* (msgs = null \vee (\mathsf{lst}(msgs, \pi_2(Y)) * \circledast_{(c,m) \in Y} \mathsf{pending}(t, P, Q, m, c) * \pi_1(Y) = X))\}$
    $\{\mathtt{this}.join \mapsto j * \mathtt{this}.channels \mapsto chs * \mathsf{lst}_r(chs, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)$
         $* (msgs = null \vee (\mathsf{lst}(msgs, \pi_2(Y)) * \circledast_{(c,m) \in Y} \mathsf{pending}(t, P, Q, m, c) * \pi_1(Y) = X))\}$
    $\{\mathsf{pattern}_{\mathsf{internal}}(\mathtt{this}, j, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)$
         $* (msgs = null \vee (\mathsf{lst}(msgs, \pi_2(Y)) * \circledast_{(c,m) \in Y} \mathsf{pending}(t, P, Q, m, c) * \pi_1(Y) = X))\}$
  return msgs;
}

internal void Push(Pair<Channel, Message> p) {
  Channel ch; Message msg;
```

```
    ch = p.Fst();
    msg = p.Snd();
    ch.Push(msg);
  }

  internal B pi2<A,B>(Pair<A,B> p) {
    return p.snd;
  }

  public Pattern And(Channel ch) {
    List<Channel> chs;
      $\{\mathsf{pattern}(\mathtt{this}, j, X) * \mathsf{chan}(ch, j)\}$
    chs = this.channels;
      $\{\mathtt{this}.channels \mapsto chs * \mathsf{lst}(chs, X) * \circledast_{c \in X} \mathsf{chan}(c, j) * \mathsf{chan}(ch, j)\}$
    chs.Push(ch);
      $\{\mathtt{this}.channels \mapsto chs * \mathsf{lst}(chs, \{ch\} \cup X) * \circledast_{c \in \{ch\} \cup X} \mathsf{chan}(c, j)\}$
      $\{\mathsf{pattern}(\mathtt{this}, j, \{ch\} \cup X)\}$
    return this;
  }

  public void Do(Action cont) {
    Chord chord = new Chord(this, cont);
    join.chords.Push(chord);
  }
}
```

4.4 The AsyncChannel, SyncChannel, Chord and Join classes

For the Message, MessagePool and Pattern classes it was possible to define each of their representation predicates independently of the rest of the joins library. However, this is not the case for the AsyncChannel, SyncChannel, Chord and Join classes, due to the possibility of the reentrant continuations. In this section we will thus define the remaining representation predicates up front, followed by proof outlines for the rest of the implementation.

Predicate definitions

We start by defining the internal representation predicates for joins instances in the third phase. At this point in time all channels and chords have already been initialized and the only way to interact with the joins instance is by sending messages. Except for the underlying message pools, all the state maintained by the joins instance is thus fixed. It is thus sufficient for most of these representation predicates to only assert read-only permission, allowing them to be freely duplicated.

The chord representation predicate, $\mathsf{chord}(f, P, Q, c, j)$, asserts that c refers to a chord registered with joins instance j. Chord continuations are specified in terms of the $\mathsf{join}_{\mathsf{call}}(-)$ predicate, which in turn is defined in terms of the $\mathsf{chord}(-)$ predicate. We thus initially define the $\mathsf{chord}(-)$ predicate in terms of an abstract $\mathsf{join}_{\mathsf{call}}(-)$ predicate f, closing the loop using guarded recursion in the definition of the $\mathsf{join}_{\mathsf{call}}(-)$ predicate below.

$$\begin{aligned}
\mathsf{chord}(f, P, Q, c, j) \stackrel{\text{def}}{=} \exists x, y : \text{Val}.\ \exists X : \mathcal{P}_m(\text{Val}).&\\
& c.pat \mapsto x * \mathsf{pattern}_{\mathsf{internal}}(x, j, X)\\
& * \ c.cont \mapsto y\\
& * \ (\forall Y \in \mathcal{P}_m(\text{Val} \times \text{Val}).\ \pi_{\mathsf{chan}}(Y) = X \ \Rightarrow\\
& \qquad \triangleright\, y \mapsto \{f(j) * \circledast_{y \in Y} P(y)\}\{f(j) * \circledast_{y \in Y} Q(y)\})
\end{aligned}$$

The internal join representation predicate, $\mathsf{join}_{\mathsf{internal}}(t, f, P, Q, j)$, asserts that j refers to a join instance.

$$\begin{aligned}
\mathsf{join}_{\mathsf{internal}}(t, f, P, Q, j) \stackrel{\text{def}}{=} \exists \mathsf{x}, \mathsf{y} : \text{Val}.\ \exists X, Y : \mathcal{P}_m(\text{Val}).&\\
& j.channels \mapsto \mathsf{x} * \mathsf{lst}_r(\mathsf{x}, X)\\
& * \ j.chords \mapsto \mathsf{y} * \mathsf{lst}_r(\mathsf{y}, Y)\\
& * \circledast_{c \in X} \mathsf{chan}(c, j) * \mathsf{pool}(t, P, Q, c)\\
& * \circledast_{c \in Y} \mathsf{chord}(f, P, Q, c, j)\\
& * \ (\forall a : \text{Val}.\ \forall \mathsf{x} \in X.\ \mathsf{x} : \mathsf{AsyncChannel} \Rightarrow Q(a, \mathsf{x}) = \text{emp})
\end{aligned}$$

We can now define the $\mathsf{join}_{\mathsf{call}}(-)$ predicate by guarded recursion:

$$\begin{aligned}
\mathsf{join}_{\mathsf{call}}(t, P, Q) \stackrel{\text{def}}{=} \boldsymbol{\mathit{fix}}(\lambda f : \text{Val} \to \text{Prop}.\ \lambda j : \text{Val}.\ \exists l : \text{Val}.\ \exists t_1, t_2, t_3 \in \text{RType}.&\\
& t \leq t_1 * t \leq t_2 * t \leq t_3 * j_{reg_l} \mapsto t_1 * j_{reg_m} \mapsto t_2 * j_{reg_c} \mapsto t_3\\
& * \ j.lock \mapsto l * \text{isLock}(t_1, l, \mathsf{join}_{\mathsf{internal}}(t_2, f, P, Q, j)))
\end{aligned}$$

This is well-defined as the occurrence of f in $\mathsf{chord}(-)$ is guarded by $\triangleright$. Furthermore, the lock specification (See Section 4.6) requires that the resource invariant – in this case $\mathsf{join}_{\mathsf{internal}}(t_2, f, P, Q, j)$ – is expressible using first-order protocols. By assumption, the channel pre- and post-conditions are uniformly expressible using first-order protocols. Hence, for a fixed t : RType, $\mathsf{join}_{\mathsf{internal}}(t, f, P, Q, j)$ is expressible using state-independent protocols, as the parameterized message protocols used in the $\mathsf{pool}(-)$ predicate, the $\mathsf{chan}(-)$ predicate and the protocols used by the channel pre- and post-conditions, can all be pulled out under all the existential quantifiers. Formally, we prove that,

$$\begin{aligned}
&\forall t : \text{RType}.\ \forall f : \text{Val} \to \text{Prop}.\ \forall j : \text{Val}.\ \exists S, T : \text{Prop}.\\
&\qquad noprotocol(S) \wedge nostate(T) \wedge (\mathsf{join}_{\mathsf{internal}}(t, f, P, Q, j) \Leftrightarrow S * T)
\end{aligned}$$

This is provable without any assumptions about f, as f is only used in the pre- and post-condition of a nested triple. Stability of $\mathsf{join}_{\mathsf{call}}(-)$ thus follows from the stability of

$\mathsf{isLock}(-)$. To simplify the notation, in the following proofs we will use $\mathsf{join}_{\mathsf{call}}(t, P, Q, j)$ and $\mathsf{join}_{\mathsf{call}}(t, P, Q)(j)$ interchangeably.

In the running phase no new chords or channels can be registered with the join instance and read-only permissions thus suffices. In the initialization phase this is not the case and $\mathsf{join}_{\mathsf{init\text{-}ch}}(-)$ and $\mathsf{join}_{\mathsf{init\text{-}pat}}(-)$ thus assert full ownership of the channel and chord list:

$$\begin{aligned}\mathsf{join}_{\mathsf{init\text{-}ch}}(t, C_A, C_S, j) \overset{\text{def}}{=} \;& \exists x, y, l : \text{Val}.\ \exists t_l, t_c \in \text{RType}. \\ & j.\mathsf{channels} \mapsto x * \mathsf{lst}(x, C_A \cup C_S) * \circledast_{c \in C_A \cup C_S} \mathsf{chan}(c, j) \\ & * \ j.\mathsf{chords} \mapsto y * \mathsf{lst}(y, \emptyset) \\ & * \ j.\mathsf{lock} \mapsto l * \mathsf{lock}(t_l, l) \\ & * \ j_{reg_l} \mapsto t_l * j_{reg_m} \mapsto \_ * j_{reg_c} \Mapsto t_c * t \leq t_c * t \leq t_l \\ & * \ (\forall c \in C_A.\ c : \mathsf{AsyncChannel}) \\ & * \ (\forall c \in C_S.\ c : \mathsf{SyncChannel}) \\ & * \ j_{chans} \mapsto^{t_c}_{=} C_A \cup C_S\end{aligned}$$

Here $j_{chans} \mapsto^{t}_{=} C_A \cup C_S$ asserts that the phantom field *chans* contains exactly $C_A \cup C_S$ *and* full ownership of the [ADD] action:

$$j_{chans} \mapsto^{t}_{=} X \overset{\text{def}}{=} \exists r : \text{RId}.\ \boxed{j_{chans} \mapsto X}^{r,t,j}_{I} * [\text{ADD}]^{r}_{1}$$

where I is the previously defined protocol from Section 4.3. The $\mathsf{join}_{\mathsf{init\text{-}ch}}(-)$ further asserts that $j.\mathsf{lock}$ refers to an uninitialized lock (See Section 4.6), as the lock invariant depends on the channel pre- and post-conditions provided by the client in the view-shift to $\mathsf{join}_{\mathsf{init\text{-}pat}}(-)$. The lock is initialized in the view shift from $\mathsf{join}_{\mathsf{init\text{-}pat}}(-)$ to $\mathsf{join}_{\mathsf{call}}(-)$.

Chord continuations can send messages on channels on their own join instance. In the definition of $\mathsf{join}_{\mathsf{init\text{-}pat}}(-)$ we thus use the previously defined $\mathsf{join}_{\mathsf{call}}(-)$ predicate to give meaning to chords:

$$\begin{aligned}\mathsf{join}_{\mathsf{init\text{-}pat}}(t, P, Q, j) \overset{\text{def}}{=} \;& \exists x, y : \text{Val}.\ \exists X, Y : \mathcal{P}_m(\text{Val}).\ \exists t_l, t_c \in \text{RType}. \\ & j.\mathsf{channels} \mapsto x * \mathsf{lst}(x, X) * \circledast_{c \in X} \mathsf{chan}(c, j) \\ & * \ j.\mathsf{chords} \mapsto y * \mathsf{lst}(y, Y) * \circledast_{c \in Y} \mathsf{chord}(\mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, c, j) \\ & * \ j.\mathsf{lock} \mapsto l * \mathsf{lock}(t_l, l) \\ & * \ j_{reg_l} \mapsto t_l * j_{reg_m} \mapsto \_ * j_{reg_c} \Mapsto t_c * t \leq t_l * t \leq t_c \\ & * \ (\forall a : \text{Val}.\ \forall c \in X.\ c : \mathsf{AsyncChannel} \Rightarrow Q(a, c) = \text{emp}) \\ & * \ j_{chans} \mapsto^{t_c}_{=} X\end{aligned}$$

Proof outlines

```
internal class Chord {
  internal Pattern pat;
```

```
    internal Action cont;

    internal Chord(Pattern pat, Action cont) {
            this.cont = cont;
            this.pat = pat;
    }
}

public class AsyncChannel : MessagePool {
    Join join;

    public AsyncChannel(Join j) {
        List<Message> chans;
            ${\mathsf{join}_{\mathsf{init\text{-}ch}}(t, C_A, C_S, \mathsf{j}) * \mathtt{this} : \mathsf{AsyncChannel} * \mathtt{this}.join \mapsto \mathtt{null}}$
            ${\mathsf{join}_{\mathsf{init\text{-}ch}}(t, C_A, C_S, \mathsf{j}) * \mathtt{this} : \mathsf{AsyncChannel} * \mathtt{this}.join \mapsto \mathtt{null} * \mathtt{this} \notin C_A \cup C_S}$
        this.join = j;
            ${\mathsf{join}_{\mathsf{init\text{-}ch}}(t, C_A, C_S, \mathsf{j}) * \mathtt{this} : \mathsf{AsyncChannel} * \mathtt{this}.join \mapsto \mathsf{j} * \mathtt{this} \notin C_A \cup C_S}$
        chans = join.channels;
            ${\mathsf{j}.channels \mapsto chans * \mathsf{lst}(chans, C_A \cup C_S) * \circledast_{c \in C_A \cup C_S} \mathsf{chan}(c, \mathsf{j}) * \mathtt{this} \notin C_A \cup C_S$
              $* \mathsf{j}.chords \mapsto y * \mathsf{lst}(y, \emptyset) * (\forall c \in C_A.\ c : \mathsf{AsyncChannel}) * (\forall c \in C_S.\ c : \mathsf{SyncChannel})$
              $* \mathsf{j}_{reg_l} \mapsto \_ * \mathsf{j}_{reg_m} \mapsto \_ * \mathsf{j}_{reg_c} \Mapsto t_c * t \leq t_c$
              $* \mathsf{j}_{chans} \mapsto^{t_c}_{=} C_A \cup C_S * \mathtt{this} : \mathsf{AsyncChannel} * \mathtt{this}.join \mapsto \mathsf{j}}$
        chans.Push(this);
            ${\mathsf{j}.channels \mapsto chans * \mathsf{lst}(chans, \{\mathtt{this}\} \cup C_A \cup C_S) * \circledast_{c \in C_A \cup C_S} \mathsf{chan}(c, \mathsf{j}) * \mathtt{this} \notin C_A \cup C_S$
              $* \mathsf{j}.chords \mapsto^{t} y * \mathsf{lst}(y, \emptyset) * (\forall c \in C_A.\ c : \mathsf{AsyncChannel}) * (\forall c \in C_S.\ c : \mathsf{SyncChannel})$
              $* \mathsf{j}_{reg_l} \mapsto \_ * \mathsf{j}_{reg_m} \mapsto \_ * \mathsf{j}_{reg_c} \Mapsto t_c * t \leq t_c$
              $* \mathsf{j}_{chans} \mapsto^{t_c}_{=} C_A \cup C_S * \mathtt{this} : \mathsf{AsyncChannel} * \mathtt{this}.join \mapsto \mathsf{j}}$
            ${\mathsf{j}.channels \mapsto chans * \mathsf{lst}(chans, \{\mathtt{this}\} \cup C_A \cup C_S) * \circledast_{c \in C_A \cup C_S} \mathsf{chan}(c, \mathsf{j}) * \mathtt{this} \notin C_A \cup C_S$
              $* \mathsf{j}.chords \mapsto y * \mathsf{lst}(y, \emptyset) * (\forall c \in C_A.\ c : \mathsf{AsyncChannel}) * (\forall c \in C_S.\ c : \mathsf{SyncChannel})$
              $* \mathsf{j}_{reg_l} \mapsto \_ * \mathsf{j}_{reg_m} \mapsto \_ * \mathsf{j}_{reg_c} \Mapsto t_c * t \leq t_c$
              $* \mathsf{j}_{chans} \mapsto^{t}_{=} \{\mathtt{this}\} \cup C_A \cup C_S * \mathtt{this} : \mathsf{AsyncChannel} * \mathsf{chan}(\mathtt{this}, j)}$
            ${\mathsf{join}_{\mathsf{init\text{-}ch}}(t, C_A \cup \{\mathtt{this}\}, C_S, \mathsf{j}) * \mathsf{chan}(\mathtt{this}, \mathsf{j}) * \mathtt{this} \notin C_A \cup C_S}$
    }

    public void call() {
        Join j; Lock l;
            ${\mathtt{this} : \mathsf{AsyncChannel} * \mathsf{join}_{\mathsf{call}}(t, P, Q, j) * \mathsf{chan}(\mathtt{this}, j) * P(a, \mathtt{this})}$
            ${\mathsf{join}_{\mathsf{call}}(t, P, Q, j) * \mathtt{this}.join \mapsto j * P(a, \mathtt{this}) * Q(a, \mathtt{this})}$
        j = this.join;
            ${\mathsf{join}_{\mathsf{call}}(t, P, Q, \mathsf{j}) * \mathtt{this}.join \mapsto \mathsf{j} * P(a, \mathtt{this}) * Q(a, \mathtt{this})}$
        l = j.lock;
        l.Acquire();
            ${\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}))$
```

```
      $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}) * \mathtt{this}.\mathit{join} \mapsto \mathsf{j} * P(a, \mathtt{this}) * Q(a, \mathtt{this})\}$
      $\{\mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}) * P(a, \mathtt{this})\}$
        $\{\mathsf{pool}(t_m, P, Q, \mathtt{this}) * P(a, \mathtt{this})\}$
    AddMessage();
        $\{\exists x : \mathrm{Val}.\ \mathsf{pool}(t_m, P, Q, \mathtt{this}) * \mathsf{msg}(t_m, P, Q, a, x, \mathtt{this})\}$
      $\{\mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j})\}$
    $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}))$
      $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}) * Q(a, \mathtt{this})\}$
    l.Release();
    $\{\mathsf{join}_{\mathsf{call}}(t, P, Q, \mathsf{j}) * Q(a, \mathtt{this})\}$
  }
}
```

Since the channel pre- and post-condition predicates P and Q are indexed by channel references, when creating a new channel, the client needs to know the newly created channel is distinct from existing channels ($\mathtt{this} \notin C_A \cup C_S$). Intuitively, this should obviously hold in the channel constructor; formally, we use the fact that $\mathtt{this}.\mathsf{join} \mapsto \mathtt{null}$ and that all channels registered with a join instance refers back to that join instance (See the definition of the $\mathsf{chan}(-)$ predicate).

```
public class SyncChannel : MessagePool {
  Join join;

  public SyncChannel(Join j) {
    List<Message> chans;
      $\{\mathsf{join}_{\mathsf{init\text{-}ch}}(t, C_A, C_S, \mathsf{j}) * \mathtt{this} : \mathsf{SyncChannel} * \mathtt{this}.\mathit{join} \mapsto \mathtt{null}\}$
      $\{\mathsf{join}_{\mathsf{init\text{-}ch}}(t, C_A, C_S, \mathsf{j}) * \mathtt{this} : \mathsf{SyncChannel} * \mathtt{this}.\mathit{join} \mapsto \mathtt{null} * \mathtt{this} \notin C_A \cup C_S\}$
    this.join = j;
      $\{\mathsf{join}_{\mathsf{init\text{-}ch}}(t, C_A, C_S, \mathsf{j}) * \mathtt{this} : \mathsf{SyncChannel} * \mathtt{this}.\mathit{join} \mapsto \mathsf{j} * \mathtt{this} \notin C_A \cup C_S\}$
    chans = join.channels;
      $\{\mathsf{j}.\mathit{channels} \mapsto \mathit{chans} * \mathsf{lst}(\mathit{chans}, C_A \cup C_S) * \circledast_{c \in C_A \cup C_S} \mathsf{chan}(c, \mathsf{j}) * \mathtt{this} \notin C_A \cup C_S$
        $* \mathsf{j}.\mathit{chords} \mapsto y * \mathsf{lst}(y, \emptyset) * (\forall c \in C_A.\ c : \mathsf{AsyncChannel}) * (\forall c \in C_S.\ c : \mathsf{SyncChannel})$
        $* \mathsf{j}_{reg_l} \mapsto \_ * \mathsf{j}_{reg_m} \mapsto \_ * \mathsf{j}_{reg_c} \Mapsto t_c * t \leq t_c$
        $* \mathsf{j}_{chans} \mapsto^{t_c}_{=} C_A \cup C_S * \mathtt{this} : \mathsf{SyncChannel} * \mathtt{this}.\mathit{join} \mapsto \mathsf{j}\}$
    chans.Push(this);
      $\{\mathsf{j}.\mathit{channels} \mapsto \mathit{chans} * \mathsf{lst}(\mathit{chans}, \{\mathtt{this}\} \cup C_A \cup C_S) * \circledast_{c \in C_A \cup C_S} \mathsf{chan}(c, \mathsf{j}) * \mathtt{this} \notin C_A \cup C_S$
        $* \mathsf{j}.\mathit{chords} \mapsto y * \mathsf{lst}(y, \emptyset) * (\forall c \in C_A.\ c : \mathsf{AsyncChannel}) * (\forall c \in C_S.\ c : \mathsf{SyncChannel})$
        $* \mathsf{j}_{reg_l} \mapsto \_ * \mathsf{j}_{reg_m} \mapsto \_ * \mathsf{j}_{reg_c} \Mapsto t_c * t \leq t_c$
        $* \mathsf{j}_{chans} \mapsto^{t_c}_{=} C_A \cup C_S * \mathtt{this} : \mathsf{SyncChannel} * \mathtt{this}.\mathit{join} \mapsto \mathsf{j}\}$
      $\{\mathsf{j}.\mathit{channels} \mapsto \mathit{chans} * \mathsf{lst}(\mathit{chans}, \{\mathtt{this}\} \cup C_A \cup C_S) * \circledast_{c \in C_A \cup C_S} \mathsf{chan}(c, \mathsf{j}) * \mathtt{this} \notin C_A \cup C_S$
        $* \mathsf{j}.\mathit{chords} \mapsto y * \mathsf{lst}(y, \emptyset) * (\forall c \in C_A.\ c : \mathsf{AsyncChannel}) * (\forall c \in C_S.\ c : \mathsf{SyncChannel})$
        $* \mathsf{j}_{reg_l} \mapsto \_ * \mathsf{j}_{reg_m} \mapsto \_ * \mathsf{j}_{reg_c} \Mapsto t_c * t \leq t_c$
        $* \mathsf{j}_{chans} \mapsto^{t_c}_{=} \{\mathtt{this}\} \cup C_A \cup C_S * \mathtt{this} : \mathsf{SyncChannel} * \mathsf{chan}(\mathtt{this}, j)\}$
      $\{\mathsf{join}_{\mathsf{init\text{-}ch}}(t, C_A, C_S \cup \{\mathtt{this}\}, \mathsf{j}) * \mathsf{chan}(\mathtt{this}, \mathsf{j}) * \mathtt{this} \notin C_A \cup C_S\}$
  }

  public void call() {
    Join j; Message msg; bool done; Lock l;
      $\{\mathtt{this} : \mathsf{SyncChannel} * \mathsf{join}_{\mathsf{call}}(t, P, Q, j) * \mathsf{chan}(\mathtt{this}, j) * P(a, \mathtt{this})\}$
    j = this.join;
      $\{\mathsf{join}_{\mathsf{call}}(t, P, Q, \mathsf{j}) * \mathsf{chan}(\mathtt{this}, \mathsf{j}) * P(a, \mathtt{this})\}$
    l = j.lock;
    l.Acquire();
      $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}))$
        $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}) * \mathsf{chan}(\mathtt{this}, \mathsf{j}) * P(a, \mathtt{this})\}$
        $\{\mathsf{pool}(t_m, P, Q, \mathtt{this}) * P(a, \mathtt{this})\}$
    Message msg = AddMessage();
        $\{\mathsf{pool}(t_m, P, Q, \mathtt{this}) * \mathsf{msg}(t_m, P, Q, a, msg, \mathtt{this})\}$
      $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}))$
        $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}) * \mathsf{msg}(t_m, P, Q, a, msg, \mathtt{this})\}$
    done = false;
      $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}))$
         $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j})$
         $* (done * Q(a, \mathtt{this}) \vee \neg done * \mathsf{msg}(t_m, P, Q, a, \mathsf{msg}, \mathtt{this}))\}$
    while (!done) {
        $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}))$
          $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}) * \mathsf{msg}(t_m, P, Q, a, msg, \mathtt{this}))\}$
      j.checkChords();
        $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}))$
          $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}) * \mathsf{msg}(t_m, P, Q, a, msg, \mathtt{this}))\}$
      if (msg.status == Status.Released) {
          $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}))$
            $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}) * Q(a, \mathtt{this})\}$
        done = true;
      } else {
          $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}))$
            $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}) * \mathsf{msg}(t \cdot \text{“2”}, P, Q, a, msg, \mathtt{this}))\}$
        l.Release();
          $\{\mathsf{join}_{\mathsf{call}}(t, P, Q, \mathsf{j}) * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{msg}(t_m, P, Q, a, msg, \mathtt{this}))\}$
        Thread.Sleep(1);
          $\{\mathsf{join}_{\mathsf{call}}(t, P, Q, \mathsf{j}) * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{msg}(t_m, P, Q, a, msg, \mathtt{this}))\}$
        l.Acquire();
          $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}))$
             $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}) * \mathsf{msg}(t_m, P, Q, a, msg, \mathtt{this}))\}$
      }
    }
```

```
      $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}))$
        $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathsf{j}) * Q(a, \mathtt{this})\}$
    l.Release();
      $\{\mathsf{join}_{\mathsf{call}}(t, P, Q, \mathsf{j}) * Q(a, \mathtt{this})\}$
  }
}
```

In the proof of the SyncChannel.Call method, when the busy loop releases the lock, the client keeps some fractional permission of the read-only j_{reg_m} phantom field containing the region type for message protocols. This ensures that the $\mathsf{msg}(-)$ assertion and the $\mathsf{pool}(-)$ assertion inside $\mathsf{join}_{\mathsf{internal}}(-)$ refers to the same region type for message protocols.

4.5 The Join class

```
public class Join {
  public LinkedList<Chord> chords;
  public LinkedList<Channel> channels;
  public Lock lock;

  public Join() {
      $\{\mathtt{this}_{reg_l} \mapsto \_ * \mathtt{this}_{reg_c} \mapsto \_ * \mathtt{this}_{reg_m} \mapsto \_ * \mathtt{this}_{chans} \mapsto \_$
        $* \mathtt{this.channels} \mapsto \mathtt{null} * \mathtt{this.chords} \mapsto \mathtt{null} * \mathtt{this.lock} \mapsto \mathtt{null}\}$
    channels = new LinkedList<Channel>();
    chords = new LinkedList<Chord>();
    lock = new Lock();
      $\{\mathtt{this}_{reg_l} \mapsto \_ * \mathtt{this}_{reg_c} \mapsto \_ * \mathtt{this}_{reg_m} \mapsto \_ * \mathtt{this}_{chans} \mapsto \_$
        $* \mathtt{this.channels} \mapsto c * \mathtt{this.chords} \mapsto ch * \mathtt{this.lock} \mapsto l$
        $* lst(c, \emptyset) * lst(ch, \emptyset) * \mathsf{lock}(t_l, l) * t \leq t_l\}$
      $\{\mathtt{this}_{reg_l} \mapsto t_l * \mathtt{this}_{reg_c} \mapsto t_c * \mathtt{this}_{reg_m} \mapsto \_ * \mathtt{this}_{chans} \mapsto \emptyset$
        $* \mathtt{this.channels} \mapsto c * \mathtt{this.chords} \mapsto ch * \mathtt{this.lock} \mapsto l$
        $* lst(c, \emptyset) * lst(ch, \emptyset) * \mathsf{lock}(t_l, l) * t \leq t_l * t \leq t_c\}$
      $\{\mathsf{join}_{\mathsf{init\text{-}ch}}(t, \emptyset, \emptyset, \mathtt{this})\}$
  }

  public Chord When(SyncChannel ch) {
    Chord chord = new Chord(ch, this);
    return chord;
  }

  internal void checkChord(Chord c) {
    List<Message> consumed; Action act; Pattern pat;
      $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathtt{this}))$
          $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathtt{this})$
          $* c \in Y * \circledast_{c \in Y} \mathsf{chord}(\mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, c, \mathtt{this})\}$
      $\{\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathtt{this}))$
    $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}) * \mathsf{chord}(\mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, c, \mathtt{this})\}$
pat = c.pat;
act = c.cont;
  $\{\mathsf{j}_{reg_l} \mapsto t_l * \mathsf{j}_{reg_m} \mapsto t_m * \mathsf{j.lock} \mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}))$
    $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}) * \mathsf{pattern}_{\mathsf{internal}}(pat, \mathtt{this}, X)$
    $* \forall Y \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \pi_{\mathsf{chan}}(Y) = X \Rightarrow$
      $\triangleright act \mapsto \{\mathsf{join}_{\mathsf{call}}(t,P,Q)(\mathtt{this}) * \circledast_{y \in Y} P(y)\}\{\mathsf{join}_{\mathsf{call}}(t,P,Q)(\mathtt{this}) * \circledast_{y \in Y} Q(y)\}\}$
consumed = pat.Matches();
  $\{\mathsf{j}_{reg_l} \mapsto t_l * \mathsf{j}_{reg_m} \mapsto t_m * \mathsf{j.lock} \mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}))$
    $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}) * \mathsf{pattern}_{\mathsf{internal}}(pat, \mathtt{this}, X)$
    $* \forall Y \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \pi_{\mathsf{chan}}(Y) = X \Rightarrow$
      $\triangleright act \mapsto \{\mathsf{join}_{\mathsf{call}}(t,P,Q)(\mathtt{this}) * \circledast_{y \in Y} P(y)\}\{\mathsf{join}_{\mathsf{call}}(t,P,Q)(\mathtt{this}) * \circledast_{y \in Y} Q(y)\}$
    $* (consumed = \mathtt{null} \vee (\mathsf{lst}(consumed, \pi_2(Y)) * \circledast_{(c,m) \in Y} \mathsf{pending}(t \cdot \text{“2”}, P, Q, m, c) * \pi_1(Y) = X))$
if (consumed != null) {
    $\{\mathsf{j}_{reg_l} \mapsto t_l * \mathsf{j}_{reg_m} \mapsto t_m * \mathsf{j.lock} \mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}))$
      $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}) * \mathsf{pattern}_{\mathsf{internal}}(pat, \mathtt{this}, X)$
      $* \forall Y \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \pi_{\mathsf{chan}}(Y) = X \Rightarrow$
        $\triangleright act \mapsto \{\mathsf{join}_{\mathsf{call}}(t,P,Q)(\mathtt{this}) * \circledast_{y \in Y} P(y)\}\{\mathsf{join}_{\mathsf{call}}(t,P,Q)(\mathtt{this}) * \circledast_{y \in Y} Q(y)\}$
      $* \mathsf{lst}(consumed, \pi_2(Y)) * \circledast_{(c,m) \in Y} \mathsf{pending}(t_m, P, Q, m, c) * \pi_1(Y) = X\}$
    $\{\mathsf{j}_{reg_l} \mapsto t_l * \mathsf{j}_{reg_m} \mapsto t_m * \mathsf{j.lock} \mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}))$
      $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this})$
      $* \forall Y \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \pi_{\mathsf{chan}}(Y) = X \Rightarrow$
        $\triangleright act \mapsto \{\mathsf{join}_{\mathsf{call}}(t,P,Q)(\mathtt{this}) * \circledast_{y \in Y} P(y)\}\{\mathsf{join}_{\mathsf{call}}(t,P,Q)(\mathtt{this}) * \circledast_{y \in Y} Q(y)\}$
      $* \mathsf{lst}(consumed, \{m \mid (c,m,a) \in Z\}) * \{c \mid (c,m,a) \in Z\} = X$
      $* \ \circledast_{(c,m,a) \in Z}\ P(a,c) * \mathsf{removed}(t_m, P, Q, a, m, c)\}$
  lock.Release();
    $\{\mathsf{join}_{\mathsf{call}}(t, P, Q, \mathtt{this}) * \mathsf{j}_{reg_m} \mapsto t_m$
      $* \ \triangleright act \mapsto \begin{array}{l} (\text{-}).\ \{\mathsf{join}_{\mathsf{call}}(t,P,Q)(\mathtt{this}) * \circledast_{y \in \{(a,c) \mid (c,m,a) \in Z\}} P(y)\} \\ \quad \{\mathsf{join}_{\mathsf{call}}(t,P,Q)(\mathtt{this}) * \circledast_{y \in \{(a,c) \mid (c,m,a) \in Z\}} Q(y)\} \end{array}$
      $* \mathsf{lst}(consumed, \{m \mid (c,m,a) \in Z\}) * \{c \mid (c,m,a) \in Z\} = X$
      $* \ \circledast_{(c,m,a) \in Z}\ P(a,c) * \mathsf{removed}(t_m, P, Q, a, m, c)\}$
  act();
    $\{\mathsf{join}_{\mathsf{call}}(t, P, Q, \mathtt{this}) * \mathsf{j}_{reg_m} \mapsto t_m$
      $* \mathsf{lst}(consumed, \{m \mid (c,m,a) \in Z\})$
      $* \ \circledast_{(c,m,a) \in Z}\ Q(a,c) * \mathsf{removed}(t_m, P, Q, a, m, c)\}$
  lock.Acquire();
    $\{\mathsf{j}_{reg_l} \mapsto t_l * \mathsf{j}_{reg_m} \mapsto t_m * \mathsf{j.lock} \mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}))$
      $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}) * \mathsf{lst}(consumed, \{m \mid (c,m,a) \in Z\})$
      $* \ \circledast_{(c,m,a) \in Z}\ Q(a,c) * \mathsf{removed}(t_m, P, Q, a, m, c)\}$
  consumed.ForEach(Release);
    $\{\mathsf{j}_{reg_l} \mapsto t_l * \mathsf{j}_{reg_m} \mapsto t_m * \mathsf{j.lock} \mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}))$
      $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t,P,Q), P, Q, \mathtt{this}) * \mathsf{lst}(consumed, \{m \mid (c,m,a) \in Z\})\}$
    }
      ${\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathtt{this}))$
         $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathtt{this})}$
  }

  internal void checkChords() {
    LinkedList<Chord> chords;
      ${\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathtt{this}))$
         $* \mathsf{join}_{\mathsf{internal}}(t, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathtt{this})}$
    chords = this.chords;
      ${\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathtt{this}))$
         $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathtt{this})$
         $* \mathsf{lst}_r(chords, Y) * \circledast_{c \in Y} \mathsf{chord}(\mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, c, \mathtt{this})}$
    chords.ForEach(checkChord);
      ${\mathsf{j}_{reg_l} \Mapsto t_l * \mathsf{j}_{reg_m} \Mapsto t_m * \mathsf{j.lock} \Mapsto \mathsf{l} * \mathrm{locked}(t_m, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, \mathtt{this}))$
         $* \mathsf{join}_{\mathsf{internal}}(t_m, \mathsf{join}_{\mathsf{call}}(t, P, Q), P, Q, j)}$
  }

  public void Release(Message msg) {
    msg.status = Status.Released;
  }
}
```

4.6 Auxiliary classes

The above proof outlines assume auxiliary classes Lock, List and Pair satisfying the following specifications.

The Lock class

The Lock class implements a spin-lock, using a compare-and-swap to atomically acquire the lock. The standard separation logic specification of a lock, associates a resource invariant R with each lock, which is transferred to the client upon acquiring the lock, and transferred back upon releasing the lock. Since the resource invariant R might itself assert ownership of resourced shared using CAP, we require that R is stable, that it is independent of the lock region type (picked by the client), and that it is expressible using state-independent protocols. To support the joins' view-shift from $\mathsf{join}_{\mathsf{init\text{-}pat}}(-)$ to $\mathsf{join}_{\mathsf{call}}(-)$ we further introduce a corresponding lock initialization phase, allowing the resource invariant R to be picked using a view-shift. We thus assume the following specification for the Lock library.

$$
\begin{array}{l}
\exists \mathsf{lock} : \text{RType} \times \text{Val} \to \text{Prop}.\ \exists \mathsf{islock}, \mathsf{locked} : \text{RType} \times \text{Prop} \times \text{Val} \to \text{Prop}. \\
\quad \forall \mathsf{t} \in \text{RType}.\ \forall \mathsf{R} : \text{Prop}.\ \mathsf{indep_t}(\mathsf{R}) \land \mathsf{sip}(\mathsf{R}) \land \mathsf{stable}(\mathsf{R}) \ \Rightarrow \\
\qquad \mathsf{new\ Lock()} : (\text{-}).\ \{\text{emp}\}\ \{\mathsf{ret}.\ \exists \mathsf{s} : \text{RType}.\ \mathsf{lock}(\mathsf{s}, \mathsf{ret}) * \mathsf{t} \leq \mathsf{s}\} \\
\qquad \mathsf{Lock.Acquire} : (\text{-}).\ \{\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathtt{this})\}\ \{\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathtt{this}) * \mathsf{R}\} \\
\qquad \mathsf{Lock.Release} : (\text{-}).\ \{\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathtt{this}) * \mathsf{R}\}\ \{\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathtt{this})\} \\
\qquad \forall \mathsf{x} : \text{Val}.\ \mathsf{R} * \mathsf{lock}(\mathsf{t}, \mathsf{x}) \sqsubseteq \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}) \\
\qquad \forall \mathsf{x} : \text{Val}.\ \mathsf{stable}(\mathsf{lock}(\mathsf{t}, \mathsf{x})) \land \mathsf{stable}(\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x})) \land \mathsf{stable}(\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathsf{x})) \\
\qquad \textit{\textbf{valid}}\ (\forall \mathsf{x} : \text{Val}.\ \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}) \Leftrightarrow \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}) * \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}))
\end{array}
$$

Here the $\mathsf{lock}(\mathsf{t}, \mathsf{x})$ predicate asserts that x refers to an uninitialized and unlocked lock. The lock is initialized by picking a resource invariant R using the view shift:

$$\mathsf{R} * \mathsf{lock}(\mathsf{t}, \mathsf{x}) \sqsubseteq \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x})$$

The $\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x})$ predicate asserts that x refers to an initialized lock with resource invariant R. The $\mathsf{isLock}(-)$ predicate is freely duplicable, allowing multiple clients to use the same lock. Lastly, the $\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathsf{x})$ predicate asserts that x refers to an initialized and locked lock with resource invariant R.

With the exception of the delayed choice of resource invariant R, this specification matches the Lock specification from the Example section in [3]. The proof is thus exactly the same, except that the lock region is created in the view-shift when the resource invariant is picked, instead of in the constructor. We will thus not repeat the proof. We refer the interested reader to the Example section in [3].

List class

In the case of the List class, which implements a linked list library, we assume a slightly weaker specification than usual. In particular, for our purposes, we do not care about the ordering of elements, only how many times each element appears in a given list. We thus represent the abstract state of a list as a multiset of elements. Since a lot of the lists maintained by join instances are read-only once the join instance transitions to the call phase, we index the list representation predicate with a fractional permission, to permit read-only sharing.

$$
\begin{array}{l}
\exists \mathsf{lst} : \mathrm{Val} \times \mathrm{Perm} \times \mathcal{P}_m(\mathrm{Val}) \to \mathrm{Prop}.\\
\quad \forall \mathsf{I} : \mathcal{P}_m(\mathrm{Val}) \to \mathrm{Prop}.\ \forall \pi : \mathrm{Perm}.\ \forall \mathsf{X} : \mathcal{P}_m(\mathrm{Val}).\\
\qquad \mathsf{new\ List} : (\text{-}).\ \{\mathrm{emp}\}\ \{\mathsf{ret}.\ \mathsf{lst}(\mathsf{ret}, 1, \mathsf{X})\}\\
\qquad \mathsf{List.Push} : (\mathsf{y}).\ \{\mathsf{lst}(\mathtt{this}, 1, \mathsf{X})\}\ \{\mathsf{lst}(\mathtt{this}, 1, \{\mathsf{y}\} \cup \mathsf{X})\}\\
\qquad \mathsf{List.Pop} : (\text{-}).\ \{\mathsf{lst}(\mathtt{this}, 1, \mathsf{X}) * \mathsf{X} \neq \emptyset\}\\
\qquad\qquad\qquad \{\mathsf{ret}.\ \exists \mathsf{Y} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{lst}(\mathtt{this}, 1, \mathsf{Y}) * \mathsf{X} = \{\mathsf{ret}\} \cup \mathsf{Y}\}\\
\qquad \mathsf{List.Count} : (\text{-}).\ \{\mathsf{lst}(\mathtt{this}, \pi, \mathsf{X})\}\ \{\mathsf{ret}.\ \mathsf{lst}(\mathtt{this}, \pi, \mathsf{X}) * \mathsf{ret} = |\mathsf{X}|\}\\
\qquad \mathsf{List.ForEach} : (\mathsf{f}).\ \left\{ \mathsf{lst}(\mathtt{this}, \pi, \mathsf{X}) * \mathsf{I}(\emptyset) * \forall \mathsf{Y} \in \mathcal{P}_m(\mathrm{Val}).\ \mathsf{f} \mapsto \begin{array}{r} (\mathsf{x}).\ \{\mathsf{I}(\mathsf{Y}) * \mathsf{x} \in \mathsf{X}\} \\ \{\mathsf{I}(\{\mathsf{x}\} \cup \mathsf{Y})\} \end{array} \right\}\\
\qquad\qquad\qquad \{\mathsf{lst}(\mathtt{this}, \pi, \mathsf{X}) * \mathsf{I}(\mathsf{X})\}\\
\qquad \mathsf{List.Map} : (\mathsf{f}).\ \left\{ \mathsf{lst}(\mathtt{this}, \pi, \mathsf{X}) * \mathsf{I}(\emptyset) * \forall \mathsf{Y} \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \mathsf{f} \mapsto \begin{array}{r} (\mathsf{x}).\ \{\mathsf{I}(\mathsf{Y}) * \mathsf{x} \in \mathsf{X}\} \\ \{\mathsf{I}(\{(\mathsf{x}, \mathsf{ret})\} \cup \mathsf{Y})\} \end{array} \right\}\\
\qquad\qquad\qquad \{\mathsf{ret}.\ \mathsf{lst}(\mathtt{this}, \pi, \mathsf{X}) * \exists \mathsf{Y} : \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \mathsf{lst}(\mathsf{ret}, 1, \pi_2(\mathsf{Y})) * \mathsf{I}(\mathsf{Y}) * \pi_1(\mathsf{Y}) = \mathsf{X}\}\\
\qquad \textbf{\textit{valid}}\ (\forall \mathsf{x} : \mathrm{Val}.\ \forall p, q : \mathrm{Perm}.\\
\qquad\qquad (\exists r : \mathrm{Perm}.\ p + q = r * \mathsf{lst}(\mathsf{x}, r, \mathsf{X})) \Leftrightarrow \mathsf{lst}(\mathsf{x}, p, \mathsf{X}) * \mathsf{lst}(\mathsf{x}, q, \mathsf{X}))\\
\qquad \mathsf{stable}(\mathsf{lst}(\mathsf{x}, \pi, \mathsf{X}))
\end{array}
$$

The list representation predicate, $\mathsf{lst}(\mathsf{x}, \pi, \mathsf{X})$, thus asserts that x refers to a linked list containing elements X, with fractional permission π. Modifying a list requires full ownership, whereas querying (Count) and iterating (ForEach, Map) only requires read-only permission. In the proof outlines we use $\mathsf{lst}_r(\mathsf{x}, \mathsf{X})$ as shorthand for $\exists \pi : \mathrm{Perm}.\ \mathsf{lst}(\mathsf{x}, \pi, \mathsf{X})$ and $\mathsf{lst}(\mathsf{x}, \mathsf{X})$ as shorthand for $\mathsf{lst}(\mathsf{x}, 1, \mathsf{X})$. We thus have that

$$\mathsf{lst}_r(\mathsf{x}, \mathsf{X}) \Leftrightarrow \mathsf{lst}_r(\mathsf{x}, \mathsf{X}) * \mathsf{lst}_r(\mathsf{x}, \mathsf{X})$$

We omit the completely standard higher-order separation logic proof that a singly-linked list implementation satisfies the above specification.

Pair class

Lastly, the Pair class implements pairing.

$$
\begin{array}{l}
\exists \mathsf{pair} : \mathrm{Val} \times \mathrm{Val} \times \mathrm{Val} \to \mathrm{Prop}.\\
\quad \forall \mathsf{x}, \mathsf{y}, \mathsf{z} : \mathrm{Val}.\\
\qquad \mathsf{new\ Pair} : (\mathsf{x},\ \mathsf{y}).\ \{\mathrm{emp}\}\ \{\mathsf{ret}.\ \mathsf{pair}(\mathsf{ret}, \mathsf{x}, \mathsf{y})\}\\
\qquad \mathsf{Pair.Fst} : (\text{-}).\ \{\mathsf{pair}(\mathtt{this}, \mathsf{x}, \mathsf{y})\}\ \{\mathsf{ret}.\ \mathsf{pair}(\mathtt{this}, \mathsf{x}, \mathsf{y}) * \mathsf{ret} = \mathsf{x}\}\\
\qquad \mathsf{Pair.Snd} : (\text{-}).\ \{\mathsf{pair}(\mathtt{this}, \mathsf{x}, \mathsf{y})\}\ \{\mathsf{ret}.\ \mathsf{pair}(\mathtt{this}, \mathsf{x}, \mathsf{y}) * \mathsf{ret} = \mathsf{y}\}\\
\qquad \mathsf{stable}(\mathsf{pair}(\mathsf{z}, \mathsf{x}, \mathsf{y}))
\end{array}
$$

The pair representation predicate, $\mathsf{pair}(\mathsf{x},\mathsf{y},\mathsf{z})$, asserts that x refers to a pair consisting of y and z.

We use $\mathsf{lst}_{pair}(\mathsf{x},\mathsf{X})$ as shorthand for,

$$\begin{aligned}\mathsf{lst}_{pair}(\mathsf{x},\mathsf{X}) \overset{\text{def}}{=} {} & \exists \mathsf{Y} : \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val} \times \mathrm{Val}). \\ & \mathsf{lst}(\mathsf{x}, \pi_1(\mathsf{Y})) * \circledast_{(\mathsf{a},\mathsf{y},\mathsf{z}) \in \mathsf{Y}} \mathsf{pair}(\mathsf{a},\mathsf{y},\mathsf{z}) * \{(\mathsf{y},\mathsf{z}) \mid (\mathsf{a},\mathsf{y},\mathsf{z}) \in \mathsf{Y}\} = \mathsf{X}\end{aligned}$$

where $\mathsf{X} \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val})$, to specify lists of pairs.

References

[1] K. Svendsen, L. Birkedal, and M. Parkinson. Joined-up Thinking: A Specification of the Joins Library in Higher-Order Separation Logic. Submitted for publication.

[2] K. Svendsen, L. Birkedal, and M. Parkinson. Verifying Generics and Delegates. In *Proceedings of ECOOP*, pages 175–199, 2010.

[3] K. Svendsen, L. Birkedal, and M. Parkinson. Higher-order concurrent abstract predicates. Technical report, IT University of Copenhagen, 2012.