Verifying Custom Synchronization Constructs Using Higher-Order Separation Logic

MIKE DODDS, University of York, UK
SURESH JAGANNATHAN, Purdue University, Indiana
MATTHEW J. PARKINSON, Microsoft Research, UK
KASPER SVENDSEN and LARS BIRKEDAL, Aarhus University, Denmark

Synchronization constructs lie at the heart of any reliable concurrent program. Many such constructs are standard (e.g., locks, queues, stacks, and hash-tables). However, many concurrent applications require custom synchronization constructs with special-purpose behavior. These constructs present a significant challenge for verification. Like standard constructs, they rely on subtle racy behavior, but unlike standard constructs, they may not have well-understood abstract interfaces. As they are custom built, such constructs are also far more likely to be unreliable.

This article examines the formal specification and verification of custom synchronization constructs. Our target is a library of channels used in automated parallelization to enforce sequential behavior between program statements. Our high-level specification captures the conditions necessary for correct execution; these conditions reflect program dependencies necessary to ensure sequential behavior. We connect the high-level specification with the low-level library implementation to prove that a client's requirements are satisfied. Significantly, we can reason about program and library correctness without breaking abstraction boundaries.

To achieve this, we use a program logic called iCAP (impredicative Concurrent Abstract Predicates) based on separation logic. iCAP supports both high-level abstraction and low-level reasoning about races. We use this to show that our high-level channel specification abstracts three different, increasingly complex low-level implementations of the library. iCAP's support for higher-order reasoning lets us prove that sequential dependencies are respected, while iCAP's next-generation semantic model lets us avoid ugly problems with cyclic dependencies.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*Correctness proofs*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Separation logic, concurrent abstract predicates, concurrency

ACM Reference Format:
Mike Dodds, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birkedal. 2016. Verifying custom synchronization constructs using higher-order separation logic. ACM Trans. Program. Lang. Syst. 38, 2, Article 4 (January 2016), 72 pages.
DOI: http://dx.doi.org/10.1145/2818638

Dodds was supported in part by EPSRC grants EP/H010815/1, EP/H005633/1, and EP/F036345. Svendsen was supported in part by the Danish Council for Independent Research project DFF – 4181-00273. Birkedal was supported in part by the ModuRes Sapere Aude Advanced Grant from the Danish Council for Independent Research for the Natural Sciences (FNU).
Authors' addresses: M. Dodds, Dept. of Comp. Science, University of York, York, UK; email: mike.dodds@york.ac.uk; S. Jagannathan, Dept. of Comp. Science, Purdue University, West Lafayette, IN, USA; email: suresh@cs.purdue.edu; M. J. Parkinson, Microsoft Research Cambridge, Cambridge, UK; email: mattpark@microsoft.com; K. Svendsen and L. Birkedal, Dept. of Comp. Science, Aarhus University, Aarhus, Denmark; emails: {ksvendsen, birkedal}@cs.au.dk.

ACM 0164-0925/2016/01-ART4 $15.00
DOI: http://dx.doi.org/10.1145/2818638

1. INTRODUCTION

Concurrent programming is challenging because it requires programmers to parcel work into useful units and weave suitable concurrency control to coordinate access to shared data. Coordination is generally performed by *synchronization constructs*. In order for programmers to build and reason about concurrent programs, it is essential that these synchronization constructs hide implementation details behind specifications, allowing clients to reason about correctness in terms of abstract, rather than concrete, behavior.

For standard synchronization constructs (e.g., locks, queues, stacks), abstract specifications are well studied. However, many concurrent applications depend on nonstandard, custom synchronization constructs. These may have poorly defined abstract interfaces while at the same time depending on complex racy implementation behavior. Verifying these constructs requires a technique that can build up strong abstractions, reason about the logical distribution of data between threads, and at the same time deal with the intricacies of low-level concurrency. This is our objective in this article.

Our target is to verify one such custom concurrency construct: barriers used for automated parallelization. In *deterministic* parallelization, code regions in a sequential program are executed concurrently. While the parallelized program is internally nondeterministic, control constructs are used to ensure that it exhibits the same deterministic observable behavior as its sequential counterpart. Automatic parallelization of this kind has been well studied for loop-intensive numerical computations. However, it is also possible to extract parallelism from irregularly structured sequential programs [Bocchino et al. 2009; Rinard and Lam 1992; Welc et al. 2005].

One way to implement deterministic parallelism is through compiler-injected barriers [Navabi et al. 2008]. We can think of these barriers as enforcing the original sequential program dependencies on shared resources. A resource could be any program variable, data structure, memory region, lock, and so forth for which resource ownership guarantees are essential in order to enforce deterministic semantics. While the intuition behind using such barriers is quite simple, there are many possible implementations, and verifying that an implementation enforces the correct behavior is challenging for several reasons:

—*Custom data structures*. To enable the maximum level of parallelism, barriers are implemented using custom data structures that collect and summarize signals.
—*Nonlocal signaling*. The patterns of signaling in a barrier implementation are highly nonlocal. To access a resource, a barrier must wait until all logically preceding threads have indicated that it is safe to do so. However, threads are locally unaware of this context.
—*Out-of-order signaling*. The parallelization process will strive to identify the earliest point in a thread's execution path from where a resource is no longer required. In some cases, this means threads can release resources without ever acquiring them, so that subsequent signaling of this resource by its predecessor can bypass it altogether.
—*Shared read access*. Barriers may treat reads and writes differently to ensure preservation of sequential behavior. Although many reads can be performed concurrently, they must be sequentialized with respect to writes. Moreover, reads must be sequentialized with respect to other reads if there is an intervening write.
—*Higher-order specifications*. Abstractly, channels can be used to control access to any kind of resource for which ownership is important. Thus, the natural specification is higher order: the resource is a parameter to the specification. Channels may control access to other channels or even later stages of the same channel.

In this article, we show how to tackle these verification challenges. We use *impredicative Concurrent Abstract Predicates* (iCAP), a recent program logic that enables abstract,

higher-order reasoning about concurrent libraries [Svendsen and Birkedal 2014a]. This allows us to reason about both high-level properties and low-level implementation details. iCAP supports fine-grained reasoning about concurrent behavior, meaning that each thread can be permitted exactly the behavior it needs. Furthermore, reasoning in iCAP is *local*, meaning even shared state can be encapsulated and abstracted.

The result of our work is a verified high-level specification for barriers, independent of their low-level implementation. Using iCAP, we have proved that three very different low-level implementations satisfy the same high-level specification. In the presence of runtime thread creation and dynamic (heap-allocated) data, our specification must be both generic and dynamic, since it must be possible to construct barriers at runtime that control access to arbitrary resources. To allow this, we use iCAP's higher-order quantification mechanism to encode complex patterns of resource redistribution. It is worth emphasizing that the barriers we look at were not designed with verification in mind; we have developed the specification to suit the application, not vice versa.

In this article, we focus on just the verification problem for barriers, but in a companion article, we define a parallelizing program analysis that injects appropriate barriers [Botinčan et al. 2013]. Our work here contributes to the eventual goal of a fully specified and verified system for deterministic parallelism. More generally, access to concurrent data is often controlled by custom synchronization constructs, and our work in this article demonstrates how to reason soundly about such bespoke concurrency constructs.

Contribution

This article substantially revises and expands our conference paper [Dodds et al. 2011]. Our main contributions relative to Dodds et al. [2011] are as follows:

—A revised higher-order abstract specification for the custom synchronization barriers used in deterministic parallelism. Our new specification is cleaner and more general.
—New proofs of this specification for simplified, out-of-order, and summarizing barrier implementations, written using the iCAP proof system [Svendsen and Birkedal 2014a]. The first two implementations were proved correct in Dodds et al. [2011], while the proof of the summarizing version is entirely novel.
—A self-contained presentation of the formal iCAP proof system and a tutorial on how to use iCAP to verify concurrent software. The formal iCAP proof system is not presented nor explained in the iCAP conference paper [Svendsen and Birkedal 2014a].
—An encoding in iCAP of constructs we call *saved propositions*. These serve some of the functions of auxiliary variables capable of storing propositions and allow us to reason about resource transfer and splitting without altering the proof system.
—A new application of Wickerson et al.'s *explicit stabilization* [Wickerson et al. 2010] to reason about the stability of complex separation logic assertions.
—A verified application of our barriers: a tree-based key-value store. This demonstrates a nontrivial dynamic pattern of signaling, as signals are activated at branches only if they are visited by a thread. It also demonstrates the need for higher-order reasoning: in the logical definition of a tree, the induction passes through a higher-order parameter.

This article also corrects a subtle logical problem in Dodds et al. [2011], discovered by Svendsen a year after publication. This arose as a result of a circularity in the model of higher-order propositions, which rendered several important reasoning steps unsound. The details are discussed in Section 8, but we emphasize that this problem could not have been solved by the higher-order separation logics available in 2011. The

development of iCAP was in part motivated by resolving this kind of problem; in this article, we show that iCAP can be used to verify tricky practical algorithms.

Article Structure

Section 2 discusses related work. Section 3 introduces the behavior of barriers informally and defines our abstract specification. It also discusses an example application, a tree-based key-value store. Section 4 gives a very simple barrier implementation and shows how it can be verified with respect to the core of the specification. This section also serves as a tutorial introduction to iCAP, the logic we use for verification. Section 5 discusses how the specification can be extended to cover the splitting of resources offered by a channel. Section 6 gives a more complicated implementation where channels are arranged into chains and verifies the full abstract specification. Section 7 gives an optimized implementation where signals between channels are summarized and verifies it. Section 8 explores the problems with our conference paper [Dodds et al. 2011] and how we have addressed them. Some of the subsidiary lemmas are proved in full in the appendices.

2. RELATED WORK

iCAP is a new logic for verifying complicated concurrent algorithms [Svendsen and Birkedal 2014a, 2014b]. Although we have focussed in this article on barriers used for deterministic parallelism [Welc et al. 2005; Berger et al. 2010; Bocchino et al. 2009; Navabi et al. 2008], our intention is to illustrate how iCAP can be used to specify and verify novel concurrency constructs in general.

Prior to 2011, most work on concurrent separation logic considered concurrency constructs as primitive in the logic. This begins with O'Hearn's work on concurrent separation logic [O'Hearn 2007], which takes statically allocated locks as a primitive. CSL has been extended to deal with dynamically allocated locks [Gotsman et al. 2007; Hobor et al. 2008; Jacobs and Piessens 2009] and re-entrant locks [Haack et al. 2008]. Others have extended separation logic or similar logics with primitive channels [Hoare and O'Hearn 2008; Bell et al. 2009; Villard et al. 2010; Leino et al. 2010] and event-driven programs [Krishnaswami et al. 2010]. There are important disadvantages to handling each distinct concurrency construct with a new custom logic:

—Developing a custom logic might be acceptable for standard synchronization constructs such as locks, but it is infeasible for every domain-specific construct.
—Embedding each construct as primitive in the logic provides no means for verifying implementations of the construct.
—Each custom logic handles one fixed kind of construct, with no means of verifying programs that use multiple concurrency constructs.

iCAP solves all three problems. New synchronization constructs can be introduced as libraries and given abstract specifications that abstract over the internal data representation and state through abstract predicates. Implementations can be verified against these abstract specifications by giving these predicates concrete definitions (our article does precisely this for barriers). As new constructs can be freely introduced as libraries, clients are free to combine multiple concurrency constructs as needed. Furthermore, using iCAP's higher-order quantification, specifications can abstract over arbitrary predicates, including those defined by other concurrency constructs. This allows us to support separate reasoning about each construct while still allowing them to interact cleanly. For instance, abstract lock predicates defined by a lock library can freely be transferred through our channels. (See Dinsdale-Young et al. [2010] for an example of such a lock specification.)

iCAP descends from our earlier Concurrent Abstract Predicates (CAP) logic [Dinsdale-Young et al. 2010]. CAP combined the explicit treatment of concurrent interference from rely-guarantee [Jones 1983; Feng et al. 2007; Vafeiadis 2007] and abstraction through abstract predicates [Parkinson and Bierman 2005], with a rich system of protocols based on capabilities [Dodds et al. 2009]. iCAP extends CAP with higher-order propositions and an improved system of concurrent protocols [Svendsen and Birkedal 2014a]. iCAP's step-indexed semantics is supported by an underlying theory called the *topos of trees* [Birkedal et al. 2012].

Recent years have seen a great deal of work on concurrent logics, many of which take inspiration from CAP. Complex concurrency constructs have been verified before in CAP-like logics, for example, concurrent B-trees in da Rocha Pinto et al. [2011]. The proof in da Rocha Pinto et al. [2011] is mostly concerned with complex manipulations of the B-tree structure. In comparison, our barrier implementations are relatively simple, and a large proportion of our proof concerns changes in *ownership* to support our higher-order specification. The verification of the Joins library in Svendsen et al. [2013] has similarities to our work. Both articles deal with barriers using higher-order separation logic. However, the implementations and specifications are substantially different; for example, our implementation permits chains of channels, and our specification deals with resource splitting. The authors of Svendsen et al. [2013] are also coauthors on this article, and iCAP was largely developed as a improvement on the HOCAP logic introduced there.

Two significant alternative logics to iCAP are CaReSL [Turon et al. 2013] and TaDA [da Rocha Pinto et al. 2014]. Like iCAP, both extend CAP with richer protocols. Unlike iCAP, both are primarily aimed at proving atomicity/linearizability and confine themselves to second-order logic only. This makes them less suitable for our purposes. It is plausible that many of the proofs in this article could be recast into these logics. However, we would have to constrain the higher-order parameters from our specification with some kind of explicit stratification. We would expect proofs to be significantly more complex as a result of the bookkeeping needed to track this stratification.

Another logic aimed at fine-grained concurrent data structures is FSCL [Nanevski et al. 2014]. This is defined through a shallow embedding into Coq's Calculus of Inductive Constructions, and thus supports definition of higher-order specifications. However, FCSL's reasoning principles for higher-order specifications are weaker than those of iCAP. In particular, FCSL lacks support for impredicative protocols and assertions in the heap, both of which result in circularities that would manifest as universe inconsistency errors in Coq. Thus, while FCSL can define our proposed barrier specification, we believe that FCSL would be unable to verify an implementation against it. Impredicative protocols and assertions in the heap are fundamental to the generic higher-order specifications that we require. For example, in Section 3.6, we apply our barrier specification in a key-value store: here the inductive definition of a tree passes through a higher-order parameter, representing the fact that subtrees must be acquired through channel communication.

3. A SPECIFICATION FOR DETERMINISTIC PARALLELISM

In this section, we describe the intuitive behavior of a library of barriers for enforcing deterministic parallelism that forms our case study. Based on this, we define a high-level specification for barriers—the full abstract specification is given in Section 3.4. These barriers are based on the ones used for deterministic parallelism in Navabi et al. [2008]. In Botinčan et al. [2013], we use our abstract specification in a proof-based parallelizing analysis that is guaranteed to preserve sequential behavior.

We assume that code sections believed to be amenable for parallelization have been identified, and the program split accordingly into threads. We assume a total logical

ordering on threads, such that executing the threads serially in the logical order gives the same result as the original (unparallelized) program.

Barriers are associated with resources (e.g., program variables, data structures, etc.) that are to be shared between concurrently executing program segments. There are two sorts of barriers. A *signal* barrier notifies logically later threads that the current thread will no longer use the resource. A *wait* barrier blocks until all logically prior threads have signaled that they will no longer use the resource (i.e., have issued signals).

We assume barriers are injected by an analysis that ensures that all salient data dependencies in the sequential program are respected. For example, suppose we run two instances of the function `f` in sequence (here `sleep(rand())` waits for an unknown period of time).

```
void f(int *x, int *y, int v) {
   if(*x < 10) {
     *y = *y + v;
     *x = *x + v;
     sleep(rand());
   } else {
     sleep(rand());
   }
}
```

```
*x = 0;
*y = 0;

f(x,y,5);
f(x,y,11);
```

When this program terminates, location `x` and `y` will both hold 16.

The second call to `f` will wait for the first call to finish its arbitrarily long `sleep`, even though the first call will do nothing more once it wakes. An analysis could parallelize this function by passing control between the two earlier. The parallelized functions `f1` and `f2` are given next. We run both concurrently, but require that `f1` passes control of `x` and `y` to `f2` *before* sleeping, allowing `f2` to continue executing.

FUNCTION DEFINITIONS:

```
f1(x,y,v,i) {
   if(*x < 10) {
     *y = *y + v;
     *x = *x + v;
     signal(i);
     sleep(rand());
   } else {
     signal(i);
     sleep(rand());
   }
}
```

```
f2(x,y,v,i) {
   wait(i);
   if(*x < 10) {
     *y = *y + v;
     *x = *x + v;
     sleep(rand());
   } else {
     sleep(rand());
   }
}
```

PROGRAM BODY:

```
*x = 0; *y = 0;
chan *i = newchan();

f1(x,y,5,i) || f2(x,y,11,i);
```

The barriers in `f1` and `f2` ensure that the two threads wait exactly until the resources they require can be safely modified, without violating sequential program dependencies. The correct ordering is enforced by barriers that communicate through a channel; in the example, `newchan` creates the channel `i`. Assuming the barriers are correctly implemented, the resulting behavior is equivalent to the original sequential program, with `x` and `y` both holding 16.

3.1. Verifying a Client Program

How can we verify that our parallelized program based on `f1` and `f2` satisfies the same specification as the original sequential program? Typically (e.g., in Navabi et al. [2008]), one would incorporate signaling machinery as part of a parallelization program analysis. Clients would then reason about program behavior using the operational semantics of the barrier implementation. Validating the correctness of parallelization with respect to the sequential program semantics would therefore require a detailed knowledge of the barrier implementation. Any changes to the implementation could entail reproving the correctness of the parallelization analysis.

In contrast, we reason about program behavior in terms of abstract specifications for `signal`, `wait`, and `newchan`. Such an approach has the advantages that (1) implementors can modify their underlying implementation and be sure that relevant program properties are preserved by the implementation, and (2) client proofs (in this case, proofs involving compiler correctness) can be completed without knowledge of the underlying implementation.

We will reason about `f1` and `f2` using separation logic, which lets us precisely control the allocation of resources to threads over time. Assertions in separation logic denote resources: heap cells and data structures, but also abstract resources like channel ends. For example, we write the following assertion to denote that `x` points to value v and `y` to value v': $\mathtt{x} \mapsto v * \mathtt{y} \mapsto v'$. The *separating conjunction* $*$ asserts that `x` and `y` are distinct. As well as capturing information about the current state of resources, assertions in separation logic also capture *ownership*. Thus, the assertion $\mathtt{x} \mapsto v * \mathtt{y} \mapsto v'$ in an invariant for a thread implicitly states that the thread has exclusive access to `x` and `y`.

To reason about the parallel composition of threads, we can use the Par rule of concurrent separation logic [O'Hearn 2007]:

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\ \textsc{Par}$$

To verify `f1` and `f2`, we must encode the fact that `f1` gives up access to `x` and `y` by calling `signal(i)`, while `f2` retrieves access to them by calling `wait(i)`. We encode these two facts with two predicates, recv and send, corresponding to the *promised resource*, the resource that can be acquired from logically earlier threads, and the *required resource*, the resource that must be supplied to logically later threads. We read these as follows:

- $\mathsf{recv}(\mathtt{i}, P)$ – By calling `wait` on `i`, the thread will acquire a resource satisfying the assertion P.
- $\mathsf{send}(\mathtt{i}, P)$ – By calling `signal` on `i` when holding a resource satisfying P, the thread will lose the resource P.

These predicates are *abstract*; each instantiation of the library will define them differently. The client only depends on an abstract specification that captures their intuitive meaning:

$$\begin{array}{rcl} \{\mathsf{emp}\} & \mathtt{i = newchan()} & \{\mathsf{send}(\mathtt{i}, P) * \mathsf{recv}(\mathtt{i}, P)\} \\ \{\mathsf{send}(\mathtt{i}, P) * \mathsf{stable}(P) * P\} & \mathtt{signal(i)} & \{\mathsf{emp}\} \\ \{\mathsf{recv}(\mathtt{i}, P) * \mathsf{stable}(P)\} & \mathtt{wait(i)} & \{P\} \end{array}$$

The assigned variable `i` stands for `newchan`'s return value, that is, the address of the new channel. We also use this notation in the specification of `extend`, later.

This specification of `newchan` is implicitly universally quantified for all assertions P, meaning that we can construct a channel for *any* iCAP assertion. The same is true for other operation specifications: unless otherwise stated, predicates are universally quantified and thus can be chosen freely.

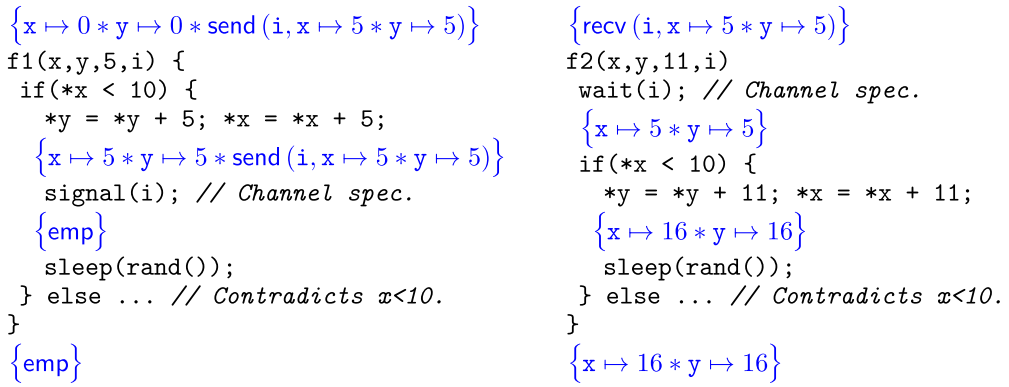


Fig. 1. Proofs for `f1` (left) and `f2` (right).

In general, the universally quantified P can be instantiated with assertions about shared resources. In this case, we need to establish that these assertions are *stable*, that is, invariant under changes potentially performed by other threads. This is expressed by the stability assertion, $\mathsf{stable}(P)$, in the preconditions of `signal` and `wait`. Stability will be explained in Section 4.1. For the moment, note that if P is a thread-local assertion, such as $x \mapsto v$, then P is trivially stable, as these assertions assert exclusive ownership of the underlying resource.

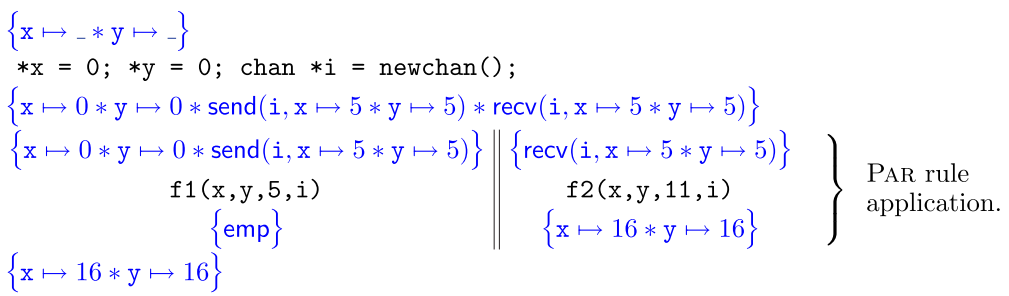
Note that we do not require stability in the specification of `newchan`; that is, the proposition P in $\{\mathsf{send}(\mathtt{i}, P) * \mathsf{recv}(\mathtt{i}, P)\}$ need not be invariant. This gives us more freedom, because P can be dynamically modified before a corresponding resource is supplied; see later sections on splitting and renunciation. We only require that when a resource is eventually supplied or received, it is stable.

New recv and send predicates can be constructed at runtime using `newchan`, meaning we can construct an arbitrarily large number of channels for use in the program. Given these two predicates, we can give the following specifications for `f1` and `f2`. (Here we specialize to the particular parameter values of 5/11; it would be easy to generalize.)

$$\begin{array}{lll} \{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathsf{send}(\mathtt{i},\ \mathtt{x} \mapsto 5 * \mathtt{y} \mapsto 5)\} & \mathtt{f1}(\mathtt{x}, \mathtt{y}, 5, \mathtt{i}) & \{\mathsf{emp}\} \\ \{\mathsf{recv}(\mathtt{i},\ \mathtt{x} \mapsto 5 * \mathtt{y} \mapsto 5)\} & \mathtt{f2}(\mathtt{x}, \mathtt{y}, 11, \mathtt{i}) & \{\mathtt{x} \mapsto 16 * \mathtt{y} \mapsto 16\} \end{array}$$

The send predicate in the specification for `f1` says that the thread must supply the resources `x` and `y` such that they both contain the value 5. The specification for `f2` says that the thread can receive `x` and `y` containing the value 5. Figure 1 gives sketch proofs for these two specifications.

Given this specification, the proof for the main program goes through as follows:



This proof establishes that the parallelized version of the program satisfies the same specification as the sequential original.

3.2. Splitting Waiters

It is often useful for several threads to receive resources via the same channel. This kind of sharing is safe as long as the promised resources are split disjointly. It would be unsafe for two threads to both gain access to `x` at the same time, but it is safe for one thread to access `x` while another accesses `y`. Consider the following three threads:

```
*x = *y + 1;  ||  wait(i);  ||  wait(i);
signal(i)     ||  z = *x    ||  *y = 4
```

The first thread signals on `i` to indicate that it has finished with both `x` and `y`. The other two threads both wait on this signal, and each uses a different aspect of the promised resource.

To support splitting, we add a property to the specification allowing threads to divide promised resources:

$$\{\mathsf{recv}(a, P * Q) * \mathsf{stable}(P * Q)\} \quad \langle \texttt{skip} \rangle \quad \{\mathsf{recv}(a, P) * \mathsf{recv}(a, Q)\}$$

This axiom states that when a thread has been promised a resource that consists of two parts, access can be split between two threads, potentially before the resource is available. This is achieved by splitting a single promise for a resource consisting of two disjoint parts into two promises, one for each part.

Note that the splitting property is *not* a logical entailment—applying it requires an operational step, `skip`. This is because the property manipulates a shared higher-order resource: $\mathsf{recv}(a, P)$. To avoid problems with self-reference, iCAP requires that such manipulations correspond with operational steps—this anomaly is discussed when we introduce iCAP in Section 4.1. Thus, we have to assume that every application of the splitting specification is associated with a `skip`. We discuss whether this assumption is justified in Section 4.1.

3.3. Chains and Renunciation

To allow many threads to access related resources in sequence, we can construct a *chain* of channels. A `wait` barrier called on a channel waits for `signal` barriers on *all* preceding channels. We use the ordering in a chain to model the logical ordering between a sequence of parallelized threads. A chain initially consists of a singleton channel constructed using `newchan`. We introduce an operation `extend`, which takes as its argument an existing channel and creates a new channel immediately preceding it in the chain.

Connecting channels into chains creates a new opportunity for parallelism: the ability to *renounce* access to a resource without acquiring it first. In the simple specification given earlier, a thread can only call `signal` if it has acquired the required resource from its predecessors. However, this is often unnecessary. For example, consider the following three threads:

```
                  ||  if (b != 0) {  ||
                  ||    wait(iEnd);  ||
*x = *x + a;      ||    *x = *x + b; ||  wait(jEnd);
signal(iStart)    ||  }              ||  r = *x
                  ||  signal(jStart); ||
```

Here we give the start and end points of channels distinct names: `iStart` and `iEnd` are the start and end points of channel `i` (`j`, respectively). The channels `i` and `j` are

arranged in a chain. The second thread waits on `iEnd` only if it needs to access `x`. Otherwise, it signals immediately, even if the first thread has not signaled. Without renunciation, the thread would have to insert a wait confirming that the first thread had signaled.

Chains. To support chains in our specification, we introduce an order predicate "$\_ \prec \_$", which represents the order between links in the chain. $x \prec y$ asserts that the channel x is earlier in the chain than channel y. We use two axioms about the ordering of channels:

$$x \prec y \Rightarrow x \prec y * x \prec y \qquad (\textit{duplication})$$
$$x \prec y * y \prec z \Rightarrow x \prec z \qquad (\textit{transitivity})$$

The precondition for the three-thread example earlier would therefore be as follows. The predicate $\texttt{iEnd} \prec \texttt{jStart}$ represents the relationship between the two channels `i`/`j`:

$$\left\{\begin{array}{l}\mathsf{send}(\texttt{iStart}, x \mapsto \_) * \mathsf{recv}(\texttt{iEnd}, x \mapsto \_) * \\ \mathsf{send}(\texttt{jStart}, x \mapsto \_) * \mathsf{recv}(\texttt{jEnd}, x \mapsto \_) * \texttt{iEnd} \prec \texttt{jStart}\end{array}\right\}$$

The abstract specification of `extend` takes a send predicate and a set of order predicates about earlier channels E, and a set of order predicates about later channels L. The function returns a pair of channels `(a,b)` that are later than all the channels before `x` and before all the channels after `x`, and `a` is before `b` in the chain. Note this means a single channel's start and end point may have different identifiers. It also creates recv, send, and order predicates representing the new channel.

$$\left\{\begin{array}{l}\mathsf{send}(\mathtt{x}, P) * \\ \circledast_{e \in E}\, e \prec \mathtt{x} * \circledast_{l \in L}\, \mathtt{x} \prec l\end{array}\right\} \quad (\mathtt{a}, \mathtt{b}) = \texttt{extend(x)} \quad \left\{\begin{array}{l}\mathsf{send}(\mathtt{a}, Q) * \mathsf{recv}(\mathtt{a}, Q) * \mathsf{send}(\mathtt{b}, P) \\ * \mathtt{a} \prec \mathtt{b} * \circledast_{e \in E}\, e \prec \mathtt{a} * \circledast_{l \in L}\, \mathtt{b} \prec l\end{array}\right\}$$

Using this, we can construct the precondition for the three-thread example as follows:

```
{emp}
jEnd = newchan();
{send(jEnd, x ↦ _) * recv(jEnd, x ↦ _)}
(iEnd, jStart) = extend(jEnd);
iStart = iEnd;
{send(iStart, x ↦ _) * recv(iEnd, x ↦ _) *
 send(jStart, x ↦ _) * recv(jEnd, x ↦ _) * iEnd ≺ jStart}
```

Renunciation. To support renunciation, we add an axiom allowing threads to satisfy required resources using earlier promised resources:

$$\{\mathsf{recv}(x, P) * \mathsf{send}(y, P * Q) * x \prec y\} \quad \langle \texttt{skip} \rangle \quad \{\mathsf{send}(y, Q)\}$$

By giving up the ability to acquire the P resource on the x channel, we can forward the P resource to partially discharge our send obligation on a subsequent channel y. If the initial send obligation on y requires us to supply a resource with two disjoint parts P and Q, after renouncing P to y, the obligation on y reduces to Q.

In the three-thread example, we can use this specification to justify signaling on `jStart` without waiting on `iEnd`:

```
{recv(iEnd, x ↦ _) * send(jStart, x ↦ _) * iEnd ≺ jStart}
// Renunciation axiom
{send(jStart, emp)}
signal(jStart); // signal specification
{emp}
```

SPECIFICATIONS:

$$\{\mathsf{emp}\}\quad \texttt{i = newchan()}\quad \{\mathsf{recv}(\texttt{i}, P) * \mathsf{send}(\texttt{i}, P)\}$$

$$\{\mathsf{send}(\texttt{i}, P) * \mathsf{stable}(P) * P\}\quad \texttt{signal(i)}\quad \{\mathsf{emp}\}$$

$$\{\mathsf{recv}(\texttt{i}, P) * \mathsf{stable}(\mathsf{P})\}\quad \texttt{wait(i)}\quad \{P\}$$

$$\left\{\begin{array}{l}\mathsf{send}(\texttt{x}, P) * \\ \circledast_{e\in E}\, e \prec \texttt{x} * \circledast_{l\in L}\, \texttt{x} \prec l\end{array}\right\}\quad \texttt{(b,a)=extend(x)}\quad \left\{\begin{array}{l}\mathsf{send}(\texttt{b}, Q) * \mathsf{recv}(\texttt{b}, Q) * \mathsf{send}(\texttt{a}, P) \\ * \texttt{b} \prec \texttt{a} * \circledast_{e\in E}\, e \prec \texttt{b} * \circledast_{l\in L}\, \texttt{a} \prec l\end{array}\right\}$$

AXIOMS:

$$x \prec y \implies x \prec y * x \prec y$$

$$x \prec y * y \prec z \implies x \prec z$$

$$\{\mathsf{recv}(x, P) * \mathsf{send}(y, P * Q) * x \prec y\}\quad \langle\texttt{skip}\rangle\quad \{\mathsf{send}(y, Q)\}$$

$$\{\mathsf{recv}(a, P * Q) * \mathsf{stable}(P * Q)\}\quad \langle\texttt{skip}\rangle\quad \{\mathsf{recv}(a, P) * \mathsf{recv}(a, Q)\}$$

Fig. 2. Abstract specification for deterministic parallelism.

3.4. Full Abstract Specification

Figure 2 shows our client-facing abstract specification for deterministic parallelism. It introduces the extra predicates and axioms to support chains, renunciation, and splitting.

Note that in Section 5 we define a more general specification that is more convenient when verifying channel implementations. Specifically, it uses *explicit stabilization* rather than stable assertions (see Section 4.1). The client-facing specification given in Figure 2 is less general, but also less complex, and sufficient to verify all our examples.

3.5. Adding Forward Extension

The specification in Figure 2 allows chain extension "backwards," by inserting a channel immediately before an existing send. However, in order to impose a sequential order, it is often useful to allow extension in the other direction, at the end of the chain. Fortunately, it is simple to implement a wrapper library with this behavior on top of our abstract specification. We use an object called seqChan to represent the end of the chain, with specification as follows:

$$\{\mathsf{seqChan}(\texttt{x}, P)\}\ \texttt{cw, cs = extendSC(x)}\ \{\mathsf{recv}(\texttt{cw}, P) * \mathsf{send}(\texttt{cs}, Q) * \texttt{cw} \prec \texttt{cs} * \mathsf{seqChan}(\texttt{x}, Q)\}$$

$$\{P\}\ \texttt{sc = newSeqChan()}\quad \{\mathsf{seqChan}(\texttt{sc}, P)\}$$

To implement a seqChan object, we store a pair of channels. The first, cWait, is the recv point for the resource P. The second, cEnd, is a dummy send channel used to allow extension. In order to extend, the wrapper library calls extend(cEnd)—this yields a new channel that can replace cWait. The seqChan object is thus defined as follows:

$$\mathsf{seqChan}(\texttt{x}, P) \triangleq \exists cw, ce.\ \texttt{x.cWait} \mapsto cw * \texttt{x.cEnd} \mapsto ce * \mathsf{recv}(cw, P) * \mathsf{send}(ce, \_) * cw \prec ce$$

The details of the implementation and verification of the wrapper library are given in Figure 3. Note that all the reasoning in this proof uses the abstract specification in Figure 2—we can use this specification to build and verify richer families of synchronization constructs.

3.6. Example: Concurrent Key-Value Store

We will now use our channels to implement a simple tree-based key-value store. We will focus on two methods: an add/update method that inserts a key-value pair, and a reverse lookup method that returns the key associated with a particular value if it exists. These

```
1  struct seqChan {
2    chan * cWait;
3    chan * cEnd;
4  }
5
6  seqChan * newSeqChan() {
7    chan c1,c2;
8  {P}
9    seqChan * sc = new(seqChan);
10 {sc.cWait ↦ _ * sc.cEnd ↦ _ * P}
11   c2 = newchan();
12 {sc.cWait ↦ _ * sc.cEnd ↦ _ * P * recv(c2, Q) * send(c2, Q)}
13   c1,c2 = extend(c2);
14 {sc.cWait ↦ _ * sc.cEnd ↦ _ * P * recv(_, Q) * send(c1, P) * recv(c1, P) * send(c2, Q) * c1 ≺ c2}
15   signal(c1);
16 {sc.cWait ↦ _ * sc.cEnd ↦ _ * recv(_, Q) * recv(c1, P) * send(c2, Q) * c1 ≺ c2}
17   sc->cWait = c1;
18   sc->cEnd = c2;
19 {sc.cWait ↦ c1 * sc.cEnd ↦ c2 * recv(_, Q) * recv(c1, P) * send(c2, Q) * c1 ≺ c2}
20   // Drop useless recv predicate for end of the channel
21 {seqChan(sc, P)}
22   return sc;
23 }
24
25 (chan *,chan *) extendSC(seqChan* sc) {
26   chan ce,cs,cw;
27 {seqChan(sc, P)}
28 {∃cw, ce. sc.cWait ↦ cw * sc.cEnd ↦ ce * recv(cw, P) * send(ce, _) * cw ≺ ce}
29   cw = sc->cWait;
30 {∃ce. sc.cWait ↦ cw * sc.cEnd ↦ ce * recv(cw, P) * send(ce, _) * cw ≺ ce}
31   cs,ce = extend(sc->cEnd);
32 {sc.cWait ↦ cw * sc.cEnd ↦ _ * recv(cw, P) * send(cs, Q) * recv(cs, Q)
   * send(ce, _) * cw ≺ cs * cs ≺ ce}
33   sc->cWait = cs;
34   sc->cEnd = ce;
35 {sc.cWait ↦ cs * sc.cEnd ↦ ce * recv(cw, P) * send(cs, Q) * recv(cs, Q)
   * send(ce, _) * cw ≺ cs * cs ≺ ce}
36 {recv(cw, P) * send(cs, Q) * cw ≺ cs * seqChan(sc, Q)}
37   return (cw,cs);
38 }
```

Fig. 3. Source code and proof outlines for seq channels.

methods are called sequentially, but our implementation permits concurrent updates and lookups internally within the data structure. The methods have the following specification:

$\{\mathsf{Tree}(\mathtt{x}, m)\}$ `add(x, k, v);` $\{\mathsf{Tree}(\mathtt{x}, m[\mathtt{k} \mapsto \mathtt{v}])\}$
$\{\mathsf{Tree}(\mathtt{x}, m)\}$ `k = revlookup(x, v);` $\{\mathsf{Tree}(\mathtt{x}, m) * (m(\mathtt{k}) = \mathtt{v} \vee (\mathtt{k} = -1 \wedge \mathtt{v} \notin img(m)))\}$

The second parameter of the Tree predicate represents the contents of the key-value store; that is, it is a partial function $m : \mathsf{Key} \rightharpoonup \mathsf{Val}$.

This specification requires that methods are called sequentially and indeed makes it appear that each method runs entirely sequentially. For example, the postcondition of `add(x,k,v)` is a Tree updated so that the store maps key `k` to value `v`. However, internally, the `add` operation forks a new thread to apply the update, then immediately returns. In other words, internally `add` and `revlookup` operations run concurrently, but our channels enforce the illusion of sequential access.

Algorithm. Keys and values are stored in a standard binary tree data structure, with left and right subtrees partitioned according to the value stored at a given node. Each subtree stores a seqChan in field `seqc`, which is used to sequentialize concurrent access to it.

```
struct Node {
   int key;
   int val;
   Tree * left;
   Tree * right;
};

  struct Tree {
   Node * node;
   seqChan * seqc;
}
```

```
Tree * newTree() {
   Tree * t = new(Tree);
   t->node = null;
   t->seqc = newSeqChan();
 }

Node * mkNode(int k, int v) {
   Node * n = new(Node);
   n->val = v;
   n->key = k;
   n->left = newTree();
   n->right = newTree();
}
```

The two main methods are defined as follows. Both call `extendSC` to register the current operation in the chain, then call a concurrent helper method that uses the channels to synchronize. In the case of `addNode`, the helper method is forked into another thread. These helper methods, `addNode` and `revlookupNode`, are defined in Figure 4.

```
add(Tree* t, int k, int v) {
  cw,cs = extendSC(t->seqc);
  fork(addNode(&(t->node),k,v,cw,cs));
}

int revlookup(Tree* t, int v) {
  cw,cs = extendSC(t->seqc);
  k = revlookupNode(&(t->node), v, cw, cs);
}
```

Channel operations are highlighted red in Figure 4. Both methods can be understood sequentially, if these operations are ignored:[1]

—`addNode()` searches from the root for the position of the chosen key—the order on nodes allows it to locate a unique position. On finding the appropriate subtree, it either creates a new node if one is absent or updates the value if a node exists.

[1]We might imagine the program being parallelized by injecting these constructs, either by an expert or a sufficiently clever automated analysis.

```
addNode(Tree * curr, int k, int v, chan *cw, chan *cs) {
  chan c1,c2;
  while(true) {
    wait(cw);
    if(curr->node == null) {
      Node * n = mkNode(k,v);
      curr->node = n;
      signal(cs);
      return;
    } else if (curr->node->key < k) {
      curr = curr->node->left;
    } else if (curr->node->key > k) {
      curr = curr->node->right;
    } else {
      curr->node->val = v;
      signal(cs);
      return;
    }
    c1,c2 = extendSC(curr->seqc);
    signal(cs);
    cw = c1;
    cs = c2;
  }
}


int revlookupNode(Tree * curr, int v, chan cw, chan cs) {
    wait(cw);
    if(curr->node == null) {
      signal(cs);
      return -1;
    } else {
        if (curr->node->val == v) {
          signal(cs);
          return k;
        }
        cl = curr->left;
        cr = curr->right;
        c1,c2 = extendSC(cl->seqc);
        c3,c4 = extendSC(cr->seqc);
        signal(cs);
        r = revlookupNode(cl, v, c1, c2)
        if(r == -1) {
           return revlookupNode(cr, v, c3, c4)
        } else {
           signal(c4); //Renunciation
           return r;
        }
    }
}
```

Fig. 4. Concurrent helper methods for the key-value store.

—`revLookupNode()` searches the entire tree for a key associated with the value. It checks the current node and then calls itself recursively on the left and right subtrees. Finding an appropriate value causes the function to return and the search to end.

The channel operations enforce sequential access for both operations when searching down the tree. Before accessing a subtree, an operation must register its presence using `extendSC` and then wait for all preceding threads using `wait`.

For example, `addNode` first waits for any preceding operations accessing this subtree (line 4). After taking a step left or right down the tree, the method registers its presence in the new subtree by extending the chain (line 19). It then signals it has left the parent node, allowing other operations to access it (line 20).

`revlookupNode` follows a similar pattern, but an operation registers its presence in both left and right subtrees, because it may need to search both (lines 22–23). If the value is located in the left subtree, then the right subtree will never be searched and can be signaled without waiting (line 42)—in other words, this is an example of renunciation.

Note that the need to wait for a return value sequentializes calls to `revlookup`. However, this restriction could be lifted with a little more work: `revlookup` could return a channel identifier through which its result value could be passed. A client program could then initiate multiple calls to `revlookup` in sequence and wait for all the results at once. Internally, these calls would run in parallel in precisely the same way as the current implementation. This would break the illusion of sequential execution, as clients would need to manage return-value channels. However, this specification could be verified with largely identical internal reasoning to the current version.

Verifying the key-value store. The predicate Tree representing the store data structure is defined as follows:

$$\begin{aligned}
\mathsf{Tree}(x, m) &\triangleq \exists c.\mathsf{seqChan}(c, \exists r.\, x.\mathtt{node} \mapsto r * \mathsf{Node}(r, m)) * x.\mathtt{seqc} \mapsto c \\
\mathsf{Node}(x, m) &\triangleq \exists l, r, k.\, x.\mathtt{key} \mapsto k * x.\mathtt{val} \mapsto v * x.\mathtt{left} \mapsto l * x.\mathtt{right} \mapsto r * \\
&\quad \mathsf{Tree}(l, m_l) * \mathsf{Tree}(r, m_r) * m_l; [k : v]; m_r = m * keys(m_l) < k < keys(m_r) \\
\mathsf{Node}(\mathtt{null}, m) &\triangleq m = \emptyset
\end{aligned}$$

The Tree predicate consists of a pointer to a seqChan predicate, which promises the rest of the tree. Thus, by using `extendSC` and `wait`, a thread can gain access to the remainder of the tree. The Node predicate splits the map m between its key/value and the left and right subtrees. Each subtree is represented recursively by a Tree predicate.

The key step in the proof lifts the specification of `extendSC` to the Tree predicate. The correctness of this step follows trivially from the definition of Tree.

$\{\mathsf{Tree}(\mathtt{t}, m)\}$
`cw,cs = extendSC(t->seqc);`
$\left\{\begin{array}{l}\mathsf{Tree}(\mathtt{t}, m') * \mathsf{recv}(\mathtt{cw}, \exists x.\ \ \mathtt{curr.node} \mapsto x * \mathsf{Node}(x, m)) \\ * \ \mathsf{send}(\mathtt{cs}, \exists x.\ \ \mathtt{curr.node} \mapsto x * \mathsf{Node}(x, m')) * \mathtt{cw} \prec \mathtt{cs}\end{array}\right\}$

In the precondition, the current thread is promised a tree containing map m. This proof step promises that m will then be updated to some other map m'. This m' can be chosen as needed to represent the correct update—the choice is entirely arbitrary. However, before the channel can be signaled, the channel specification requires that the map m' is established, which enforces the obligation to make a reasonable choice. This step also gives a handle to the old promise, `cw`, and a handle used to establish the new promise, `cs`.

This proof step is used in the `add` and `addNode` functions. For example, in `add`, the promised map is updated to $m[k : v]$ as follows. This guarantees a correctly updated data

structure to the next thread accessing the store but defers the actual data structure modification.

```
{Tree(t, m)}
add(Tree* t, int k, int v) {
  cw,cs = extendSC(t->seqc);
⎧Tree(t, m[k : v]) ∗ recv(cw, ∃x. t.node ↦ x ∗ Node(x, m))⎫
⎩∗ send(cs, ∃x. t.node ↦ x ∗ Node(x, m[k : v]))          ⎭
  fork(addNode(t, k, v, cw, cs));
}
{Tree(t, m[k : v])}
```

The proof outline for addNode can be found in Figure 5. Each iteration of the main loop updates one subtree. The loop invariant (line 6) consists of a recv predicate for the previous version of the subtree and a send predicate for the subtree updated with the key/value pair. Calling wait grants access to the previous version of the subtree. The algorithm then examines the node contents and branches on the results.

If the current node is null, then the map m must be empty. Therefore, send can be satisfied by just creating a single node containing the key/value pair (line 14). The case where the node already exists is similar.

In the recursive case, the algorithm first promises to update the tree by calling extendSC on the appropriate subtree (line 19). It then uses signal to indicate it has finished with the current node and then recurses to the appropriate subtree.

More specifically, suppose the algorithm takes the left branch. In order to signal, it must first satisfy the send on cs, which requires a Node predicate with the key/value pair inserted. It is easy to prove that the map can be partitioned into subtrees less than and greater than the current key—these two maps are written m_l and m_r (line 22). As the required key must be added to the left subtree, the right subtree need not be updated. Calling extendSC promises to update the left subtree with the key/value pair, which in turn satisfies the Node predicate required by send. This means the algorithm can call signal (line 23).

The proof structure for revlookup is similar to the one for add: by extending the chain with extendSC, the program gains the right to access the tree structure while still permitting other threads to work on the tree at the same time.

```
{Tree(t, m)}
int revlookup(Tree* t, int v) {
  cw,cs = extendSC(t->seqc);
  ⎧Tree(x, m) ∗ recv(cw, ∃x. t.node ↦ x ∗ Node(x, m))⎫
  ⎩∗ send(cs, ∃x. t.node ↦ x ∗ Node(x, m))          ⎭
  k = revlookupNode(&(t->node), v, cw, cs);
}
{Tree(x, m) ∗ (m(k) = v ∨ (k = −1 ∧ v ∉ img(m)))}
```

The proof outline for the revlookupNode helper function can be found in Figures 6 and 7. The broad structure of the reasoning is similar to addNode, albeit expressed inductively rather than iteratively. Each step down the tree retrieves a subtree by calling wait and then releases the portion of the tree it has already searched using signal. Because subtrees are associated with channels, the algorithm only accesses the portion of the tree it needs, leaving the remainder of the tree available to other threads.

One significant step is the renunciation that takes place on line 40. Here the algorithm takes the requirement to supply a subtree to c4 and satisfies it using the recv

```
1  {recv(cw, ∃x. curr.node ↦ x * Node(x, m)) * send(cs, ∃x. curr.node ↦ x * Node(x, m[k : v]))}
2  addNode(Tree * curr, int k, int v, chan *cw, chan *cs) {
3    Tree * t;
4    chan c1,c2;
5    while(true) {
6  {recv(cw, ∃x. curr.node ↦ x * Node(x, m)) * send(cs, ∃x. curr.node ↦ x * Node(x, m[k : v]))}
7      wait(cw);
8  {∃x. curr.node ↦ x * Node(x, m) * send(cs, ∃x. curr.node ↦ x * Node(x, m[k : v]))}
9      if(curr->node == null) {
10   {curr.node ↦ null * m = ∅ * send(cs, ∃x. curr.node ↦ x * Node(x, m[k : v]))}
11       Node * n = mkNode(k,v);
12       curr->node = n;
13   {∃x. curr.node ↦ x * Node(x, [k : v]) * send(cs, ∃x. curr.node ↦ x * Node(x, [k : v]))}
14       signal(cs);
15       return;
16     } else if (curr->node->key < k) {
17       curr = curr->node->left;
18 {∃c, x. c.node ↦ x * ∃r, k′, v′. x.key ↦ k′ * x.val ↦ v′ * x.left ↦ curr * x.right ↦ r
   * Tree(curr, m_l) * Tree(r, m_r) * m_l; [k′ : v′]; m_r = m * keys(m_l) < k′ < keys(m_r)
   * k < k′ * send(cs, ∃x. c.node ↦ x * Node(x, m[k : v]))}
19       c1,c2 = extendSC(curr->seqc);
20 {∃c, x. c.node ↦ x * ∃r, k′, v′. x.key ↦ k′ * x.val ↦ v′ * x.left ↦ curr * x.right ↦ r
   * Tree(curr, m_l[k : v]) * Tree(r, m_r) * m_l[k : v]; [k′ : v′]; m_r = m * keys(m_l) < k′ < keys(m_r)
   * send(cs, ∃x. c.node ↦ x * Node(x, m[k : v]))
   * recv(c1, ∃x. curr.node ↦ x * Node(x, m_l)) * send(c2, ∃x. curr.node ↦ x * Node(x, m_l[k : v]))}
21       // definition of Node predicate
22 {∃c, x. c.node ↦ x * Node(x, m[k : v]) * send(cs, ∃x. c.node ↦ x * Node(x, m[k : v]))
   * recv(c1, ∃x. curr.node ↦ x * Node(x, m_l)) * send(c2, ∃x. curr.node ↦ x * Node(x, m_l[k : v]))}
23       signal(cs);
24 {recv(c1, ∃x. curr.node ↦ x * Node(x, m_l)) * send(c2, ∃x. curr.node ↦ x * Node(x, m_l[k : v]))}
25       cw = c1;
26       cs = c2;
27 {recv(cw, ∃x. curr.node ↦ x * Node(x, m_l)) * send(cs, ∃x. curr.node ↦ x * Node(x, m_l[k : v]))}
28     } else if (curr->node->key > k) {
29       // Similar to left side
30       ...
31     } else {
32       curr->node->val = v;
33       signal(cs);
34       return;
35     }
36   }
37 }
```

Fig. 5. Proof outline for addNode.

predicate for c3. In other words, the resource is released without ever being acquired by the thread.

Summary. Thus, using our abstract specification for channels, we have verified the behavior of the parallelized key-value store. Crucially, even though this program features many threads running at once, with complex communication between threads, each individual thread is able to reason locally, without dealing with other threads or

```
1  {recv(cw, ∃x. curr.node ↦ x * Node(x,m)) * send(cs, ∃x. curr.node ↦ x * Node(x,m))}
2  int revlookupNode(Tree * curr, int v, chan cw, chan cs) {
3      wait(cw);
4  {∃x. curr.node ↦ x * Node(x,m) * send(cs, ∃x. curr.node ↦ x * Node(x,m))}
5      if(curr->node == null) {
6    {∃x. curr.node ↦ x * Node(x,m) * send(cs, ∃x. curr.node ↦ x * Node(x,m)) ∧ m = ∅}
7        signal(cs);
8    {v ∉ img(m)}
9        return -1;
10     } else {
11 {∃x. curr.node ↦ x * Node(x,m) * send(cs, ∃x. curr.node ↦ x * Node(x,m))}
12         if (curr->node->val == v) {
13           k = curr->node->key;
14       {∃x. curr.node ↦ x * Node(x,m) * send(cs, ∃x. curr.node ↦ x * Node(x,m)) ∧ m = [k : v]}
15           signal(cs);
16       {m(k) = v}
17           return k;
18         }
19 {∃x. curr.node ↦ x * Node(x,m) * send(cs, ∃x. curr.node ↦ x * Node(x,m))}
20         cl = curr->left;
21         cr = curr->right;
22         c1,c2 = extendSC(cl->seqc);
23         c3,c4 = extendSC(cr->seqc);
24 { ∃x. curr.node ↦ x * Node(x,m) * send(cs, ∃x. curr.node ↦ x * Node(x,m))
   * recv(c1, ∃x. cl.node ↦ x * Node(x,m_l)) * send(c2, ∃x. cl.node ↦ x * Node(x,m_l))
   * recv(c3, ∃x. cr.node ↦ x * Node(x,m_r)) * send(c4, ∃x. cr.node ↦ x * Node(x,m_r))
   * ∃k', v'. m = m_l; [k' : v']; m_r * v ≠ v' }
25         signal(cs);
26 { recv(c1, ∃x. cl.node ↦ x * Node(x,m_l)) * send(c2, ∃x. cl.node ↦ x * Node(x,m_l))
   * recv(c3, ∃x. cr.node ↦ x * Node(x,m_r)) * send(c4, ∃x. cr.node ↦ x * Node(x,m_r))
   * ∃k', v'. m = m_l; [k' : v']; m_r }
27         r = revlookupNode(cl, v, c1, c2)
28 {recv(c3, ∃x. cr.node ↦ x * Node(x,m_r)) * send(c4, ∃x. cr.node ↦ x * Node(x,m_r))
   * ∃k', v'. m = m_l; [k' : v']; m_r * v ≠ v' * (m_l(r) = v ∨ (v ∉ img(m_l) ∧ r = −1))}
29     }
30 }
```

Fig. 6. Proof outline for `revlookupNode` (completed in Figure 7).

the implementation of the barriers. Furthermore, we are able to verify a function specification that hides the existence of internal concurrency. In Botinčan et al. [2013], we use the same abstract specification as the basis for a general parallelization analysis.

4. PROOF STRATEGY

This section provides an intuitive introduction to our general proof approach, and to iCAP, the reasoning system our proofs are based on. We introduce iCAP using a tutorial-style presentation by verifying a simple channel implementation against a simplified barrier specification consisting of strengthened versions of the first three axioms of our

```
31  {recv(c3, ∃x. cr.node ↦ x * Node(x, m_r)) * send(c4, ∃x. cr.node ↦ x * Node(x, m_r))
    * ∃k', v'. m = m_l; [k' : v']; m_r * v ≠ v' * (m_l(r) = v ∨ (v ∉ img(m_l) ∧ r = −1))}
32          if(r == -1) {
33  {recv(c3, ∃x. cr.node ↦ x * Node(x, m_r)) * send(c4, ∃x. cr.node ↦ x * Node(x, m_r))
    * ∃k', v'. m = m_l; [k' : v']; m_r * v ≠ v' * (m_l(−1) = v ∨ (v ∉ img(m_l)))}
34              r = revlookupNode(cr, v, c3, c4)
35  {∃k', v'. m = m_l; [k' : v']; m_r * v ≠ v' * (m_l(−1) = v ∨ (v ∉ img(m_l)))
    * (m_r(r) = v ∨ (v ∉ img(m_r) ∧ r = −1))}
36  {(m(−1) = v ∨ (v ∉ img(m) ∧ r = −1)) ∨ (m(r) = v)}
37              return r;
38          } else {
39  {recv(c3, ∃x. cr.node ↦ x * Node(x, m_r)) * send(c4, ∃x. cr.node ↦ x * Node(x, m_r))
    * ∃k', v'. m = m_l; [k' : v']; m_r * v ≠ v' * m_l(r) = v}
40              <skip> // Renunciation
41  {send(c4, true) * ∃k', v'. m = m_l; [k' : v']; m_r * v ≠ v' * m_l(r) = v}
42              signal(c4);
43  {∃k', v'. m = m_l; [k' : v']; m_r * v ≠ v' * m_l(r) = v}
44  {m(r) = v}
45              return r;
46          }
47      }
48  }
```

Fig. 7. Proof outline for `revlookupNode` (continued from Figure 6).

```
struct chan {
  int flag;
}

chan *newchan() {
  chan *x := new(chan);
  x->flag := 0;
  return x;
}

signal(chan *x) {
  x->flag := 1;
}

wait(chan *x) {
  while(x->flag == 0)
    skip;
}
```

Fig. 8. Implementation of the barrier library.

abstract specification:

$$\begin{array}{rcl} \{\mathsf{emp}\} & \mathtt{i} = \mathtt{newchan}() & \{\mathsf{recv}(\mathtt{i}, P) * \mathsf{send}(\mathtt{i}, P)\} \\ \{\mathsf{send}(\mathtt{i}, P) * \mathsf{stable}(P) * P\} & \mathtt{signal(i)} & \{\mathsf{emp}\} \\ \{\mathsf{recv}(\mathtt{i}, P) * \mathsf{stable}(P)\} & \mathtt{wait(i)} & \{P\} \end{array}$$

By avoiding splitting, chain extension, and renunciation, we can illustrate the basic features of iCAP in a straightforward manner.

Figure 8 gives a simple barrier implementation. Each channel has a `flag` field representing the current state of the channel. Each send / recv pair is associated with one such structure in the heap. The `signal` simply sets the flag, while the `wait` loops until

the flag is set. In Sections 5 and 6, we reintroduce the necessary extra reasoning to verify our full abstract specification.

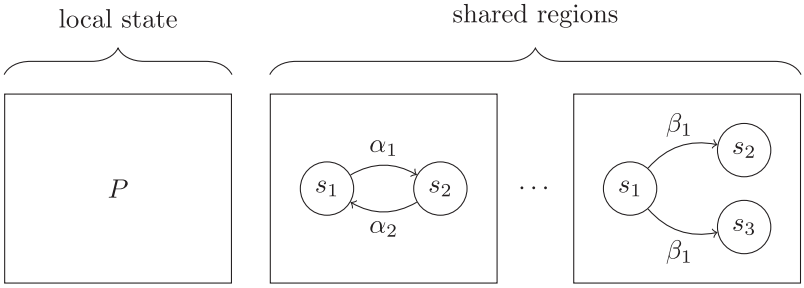
4.1. iCAP Tutorial

iCAP is a separation logic variant intended for verifying concurrent higher-order programs [Svendsen and Birkedal 2014a, 2014b]. In this section, we introduce the iCAP concepts and proof rules needed to understand the proofs in this article. The full iCAP proof system is included in Appendix C.

We do not present iCAP's step-indexed semantics in this article. Instead, we reason entirely using iCAP's proof rules and treat the semantics as a black box. The soundness of these proof rules is proved in the iCAP article [Svendsen and Birkedal 2014a] with some auxiliary lemmas proved in the associated technical report [Svendsen and Birkedal 2014b].

In this section, we employ a tutorial-style presentation, using the verification of the simplified barrier to motivate and illustrate concepts and proof rules as they are introduced. When presenting proof rules, we will elide typing contexts and logical contexts.

Regions. To handle concurrency, iCAP extends separation logic with *regions* containing resources shared between threads. Conceptually, the state in iCAP consists of a local component and a finite number of shared regions, each governed by a *protocol*, as illustrated here:



In the state just illustrated, we own the local resource P exclusively and can thus access it nonatomically. Resources owned by regions are shared between every thread and must therefore be accessed atomically and updated according to the protocol of the given region.

Protocols require that threads sharing a region behave as expected: for example, only releasing a lock once it has been acquired. They are expressed as a *labeled transition system* and an *interpretation function*. The labeled transition system defines the abstract states of the given region and describes how the abstract state may evolve, while the interpretation function maps each abstract state to a corresponding concrete heap resource. An assertion about a region has the following form:

$$\mathsf{region}(R, T, I, r)$$

In this assertion, R is the set of abstract states that the region could currently occupy. These possible states are taken from a larger set, fixed when the region is created. I maps from abstract states to invariants, also written in iCAP's assertion language. Intuitively, $I(x)$ describes the resources contained in the region when it is in abstract state $x \in R$. To allow multiple distinct regions, r is a unique identifier for this region.

The remaining field, T, is a transition relation over abstract states, with transitions labeled with *actions*. T and I express the protocol that all threads must adhere to when accessing the region. Threads are only allowed to move a region from one abstract state

to another if there exists a path in the labeled transition system T labeled with actions *permitted* to the thread. Permitted actions are tracked using *tokens*. These are linear objects created at the same time as a region. By issuing threads different tokens, we grant them different abilities over the shared region. For instance, the following assertion asserts ownership of a set and change token: $[\mathsf{set}]_i^{r_1} * [\mathsf{change}]_j^{r_2}$. Here r_1 and r_2 are the identifiers for the associated regions, while i and j are fractional parameters tracking how ownership of each token is shared.

A token with full permission ($i = 1$) asserts exclusive ownership of the action and thus ensures that no other thread can use the given action to change the abstract state of the shared region. Partial permission ($i < 1$) allows the owner to use the action but does not exclude other currently executing threads from also using the action. Tokens can be split if the permission value is preserved: $[\alpha]_{i+j}^r \Leftrightarrow [\alpha]_i^r * [\alpha]_j^r$. In contrast, region assertions can be duplicated arbitrarily:

$$\mathsf{region}(R, T, I, r) \Rightarrow \mathsf{region}(R, T, I, r) * \mathsf{region}(R, T, I, r)$$

This allows arbitrarily many threads to access a shared region, but the ability to modify the region is restricted to threads holding the appropriate tokens.

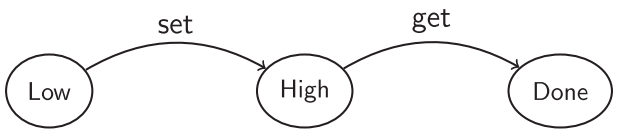
Defining send *and* recv. To verify the implementation shown in Figure 8 with respect to the abstract specification, we must first give concrete definitions to the abstract predicates send and recv.

The idea is to introduce a shared region for each channel, governing the internal state of that channel (the flag field) and ownership of the promised resource P. The shared region will have three possible abstract states: Low, High, and Done. In the Low state, the flag is low and the promised resource has yet to be provided. In the High state, the flag is high and the promised resource has been sent but not yet received and is thus conceptually owned by the channel. Lastly, in the Done state, the flag is high and the promised resource has been sent and received. Each abstract state is associated with an invariant by the interpretation function I_b, defined as follows:

$$\begin{aligned} I_\mathsf{b}(x, P)(\mathsf{Low}) &\triangleq x.\mathsf{flag} \mapsto 0 \\ I_\mathsf{b}(x, P)(\mathsf{High}) &\triangleq x.\mathsf{flag} \mapsto 1 * \mathsf{stable}(P) * P \\ I_\mathsf{b}(x, P)(\mathsf{Done}) &\triangleq x.\mathsf{flag} \mapsto 1 \end{aligned}$$

Here x is the location of the channel, while P is the resource controlled by the channel. Note that to move from the Low to the High abstract state, the client must transfer ownership of P to the shared region, in addition to setting the flag. Likewise, to move from the High to the Done abstract state, the client may transfer P out of the shared region and into its local state.

We want to ensure that only the sender (i.e., the owner of the send resource) is allowed to transition the abstract state from Low to High and only the receiver from High to Done. This will force the sender to transfer P to the shared region upon setting the flag and allow the receiver to take ownership of P once the flag has been set. In iCAP, we express this by labeling the Low to High transition with a set action and giving the sender exclusive ownership of the set action. The transition relation T_b thus has the following form. Each transition corresponds to an operation that can be performed on the channel.



Using these definitions, we can give the predicates send and recv an interpretation:

$$\mathsf{send}(x, P) \triangleq \exists r.\ \mathsf{region}(\{\mathsf{Low}\}, T_\mathsf{b}, I_\mathsf{b}(x, P), r) * [\mathsf{set}]_1^r$$

$$\mathsf{recv}(x, P) \triangleq \exists r.\ \mathsf{region}(\{\mathsf{Low}, \mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(x, P), r) * [\mathsf{get}]_1^r$$

The send predicate asserts that the abstract state is Low, since the promised resource has not been sent yet. On the other hand, recv asserts that the abstract state is either Low or High, but not Done, as the receiver does not know whether the promised resource has been sent yet, only that it has not been received yet. $[\mathsf{set}]_1^r$ and $[\mathsf{get}]_1^r$ are tokens allowing the thread to take particular transitions in T_b. The send predicate allows the sender to set the flag and supply the promised resource, while recv allows the receiver to retrieve the promised resource.

Stability. Region assertions allow us to describe our knowledge about the current abstract state of a region. However, since regions are shared, concurrently executing threads may update the abstract state of regions, invalidating our region assertions in the process. An assertion is said to be *stable* if it is invariant under possible interference from the environment. Since the concrete send resource defined previously asserts exclusive ownership of the set action, the environment cannot use this transition to update the abstract state. Hence, if we own the send resource, the environment cannot invalidate our knowledge that the current abstract state is Low; thus, the send resource is stable.

More generally, a region assertion, $\mathsf{region}(R, T, I, r)$, is stable if R is closed under all transitions in T labeled with actions potentially owned by the environment. This is captured by the following iCAP proof rule:

$$(\forall \alpha \notin A.\ \forall x \in X.\ T(\alpha)(x) \subseteq X) \Rightarrow \mathsf{stable}(\mathsf{region}(X, T, I, r) * \circledast_{\alpha \in A}[\alpha]_1^r)$$

Since the assertion asserts exclusive ownership of all the action in A ($\circledast_{\alpha \in A}[\alpha]_1^r$), the environment cannot own any actions in A. Hence, the region assertion is stable if X is closed under any transitions α not in A, as expressed by the assumption.

Stability is closed under several of the usual connectives of higher-order separation logic ($\bot, \top, \vee, \wedge, \forall, \exists, =_\tau, \mathsf{emp}, *$) but generally not under implication and separating implication. To avoid reasoning about the stability of separating implication, we use Wickerson et al.'s explicit stabilization operators, $\lfloor - \rfloor$ and $\lceil - \rceil$, instead of requiring P to be stable [Wickerson et al. 2010]. We explicitly stabilize using $\lfloor P \rfloor$ and $\lceil P \rceil$, which are stable by construction for any assertion P. $\lfloor P \rfloor$ stands for the weakest assertion stronger than P that is stable and $\lceil P \rceil$ for the strongest assertion weaker than P that is stable. Thus, if P is already stable, explicitly stabilizing P does nothing:

$$\mathsf{stable}(P) \Rightarrow (\lfloor P \rfloor \iff P \iff \lceil P \rceil)$$

However, in general, only the right implications hold: $\lfloor P \rfloor \Rightarrow P \Rightarrow \lceil P \rceil$. Explicit stabilization operators are semidistributive over separating conjunctions:

$$\lfloor P \rfloor * \lfloor Q \rfloor \Rightarrow \lfloor P * Q \rfloor \qquad\qquad \lceil P * Q \rceil \Rightarrow \lceil P \rceil * \lceil Q \rceil$$

As a result, they are easy to move around in proofs.

The concrete recv resource defined earlier is thus also easily shown to be stable, as {Low, High} is closed under the set transition, which is the only transition potentially owned by the environment.

When reasoning about nonatomic statements, iCAP requires that the pre- and postcondition are stable, to account for possible interference from the environment. We use standard Hoare triples, written $\{P\}\ C\ \{Q\}$, when reasoning about nonatomic statements, and angled triples, written $\langle P \rangle\ C\ \langle Q \rangle$, when reasoning about atomic statements.

For every standard Hoare triple, $\{P\}\ C\ \{Q\}$, we implicitly have to prove the stability of P and Q. For the majority of the proof outlines in this article, these stability proofs are trivial and will be omitted. The following structural rule allows us to switch from atomic to nonatomic triples:

$$\mathsf{atomic}(C) \wedge \mathsf{stable}(P) \wedge \mathsf{stable}(Q) \wedge \langle P\rangle\ C\ \langle Q\rangle \Rightarrow \{P\}\ C\ \{Q\}$$

The predicate atomic holds for any command that is assumed to be atomic by iCAP: these are CAS, field read, field assignment, and stack assignment.

Higher-order shared resources. The idea of shared resources that must satisfy an invariant or evolve following a protocol could equally be expressed in prior logics such as CAP or RGSep [Vafeiadis and Parkinson 2007; Dinsdale-Young et al. 2010]. The distinction with iCAP is that it is based on a higher-order separation logic and supports shared higher-order resources—that is, shared regions containing shared resources. For instance, the send resource defined earlier is parametric in the promised resource P, which the client is free to instantiate with a shared resource—for example, another synchronization construct.

It is well known that reasoning about shared higher-order resources is difficult (e.g., see the problems with our previous article, discussed in Section 8). Intuitively, this is because the semantics of protocols is defined in terms of the semantics of assertions, but assertions are defined in terms of protocols. To avoid this problematic circularity, iCAP stratifies the construction of the semantic domain of protocols using step-indexing. As a result of this stratification, assertions and specifications are only given meaning with respect to a step-index.

In the case of self-referential protocols, step-indexing breaks the circularities by interpreting a region assertion at step-index $n+1$ in terms of the semantics of the region interpretation at step-index n. For instance, a region assertion $\mathsf{region}(\{x\}, T, I, r)$ holds at step-index $n+1$ if the shared resources owned by region r satisfies $I(x)$ at step-index n. To capture this syntactically, iCAP introduces a "later" modality, written $\triangleright$. The assertion $\triangleright P$ holds at step-index $n+1$ if P holds at step-index n. The region assertion $\mathsf{region}(\{x\}, T, I, r)$ thus expresses that the shared region r currently owns the resources described by $\triangleright I(x)$.

The semantics of Hoare triples connects the step-indexes of the pre- and postcondition with execution steps. Disregarding environmental interference, a specification $\{P\}C\{Q\}$ is valid if for any number of steps n and any initial state in P at step-index n, if C terminates in $k \leq n$ steps, then the terminal states satisfies Q at step-index $n-k$. The effects of this stratification and connection of step-indices with operational steps are that reasoning involving shared resources requires a corresponding step in the program execution. Intuitively, the logic must "wait a step" before examining resources transferred out of shared regions to prevent cycles in the reasoning. Formally, this is expressed by the frame rule for atomic commands (AFrame) given next, which allows the removal of a later from the frame. In particular, if $\triangleright P$ holds for the frame at step-index n, then P holds for the frame at step-index $n-1$. Thus, since the atomic command requires exactly one step of execution, P holds in the postcondition, as the postcondition is evaluated at step-index $n-1$.

$$\mathsf{stable}(R) \wedge \langle P\rangle\ C\ \langle Q\rangle \Rightarrow \langle P * \triangleright R\rangle\ C\ \langle Q * R\rangle \qquad (\textsc{AFrame})$$

One intuition for the step-indexed model is that an assertion P holds at step-index n if the assertion that it makes about the state cannot be invalidated with n or fewer steps of execution. Consequently, if an assertion P holds at step-index $n+1$, it also holds at step-index n, since any assertion that can be invalidated in n or fewer steps can also be invalidated in $n+1$ or less steps. This property is expressed by the (SMono)

rule given next, which expresses that if P holds now, it also holds with one less step of execution left ($\triangleright P$).

$$P \Rightarrow \triangleright P \qquad \text{(SMono)}$$

Using (SMono) and the rule of consequence, one can derive the standard CAP frame rule. The assertion $\triangleright P$ is stable if P is stable.

The effect of introducing $\triangleright$ is that accesses to shared resources in regions coincide with operational steps in the program. This breaks the circularity and gives iCAP a well-defined semantics.

However, it also means that splitting and renunciation must be associated with an explicit `skip` instruction to justify the transfer of shared resources in and out of regions. While these `skip` instructions are crucial to the well-definedness of protocols, they can typically be eliminated once we consider a whole program. In particular, for pre- and postconditions expressible in first-order separation logic, iCAP is adequate with respect to first-order separation logic [Svendsen and Birkedal 2014b, Theorem 1] and in first-order separation logic `skip` instructions can freely be eliminated. Hence, if P and Q are expressible in first-order separation logic and $\vdash_{iCAP} \{P\}\ C\ \{Q\}$, then $\vdash_{SL} \{P\}\ \tilde{C}\ \{Q\}$, where $\tilde{C}$ is C stripped of `skip` instructions.

The later operator distributes over conjunction, disjunction, separating conjunction, and stabilization brackets. It semidistributes over implication and separating implication, that is:

$$\triangleright(P \Rightarrow Q) \Rightarrow (\triangleright P \Rightarrow \triangleright Q) \qquad \triangleright(P \mathrel{-\!\!*} Q) \Rightarrow (\triangleright P \mathrel{-\!\!*} \triangleright Q)$$

Later also distributes over existential and universal quantification over nonempty types. See Appendix C.2.2 for proof rules. In proof outlines, we sometimes apply these properties silently.

Reasoning about shared regions. All statements that access resources owned by shared regions must be atomic and obey the protocols of the regions involved. This is achieved using structural rules that allow shared resources to be treated as local resources for the duration of an atomic statement. We refer to these rules as "region opening" rules and as "entering" and "exiting" a shared region when applying these rules in proof outlines.

Conceptually, these rules require us to prove (1) that we own sufficient action permissions to justify any potential updates of the abstract state and (2) that we transfer the appropriate resources back to the shared region after the atomic statement. To illustrate, consider verifying the signal method of the simplified barrier implementation:

$$\{\mathsf{send}(\mathtt{i}, P) * \mathsf{stable}(P) * P\}\ \mathtt{i->flag\ :=\ 1}\{\mathsf{emp}\}$$

Since raising the flag is an atomic statement and both the pre- and postcondition are stable, we can switch to atomic triples. This frees us from reasoning about stability and possible interference during the execution of the atomic statement. After unfolding the send resource, we are thus left with the following proof obligation:

$$\langle \mathsf{region}(\{\mathsf{Low}\}, T_{\mathsf{b}}, I_{\mathsf{b}}(\mathtt{x}, P), r) * [\mathsf{set}]^r_1 * \mathsf{stable}(P) * P \rangle \mathtt{x->flag\ :=\ 1}\ \langle \mathsf{emp} \rangle$$

The precondition asserts that the flag field is owned by the shared region r. To update the field, we are thus forced to open the shared region governing the given channel and thus to respect its protocol. Intuitively, upon raising the flag, the abstract state changes from Low to High. We can thus strengthen the proof obligation:

$$\langle \mathsf{region}(\{\mathsf{Low}\}, T_{\mathsf{b}}, I_{\mathsf{b}}(\mathtt{x}, P), r) * [\mathsf{set}]^r_1 * \mathsf{stable}(P) * P \rangle$$
$$\mathtt{x->flag\ :=\ 1}$$
$$\langle \mathsf{region}(\{\mathsf{High}\}, T_{\mathsf{b}}, I_{\mathsf{b}}(\mathtt{x}, P), r) \rangle$$

Note that this postcondition is not stable under get transitions. However, since we are reasoning about an atomic statement, the postcondition is not required to be stable.

To discharge the previous proof obligation, we first have to prove that we are allowed to update the abstract state from Low to High. Since the precondition asserts exclusive ownership of the set transition, this reduces to proving that there exists a set-labeled path from Low to High in T_b, which is true by definition. Second, we must prove that `x->flag := 1` does indeed transform the resources associated with abstract state Low to those associated with abstract state High, according to $I_b(x, P)$. We are thus left with the following proof obligation:

$$\langle \triangleright I_b(x, P)(\mathsf{Low}) * [\mathsf{set}]^r_1 * \mathsf{stable}(P) * P \rangle \ \texttt{x->flag := 1} \ \langle \triangleright I_b(x, P)(\mathsf{High}) \rangle$$

Given local ownership of the shared resources for the abstract state Low, we must transfer back resources corresponding to abstract state High after the execution of the atomic statement. Since protocols are implicitly interpreted one step later, the shared resources are only available one step later in the precondition and only have to be provided one step later in the postcondition. After unfolding $I_b(x, P)$ and applying SMono, we are left with the following proof obligation:

$$\langle (\triangleright x.\mathsf{flag} \mapsto 0) * [\mathsf{set}]^r_1 * \mathsf{stable}(P) * P \rangle \ \texttt{x->flag := 1} \langle x.\mathsf{flag} \mapsto 1 * \mathsf{stable}(P) * P \rangle$$

Conceptually, this is provable because step-indexing only affects assertions that can generate problematic circularities in the domain. Primitive points-to assertions such as $x.f \mapsto v$ only affect addresses and values, and are thus independent of the step-indexing. This is captured by the structural (LPoints) rule given next, which allows us to remove a $\triangleright$ from the precondition of a points-to assertion.

$$\langle x.f \mapsto y \rangle \ C \ \langle Q \rangle \Rightarrow \langle \triangleright x.f \mapsto y \rangle \ C \ \langle Q \rangle \qquad \text{(LPoints)}$$

The general iCAP proof rule for accessing shared regions is given as follows:

$$\frac{\begin{array}{c}\forall x \in X.\,(x,\ f(x)) \in \overline{T(A)} \\ \forall x \in X.\,\langle P * \circledast_{\alpha \in A}[\alpha]^r_{g(\alpha)} * \triangleright I(x) \rangle \ C \ \langle Q(x) * \triangleright I(f(x)) \rangle^{\mathcal{E}}\end{array}}{\langle P * \circledast_{\alpha \in A}[\alpha]^r_{g(\alpha)} * \mathsf{region}(X, T, I, r) \rangle \ C \ \langle \exists x.\, Q(x) * \mathsf{region}(\{f(x)\}, T, I, r) \rangle^{\mathcal{E} \uplus \{r\}}} \qquad \text{(AOpen)}$$

It generalizes the previous example by considering a set of possible initial abstract states X and allowing the client to transfer ownership of resources in and out of the shared region using P and Q. To apply the rule, we must define a function f that for every possible initial abstract, state $x \in X$ defines the desired terminal abstract state $f(x)$. The first premise asserts that for every possible initial abstract state x, there exists a path from x to $f(x)$ labeled with actions from the set A (we use $\overline{R}$ as notation for the reflexive, transitive closure of the relation R). Since the precondition asserts nonexclusive ownership of every action in A, this ensures that we are allowed to update the abstract state from x to $f(x)$ for every possible initial state $x \in X$. The g function, which records the fractional ownership of each action $\alpha \in A$, ensures that we own the same action fractions in the assumption and conclusion of the rule. We implicitly require that $g(\alpha) \neq 0$ for all $\alpha \in A$. The second premise ensures that C does indeed transform the resources associated with the abstract state x to $f(x)$ for every possible initial abstract state $x \in X$.

Since opening a shared region grants local ownership of the region's current resources, in general, it would be unsound to have two nested openings of the same region in the proof tree, as this would duplicate the region's resources. To avoid this, we annotate the postcondition of atomic triples with a *region mask*, $\mathcal{E}$, of regions that may be opened. To open the region r, the previous proof rule requires that r is in the

region mask and removes it from the region mask in the premise, to ensure r is not opened again.

Verifying `signal` *and* `wait`. Figures 10 and 11 sketch proofs for `signal()` and `wait()`, respectively. The proof of `wait` (Figure 11) uses similar reasoning to `signal`. The main difference is that the region has two initial abstract states, Low and High, and that the thread holds the token get, allowing it to transition from High to Done. We deal with the Low and High cases separately—see the bottom left and right of Figure 11. In the Low case, the resource has not been sent yet and we close the region in the Low state again. In the High case, the resource has been sent and we use the thread's get token to take ownership of P and close the region in the Done state.

iCAP is an intuitionistic separation logic, meaning it admits weakening, and in particular that $P * Q \Rightarrow Q$. We often use this property to dispose of unwanted predicates—intuitively, we can forget they exist. For example, we delete a redundant set token on line 11 of Figure 10.

View shifts. The region opening rule given earlier allows a region to be opened for the duration of an atomic statement. It is also possible to open a shared region and close it immediately before the next statement is executed. This is, for instance, useful to abstractly describe transfer of resources in and out of shared regions. iCAP expresses such updates of the abstract state that do not affect the concrete state using the *view-shift* operator, $\sqsubseteq$.

For instance, the following view shift expresses that if the channel is in the abstract state High and we own the get action, then we can change the abstract state to Done and take ownership of $\triangleright P$.

$$\mathsf{region}(\{\mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(x, P), r) * [\mathsf{get}]^r_1 \sqsubseteq \mathsf{region}(\{\mathsf{Done}\}, T_\mathsf{b}, I_\mathsf{b}(x, P), r) * [\mathsf{get}]^r_1 * \triangleright P$$

View shifts generalize standard implication and satisfy a generalized rule of consequence:

$$P_1 \sqsubseteq P_2 \wedge \{P_2\}\ C\ \{Q_2\} \wedge Q_2 \sqsubseteq Q_1 \Rightarrow \{P_1\}\ C\ \{Q_1\} \qquad (\textsc{AConsq})$$

We can thus use view shifts to factor manipulations of shared resources that do not affect the concrete state. The view-shift region opening rule is very similar to the region opening rule for atomic statements and also requires proving that any updates of the abstract state are permitted and that resources are transformed correctly:

$$\frac{\forall x \in X.\ (x,\ f(x)) \in \overline{T(A)}\ \forall x \in X.\ P * \circledast_{\alpha \in A}[\alpha]^r_{g(\alpha)} * \triangleright I(x) \rangle \sqsubseteq^{\mathcal{E}} Q * \triangleright I(f(x))}{P * \circledast_{\alpha \in A}[\alpha]^r_{g(\alpha)} * \mathsf{region}(X, T, I, r) \rangle \sqsubseteq^{\mathcal{E} \uplus \{r\}} Q * \mathsf{region}(\{f(x)\}, T, I, r)} \qquad (\textsc{VOpen})$$

Just as for atomic commands, we need to ensure that already opened regions cannot be opened again. We thus index view shifts with a region mask annotation ($\mathcal{E}$) that describes the set of regions that may be opened.

Allocation of shared regions is also described using view shifts. To allocate a shared region in initial abstract state x with interpretation map I, we must transfer the resource $I(x)$ to the shared region. This is expressed by the following view-shift axiom:

$$(\forall x.\ \mathsf{stable}(I(x))) \Rightarrow I(x) \sqsubseteq \exists r.\ \mathsf{region}(\{x\}, T, I, r) * \circledast_{\alpha \in A}[\alpha]^r_1 \qquad (\textsc{VAlloc})$$

The resources owned by a shared region must always be stable. This is enforced upon allocation of the region, as expressed by the previous axiom. Upon allocating a shared region, we can take exclusive ownership of any set of actions A on r.

While iCAP supports recursively defined higher-order shared resources, soundness of iCAP depends crucially on the fact that the transition systems associated with each

```
1  {emp}
2  newchan() {
3    chan *x := new(chan);
4    x->flag := 0;
5    {x.flag ↦ 0}
6    // introduce a later using SMono
7    ⟨▷I_b(x, P)(Low)⟩
8    // allocate new channel region in the Low state
9    ⟨∃r. region({Low}, T_b, I_b(x, P), r) ∗ [set]^r_1 ∗ [get]^r_1⟩
10   // duplicate and weaken region assertion
11   ⟨∃r. region({Low}, T_b, I_b(x, P), r) ∗ [set]^r_1 ∗ region({Low, High}, T_b, I_b(x, P), r) ∗ [get]^r_1⟩
12   // definition of recv and send predicates
13   {recv(x, P) ∗ send(x, P)}
14   return x;
15 }
16 {recv(ret, P) ∗ send(ret, P)}
```

Fig. 9. Proof of `newchan()` using simple predicate definitions.

region are *not* recursively defined. This is enforced when allocating a new region. Formally, the VALLOC rule includes a side condition on the labeled transition system, T, which is given as relation on abstract states indexed by an action identifier to enforce this. This side condition is trivially satisfied by every labeled transition system expressible in first-order logic with equality. Since this is the case for all the transition systems employed in this article, we will not go into detail about this technical side condition. We refer the reader to the iCAP article for details [Svendsen and Birkedal 2014a].

Verifying `newchan`. Figure 9 shows a sketch proof of `newchan`. First, the concrete channel data structure is allocated. This allows ownership of the `flag` field to be transferred to a newly allocated channel region, using the VALLOC and ACONSQ rule.

5. SPLITTING CHANNELS

In this section, we extend our proof of the simple implementation to cover *splitting*—intuitively, promised resources can be divided between threads before they are sent. As the simple implementation sequentializes signaling, we leave extension and renunciation to Section 6.

Strengthened abstract specification. To manage promised split resources inside the proof, we use the separating implication, $-\!\!*$. However, the stability assertion stable is difficult to reason about because it is does not distribute with respect to $*$. Explicit stabilization operators, $\lceil - \rceil$ and $\lfloor - \rfloor$, are easier to reason about, and we therefore define a channel specification written in terms of explicit stabilization, from which our client-facing abstract specification can be derived. This strengthened specification is given in Figure 12.

To derive the weaker specification given in Section 3.4 from this stronger one, we define $\mathsf{send}_w(x, P)$ in the weaker specification as $\exists P'.\, \mathsf{send}_s(x, P') \wedge \mathsf{valid}(P \Rightarrow P')$, using the send_s predicate from the stronger specification. The valid predicate allows us to utilize facts about iCAP axioms in our library definitions and specifications. For further discussion, see Appendix C.2.1. Predicates other than send are lifted to the new specification without modification.

$\{\mathsf{send}(\mathtt{x}, P) * \mathsf{stable}(P) * P\}$
```
signal(chan *x) {
```
  // definition of send.
  $\langle \mathsf{region}(\{\mathsf{Low}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * [\mathsf{set}]_1^r * \mathsf{stable}(P) * P \rangle$
  // enter the region.
  $\langle \triangleright(\mathtt{x}.\mathrm{flag} \mapsto 0) * [\mathsf{set}]_1^r * \mathsf{stable}(P) * P \rangle$
  // drop $\triangleright$ *using* $LPoints$
  $\langle \mathtt{x}.\mathrm{flag} \mapsto 0 * [\mathsf{set}]_1^r * \mathsf{stable}(P) * P \rangle$
```
  x->flag := 1;
```
  $\langle \mathtt{x}.\mathrm{flag} \mapsto 1 * [\mathsf{set}]_1^r * \mathsf{stable}(P) * P \rangle$
  // Close region, perform set *transition. Delete* set *token using weakening.*
  $\langle \mathsf{region}(\{\mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) \rangle$
  // stabilise the region's abstract state.
  $\{\mathsf{region}(\{\mathsf{High}, \mathsf{Done}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r)\}$
```
}
```
$\{\mathsf{emp}\}$

Fig. 10. Proof of `signal()` using simple predicate definitions.

Proof body:

$\{\mathsf{recv}(\mathtt{x}, P) * \mathsf{stable}(P)\}$
```
wait(chan *x) {
  int b;
  do {
```
    $\{\mathsf{region}(\{\mathsf{Low}, \mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * [\mathsf{get}]_1^r * \mathsf{stable}(P)\}$
    // case-split on Low / High.
    $\langle (\mathsf{region}(\{\mathsf{Low}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) \vee \mathsf{region}(\{\mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r)) * [\mathsf{get}]_1^r * \mathsf{stable}(P) \rangle$
```
    b = x->flag; // Low & High cases given below.
```
    $\left\langle \begin{array}{l} (\mathsf{region}(\{\mathsf{Low}, \mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * [\mathsf{get}]_1^r \wedge \mathtt{b} = 0) \ \vee \\ (\mathsf{region}(\{\mathsf{Done}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * [\mathsf{get}]_1^r * \mathsf{stable}(P) * P \wedge \mathtt{b} = 1) \end{array} \right\rangle$
```
  } while (b == 0)
} // Delete region assertions using weakening.
```
$\{P\}$

Low case:

$\langle \mathsf{region}(\{\mathsf{Low}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * [\mathsf{get}]_1^r \rangle$
// open region
$\langle \triangleright(\mathtt{x}.\mathrm{flag} \mapsto 0) * [\mathsf{get}]_1^r \rangle$
// drop $\triangleright$ *and read value.*
```
b = x->flag;
```
$\langle \mathtt{x}.\mathrm{flag} \mapsto 0 * [\mathsf{get}]_1^r \wedge \mathtt{b} = 0 \rangle$
// Close region with get *transition.*
// Stabilise region state.
$\left\langle \begin{array}{l} \mathsf{region}(\{\mathsf{Low}, \mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) \\ \qquad * [\mathsf{get}]_1^r \wedge \mathtt{b} = 0 \end{array} \right\rangle$

High case:

$\langle \mathsf{region}(\{\mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * [\mathsf{get}]_1^r \rangle$
// open region
$\langle \triangleright(\mathtt{x}.\mathrm{flag} \mapsto 1 * \mathsf{stable}(P) * P) * [\mathsf{get}]_1^r \rangle$
// drop $\triangleright$*, and read value.*
```
b = x->flag;
```
$\langle \mathtt{x}.\mathrm{flag} \mapsto 1 * \mathsf{stable}(P) * P * [\mathsf{get}]_1^r \wedge \mathtt{b} = 1 \rangle$
// close region with get *transition.*
$\left\langle \begin{array}{l} \mathsf{region}(\{\mathsf{Done}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * \\ \mathsf{stable}(P) * P * [\mathsf{get}]_1^r \wedge \mathtt{b} = 1 \end{array} \right\rangle$

Fig. 11. Proof of `wait()` using simple predicate definitions.

SPECIFICATIONS:

$$\{\mathsf{emp}\}\quad \texttt{i = newchan()}\quad \{\mathsf{recv}(\texttt{i}, P) * \mathsf{send}(\texttt{i}, P)\}$$

$$\{\mathsf{send}(\texttt{i}, P) * \lfloor P \rfloor\}\quad \texttt{signal(i)}\quad \{\mathsf{emp}\}$$

$$\{\mathsf{recv}(\texttt{i}, P)\}\quad \texttt{wait(i)}\quad \{\lceil P \rceil\}$$

$$\left\{\begin{array}{l}\mathsf{send}(\texttt{x}, P) * \\ \circledast_{e \in E}\, e \prec \texttt{x} * \circledast_{l \in L}\, \texttt{x} \prec l\end{array}\right\}\ \texttt{(b,a)=extend(x)}\ \left\{\begin{array}{l}\mathsf{send}(\texttt{b}, Q) * \mathsf{recv}(\texttt{b}, Q) * \mathsf{send}(\texttt{a}, P) \\ * \,\texttt{b} \prec \texttt{a} * \circledast_{e \in E}\, e \prec \texttt{b} * \circledast_{l \in L}\, \texttt{a} \prec l\end{array}\right\}$$

AXIOMS:

$$x \prec y \implies x \prec y * x \prec y$$

$$x \prec y * y \prec z \implies x \prec z$$

$$\{\mathsf{recv}(x, P) * \mathsf{send}(y, Q) * x \prec y\}\quad \langle \texttt{skip} \rangle\quad \{\mathsf{send}(y, P \mathrel{-\!\!*} Q)\}$$

$$\{\mathsf{recv}(a, P) * \lfloor \lceil P \rceil \mathrel{-\!\!*} (P_1 * P_2) \rfloor\}\quad \langle \texttt{skip} \rangle\quad \{\mathsf{recv}(a, P_1) * \mathsf{recv}(a, P_2)\}$$

Fig. 12. Full specification with explicit stabilization.

Function specifications then follow directly from $\mathsf{stable}(P) \Rightarrow (\lfloor P \rfloor \Leftrightarrow P \Leftrightarrow \lceil P \rceil)$.

To derive the weakened renunciation axiom, we reason as follows:

$$\begin{array}{lll}
\{\mathsf{send}_w(x, P * Q) * \mathsf{recv}(y, P) * x \prec y\} & & \\
\iff \{\exists P'.\ \mathsf{send}_s(x, P') * \mathsf{valid}((P * Q) \Rightarrow P') * \mathsf{recv}(y, P) * x \prec y\} & & \text{(Definition)} \\
\iff \{\exists P'.\ \mathsf{send}_s(x, P') * \mathsf{valid}(Q \Rightarrow (P \mathrel{-\!\!*} P')) * \mathsf{recv}(y, P) * x \prec y\} & & \text{(Adjoint)} \\
\quad \langle \texttt{skip} \rangle & & \text{(Renunciation spec)} \\
\quad \{\exists P'.\ \mathsf{send}_s(x, P \mathrel{-\!\!*} P') * \mathsf{valid}(Q \Rightarrow (P \mathrel{-\!\!*} P'))\} & & \text{(Definition)} \\
\iff \{\mathsf{send}_w(Q)\} & &
\end{array}$$

In our strengthened specification, splitting is expressed by the following axiom:

$$\{\mathsf{recv}(a, P) * \lfloor \lceil P \rceil \mathrel{-\!\!*} (P_1 * P_2) \rfloor\}\quad \langle \texttt{skip} \rangle\quad \{\mathsf{recv}(a, P_1) * \mathsf{recv}(a, P_2)\}$$

To derive the weaker splitting axiom, note that if $\mathsf{stable}(P * Q)$, then

$$\top \Rightarrow \lfloor \top \rfloor \Rightarrow \lfloor P * Q \mathrel{-\!\!*} P * Q \rfloor \Rightarrow \lfloor \lceil P * Q \rceil \mathrel{-\!\!*} P * Q \rfloor$$

In the remainder of the article, we verify our implementations against the more general specification given in Figure 12.

Abstract state. To begin the proof, we first define the set of abstract states. In the absence of splitting, the entire promised resource is transferred to a single recipient. For each channel, the resource can therefore be either unsent, sent but not received, or received. In Section 4, we thus introduced three abstract states (Low, High, and Done) to represent these three situations.

Splitting means that promised resources can be logically divided between recipients. The abstract state must therefore also track how the promised resource has been split and (if it has been sent) which recipients have taken ownership. Intuitively, we parameterize the abstract state with sets of propositions describing the splitting. We would then have abstract states $\mathsf{Low}(\mathcal{I})$ and $\mathsf{High}(\mathcal{I})$, with $\mathcal{I} \in \mathcal{P}_{fin}(\mathsf{Prop})$. Each $P \in \mathcal{I}$ represents a promise to a recipient. In the abstract state $\mathsf{Low}(\mathcal{I})$, the whole resource, which has yet to be sent, has been split into a set of promised resources $\mathcal{I}$. In the abstract state $\mathsf{High}(\mathcal{I})$, the entire promised resource has been sent, and the portions still in $\mathcal{I}$ have yet to be received.

In the presence of splitting, $\mathsf{recv}(a, P)$ only confers the right to receive and take ownership of the portion of the resource represented by P. We capture this by indexing the split and receive transition with a proposition describing the associated resource.

We thus have two transitions: send for sending the entire promised resource and change_P for splitting the resource P or taking ownership of P.

$$\begin{aligned}
\mathsf{send} &: \quad \mathsf{Low}(\mathcal{I}) \rightsquigarrow \mathsf{High}(\mathcal{I}) \\
\mathsf{change}_P &: \ \mathsf{Low}(\mathcal{I} \uplus \{P\}) \rightsquigarrow \mathsf{Low}(\mathcal{I} \uplus \{P_1, P_2\}) \\
\mathsf{change}_P &: \ \mathsf{High}(\mathcal{I} \uplus \{P\}) \rightsquigarrow \mathsf{High}(\mathcal{I} \uplus \{P_1, P_2\}) \\
\mathsf{change}_P &: \ \mathsf{High}(\mathcal{I} \uplus \{P\}) \rightsquigarrow \mathsf{High}(\mathcal{I})
\end{aligned}$$

Note that when splitting P using the change_P transition, the transition system does not enforce that $P \mathrel{-\!\!*} P_1 * P_2$. Rather, this will be enforced by the interpretation function for the abstract states.

The transition system described in this section unfortunately cannot be expressed directly in iCAP. This is because iCAP's abstract states and transitions cannot be directly indexed by propositions. It is unclear how this restriction could be lifted in iCAP's step-indexing framework.

Instead, iCAP supports *saved propositions*, an encoding that allows propositions to be associated with identifiers and stored. To formalize the previous transition system, we index states and transitions with these identifiers. This skirts the restrictions imposed by step-indexing and allows the reasoning we want. In the following section, we introduce the necessary logical machinery.

5.1. Saved Propositions

A saved proposition, written $r \overset{\pi}{\Longmapsto} P$, associates an identifier r with a proposition P. By introducing the indirection from identifiers to propositions, we lose some properties. Most importantly, we cannot easily unify saved propositions: given $r \overset{\pi_1}{\Longmapsto} P$ and $r \overset{\pi_2}{\Longmapsto} Q$, in general, it does not hold that $P = Q$. However, saved propositions still satisfy enough properties that we can verify the splitting axiom.

In addition to the identifier r and proposition P, we also have a fractional parameter $\pi \in (0, 1]$, which records how the saved proposition has been shared between threads. In other words, it serves the same role as fractional permissions for heap cells in standard separation logic [Bornat et al. 2005].

We require that saved propositions satisfy the following three properties:

$$\mathsf{emp} \Rightarrow \exists r.\ r \overset{1}{\Longmapsto} P \tag{1}$$

$$r \overset{\pi_1}{\Longmapsto} P * r \overset{\pi_2}{\Longmapsto} P \iff \begin{cases} r \overset{\pi_1+\pi_2}{\Longmapsto} P & \text{if } (\pi_1 + \pi_2) \leq 1 \\ \mathsf{false} & \text{otherwise} \end{cases} \tag{2}$$

$$r \overset{\pi_1}{\Longmapsto} P * r \overset{\pi_2}{\Longmapsto} Q \Rightarrow r \overset{\pi_1}{\Longmapsto} P * r \overset{\pi_2}{\Longmapsto} Q * (\triangleright P \Rightarrow \triangleright Q) \tag{3}$$

Property (1) allows us to create a saved proposition for an arbitrary proposition P. *Property* (2) says that saved propositions are linear, meaning we can split and join them without worrying about unwanted duplication. Observe that the fractions π_1 and π_2 are used to track splitting. *Property* (3) says that holding two saved propositions in the same region allows us to convert from one to another. This is a corrected version of the unification property discussed earlier. The iCAP later operator "$\triangleright$" is needed because we use shared regions internally in the definition of saved propositions.

The following additional property is derivable from Property (3), proved in Lemma A.1.

$$\begin{pmatrix} r \overset{\pi_1}{\Longmapsto} P * r \overset{\pi_2}{\Longmapsto} Q * \\ (X \mathrel{-\!\!*} \triangleright(Q * Y)) * (P \mathrel{-\!\!*} Z) \end{pmatrix} \Rightarrow r \overset{\pi_1}{\Longmapsto} P * r \overset{\pi_2}{\Longmapsto} Q * (X \mathrel{-\!\!*} \triangleright(Z * Y)) \tag{4}$$

This says that we can apply Property (3) inside separating implications. This is useful when modifying a resource embedded into a larger assertion.

We can encode and verify saved propositions as predicates in iCAP—they do not require any extension of the logic. Our encoding is given in Appendix A. In our encoding, region identifiers are used as identifiers for saved propositions—this is because internally saved propositions are encoded by regions.

5.2. Predicate Definitions for send and recv

Once again, we begin by defining the structure of a region. Abstract states are now terms of the form $\mathsf{Low}(\mathcal{I})$ and $\mathsf{High}(\mathcal{I})$, where $\mathcal{I}$ is a finite set of identifiers in $\mathcal{P}_{fin}(\mathsf{RId})$. We use LoHi to stand for either Low or High. The $\mathcal{I}$ parameter represents the set of outstanding obligations, that is, the resources that other threads expect to be supplied. As described earlier, we use saved propositions to give an interpretation to these sets of region identifiers. If we have the abstract state $\mathsf{Low}(\mathcal{I})$ or $\mathsf{High}(\mathcal{I})$, then each $i \in \mathcal{I}$ corresponds to a resource promised to some thread. To find out *what* resource P is expected, we examine the associated saved proposition $i \overset{\pi}{\Longmapsto} P$.

Actions in the transition relation T_{m} are of the form send and $\mathsf{change}(i)$, where i is the identifier for a saved proposition. The first action, send, sets the flag and simply moves from Low to High. The second, $\mathsf{change}(i)$, is the action both of taking the resource associated to region i when the flag is high and of splitting the resource associated with identifier i to the resource required by i_1 and i_2. These new identifiers i_1/i_2 can be chosen arbitrarily: however, the invariant mapping I_{m} ensures they are properly associated with saved propositions.

$$
\begin{aligned}
T_{\mathsf{m}}(\mathsf{send}) &\triangleq \{(\mathsf{Low}(\mathcal{I}), \mathsf{High}(\mathcal{I}))\} \\
T_{\mathsf{m}}(\mathsf{change}(i)) &\triangleq \{(\mathsf{High}(\mathcal{I} \uplus \{i\}), \mathsf{High}(\mathcal{I}))\} \cup \{(\mathsf{Low}(\mathcal{I} \uplus \{i\}), \mathsf{Low}(\mathcal{I} \uplus \{i_1, i_2\}))\} \\
&\qquad \cup \{(\mathsf{High}(\mathcal{I} \uplus \{i\}), \mathsf{High}(\mathcal{I} \uplus \{i_1, i_2\}))\}
\end{aligned}
$$

The invariants associated with Low and High are defined as follows:

$$
\begin{aligned}
I_{\mathsf{m}}(x, r, P)(\mathsf{Low}(\mathcal{I})) &\triangleq x.\mathsf{flag} \mapsto 0 * \mathsf{waiting}(P, \mathcal{I}) * \mathsf{changes}_r(\mathcal{I}) \\
I_{\mathsf{m}}(x, r, P)(\mathsf{High}(\mathcal{I})) &\triangleq x.\mathsf{flag} \mapsto 1 * \mathsf{ress}(\mathcal{I}) * \mathsf{changes}_r(\mathcal{I}) \\
&\textit{where} \\
\mathsf{waiting}(P, \mathcal{I}) &\triangleq \exists Q : \mathcal{I} \to \mathsf{Prop}.\ \lfloor (\triangleright P) \mathrel{-\!\!*} \triangleright \mathop{\circledast}_{i \in \mathcal{I}} Q(i) \rfloor * \mathop{\circledast}_{i \in \mathcal{I}} i \overset{1/2}{\Longmapsto} Q(i) \\
\mathsf{ress}(\mathcal{I}) &\triangleq \mathop{\circledast}_{i \in \mathcal{I}} \exists R.\ (i \overset{1/2}{\Longmapsto} R) * \lceil \triangleright R \rceil \\
\mathsf{changes}_r(\mathcal{I}) &\triangleq \mathop{\circledast}_{i \notin \mathcal{I}} [\mathsf{change}(i)]^r_1
\end{aligned}
$$

The definitions here use three auxiliary predicates: $\mathsf{waiting}$, standing for resources that have been promised but not supplied; ress, standing for resources once they have been supplied; and $\mathsf{changes}$, standing for change tokens for unused identifiers. The set $\mathsf{changes}$ can be seen as a "library" of tokens that are not currently in use and are currently not used by any thread. Having this library of tokens allows new saved propositions to be added when splitting.

The representation of Low consists of the flag, change tokens, and $\mathsf{waiting}$ predicate. $\mathsf{waiting}(P, \mathcal{I})$ requires the existence of a mapping Q from region identifiers to propositions representing obligations to other threads. The obligations for different threads are tied together using fractional saved propositions $i \overset{1/2}{\Longmapsto} Q(i)$. The assertion $\lfloor (\triangleright P) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}} Q(i) \rfloor$ means that supplying the resource P will satisfy each obligation $Q(i)$.

```
1  {emp}
2  chan *newchan() {
3    chan *x := new(chan);
4    x->flag := 0;
5   {x.flag ↦ 0}
6    // Create floor assertion
7   {x.flag ↦ 0 ∗ ⌊(▷P) −∗ (▷P)⌋}
8    skip; // Create saved proposition
9   {∃r1. r1 ⇛^1 P ∗ x.flag ↦ 0 ∗ ⌊(▷P) −∗ (▷P)⌋}
10   // Create channel region
11  {∃r1, r2. r1 ⇛^1/2 P ∗ creg(x, r2, P, {Low({r1})}) ∗ [send]^r2_1 ∗ [change(r1)]^r2_1}
12   // Satisfy the send predicate
13  {∃r1, r2. send(x, P) ∗ r1 ⇛^1/2 P ∗ creg(x, r2, P, {Low({r1})}) ∗ [change(r1)]^r2_1}
14 } // Satisfy the recv predicate
15  {send(x, P) ∗ recv(x, P)}
```

Fig. 13. Proof of `newchan()` w.r.t. the full specification.

The representation of the High state consists of the flag and change tokens and the ress predicate. ress($\mathcal{I}$) pairs together fractional saved propositions, $i \overset{1/2}{\Longrightarrow} R$, with resources $\lceil \triangleright R \rceil$. The other half of each saved proposition is held by the thread that has been promised the resource through the recv predicate (see later). This ensures that all threads that have been promised resources can claim them.

We use a shorthand for the region assertion in our definitions and proofs:

$$\mathsf{creg}(x, r, P, S) \triangleq \mathsf{region}(S, T_{\mathsf{m}}, I_{\mathsf{m}}(x, r, P), r)$$

The definition of the send predicate is now straightforward. It asserts that the region is in a Low state and holds the unique permission to perform the send action.

$$\mathsf{send}(x, P) \triangleq \exists r.\ \mathsf{creg}(x, r, P, \{\mathsf{Low}(\mathcal{I}) \mid \mathsf{true}\}) * [\mathsf{send}]^r_1$$

The definition of the recv(x, Q) predicate is more complex. It includes $r' \overset{1/2}{\Longrightarrow} Q$, half the permission on the saved proposition Q. It also asserts that r' is one of the identifiers recorded in the region. This ensures that the resource retrieved from the shared region is the correct one, that is, the one that was promised (see the next section for the reasoning steps involved).

$$\mathsf{recv}(x, Q) \triangleq \exists R, r, r'.\ \mathsf{creg}(x, r, R, \{\mathsf{LoHi}(\mathcal{I}) \mid r' \in \mathcal{I}\}) * r' \overset{1/2}{\Longrightarrow} Q * [\mathsf{change}(r')]^r_1$$

5.3. Proofs of `newchan()`, `signal()`, `wait()`, and the Splitting Axiom

Proving `newchan`. The proof of `newchan` (Figure 13) allocates a region containing the concrete channel state. Most steps in the proof are straightforward. The challenging ones are line 6 (creating the stabilized assertion) and line 10 (the view shift that creates the region itself). In line 6, we need the following implication:

$$\mathsf{emp} \Rightarrow \lfloor (\triangleright P) \mathrel{-\!\!*} (\triangleright P) \rfloor$$

To show this holds, we observe that for any X, $\mathsf{emp} \Longrightarrow X \mathrel{-\!\!*} X$, and that emp is always stable. The implication then follows by monotonicity of explicit stabilization brackets, $(A \Rightarrow B) \Rightarrow (\lfloor A \rfloor \Rightarrow \lfloor B \rfloor)$.

```
1  {send(x, P) ∗ ⌊P⌋}
2  signal(chan *x) {
3    {creg(x, r, P, {Low(I′) | true}) ∗ [send]^r_1 ∗ ⌊P⌋}
4      // open the region. Low(I) is an arbitrary member of {Low(I′) | true}.
5      ⟨▷I_m(x, r, P)(Low(I)) ∗ [send]^r_1 ∗ ⌊P⌋⟩
6      // invariant definition.
7      ⟨[send]^r_1 ∗ ⌊P⌋ ∗ ▷(x.flag ↦ 0 ∗ waiting(P, I) ∗ changes_r(I))⟩
8      // pull out points-to using LBin and LPoints.
9      ⟨[send]^r_1 ∗ ⌊P⌋ ∗ x.flag ↦ 0 ∗ ▷(waiting(P, I) ∗ changes_r(I))⟩
10     x->flag := 1; // rewrite flag, drop ▷ using AFrame
11     ⟨[send]^r_1 ∗ ⌊P⌋ ∗ x.flag ↦ 1 ∗ waiting(P, I) ∗ changes_r(I)⟩
12     // Lemma 5.1, add a ▷ using SMono.
13     ⟨[send]^r_1 ∗ ▷(x.flag ↦ 1 ∗ ress(I) ∗ changes_r(I))⟩
14     // Invariant definition.
15     ⟨[send]^r_1 ∗ ▷I_m(x, r, P)(High(I))⟩
16     // close the region using send transition.
17   {[send]^r_1 ∗ creg(x, r, P, {High(I′) | true})}
18 } // delete redundant assertions using weakening.
19 {emp}
```

Fig. 14. Proof of `signal()` w.r.t. the full specification.

Line 10 requires us to prove the following view shift:

$$\begin{array}{rl} r_1 \overset{1/2}{\Longrightarrow} & P * \mathrm{x} \mapsto 0 * \lfloor (\triangleright P) \mathrel{-\!\!*} (\triangleright P) \rfloor \\ \sqsubseteq & \exists r_2.\ \mathsf{creg}(\mathrm{x}, r_2, P, \{\mathsf{Low}(\{r_1\}, \emptyset)\}) * [\mathsf{send}]^{r_2}_1 * [\mathsf{change}(r_1)]^{r_2}_1 \end{array}$$

To prove this, we appeal to iCAP's VAlloc rule, which controls construction of new regions (see Section 4.1 for its definition). Intuitively, this requires that the resources available satisfy the initial abstract state, defined by I_{m}. Note that all the change tokens apart from $\mathsf{change}(r_1)$ are stored in the token library predicate changes, held inside the new region.

Proving `signal`. The proof of `signal` (Figure 14) works by opening the channel region (line 4), merging in the supplied resource $\lfloor P \rfloor$ to give the promised resources (line 12), and closing the region again (line 16). When we close the region, we also need to confirm that the transition from $\mathsf{Low}(\mathcal{I})$ to $\mathsf{High}(\mathcal{I})$ is allowed, but this is simple: it's the only transition associated to send by T_{m}. The trickiest step is the merging of the resource into the region (line 12), embodied by the following lemma.

Lemma 5.1. $\lfloor P \rfloor * \mathsf{waiting}(P, \mathcal{I}) \sqsubseteq \mathsf{ress}(\mathcal{I})$

Proof.

$$\begin{array}{ll} & \lfloor P \rfloor * \mathsf{waiting}(P, \mathcal{I}) \\ \sqsubseteq & \lfloor \triangleright P \rfloor * \exists Q : \mathcal{I} \to \mathsf{Prop}.\ \lfloor \triangleright P \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}}.\ Q(i) \rfloor * (\circledast_{i \in \mathcal{I}}.\ i \overset{1/2}{\Longrightarrow} Q(i)) \\ \sqsubseteq & \exists Q : \mathcal{I} \to \mathsf{Prop}.\ \lfloor \triangleright \circledast_{i \in \mathcal{I}}.\ Q(i) \rfloor * (\circledast_{i \in \mathcal{I}}.\ i \overset{1/2}{\Longrightarrow} Q(i)) \\ \sqsubseteq & \exists Q : \mathcal{I} \to \mathsf{Prop}.\ (\circledast_{i \in \mathcal{I}}.\ \lceil \triangleright Q(i) \rceil) * (\circledast_{i \in \mathcal{I}}.\ i \overset{1/2}{\Longrightarrow} Q(i)) \\ \sqsubseteq & (\circledast_{i \in \mathcal{I}}.\ \exists R.\ i \overset{1/2}{\Longrightarrow} R * \lceil \triangleright R \rceil) \\ \sqsubseteq & \mathsf{ress}(\mathcal{I}) \end{array}$$

```
1  {recv(x, P)}
2  wait(chan *x) {
3   {∃R, r, r'. r' ⟹^{1/2} P * [change(r')]^r_1 * creg(x, r, R, {LoHi(I') | r' ∈ I'})}
4    // Open the region. Only consider the High case.
5     ⟨r' ∈ I ∧ r' ⟹^{1/2} P * [change(r')]^r_1 * ▷I_m(x, r, R)(High(I))⟩
6      // Apply invariant definition
7      // Pull out points-to using LBin and LPoints
8     ⟨r' ∈ I ∧ r' ⟹^{1/2} P * [change(r')]^r_1 * x.flag ↦ 1 * ▷(ress(I) * changes_r(I))⟩
9      assume(x->flag == 1) // drop ▷ using AFrame, push in [change(r')] perm.
10    ⟨r' ∈ I ∧ r' ⟹^{1/2} P * x.flag ↦ 1 * ress(I) * changes_r(I \ {r'})⟩
11     // Lemma 5.2, add ▷ using SMono
12    ⟨▷⌈P⌉ * ▷(x.flag ↦ 1 * ress(I \ {r'}) * changes_r(I \ {r'}))⟩
13     // Invariant definition
14    ⟨▷⌈P⌉ * ▷I_m(x, r, R)(High(I \ {r'}))⟩
15   // close the region using change(r') transition.
16  {▷⌈P⌉ * ∃R, r. creg(x, r, R, {High(I) | true})}
17   // Delete redundant assertions using weakening.
18  {▷⌈P⌉}
19 }
```

Fig. 15. Proof of `wait()` w.r.t. the full specification.

All of the steps in this proof are in fact standard implications (this is also true of all the other view-shift lemmas unless stated explicitly). To prove the second step, we appeal to the fact that $\lfloor - \rfloor$ is semidistributive over the separating conjunction, $\lfloor A \rfloor * \lfloor B \rfloor \Rightarrow \lfloor A * B \rfloor$, and *modus ponens* for the separating implication, $A * (A \mathrel{-\!\!*} B) \Rightarrow B$. The third step follows from the fact that $\lceil - \rceil$ is weaker than $\lfloor - \rfloor$, and $\lceil - \rceil$ is semidistributive over the separating conjunction, $\lceil A * B \rceil \Rightarrow \lceil A \rceil * \lceil B \rceil$. □

Proving `wait`. In the proof of `wait` (Figure 15), we open the shared region (line 4), extract the required resource (line 11), and close the region again (line 15). For simplicity, we assume that the abstract state is High; if not, the algorithm spins doing nothing until it is the case. Each promised resource is associated with a region identifier i in the set $\mathcal{I}$; removing the resource is modeled abstractly by removing i. This abstract transition is allowed by the [change] permission. The key step in the proof is extracting the resource (line 11), embodied by the following lemma.

LEMMA 5.2. $r \overset{1/2}{\Longmapsto} P * \mathsf{ress}(\mathcal{I}) \wedge r \in \mathcal{I} \;\sqsubseteq\; \mathsf{ress}(\mathcal{I} \setminus \{r\}) * \triangleright \lceil P \rceil$

PROOF.
$$
\begin{aligned}
& r \overset{1/2}{\Longmapsto} P * \mathsf{ress}(\mathcal{I} \uplus \{r\}) \\
& \sqsubseteq\; r \overset{1/2}{\Longmapsto} P * \mathsf{ress}(\mathcal{I}) * \exists R.\, r \overset{1/2}{\Longmapsto} R * \lceil \triangleright R \rceil && \text{(Definition of ress)} \\
& \sqsubseteq\; r \overset{1/2}{\Longmapsto} P * \mathsf{ress}(\mathcal{I}) * \exists R.\, r \overset{1/2}{\Longmapsto} R * \lceil \triangleright P \rceil && \text{(Property 3, mono of } \lceil - \rceil) \\
& \sqsubseteq\; \mathsf{ress}(\mathcal{I}) * \triangleright \lceil P \rceil && (\lceil \triangleright P \rceil \Rightarrow \triangleright \lceil P \rceil) \quad \square
\end{aligned}
$$

Proving the splitting axiom. In our specification, splitting must always be associated with a `skip` step. It should now be clear why we need this: a `skip` step allows us to

$\{\mathsf{recv}(\mathtt{x}, P) * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor\}$

$\{\exists R, r, r'.\, r' \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(r')]^r_1 * \mathsf{creg}(\mathtt{x}, r, R, \{\mathsf{LoHi}(\mathcal{I}') \mid r' \in \mathcal{I}'\}) * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor\}$

// Open the region using an arbitrary state containing r'

$\langle \exists R, r, r'.\, r' \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(r')]^r_1 * \triangleright I_{\mathsf{m}}(\mathtt{x}, r, R)(\mathsf{LoHi}(\mathcal{I} \uplus \{r'\})) * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor \rangle$

skip *// Use AFrame to remove $\triangleright$*

$\langle \exists R, r, r'.\, r' \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(r')]^r_1 * I_{\mathsf{m}}(\mathtt{x}, r, R)(\mathsf{LoHi}(\mathcal{I} \uplus \{r'\})) * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor \rangle$

// Lemma 5.5

$\langle \exists R, r, r_1, r_2.\, r_1 \overset{1/2}{\Longrightarrow} P_1 * r_1 \overset{1/2}{\Longrightarrow} P_2 * [\mathsf{change}(r_1)]^r_1 * [\mathsf{change}(r_2)]^r_1 * I_{\mathsf{m}}(\mathtt{x}, r, R)(\mathsf{LoHi}(\mathcal{I} \uplus \{r_1, r_2\})) \rangle$

// Close the region using $\mathsf{change}(r')$ *transition.*

$\{\exists R, r, r_1, r_2.\, r_1 \overset{1/2}{\Longrightarrow} P_1 * r_1 \overset{1/2}{\Longrightarrow} P_2 * [\mathsf{change}(r_1)]^r_1 * [\mathsf{change}(r_2)]^r_1 * \mathsf{creg}(\mathtt{x}, r, R, \{\mathsf{LoHi}(\mathcal{I}') \mid r_1, r_2 \in \mathcal{I}'\})\}$

// Definition of predicates.

$\{\mathsf{recv}(\mathtt{x}, P_1) * \mathsf{recv}(\mathtt{x}, P_2)\}$

Fig. 16. Proof outline for splitting axiom.

enter the shared region and get rid of $\triangleright$. We present the proof outline in Figure 16. The core of the proof is two lemmas that express splitting in the Low and High cases.

LEMMA 5.3 (LOW SPLITTING).

$$\begin{aligned} & r \overset{1/2}{\Longrightarrow} P * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor * \mathsf{waiting}(R, \mathcal{I} \uplus r) \\ \sqsubseteq\ & \exists r_1, r_2.\, r_1 \overset{1/2}{\Longrightarrow} P_1 * r_2 \overset{1/2}{\Longrightarrow} P_2 * \mathsf{waiting}(R, \mathcal{I} \uplus \{r_1, r_2\}) \end{aligned}$$

PROOF.

$$\begin{aligned} & r \overset{1/2}{\Longrightarrow} P * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor * \mathsf{waiting}(R, \mathcal{I} \uplus r) \\ & \qquad \text{(Predicate definitions, extract saved propositions)} \\ \sqsubseteq\ & \exists Q'.\ r \overset{1/2}{\Longrightarrow} P * r \overset{1/2}{\Longrightarrow} Q' * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor * \\ & \exists Q\colon \mathcal{I} \to \mathsf{Prop}.\ \lfloor (\triangleright R) \mathrel{-\!\!*} \triangleright Q' * \triangleright \circledast_{i \in \mathcal{I}}.\ Q(i) \rfloor * (\circledast_{i \in \mathcal{I}}.\ i \overset{1/2}{\Longrightarrow} Q(i)) \\ & \qquad \text{(Unify } P/Q', \text{ apply } P \implies \lceil P \rceil) \\ \sqsubseteq\ & \lfloor \lceil \triangleright P \rceil \mathrel{-\!\!*} (\triangleright P_1) * (\triangleright P_2) \rfloor * \\ & \exists Q\colon \mathcal{I} \to \mathsf{Prop}.\ \lfloor (\triangleright R) \mathrel{-\!\!*} \lceil \triangleright P \rceil * \triangleright \circledast_{i \in \mathcal{I}}.\ Q(i) \rfloor * (\circledast_{i \in \mathcal{I}}.\ i \overset{1/2}{\Longrightarrow} Q(i)) \\ & \qquad \text{(Modus ponens, create saved props } r_1/r_2) \\ \sqsubseteq\ & \exists r_1, r_2.\, r_1 \overset{1}{\Rrightarrow} P_1 * r_2 \overset{1}{\Rrightarrow} P_2 \wedge r_1, r_2 \notin \mathcal{I} * \\ & \exists Q\colon \mathcal{I} \to \mathsf{Prop}.\ \lfloor (\triangleright R) \mathrel{-\!\!*} (\triangleright P_1) * (\triangleright P_2) * \triangleright \circledast_{i \in \mathcal{I}}.\ Q(i) \rfloor * (\circledast_{i \in \mathcal{I}}.\ i \overset{1/2}{\Longrightarrow} Q(i)) \\ & \qquad \text{(Push saved props into } \mathcal{I}) \\ \sqsubseteq\ & \exists r_1, r_2.\, r_1 \overset{1/2}{\Longrightarrow} P_1 * r_2 \overset{1/2}{\Longrightarrow} P_2 * \\ & \exists Q\colon \mathcal{I} \uplus \{r_1, r_2\} \to \mathsf{Prop}.\ \lfloor (\triangleright R) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I} \uplus \{r_1, r_2\}}.\ Q(i) \rfloor * (\circledast_{i \in \mathcal{I} \uplus \{r_1, r_2\}}.\ i \overset{1/2}{\Longrightarrow} Q(i)) \\ & \qquad \text{(Predicate definitions)} \\ \sqsubseteq\ & \exists r_1, r_2.\, r_1 \overset{1/2}{\Longrightarrow} P_1 * r_2 \overset{1/2}{\Longrightarrow} P_2 * \mathsf{waiting}(R, \mathcal{I} \uplus \{r_1, r_2\}) \quad \square \end{aligned}$$

LEMMA 5.4 (HIGH SPLITTING).

$$\begin{aligned} & r \overset{1/2}{\Longrightarrow} P * \lfloor \ulcorner P \urcorner \mathrel{-\!\!*} P_1 * P_2 \rfloor * \mathsf{ress}(\mathcal{I} \uplus r) \\ \sqsubseteq\ & \exists r_1, r_2.\ r_1 \overset{1/2}{\Longrightarrow} P_1 * r_2 \overset{1/2}{\Longrightarrow} P_2 * \mathsf{ress}(\mathcal{I} \uplus \{r_1, r_2\}) \end{aligned}$$

PROOF.

$$\begin{aligned} & r \overset{1/2}{\Longrightarrow} P * \lfloor \ulcorner P \urcorner \mathrel{-\!\!*} P_1 * P_2 \rfloor * \mathsf{ress}(\mathcal{I} \uplus r) \\ \sqsubseteq\ & (\ulcorner P \urcorner \mathrel{-\!\!*} P_1 * P_2) * \mathsf{ress}(\mathcal{I}) * \triangleright \ulcorner P \urcorner && \text{(Lemma 5.2)} \\ \sqsubseteq\ & ((\triangleright \ulcorner P \urcorner) \mathrel{-\!\!*} (\triangleright P_1) * (\triangleright P_2)) * \mathsf{ress}(\mathcal{I}) * \triangleright \ulcorner P \urcorner && \text{(SMono, dist } \triangleright \text{ over } \mathrel{-\!\!*}) \\ \sqsubseteq\ & \mathsf{ress}(\mathcal{I}) * (\triangleright P_1) * (\triangleright P_2) && \text{(Modus ponens)} \\ \sqsubseteq\ & \mathsf{ress}(\mathcal{I}) * (\triangleright P_1) * (\triangleright P_2) * \exists r_1, r_2.\ r_1 \overset{1}{\Longrightarrow} P_1 * r_2 \overset{1}{\Longrightarrow} P_2 && \text{(Property 1)} \\ \sqsubseteq\ & \mathsf{ress}(\mathcal{I} \uplus \{r_1, r_2\}) * \exists r_1, r_2.\ r_1 \overset{1/2}{\Longrightarrow} P_1 * r_2 \overset{1/2}{\Longrightarrow} P_2 && \text{(Sublemma)} \end{aligned}$$

The last step in building the new ress predicate consists of two applications of the following sublemma:

$$\begin{aligned} & \mathsf{ress}(\mathcal{I}) * \triangleright P * r \overset{1}{\Longrightarrow} P \\ \sqsubseteq\ & (\circledast_{i \in \mathcal{I}} \exists Q.\ i \overset{1/2}{\Longrightarrow} Q * \ulcorner \triangleright Q \urcorner) * \ulcorner \triangleright P \urcorner * r \overset{1}{\Longrightarrow} P \\ \sqsubseteq\ & (\circledast_{i \in \mathcal{I} \uplus \{r\}} \exists Q.\ i \overset{1/2}{\Longrightarrow} Q * \ulcorner \triangleright Q \urcorner) * r \overset{1/2}{\Longrightarrow} P \\ \sqsubseteq\ & \mathsf{ress}(\mathcal{I} \uplus \{r\}) * r \overset{1/2}{\Longrightarrow} P \quad \square \end{aligned}$$

These two lemmas are combined as follows.

LEMMA 5.5.

$$\begin{aligned} & r' \overset{1/2}{\Longrightarrow} P * I_{\mathsf{m}}(x, r, R)(\mathsf{LoHi}(\mathcal{I} \uplus \{r'\})) * [\mathsf{change}(r')]^1_r * \lfloor \ulcorner P \urcorner \mathrel{-\!\!*} P_1 * P_2 \rfloor \\ & \quad \sqsubseteq \\ & \exists r_1, r_2.\ [\mathsf{change}(r_1)]^1_r * r_1 \overset{1/2}{\Longrightarrow} P_1 * [\mathsf{change}(r_2)]^1_r * r_2 \overset{1/2}{\Longrightarrow} P_2 * \\ & \qquad \triangleright I_{\mathsf{m}}(x, r, R)(\mathsf{LoHi}(\mathcal{I} \uplus \{r_1, r_2\})) \end{aligned}$$

PROOF. We case-split on whether LoHi is Low or High. The two proofs are given by Lemmas 5.3 and 5.4 and some rearrangement of the change permissions. □

6. CHAINS AND RENUNCIATION

The simple barrier implementation verified in Sections 4 and 5 does not consider an order of channels. In this section, we verify an implementation that supports chains of channels and early renunciation. Recall that renunciation is expressed by the following axiom:

$$\{\mathsf{recv}(x, P) * \mathsf{send}(y, Q) * x \prec y\} \quad \langle \texttt{skip} \rangle \quad \{\mathsf{send}(y, P \mathrel{-\!\!*} Q)\}$$

In the chain of channels, $x \prec y$ states that x is earlier than y. This axiom states that the required resource Q can be partially or totally satisfied using the earlier promised resource P (if $P \Leftrightarrow Q$, then $(P \mathrel{-\!\!*} Q) \Leftrightarrow \mathsf{emp}$, i.e., no more resources need to be supplied by the client in order for the signal to be set).

In our new implementation, channels are arranged into a chain represented by a linked list. Calls to `signal` do not block and can complete in any order consistent with the specification. However, renunciation means later channels may depend on resources promised earlier in the chain. To ensure renounced resources are available, `wait` checks all predecessors in the chain. The implementation is defined as follows:

```
struct chan {
   int flag;
   chan *prev;
}

signal(chan *x) {
   x->flag = 1;
}

wait(chan *x) {
   chan *c = x;
   while(c != NULL) {
    while(c->flag == 0) skip;
    c = c->prev;
   }
}
```

```
chan *newchan() {
   chan *x = new(chan);
   x->flag = 0;
   x->prev = NULL;
   return x;
}

extend(chan *x) {
   chan *z = new(chan);
   z->flag = 0;
   z->prev = x->prev;
   x->prev = z;
   return (z,x);
}
```

Calling `signal` sets the current channel flag to 1, then exits immediately. When `wait` is called, it blocks until every bit earlier in the chain is set. To do this, it follows `prev` fields, waiting for each `flag` field before accessing the preceding location. To add extra nodes to the chain, `extend` allocates a new channel and then inserts it immediately before the channel passed as an argument.

6.1. Abstract State

Our fundamental approach remains the same as for the previous proof. That is, shared channels are represented by abstract states, for example, $\mathsf{High}(\mathcal{I})$. Resource obligations are represented by sets of identifiers that are tied to saved propositions, for example, the members of $\mathcal{I}$. Modifications to the shared channel by the thread and environment are represented by transitions over these abstract states. Predicates send, recv are defined as constraints on the abstract state.

The difference with the new implementation is that operations access multiple channels along the chain. As a result, the abstract state cannot be a single channel: instead, it is an ordered sequence of channel nodes from the set CNode:

$$[\mathrm{node}(x),\ \mathrm{node}(y),\ \mathrm{node}(z), \ldots]$$

Here x, y, z are addresses, and channel nodes are ordered $z \prec y \prec x$. Note that the list is reversed with respect to chain order: nodes closer to the tail precede than those closer to the head. (We do this because pointers in the underlying list go in this direction.)

Each CNode plays a similar role to an individual channel in the previous section. Therefore, each has a state High / Low and a set $\mathcal{I}$ representing splittings of the promised resource. In addition, to handle renunciation, each node records a set $\mathcal{W}$ of identifiers for resources promised to it through renunciation. Formally, channel nodes have the following structure:

$$\begin{array}{lll}
\mathsf{CNode} \triangleq \langle & & \\
\quad loc \in \mathsf{Addr}, & \textit{(physical address)} & \\
\quad res \in \mathsf{RId}, & \textit{(region ID for sent resource)} & \\
\quad \mathcal{I} \in \mathcal{P}_{fin}(\mathsf{RId}), & \textit{(region IDs for promised resources)} & \\
\quad flg \in \{\mathsf{High}, \mathsf{Low}\}, & \textit{(flag status)} & \\
\quad \mathcal{W} \in \mathcal{P}_{fin}(\mathsf{RId}), & \textit{(region IDs for earlier renounced resources)} & \\
\rangle &
\end{array}$$

Each CNode represents one channel in the chain, so the abstract state of the barrier is an *abstract chain* consisting of a finite sequence in CNode^+.

We assume that the CNode locations in any abstract chain are pairwise distinct. Thus, where convenient, we sometimes treat an abstract chain as a function from locations to tuples: that is, $rs(x)$ gives some tuple $(r, \mathcal{I}, f, \mathcal{W})$. Given a CNode s, we sometimes write $s.fg$, $s.\mathcal{I}$, and so forth to identify the appropriate components of the tuple. We use 0 and 1 to represent the Low and High flag state, respectively.

Most of the abstract transitions are just operations on individual channels in the chain: these are liftings of the single-channel operations defined in the previous section. For example, the operation set just involves rewriting the flag field for some channel from Low to High, leaving the state otherwise unchanged.

Renunciation is the most interesting case as it involves *two* channels in the chain. The abstract specification for renunciation uses a resource from a recv predicate to satisfy a send predicate—the former must be earlier in the chain than the latter. Correspondingly, the renun abstract transition copies an identifier from the earlier channel's promise set to the later channel's renunciation set. For example:

$$[\ldots \langle x, i, \mathcal{I}, 0, \mathcal{W}\rangle \ldots\ \langle y, i', \mathcal{I}' \uplus \{r\}, 0, \mathcal{W}'\rangle \ldots] \quad \leadsto \\ [\ldots \langle x, i'', \mathcal{I}, 0, \mathcal{W} \uplus \{r\}\rangle \ldots\ \langle y, i', \mathcal{I}' \cup \{r\}, 0, \mathcal{W}'\rangle \ldots]$$

In the proof, there will exist saved propositions $r \overset{1}{\Rightarrow\!\!\!|}\ P$ and $i \overset{1}{\Rightarrow\!\!\!|}\ Q$—the former records the promised resource from y, while the latter records the resource required to signal the channel x. Intuitively, after this transition, the resource P can no longer be claimed: it will be used to satisfy channel x. This corresponds to the send predicate disappearing in the renunciation axiom. The identifier i also changes to i''—the associated saved proposition will now be $i'' \overset{1}{\Rightarrow\!\!\!|}\ (P \mathrel{-\!\!*} Q)$, reflecting the fact that P no longer needs to be supplied.

This transition is purely abstract in the same way that splitting is: nothing has changed in the concrete representation. All that has changed is the way that the threads agree to use resources.

Chain extension also involves multiple channels. The abstract specification takes as its precondition a send and two sets E and L representing nodes earlier and later in the chain:

$$\left\{\begin{array}{l}\mathsf{send}(\mathtt{x}, P) * \\ \circledast_{e\in E}\, e \prec \mathtt{x} * \circledast_{l\in L}\, \mathtt{x} \prec l\end{array}\right\}\ (\mathtt{b}, \mathtt{a}){=}\mathtt{extend}(\mathtt{x})\ \left\{\begin{array}{l}\mathsf{send}(\mathtt{b}, Q) * \mathsf{recv}(\mathtt{b}, Q) * \mathsf{send}(\mathtt{a}, P) \\ * \mathtt{b} \prec \mathtt{a} * \circledast_{e\in E}\, e \prec \mathtt{b} * \circledast_{l\in L}\, \mathtt{a} \prec l\end{array}\right\}$$

In the abstract state, this corresponds to the following transition:

$$[\ldots \textit{ nodes in } L \ldots\ \langle \mathtt{x}, i, \mathcal{I}, 0, \mathcal{W}\rangle\ \ldots \textit{ nodes in } E \ldots] \quad \leadsto \\ [\ldots \textit{ nodes in } L \ldots\ \langle \mathtt{a}, i, \mathcal{I}, 0, \mathcal{W}\rangle,\ \langle \mathtt{b}, i', \mathcal{I}', 0, \emptyset\rangle\ \ldots \textit{ nodes in } E \ldots]$$

The new node is inserted immediately preceding the parameter x, with the remaining structure of the abstract state remaining unchanged.

6.2. Definitions and Predicates

Abstract state predicates. The introduction of renunciation makes it important that resources used later in the chain are available as promised earlier in the chain. More concretely, given an abstract chain $x \cdot xs$, every identifier in the set $x.\mathcal{W}$ must be

available from some node in xs (i.e., in some set $\mathcal{I}$):

$$\mathsf{available}([\,]) \triangleq \emptyset, \qquad \mathsf{available}(s \cdot xs) \triangleq (\mathsf{available}(xs) \setminus s.\mathcal{W}) \uplus s.\mathcal{I}$$

$$\mathsf{wf}([\,]) \triangleq \mathsf{true}, \qquad \mathsf{wf}(s \cdot xs) \triangleq \mathsf{wf}(xs) \wedge s.\mathcal{W} \subseteq \mathsf{available}(xs) \wedge s.\mathcal{I} \cap s.\mathcal{W} = \emptyset \wedge \forall s' \in xs.\ s.\mathcal{I} \cap s'.\mathcal{I} = \emptyset \wedge s.\mathcal{W} \cap s'.\mathcal{W} = \emptyset$$

The predicate available constructs the set of identifiers that have been promised earlier in the chain, and that have not been taken by some other earlier channel. Well formedness, wf, then requires that the set of identifiers available from earlier in the chain includes those required by the current channel.

We also define two predicates over abstract chains, ctrue and cconf. The first asserts that all the flags in the abstract chain have been set, while the second furthermore asserts that all sets of waited-for renounced resources are empty. When the latter holds, any resources promised by this node must be available for retrieval.

$$\mathsf{ctrue}(rs) \triangleq \forall e \in rs.\ e.\mathit{flg} = 1$$
$$\mathsf{cconf}(rs) \triangleq \forall e \in rs.\ e.\mathit{flg} = 1 \wedge e.\mathcal{W} = \emptyset$$

Finally, we use $rs_1 \xrightarrow{pr}{}^{*} rs_2$ to denote that the abstract chain rs_2 can be derived from rs_1 by canceling out renounced resources with the corresponding promises. This is used in the proof to show that renounced resources can eventually be satisfied with real resources, once all the flags in the chain have been set. $\xrightarrow{pr}{}^{*}$ is the transitive-reflexive closure of a relation that cancels a single promise to a later node using a renounced resource from an earlier node. For example:

$$[\ldots\langle x, i, \mathcal{I}, 1, \mathcal{W} \uplus \{r\}\rangle, \ldots\ \langle y, i', \mathcal{I}' \uplus \{r\}, 1, \mathcal{W}'\rangle\ \ldots] \xrightarrow{pr} [\ldots\ \langle x, i, \mathcal{I}, 1, \mathcal{W}\rangle, \ldots\ \langle y, i', \mathcal{I}', 1, \mathcal{W}'\rangle\ \ldots]$$

We define the relation formally as follows:

$$rs \xrightarrow{pr} rs' \triangleq \exists x, y, r.\ r \in rs(x).\ \mathcal{W} \wedge r \in rs(y).\ \mathcal{I} \wedge (x, y) \in \mathsf{ord}(rs) \wedge rs' = (rs \blacktriangleleft_x \blacktriangleleft_{\mathcal{W}} (\bullet \setminus \{r\})) \blacktriangleleft_y \blacktriangleleft_{\mathcal{I}} (\bullet \setminus \{r\})$$

We write $(x, y) \in \mathsf{seq}(rs)$ to say that the two addresses x and y appear adjacent in the sequence rs, and $(x, y) \in \mathsf{ord}(rs)$ to say just that they are ordered in rs.

Lenses. The notation $\blacktriangleleft$ is a *lens* allowing a single field of a chain to be updated without modifying the remainder of the chain. We use lenses to make our definitions more compact. Lenses are a notation borrowed from functional programming that we use to update one field of an object while preserving the remainder of it. By object, we mean either a tuple or a map—we treat tuples as maps from field names to values. Recall that we can also treat abstract chains as maps from locations to tuples as convenient.

We define the lens notation as follows. In the following, let x be the tuple/map we wish to update. Let i/j be values in the domain (i.e., field names for a tuple). Let f be an expression with $\bullet$ standing for the value to be updated. Then the lens notation is defined as follows:

$$x \blacktriangleleft_i f \triangleq x[i \mapsto (f[x(i)/\bullet])] \qquad x \blacktriangleleft_i \blacktriangleleft_j f \triangleq x[i \mapsto (x(i) \blacktriangleleft_j f)]$$

On the left, the value associated with index/field name f in object x is updated to $f[x(i)/\bullet]$. On the right, we show we can stack lenses, allowing us to update fields deeper inside the object—here we update field j of field i of object x.

$$
\begin{aligned}
\mathsf{renun}_c(x, rs, rs') &\triangleq rs(x).\mathit{flg} = 0 \wedge \exists r', w.\, rs' = (rs \blacktriangleleft_x \blacktriangleleft_{res} r') \blacktriangleleft_x \blacktriangleleft_{\mathcal{W}} (\bullet \uplus w)\\
\mathsf{set}_c(x, rs, rs') &\triangleq rs(x).\mathit{flg} = 0 \wedge rs' = rs \blacktriangleleft_x \blacktriangleleft_{\mathit{flg}} (1)\\
\mathsf{ext}_c(x, rs, rs') &\triangleq rs = (rs_1 \cdot a \cdot rs_2) \wedge rs' = (rs_1 \cdot a \cdot b \cdot rs_2) \wedge\\
&\quad\ a.\mathit{loc} = x \wedge a.\mathit{flg} = 0 \wedge b.\mathit{flg} = 0 \wedge b.\mathcal{W} = \emptyset\\
\mathsf{split}_c(x, r, rs, rs') &\triangleq r \in rs(x).\mathcal{I} \wedge \exists r_2, r_3.\, rs' = rs \blacktriangleleft_x \blacktriangleleft_{\mathcal{I}} ((\bullet \setminus \{r\}) \uplus \{r_2, r_3\})\\
\mathsf{sat}_c(rs, rs') &\triangleq rs = (rs_1 \cdot rs_2) \wedge rs' = (rs_1 \cdot rs_2') \wedge \mathsf{cconf}(rs_2') \wedge rs_2 \overset{pr}{\rightarrow}{}^{*}\ rs_2'\\
\mathsf{get}_c(x, r, rs, rs') &\triangleq rs = (rs_1 \cdot rs(x) \cdot rs_2) \wedge \mathsf{cconf}(rs(x) \cdot rs_2) \wedge rs' = rs \blacktriangleleft_x \blacktriangleleft_{\mathcal{I}} (\bullet \setminus \{r\})
\end{aligned}
$$

$$
\begin{aligned}
T_{\mathsf{c}}(\mathsf{send}(x)) &\triangleq \{(a, b) \mid \mathsf{wf}(b) \wedge (\mathsf{renun}_c(x, a, b) \vee \mathsf{set}_c(x, a, b) \vee \mathsf{ext}_c(x, a, b))\}\\
T_{\mathsf{c}}(\mathsf{change}(x, r)) &\triangleq \{(a, b) \mid \mathsf{wf}(b) \wedge (\mathsf{split}_c(x, r, a, b) \vee \mathsf{sat}_c(a, b) \vee \mathsf{get}_c(x, r, a, b))\}
\end{aligned}
$$

Fig. 17. Definition of T_{c}, the transition relation for the chained channel implementation.

To give an example, in the preceding section, we had the following lens expression:

$$(rs \blacktriangleleft_x \blacktriangleleft_{\mathcal{W}} (\bullet \setminus \{r\})) \blacktriangleleft_y \blacktriangleleft_{\mathcal{I}} (\bullet \setminus \{r\})$$

Note that here we are using $\mathcal{W}/\mathcal{I}$ as identifiers for particular tuple components. This expression denotes the chain rs, with identifier r removed from both the set of renounced resources at CNode x and the set of promised resources at CNode y.

Predicate definitions. As usual, we begin by defining the structure of the shared region. Abstract states have the form $\mathsf{Chain}(rs)$, where rs is an abstract chain. Actions have the form $\mathsf{send}(x)$ and $\mathsf{change}(x, r)$, where x is an address, and r a region identifier. The transition relation T_{c} is defined in Figure 17. We assume that physical addresses are used uniquely, so where convenient, we use chains as finite functions of type

$$\mathsf{Addr} \overset{\mathit{fin}}{\rightharpoonup} (\mathsf{RId} \times \mathcal{P}(\mathsf{RId}) \times \{\mathsf{High}, \mathsf{Low}\} \times \mathcal{P}(\mathsf{RId}))$$

The transition relation defines six kinds of transitions in Figure 17. For send, we have renunciation, which adds an element to $\mathcal{W}$; setting the flag; and extending the chain, which creates a new CNode b. For change, we have splitting; satisfying the renounced resource set, which sets $\mathcal{W}$ to $\emptyset$ and pulls the resources out of earlier chain elements; and pulling out a resource.

To translate from an abstract chain to a concrete invariant, we define three predicates: chainds, chainres, and unused (defined in Figure 18). The predicate chainds represents the linked list underpinning the implementation. Each link in the chain has the appropriate prev and flag values set determined by the corresponding CNode in the chain.

The predicate chainres represents the resources that are communicated through the chain. The key predicate is resource, which ties together a set of promised resources $\mathcal{I}$ and a set of resources waited for $\mathcal{W}$. Note that the set $\mathcal{W}$ of resources waited for includes both renounced resources *and* the resource supplied by the preceding channel—these are unioned by chainres. The core of the resource predicate is the following assertion:

$$\lfloor (\triangleright \circledast_{w \in \mathcal{W}} . \, R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}} . \, \lceil Q(i) \rceil \rfloor$$

Here Q and R map identifiers to propositions. Leaving aside explicit stabilization and $\triangleright$, this assertion has a straightforward intuition: supplying all the resources waited for (those with identifiers in $\mathcal{W}$) results in the resources promised (those with identifiers in $\mathcal{I}$).

When there are no resources waited for, that is, $\mathcal{W} = \emptyset$, the resource predicate can be simplified to just the promised resources:

$$\begin{aligned}
\mathsf{chainds}(x \cdot y \cdot rs) &\triangleq x.loc \mapsto \{\mathrm{prev} = y.loc;\ \mathrm{flag} = x.\mathit{flg}\} * \mathsf{chainds}(y \cdot rs)\\
\mathsf{chainds}(x \cdot \mathit{null}) &\triangleq x.loc \mapsto \{\mathrm{prev} = \mathsf{NULL};\ \mathrm{flag} = x.\mathit{flg}\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{resource}(\mathcal{I}, \mathcal{W}) \triangleq\ & \exists Q\colon \mathcal{I} \to \mathsf{Prop},\ R\colon \mathcal{W} \to \mathsf{Prop}.\\
& \circledast_{i \in \mathcal{I}}.\, i \overset{1/2}{\Longmapsto} Q(i) * \circledast_{w \in \mathcal{W}}.\, w \overset{1/2}{\Longmapsto} R(w)\\
& * \lfloor (\triangleright \circledast_{w \in \mathcal{W}}.\, R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}}.\, \lceil Q(i) \rceil \rfloor\\
\mathsf{chainres}(x \cdot rs) &\triangleq \mathsf{resource}(x.\mathcal{I}, x.\mathcal{W} \uplus \{x.res \mid \neg x.\mathit{flg}\}) * \mathsf{chainres}(rs)\\
\mathsf{chainres}(\mathit{null}) &\triangleq \mathsf{emp}
\end{aligned}$$

$$\begin{aligned}
\mathsf{uS}(rs) &\triangleq \{x \mid (x, _, _, 0, _) \in rs\}\\
\mathsf{uC}(rs) &\triangleq \{(x, i) \mid (x, _, \mathcal{I}, _, _) \in rs \wedge i \in \mathcal{I} \wedge \neg\exists (y, _, _, _, \mathcal{W}) \in rs.\, i \in \mathcal{W}\}\\
\mathsf{unused}(r, rs) &\triangleq (\circledast x \notin \mathsf{uS}(rs).\, [\mathsf{send}(x)]^r_1) * (\circledast (x, r') \notin \mathsf{uC}(rs).\, [\mathsf{change}(x, r')]^r_1)
\end{aligned}$$

Fig. 18. Predicates used in defining the state of a region.

Lemma 6.1. $\mathsf{resource}(\mathcal{I}, \emptyset) \quad \sqsubseteq \quad \circledast_{i \in \mathcal{I}}.\, \exists Q : \mathsf{Prop}.\, i \overset{1/2}{\Longmapsto} Q(i) * \lceil \triangleright Q(i) \rceil$

The unused predicate stands for the set of unused permissions (similar to changes in the previous proof). We define this using $\mathsf{uS}(rs)$, the set of used send permissions, and $\mathsf{uC}(rs)$, the set of used change permissions.

The representation function for the region, I_c, is defined as follows:

$$I_c(r)(\mathsf{Chain}(rs)) \triangleq \mathsf{chainds}(rs) * \mathsf{chainres}(rs) * \mathsf{unused}(r, rs)$$

As before, we use a shorthand for the region assertion in our definitions and proofs:

$$\mathsf{oreg}(r, S) \triangleq \mathsf{region}(S, T_c, I_c(r), r)$$

We can now define the send, recv, and ordering predicates:

$$\begin{aligned}
\mathsf{send}(x, P) \triangleq\ & \exists r_1, r_2.\, \mathsf{oreg}(r_1, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(x) = (r_2, _, 0, _)\})\\
& * r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(x)]^{r_1}_1\\
\mathsf{recv}(x, P) \triangleq\ & \exists r_1, r_2.\, \mathsf{oreg}(r_1, \{\mathsf{Chain}(rs) \mid rs(x) = (_, \mathcal{I}, _, _) \wedge r_2 \in \mathcal{I} \wedge \mathsf{wf}(rs)\})\\
& * r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(x, r_2)]^{r_1}_1\\
x \prec y \triangleq\ & \exists r.\, \mathsf{oreg}(r, \{\mathsf{Chain}(rs) \mid (y, x) \in \mathsf{ord}(rs)\})
\end{aligned}$$

The structure of the send and recv predicates is similar to the unchained proof. Both send and recv contain saved propositions for their proposition parameter P. The predicate definitions ensure the identifier for this saved proposition is embedded into the abstract chain correctly. Meanwhile, the $\prec$ predicate is a straightforward lifting of the ord predicate on abstract chains.

6.3. Proving `Signal`, `Wait`, and `Extend`

The majority of the proof concerns manipulations of resource obligations, rather than reads and writes to the underlying data structure. To help with proof clarity, as far as possible we factor reads and writes into small, separate specifications.

Proving `signal`. The sketch proof is shown in Figure 19—it is similar in structure to the one in Section 5.3. The main additional challenge is to show that resources

1 $\{\mathsf{send}(\mathtt{x}, P) * \lfloor P \rfloor\}$
2 *// Definition of* send.
3 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \lfloor P \rfloor * \\ \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(\mathtt{x}) = (r_2, \mathcal{I}, 0, \mathcal{W})\})\end{array}\right\}$
4 *// Enter the region.*
5 $\left\langle\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \lfloor P \rfloor * \\ \triangleright \left(\mathsf{chainds}(rs) * \mathsf{chainres}(rs) * \mathsf{unused}(r_1, rs) \wedge \mathsf{wf}(rs) \wedge rs(\mathtt{x}) = (r_2, \mathcal{I}, 0, \mathcal{W})\right)\end{array}\right\rangle$
6 *// Split up using chain axioms.*
7 $\left\langle\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \lfloor P \rfloor * \\ \triangleright \left(\begin{array}{l}\exists rs_1, rs_2, a.\, \mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 0\} * \mathsf{chainds}(rs_2) * \\ \mathsf{chainres}(rs) * \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge a = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W})\end{array}\right)\end{array}\right\rangle$
8 `x->flag = 1;` *// Set the flag, drop the* $\triangleright$ *using Aframe.*
9 $\left\langle\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \lfloor P \rfloor * \\ \exists rs_1, rs_2, a.\, \mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 1\} * \mathsf{chainds}(rs_2) * \\ \mathsf{chainres}(rs) * \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge a = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W})\end{array}\right\rangle$
10 *// Pull out* resource *predicate from* chainres *predicate.*
11 $\left\langle\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \lfloor P \rfloor * \\ \exists rs_1, rs_2, a.\, \mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 1\} * \mathsf{chainds}(rs_2) \\ \qquad * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r_2\}) * \mathsf{chainres}(rs_2) \\ \qquad * \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge a = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W})\end{array}\right\rangle$
12 *// Apply Lemma* 6.2 *-- use* $\lfloor P \rfloor$ *to remove* r_2 *from the second parameter of* resource.
13 $\left\langle\begin{array}{l}\exists r_1, r_2.\, [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \\ \exists rs_1, rs_2, a.\, \mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 1\} * \mathsf{chainds}(rs_2) \\ \qquad * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}, \mathcal{W}) * \mathsf{chainres}(rs_2) \\ \qquad * \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge a = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W})\end{array}\right\rangle$
14 *// Collapse the chain, add* $\triangleright$.
15 $\left\langle\begin{array}{l}\exists r_1, r_2. \\ \triangleright\, (\mathsf{chainds}(rs') * \mathsf{chainres}(rs') * \mathsf{unused}(r_1, rs') \wedge rs' = rs[\mathtt{x} \mapsto (r_2, \mathcal{I}, 1, \mathcal{W})])\end{array}\right\rangle$
16 *// Close the region using transition* set.
17 *// Well-formed structure of chain hasn't changed.*
18 $\left\{\exists r_1, r_2.\, \mathsf{oreg}(r_1, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(\mathtt{x}) = (r_2, \mathcal{I}, 1, \mathcal{W})\})\right\}$
19 *// Delete all assertions using weakening.*
20 $\{\mathsf{emp}\}$

Fig. 19. Sketch proof for `signal` with out-of-order signaling.

are supplied to the appropriate point in the chain. To do this, we use the following lemma, which says that supplying the resource $\lfloor P \rfloor$ and an associated saved proposition removes the need to supply r. This is then sufficient to allow the flag to be set.

LEMMA 6.2. $r \overset{1/2}{\Longmapsto} P * \lfloor P \rfloor * \mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r\}) \sqsubseteq \mathsf{resource}(\mathcal{I}, \mathcal{W})$

PROOF. Given in Appendix B. □

In the proof of `signal`, this lemma is used to show that the appropriate resource has been supplied (Figure 19, line 12). By factoring logical resource transfer away from the physical signaling, we simplify the proof structure considerably.

The rest of the proof consists of manipulating predicates. We pull out the node associated with `x` and set the flag (lines 1–8). Once the resource has been supplied on line 12, the remainder of the proof closes the region again.

Proving `wait`. The sketch proof is given in Figure 20. The three most important steps are checking that all preceding flags in the chain are set (lines 5–12), checking that renounced resources have been supplied (line 15), and retrieving the resource from the chain (line 20). The last two of these require helper lemmas, given later.

Resources that are renounced earlier in the chain can be used to satisfy required resources later in the chain. These resources are represented by the set $\mathcal{W}$ in the abstract state of a cnode. Renounced resources need not be supplied when `signal` is called, but they must be available before `wait` returns. To ensure this, the implementation of `wait` checks all the preceding flags in the chain. Once all preceding flags are set, all the resources should be available. However, proving this is subtle, because renounced resources may themselves be satisfied by resources renounced earlier in the chain.

To establish that the required resources are available, we use the following lemma. This says that a chainres predicate for a chain where all the flags are set can be transformed into one where pending resources have been resolved (asserted by ctrue and cconf, respectively).

LEMMA 6.3. $$\mathsf{chainres}(rs) \wedge \mathsf{wf}(rs) \wedge \mathsf{ctrue}(rs) \sqsubseteq \exists rs'.\ \mathsf{chainres}(rs') \wedge \mathsf{cconf}(rs') \wedge rs \xrightarrow{pr}{}^{*} rs' \wedge \mathsf{wf}(rs')$$

PROOF. Given in Appendix B. □

We apply this lemma on line 15 of the sketch proof.

Once we've established that the resources are available, we use the following lemma to extract the appropriate resource from the resource predicate:

LEMMA 6.4. $$\mathsf{resource}(\mathcal{I} \uplus r_2, \emptyset) * r_2 \overset{1/2}{\Longmapsto} P \sqsubseteq \mathsf{resource}(\mathcal{I}, \emptyset) * \lceil \triangleright P \rceil$$

PROOF. Given in Appendix B. □

This lemma says that, given resource and an identifier r_2 in $\mathcal{I}$ such that all required resources are available, the resource $\triangleright P$ associated with r_2 can be retrieved. We apply this lemma on line 20 of the sketch proof.

Proving `extend`. The sketch proof is given in Figure 21. The key steps in the proof are creating a new node to add to the chain (lines 3–9), stitching the new node into the chain itself (line 12), then satisfying the required invariants for the region (lines 18–22).

It is important that new saved propositions are *fresh*—that is, their identifiers have not been used elsewhere in the chain. We use the following lemma to show that new identifiers are fresh:

LEMMA 6.5.

$$\{\mathsf{oreg}(r, \mathcal{T}) * r' \overset{1}{\Longmapsto} P\} \quad \langle \mathtt{skip} \rangle \quad \{\mathsf{oreg}(r, \mathcal{T} \cap \{\mathsf{Chain}(rs) \mid r' \notin rs\}) * r' \overset{1}{\Longmapsto} P\}$$

PROOF. Each identifier r'' used in rs is associated with a fractional saved proposition $r'' \overset{1/2}{\Longmapsto} P$. We case-split on the finite set of possible equalities and appeal to the linearity of saved propositions (Property (2)). The `skip` is required by iCAP because we access the internal state of the region r. □

The following lemma uses this freshness property, along with the freshness of allocated locations, to show that we can retrieve the required permissions from unused (the "library" of unused permissions). We use this lemma on line 16 of the sketch proof.

$\{\mathsf{recv}(\mathtt{x}, P)\}$
`wait(x){`
`  chan *c = x;`
`  // Definition of` recv.
$\left\{\exists r_1, r_2.\, \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid \exists \mathcal{I}'.\, rs(\mathtt{x}) = (\_, \mathcal{I}', \_, \_) \wedge r_2 \in \mathcal{I}' \wedge \mathsf{wf}(rs)\}) \right.$
$\left. \qquad * \, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]^{r_1}_1 \wedge \mathtt{c} = \mathtt{x} \right\}$
`  while(c != NULL) {`
$\left\{\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]^{r_1}_1 * \right.$
$\left. \mathsf{oreg}\left(r_1, \left\{\mathsf{Chain}(rs) \;\middle|\; \begin{array}{r} \exists \mathcal{I}'.\, \mathsf{wf}(rs) \wedge \exists i.\, rs[i] = (\mathtt{x}, \_, \mathcal{I}' \uplus r_2, \_, \_) \wedge \\ \exists m < \mathrm{len}(rs).\, rs[m] = (\mathtt{c}, \_, \_, \_, \_) \wedge \\ \forall j.\, i \le j < m \implies rs[j] = (\_, \_, \_, 1, \_) \end{array}\right\}\right)\right\}$
`    while(!c->flag) skip;`
`    c = c->prev;`
`  }`
`  // Flags up to x are now set. Stable because flags cannot get unset.`
$\left\{\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]^{r_1}_1 * \right.$
$\left. \mathsf{oreg}\left(r_1, \left\{\mathsf{Chain}(rs) \;\middle|\; \begin{array}{l} \exists \mathcal{I}'.\, \mathsf{wf}(rs) \wedge \exists i.\, rs[i] = (\mathtt{x}, \_, \mathcal{I}' \uplus r_2, \_, \_) \wedge \\ \forall j.\, i \le j < \mathrm{len}(rs) \implies rs[j] = (\_, \_, \_, 1, \_) \end{array}\right\}\right)\right\}$
$\langle \mathtt{skip} \rangle$; `// Enter the region and drop` $\triangleright$ `using` $AFrame$.
$\left\langle \exists rs_1, rs_2, a.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]^{r_1}_1 * \mathsf{chainds}(rs) * \mathsf{chainres}(rs) * \mathsf{unused}(r_1, rs) \wedge \right.$
$\left. \mathsf{wf}(rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge a = (\mathtt{x}, r, \mathcal{I}' \uplus r_2, \_, \mathcal{W}) \wedge \mathsf{ctrue}(a \cdot rs_2) \right\rangle$
`  // Apply Lemma 6.3 to convert from` ctrue `to` cconf`, denoting that the`
`  // resource` P `is now available to be claimed.`
$\left\langle \exists rs_1, rs_2, rs_2', a.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]^{r_1}_1 * \right.$
$\left. \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1 \cdot rs_2') * \mathsf{unused}(r_1, rs) * \mathsf{wf}(rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge \right.$
$\left. a = (\mathtt{x}, r, \mathcal{I}' \uplus r_2, \_, \mathcal{W}) \wedge \mathsf{cconf}(rs_2') \wedge a \cdot rs_2 \overset{pr}{\rightarrow}{}^{*}\, rs_2' \right\rangle$
`// Pull out the` resource `predicate for chain node x.`
$\left\langle \exists rs_1, rs_2, rs_2', a.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]^{r_1}_1 * \right.$
$\left. \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}' \uplus r_2, \emptyset) * \mathsf{chainres}(rs_2') * \mathsf{unused}(r_1, rs) \wedge \right.$
$\left. \mathsf{wf}(rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge a = (\mathtt{x}, r, \mathcal{I}' \uplus r_2, \_, \mathcal{W}) \wedge \mathsf{cconf}(rs_2') \wedge a \cdot rs_2 \overset{pr}{\rightarrow}{}^{*}\, rs_2' \right\rangle$
`  // Apply Lemma 6.4 to retrieve the resource` $\lceil \triangleright P \rceil$.
$\left\langle \lceil \triangleright P \rceil * \exists rs_1, rs_2, rs_2', a.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]^{r_1}_1 * \right.$
$\left. \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}', \emptyset) * \mathsf{chainres}(rs_2') * \mathsf{unused}(r_1, rs) \wedge \right.$
$\left. \mathsf{wf}(rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge a = (\mathtt{x}, r, \mathcal{I}' \uplus r_2, \_, \mathcal{W}) \wedge \mathsf{cconf}(rs_2') \wedge a \cdot rs_2 \overset{pr}{\rightarrow}{}^{*}\, rs_2' \right\rangle$
`  // Close the region, use transitions` sat `and` get.
$\left\{\exists r_1, r_2.\, \lceil \triangleright P \rceil * \right.$
$\left. \mathsf{oreg}\left(r_1, \left\{\mathsf{Chain}(rs') \;\middle|\; \begin{array}{l} \mathsf{wf}(rs') \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge a \cdot rs_2 \overset{pr}{\rightarrow}{}^{*}\, rs_2' \wedge \\ \mathsf{cconf}(rs_2') \wedge rs' = (rs_1 \cdot rs_2') \blacktriangleleft_x \blacktriangleleft_{\mathcal{I}} (\bullet \setminus \{r_2\}) \end{array}\right\}\right)\right\}$
`}// Use` $\lceil \triangleright P \rceil \Rightarrow \triangleright \lceil P \rceil$`, and delete redundant assertions using weakening.`
$\{\triangleright \lceil P \rceil\}$

Fig. 20. Sketch proof for `wait` with out-of-order signaling.

1 $\left\{\mathsf{send}(\mathtt{x}, P) * \circledast_{e\in E}\, e \prec \mathtt{x} * \circledast_{l\in L}\, \mathtt{x} \prec l\right\}$
2 `extend(x){ // Frame off order predicates`
3 `  chan *z = new(chan); z->flag = 0;`
4 `  z->prev = x->prev; // Stable as thread holds exclusive` $[\mathsf{send}(x)]$ `permission.`
5 $\left\{\begin{array}{l}\exists r_1, r_2, x'.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \mathtt{z} \mapsto \{\mathrm{prev} = x'; \mathrm{flag} = 0\} * \\ \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(\mathtt{x}) = (r_2, \_, 0, \_) \wedge (\mathtt{x}, x') \in \mathsf{seq}(rs) \wedge \mathtt{z} \notin rs\})\end{array}\right\}$
6 `  skip; skip; // Make saved props, use Lemma` 6.5 `to show they are fresh.`
7 $\left\{\begin{array}{l}\exists r_1, r_2, x', r', r''.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \mathtt{z} \mapsto \{\mathrm{prev} = x'; \mathrm{flag} = 0\} * r' \overset{1}{\Longmapsto} Q * r'' \overset{1}{\Longmapsto} Q * \\ \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(\mathtt{x}) = (r_2, \_, 0, \_) \wedge (\mathtt{x}, x') \in \mathsf{seq}(rs) \wedge \mathtt{z}, r', r'' \notin rs\})\end{array}\right\}$
8 `  // Create a node in the chain.`
9 $\left\{\begin{array}{l}\exists r_1, r_2, x', r', r''.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \\ \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(\mathtt{x}) = (r_2, \_, 0, \_) \wedge (\mathtt{x}, x') \in \mathsf{seq}(rs) \wedge \mathtt{z}, r', r'' \notin rs\}) * \\ \qquad r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q * \mathtt{z} \mapsto \{\mathrm{prev} = x'; \mathrm{flag} = 0\} * \mathsf{resource}(\{r''\}, \{r'\})\end{array}\right\}$
10 `  // Enter the region, split the chain up.`
11 $\left\langle\begin{array}{l}\exists r_1, r_2, x', r', r'', h, rs_1, rs_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \\ \quad \triangleright \begin{pmatrix}\mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = x'; \mathrm{flag} = 0\} * \mathsf{chainds}(rs_2) * \mathsf{chainres}(rs) * \\ \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot e \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge e = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W}) \wedge \mathtt{z}, r', r'' \notin rs\end{pmatrix} * \\ \qquad r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q * \mathtt{z} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 0\} * \mathsf{resource}(\{r''\}, \{r'\})\end{array}\right\rangle$
12 `  x->prev = z; // Stitch the new cnode into the chain using LPoints / AFrame.`
13 $\left\langle\begin{array}{l}\exists r_1, r_2, x', r', r'', h, rs_1, rs_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \\ \mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = \mathtt{z}; \mathrm{flag} = 0\} * \mathsf{chainds}(rs_2) * \mathsf{chainres}(rs) * \\ \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot e \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge e = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W}) \wedge \mathtt{z}, r', r'' \notin rs * \\ r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q * \mathtt{z} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 0\} * \mathsf{resource}(\{r''\}, \{r'\})\end{array}\right\rangle$
14 `  // Fold the` chainds `predicate back up.`
15 $\left\langle\begin{array}{l}\exists r_1, r_2, x', r', r'', h, rs_1, rs_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q \\ \mathsf{chainds}(rs_1 \cdot e \cdot e' \cdot rs_2) * \mathsf{chainres}(rs_1 \cdot e \cdot e' \cdot rs_2) * \mathsf{unused}(r_1, rs) \wedge \\ rs = rs_1 \cdot e \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge e = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W}) \wedge e' = (\mathtt{z}, r', \{r''\}, 0, \emptyset) \wedge \mathtt{z}, r', r'' \notin rs\end{array}\right\rangle$
16 `  // Get` $[\mathsf{send}(z)] * [\mathsf{change}(z, r'')]$ `from` unused(r_1, rs) `using Lemma` 6.6.
17 $\left\langle\begin{array}{l}\exists r_1, r_2, r', r'', h, rs_1, rs_2. \\ \mathsf{chainds}(rs_1 \cdot e \cdot e' \cdot rs_2) * \mathsf{chainres}(rs_1 \cdot e \cdot e' \cdot rs_2) * \mathsf{unused}(r_1, rs_1 \cdot e \cdot e' \cdot rs_2) \\ \qquad \wedge rs = rs_1 \cdot e \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge e = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W}) \wedge e' = (\mathtt{z}, r', \{r''\}, 0, \emptyset) \\ \wedge\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q * [\mathsf{send}(\mathtt{z})]^{r_1}_1 * [\mathsf{change}(\mathtt{z}, r'')]^{r_1}_1\end{array}\right\rangle$
18 `  // Close the region using the` ext `transition.`
19 $\left\{\begin{array}{l}\exists r_1, r_2, r', r'', h. \\ \mathsf{oreg}\Big(r_1, \Big\{\mathsf{Chain}(rs') \;\Big|\; rs' = rs_1 \cdot (\mathtt{x}, r_2, \_, 0, \_) \cdot (\mathtt{z}, r', \mathcal{I}', 0, \_) \cdot rs_2 \wedge r'' \in \mathcal{I}' \wedge \mathsf{wf}(rs')\Big\}\Big) \\ \qquad \wedge\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q * [\mathsf{send}(\mathtt{z})]^{r_1}_1 * [\mathsf{change}(\mathtt{z}, r'')]^{r_1}_1\end{array}\right\}$
20 `  return (z,x);`
21 `} // Frame on order predicates. Fold invariant into predicates.`
22 $\left\{\mathsf{send}(\mathtt{z}, Q) * \mathsf{recv}(\mathtt{z}, Q) * \mathsf{send}(\mathtt{x}, P) * \mathtt{z} \prec \mathtt{x} * \circledast_{e\in E}\, e \prec \mathtt{z} * \circledast_{l\in L}\, \mathtt{x} \prec l\right\}$

Fig. 21. Sketch proof of `extend` with out-of-order signaling.

LEMMA 6.6.

$$rs = rs_1 \cdot (x, r_2, \mathcal{I}, 0, \mathcal{W}) \cdot rs_2 \wedge \mathsf{unused}(r_1, rs) \wedge z, r', r'' \notin rs$$
$$\sqsubseteq \quad \mathsf{unused}(r_1, rs_1 \cdot (x, r_2, \mathcal{I}, 0, \mathcal{W}) \cdot (z, r', \{r''\}, 0, \emptyset) \cdot rs_2) * [\mathsf{send}(z)]^{r_1}_1 * [\mathsf{change}(z, r'')]^{r_1}_1$$

PROOF. Appeal to the definition of unused. □

On line 18, we close the region. The resulting chain is well formed because the new region has no elements in its renunciation set $\mathcal{W}$, and the rest of the chain is preserved. The chain is stable because we hold the send permission on x and z, meaning these channels cannot be extended or renounced.

Proving `newchan`. Omitted: this proof is similar to `extend`. However, it is simpler: we only need to construct a new point in the chain, and not update the existing chain to take account of it.

6.4. Proving Renunciation and Splitting Axioms

Renunciation. The axiom is defined as follows:

$$\{\mathsf{recv}(x, P) * \mathsf{send}(y, Q) * x \prec y\} \quad \langle \texttt{skip} \rangle \quad \{\mathsf{send}(y, P \mathrel{-\!\!*} Q)\}$$

The sketch proof is given in Figure 22. Internally, each predicate contains a view on the same shared region, and the first step of the proof consists of conjoining these three views to give a single stable view on the shared structure (line 3). The remaining steps are supplying the renounced resource to the shared region (line 13) and closing the region to give a new send predicate (line 15).

In order to conjoin the regions arising from the send, recv, and order predicates, they need to operate over the same region. Although the predicates do not expose region names, we know from order predicates that all of the regions share common elements in their chain addresses. We therefore use an extra lemma to show that pairs of such regions must be the same:

LEMMA 6.7.

$$\{\mathsf{oreg}(r, \{\mathsf{Chain}(rs) \mid x \in rs\}) * \mathsf{oreg}(r', \{\mathsf{Chain}(rs') \mid x \in rs'\})\} \quad \langle \texttt{skip} \rangle \quad \{r = r'\}$$

PROOF. Given in Appendix B. □

We use this lemma on line 3, Figure 22. The conjoined region that arises from this lemma (line 6) is stable because elements cannot be reordered with respect to each other once they are in the chain, and because exclusive [chain] and [send] permissions are held for x and y, respectively.

When we push the renounced resource into the resource predicate (line 13), we use the following lemma to show that the renunciation set $\mathcal{W}$ is updated appropriately:

LEMMA 6.8.
$$\mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r\}) * r \overset{1/2}{\Longrightarrow} Q * r' \overset{1/2}{\Longrightarrow} P$$
$$\sqsubseteq \quad \exists r''.\ \mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r', r''\}) * r'' \overset{1/2}{\Longrightarrow} (P \mathrel{-\!\!*} Q)$$

PROOF. Given in Appendix B. □

Note that the identifier r used for sending resources is replaced with a fresh identifier r'' because the associated invariant is changed from Q to $P \mathrel{-\!\!*} Q$. Internally, this corresponds to deleting one saved proposition through weakening, and then creating another.

On line 15, we close the region. The resulting chain is well formed because the identifier we selected was previously unused for renunciation—we get this from the definition of unused. Furthermore, the remainder of the chain stays the same. The resulting chain is trivially stable because the exclusive [send] permission is held.

1 $\{\mathsf{recv}(x,P) * \mathsf{send}(y,Q) * x \prec y\}$
2 $\langle \mathtt{skip} \rangle$; // *Definition of* recv, send, $\prec$. *Use Lemma* 6.7 *to establish*
3 // *that all three predicates refer to the same region.*
4 $\left\{ \begin{array}{l} \exists r, r_2, r_3.\, \mathsf{oreg}(r, \{\mathsf{Chain}(rs) \mid (y,x) \in \mathsf{ord}(rs)\}) * \\ \left(\begin{array}{c} \mathsf{oreg}(r, \{\mathsf{Chain}(rs) \mid rs(x) = (\_, \mathcal{I}, \_, \_) \wedge r_2 \in \mathcal{I} \wedge \mathsf{wf}(rs)\}) \\ * \, r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(x, r_2)]^r_1 \end{array} \right) * \\ \left(\begin{array}{c} \mathsf{oreg}\,(r, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(y) = (r_3, \_, 0, \_)\}) \\ * \, r_3 \overset{1/2}{\Longrightarrow} Q * [\mathsf{send}(y)]^r_1 \end{array} \right) \end{array} \right\}$
5 // *Conjunction on regions. Stable because of the permissions held.*
6 $\left\{ \begin{array}{l} \exists r, r_2, r_3.\, r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(x, r_2)]^r_1 * r_3 \overset{1/2}{\Longrightarrow} Q * [\mathsf{send}(y)]^r_1 * \\ \mathsf{oreg}\left(r, \left\{ \mathsf{Chain}(rs) \;\middle|\; \begin{array}{l} rs = rs_1 \cdot e \cdot rs_2 \cdot e' \cdot rs_3 \wedge \mathsf{wf}(rs) \wedge \\ e = (y, r_3, \_, 0, \_) \wedge e' = (x, \_, \mathcal{I}, \_, \_) \wedge r_2 \in \mathcal{I} \end{array} \right\} \right) \end{array} \right\}$
7 $\langle \mathtt{skip} \rangle$; // *Open the region, drop* $\triangleright$ *using AFrame.*
8 $\left\langle \begin{array}{l} \exists r, r_2, r_3.\, r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(x, r_2)]^r_1 * r_3 \overset{1/2}{\Longrightarrow} Q * [\mathsf{send}(y)]^r_1 * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs) * \mathsf{unused}(r, rs) \wedge rs = rs_1 \cdot e \cdot rs_2 \cdot e' \cdot rs_3 \wedge \\ \mathsf{wf}(rs) \wedge e = (y, r_3, \mathcal{I}', 0, \mathcal{W}') \wedge e' = (x, r', \mathcal{I}, f, \mathcal{W}) \wedge r_2 \in \mathcal{I} \end{array} \right\rangle$
9 // *Pull out node.*
10 $\left\langle \begin{array}{l} \exists r, r_2, r_3.\, r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(x, r_2)]^r_1 * r_3 \overset{1/2}{\Longrightarrow} Q * [\mathsf{send}(y)]^r_1 * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}', \mathcal{W}' \uplus \{r_3\}) * \mathsf{chainres}(rs_2 \cdot e' \cdot rs_3) * \mathsf{unused}(r, rs) \wedge \\ rs = rs_1 \cdot e \cdot rs_2 \cdot e' \cdot rs_3 \wedge \mathsf{wf}(rs) \wedge e = (y, r_3, \mathcal{I}', 0, \mathcal{W}') \wedge e' = (x, r', \mathcal{I}, f, \mathcal{W}) \wedge r_2 \in \mathcal{I} \end{array} \right\rangle$
11 // *Apply Lemma* 6.8 *to supply the resource*
12 $\left\langle \begin{array}{l} \exists r, r_2, r_3.\, [\mathsf{change}(x, r_2)]^r_1 * r_4 \overset{1/2}{\Longrightarrow} (P \mathrel{-\!\!*} Q) * [\mathsf{send}(y)]^r_1 * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}', \mathcal{W}' \uplus \{r_2, r_4\}) * \mathsf{chainres}(rs_2 \cdot e' \cdot rs_3) * \mathsf{unused}(r, rs) \wedge \\ rs = rs_1 \cdot e \cdot rs_2 \cdot e' \cdot rs_3 \wedge \mathsf{wf}(rs) \wedge e = (y, r_3, \mathcal{I}', 0, \mathcal{W}') \wedge e' = (x, r', \mathcal{I}, f, \mathcal{W}) \wedge r_2 \in \mathcal{I} \end{array} \right\rangle$
13 // *Close the chain.*
14 $\left\langle \begin{array}{l} \exists r, r_2, r_4.\, r_4 \overset{1/2}{\Longrightarrow} (P \mathrel{-\!\!*} Q) * [\mathsf{send}(y)]^r_1 * \mathsf{chainds}(rs') * \mathsf{chainres}(rs') * \mathsf{unused}(r, rs') \wedge \\ rs = rs_1 \cdot e \cdot rs_2 \cdot e' \cdot rs_3 \wedge rs' = rs_1 \cdot (y, r_4, \mathcal{I}', 0, \mathcal{W}' \uplus \{r_2\}) \cdot rs_2 \cdot e' \cdot rs_3 \wedge \\ \mathsf{wf}(rs) \wedge e = (y, \_, \mathcal{I}', 0, \mathcal{W}') \wedge e' = (x, r', \mathcal{I}, f, \mathcal{W}) \wedge r_2 \in \mathcal{I} \end{array} \right\rangle$
15 // *Close region using the* renun *transition.*
16 $\left\{ \exists r, r_4.\, r_4 \overset{1/2}{\Longrightarrow} (P \mathrel{-\!\!*} Q) * [\mathsf{send}(y)]^r_1 * \mathsf{oreg}\,(r, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(y) = (r_4, \_, 0, \_)\}) \right\}$
17 // *Definition of* send.
18 $\{\mathsf{send}(y, P \mathrel{-\!\!*} Q)\}$

Fig. 22. Proof of the renunciation axiom for out-of-order implementation.

Splitting. The axiom is defined as follows:

$$\{\mathsf{recv}(a,P) * \lfloor \ulcorner P \urcorner \mathrel{-\!\!*} (P_1 * P_2) \rfloor\} \quad \langle \mathtt{skip} \rangle \quad \{\mathsf{recv}(a,P_1) * \mathsf{recv}(a,P_2)\}$$

A sketch proof is given in Figure 23. The key step is splitting one element of the promised resource set $\mathcal{I}$ for a node (line 6). To do this, we use the following lemma, which states that the saved proposition r_2 can be exchanged for new saved propositions r_3 and r_4 if the appropriate resource $\lfloor \ulcorner P \urcorner \mathrel{-\!\!*} (P_1 * P_2) \rfloor$ is supplied.

LEMMA 6.9. $r_2 \in rs(x).\mathcal{I} \wedge r_2 \overset{1/2}{\Longrightarrow} P * \lfloor \ulcorner P \urcorner \mathrel{-\!\!*} (P_1 * P_2) \rfloor * \mathsf{chainres}(rs)$
$\sqsubseteq \quad \exists rs', r_3, r_4.\, r_3, r_4 \notin rs \wedge rs' = rs \blacktriangleleft_x \blacktriangleleft_{\mathcal{I}} ((\bullet \setminus r_2) \uplus \{r_3, r_4\}) \wedge$
$r_3 \overset{1/2}{\Longrightarrow} P_1 * r_4 \overset{1/2}{\Longrightarrow} P_2 * \mathsf{chainres}(rs')$

1 $\left\{\mathsf{recv}(x,P) * \lfloor \lceil P \rceil \mathrel{-\!\!*} (P_1 * P_2) \rfloor\right\}$
2 *// Definition of* recv.
3 $\left\{\begin{array}{l}\exists r_1, r_2.\, \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid rs(x) = (\_, \mathcal{I}, \_, \_) \wedge r_2 \in \mathcal{I} \wedge \mathsf{wf}(rs)\}) \\ \qquad * r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(x, r_2)]^{r_1}_1 * \lfloor \lceil P \rceil \mathrel{-\!\!*} (P_1 * P_2) \rfloor\end{array}\right\}$
4 *// Enter the region.*
5 $\left\langle \begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(x, r_2)]^{r_1}_1 * \lfloor \lceil P \rceil \mathrel{-\!\!*} (P_1 * P_2) \rfloor * \\ \triangleright \left(\begin{array}{r}\mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I} \uplus \{r_2\}, \mathcal{W}) * \mathsf{chainres}(rs_2) \wedge \\ rs = rs_1 \cdot (x, r, \mathcal{I} \uplus \{r_2\}, f, \mathcal{W}) \cdot rs_2 \wedge \mathsf{unused}(r_1, rs)\end{array}\right)\end{array} \right\rangle$
6 $\langle \mathtt{skip} \rangle$ *// AFrame to drop* $\triangleright$*, Lemma* 6.9 *to split the region.*
7 $\left\langle \begin{array}{l}\exists r_1, r_2, r_3, r_4.\, r_3, r_4 \notin rs \wedge r_3 \overset{1/2}{\Longmapsto} P_1 * r_4 \overset{1/2}{\Longmapsto} P_2 * [\mathsf{change}(x, r_2)]^{r_1}_1 * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I} \uplus \{r_3, r_4\}, \mathcal{W}) * \mathsf{chainres}(rs_2) * \\ \qquad \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot (x, r, \mathcal{I} \uplus \{r_2\}, f, \mathcal{W}) \cdot rs_2\end{array} \right\rangle$
8 *// Close chain, pull* [change] *perms out of* unused.
9 $\left\langle \begin{array}{l}\exists r_1, r_2, r_3, r_4.\, r_3 \overset{1/2}{\Longmapsto} P_1 * r_4 \overset{1/2}{\Longmapsto} P_2 * [\mathsf{change}(x, r_3)]^{r_1}_1 * [\mathsf{change}(x, r_4)]^{r_1}_1 * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I} \uplus \{r_3, r_4\}, \mathcal{W}) * \mathsf{chainres}(rs_2) * \mathsf{unused}(r_1, rs') \\ \qquad \wedge rs = rs_1 \cdot (x, r, \mathcal{I} \uplus \{r_2\}, f, \mathcal{W}) \cdot rs_2 \wedge rs' = rs_1 \cdot (x, r, \mathcal{I} \uplus \{r_3, r_4\}, f, \mathcal{W}) \cdot rs_2\end{array} \right\rangle$
10 *// Close region using the* split *transition.*
11 $\left\{\begin{array}{l}\exists r_1, r_2, r_3, r_4.\, r_3 \overset{1/2}{\Longmapsto} P_1 * r_4 \overset{1/2}{\Longmapsto} P_2 * [\mathsf{change}(x, r_3)]^{r_1}_1 * [\mathsf{change}(x, r_4)]^{r_1}_1 * \\ \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid rs(x) = (\_, \mathcal{I}, \_, \_) \wedge r_3, r_4 \in \mathcal{I} \wedge \mathsf{wf}(rs)\})\end{array}\right\}$
12 *// Definition of* recv.
13 $\left\{\mathsf{recv}(x, P_1) * \mathsf{recv}(x, P_2)\right\}$

Fig. 23. Proof of the splitting axiom for the out-of-order implementation.

PROOF. Given in Appendix B. This proof is very similar in structure to the proof without chaining given in Section 5.3. □

7. OUT-OF-ORDER SUMMARIZATION

In this section, we consider our third, most complex, channel implementation. This implementation satisfies our abstract specification, but internally, signals propagate up a tree structure toward a shared root rather than along a linear chain. This barrier implementation corresponds closely to the one in Navabi et al. [2008]—the summarization process is how it achieves its efficiency. Verifying this implementation demonstrates that our approach scales to custom synchronization constructs developed for performance-sensitive concurrency applications.

7.1. Implementation Approach

Figure 24 shows the implementation. The data structure is an inverted tree—that is, nodes point upward toward a single common root. An instance of the data structure is illustrated in Figure 25.

All of the nodes in the tree are of type `chan_addr`, and each contains a fixed-size Boolean array. Booleans in an array correspond to channel flags, either Low or High. Flags that are leaves (i.e., that do not have a child subtree) represent the channels in the chain. Scanning these leaf flags in order gives the sequence of flags in the abstract chain.

Nonleaf flags in the tree summarize the states of multiple flags in the chain. Nodes are equipped with an `up` field containing an address and an offset—the address is the

```
typedef struct chan_hdr chan_hdr;

typedef struct chan_addr {
  chan_hdr *hdr;
  int off;
} chan_addr;

typedef struct chan_hdr {
  chan_addr up;
  int loff;
  bool flags[MAX];
} chan_hdr;

(chan_addr, chan_addr)
extend(chan_addr x){
  chan_addr nx,r;
  if(x.off == x.hdr->loff
            && x.off < MAX){
    x.hdr->flags[x.loff+1] = 0;
    x.hdr->loff++;
    r.hdr = x.hdr;
    r.off = x.off + 1;
    nx = x;
  } else {
    nh = malloc(chan_hdr);
    nh->up = x;
    nh->loff = 1;
    nx.hdr = nh;
    nx.off = 0;
    r.hdr = nh;
    r.off = 1;
  }
  return (r,nx);
}
```

```
signal(chan_addr x){
  int i;
  bool ret = FALSE;
  chan_addr a = x;
  while (a.hdr != NULL && !ret){
    a.hdr->flags[a.off] = 1;
    for(i=0; i<=a.hdr->loff; i++){
      if (a.hdr->flags[i] != 1)
        ret = TRUE;
    }
    a = a.hdr->up;
  }
}

wait(chan_addr x){
  chan_addr a = x;
  while (a.hdr != NULL) {
    for(skip; a.off<=a.hdr->loff;
                        a.off++){
      while(a.hdr->flags[a.off] != 1)
        skip;
    }
    a = a.hdr->up;
    a.off++;
  }
}
```

Fig. 24. Channel implementation with out-of-order signaling and summarization.

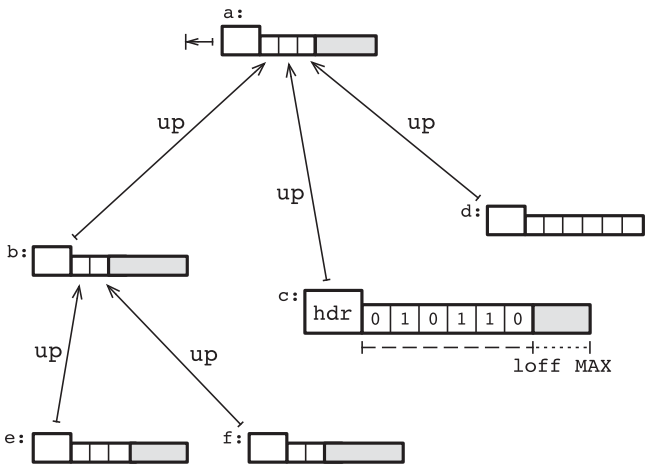
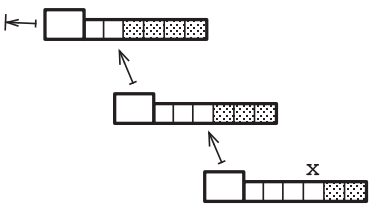


Fig. 25. Example of the summarizing channel structure.

parent node, while the offset refers to a particular "summary" flag in its array. If all the flags in a node are set, then its parent summary flag can also be set. This means that if a summary flag is set, all the flags in the subtree must also be set. Thus, `wait` only needs to read from a single flag to know that all the flags in the child subtree are set.

To `signal` a channel, the function reads the array location from the array header and writes 1 into the flag at the appropriate offset (Figure 24, line 40). `signal` then reads all the sibling flags in the same array. If any of them are unset, it exits. Otherwise, it retrieves the address to the next level in the tree and loops if it is not at the root (lines 41–45). In this way, if a flag's siblings are all set, then `signal` will set the parent summary flag. If all the siblings of the summary flag are also set, it will set its parent, and in this way iterate up the tree.

The `wait` function exploits summaries to reduce the number of flags it must test. Rather than examining all the preceding flags in the chain, `wait` just examines the summary closer to the root. The function waits on each preceding flag in the current array (Figure 24, lines 60–64—note that increasing the index moves logically earlier in the chain). It then reads the `up` address stored in the array header and loops if it is not at the root. Because it only ever climbs the tree, the function avoids the cost of iterating over the entire chain. The following diagram shows the nodes accessed when traversing up from `x`:



Arrays are allocated with a fixed maximum size `MAX` and are gradually filled when extending the chain. The current number of flags active in the array is stored in the header field `loff`. Thus, `extend` has two cases depending on whether there is room in the current array for another leaf (checked on line 17).

—If there is space, then the new leaf is inserted immediately following the current one in the array (lines 20–22).
—If no space remains in the array, then a new array is allocated, and the current flag is used as a summary. The first and second locations in the new array represent the current and newly created channels (lines 25–31).

As a result of extension, `wait` may be passed the address of a leaf flag, which is then silently converted into a summary. However, this process is sound: if this summary is set, then the appropriate child flag must also be set (this subtlety is what necessitates the predicate finalleaf in the proof next).

7.2. Proof Strategy

Abstract state. At the level of specification, the behavior of the algorithm is unchanged from Section 6; as a result, we can reuse some of the reasoning from there. The algorithm's abstract state has two parts: an *abstract chain* and a *heap map*. The former dictates the resources promised and renounced at each point in the chain, while the latter defines the underlying inverted-tree pointer structure.

An *abstract chain* is a sequence in CNode^{+}, defined exactly as in Section 6 (see p. 37). The only difference is that each node location *loc* is now a pair $\langle h, o \rangle$ consisting of

a physical address and an offset—this corresponds to the `chan_addr` type. We reuse abstract chains so that we can reuse reasoning about how resources move through the chain. However, for this implementation, an abstract chain alone is insufficient, because it does not represent the underlying inverted tree data structure.

A *heap map* represents the inverted-tree structure that controls signal propagation and summarization. Heap maps are finite, partial functions from physical addresses—each address represents a `chan_hdr`-typed object forming the tree.

$$d \in \mathsf{HMap} : \mathsf{Addr} \xrightarrow{fin} \{uhdr : \mathsf{Addr};\ \ uoff : \mathsf{Int};\ \ loff : \mathsf{Int};\ \ flags : \{0..\mathsf{MAX}\} \to \mathsf{Bool}\}$$

(To simplify the representation, we flatten the `chan_adr`-typed field "`up`" into the two subfields *uhdr* and *uoff*.)

To ensure a heap map represents a correct inverted tree, we require several well formedness properties:

—Each address-offset pair $\langle x, i\rangle$ has at most one child in d that points upward to it. This ensures leaves are uniquely identified.
—Chains of upward pointers are noncyclic and point toward a common root. This ensures that the data structure is tree shaped.
—Flags are only set in nonleaf nodes if all the corresponding leaf nodes are set. This ensures that signals are properly summarized and propagated up the tree.

Given a chain *rs* and heap map d, well formedness also requires that the two portions of the abstract state correspond. Loosely, this means that the leaves of the inverted tree defined by d are exactly the addresses in the chain *rs*.

Abstract state well formedness. In order to define formally that *rs* and d are well formed and correspond correctly, we require several auxiliary notions:

$$\begin{aligned}
\mathsf{child}_d(x, i, y) &\triangleq d(y).uhdr = x \wedge d(y).uoff = i \wedge 0 \le i \le d(x).loff \\
\mathsf{leaf}_d(\langle x, i\rangle) &\triangleq \langle x, i\rangle \in d \wedge \nexists y.\ d(y).uhdr = x \wedge d(y).uoff = i \\
\mathsf{descend}_d(\langle x, i\rangle, \langle y, j\rangle) &\triangleq \langle x, i\rangle = \langle y, j\rangle \vee \mathsf{descend}_d(\langle x, i\rangle, \langle d(y).uhdr, d(y).uoff\rangle) \\
\mathsf{isset}_d(\langle x, i\rangle) &\triangleq d(x).flags(i) = 1 \\
\mathsf{allset}_d(x) &\triangleq \forall i.\ 0 \le i \le d(x).loff \Rightarrow \mathsf{isset}_d(\langle x, i\rangle)
\end{aligned}$$

(Note we say that a location-offset pair $\langle x, i\rangle$ is in d if $x \in \mathrm{dom}(d) \wedge i \le d.loff$.)

The child, descend, and leaf predicates record corresponding structural facts about relationships in the tree. Well formedness on d (defined later) requires that paths through *uhdr* are finite, which suffices to ensure that descend is well defined. isset and allset respectively assert that a single address and a whole array have their flags set.

$$\begin{aligned}
\langle x, i\rangle <_d \langle y, j\rangle &\triangleq \exists z, ix, iy.\ \mathsf{descend}_d(\langle z, ix\rangle, \langle x, i\rangle) \wedge \mathsf{descend}_d(\langle z, iy\rangle, \langle y, j\rangle) \wedge ix > iy \\
a <_d^{\Updownarrow} b &\triangleq a <_d b \vee \mathsf{descend}_d(a, b) \vee \mathsf{descend}_d(b, a) \\
\mathsf{finalleaf}_d(x, y) &\triangleq \mathsf{descend}_d(x, y) \wedge \mathsf{leaf}_d(y) \wedge (\forall z.\ \mathsf{descend}_d(x, z) \wedge \mathsf{leaf}_d(z) \Rightarrow z <_d^{\Updownarrow} y)
\end{aligned}$$

The order predicate $<_d$ says that two addresses are ordered in the tree, meaning that they share a common ancestor array in which they are also ordered. This defines a transitive and irreflexive order. $<_d^{\Updownarrow}$ says that either two addresses are related by $<_d$ or that one is the descendant of the other (i.e., they are on the same path and one summarizes the other).

$\mathsf{finalleaf}_d(x, y)$ indicates that y is the maximal leaf according to $<_d^{\Updownarrow}$ that is summarized by x. This is useful because applications of `extend` may mean clients wait on x when the actual leaf has been superseded by y. For each x there exists at most one y satisfying finalleaf, so we generally use it as a partial function; that is, $\mathsf{finalleaf}_d(x)$ stands for the unique y such that $\mathsf{finalleaf}_d(x, y)$.

$$
\begin{aligned}
\mathsf{renun}_s(x,(rs,d),(rs',d')) &\triangleq d' = d \wedge \mathsf{renun}_c(x, rs, rs')\\
\mathsf{set}_s(x,(rs,d),(rs',d')) &\triangleq x = \langle h,o\rangle \wedge d' = d \blacktriangleleft_h \blacktriangleleft_{flags[o]} (1) \wedge \mathsf{set}_c(\langle h,o\rangle, rs, rs')\\
\mathsf{ext}_s(x,(rs,d),(rs',d')) &\triangleq rs = (rs_1 \cdot a \cdot rs_2) \wedge rs' = (rs_1 \cdot b \cdot c \cdot rs_2) \wedge a.\mathit{flg} = 0 \wedge\\
&\quad \exists y.\, b = a[\mathit{loc} \mapsto y] \wedge c.\mathit{flg} = 0 \wedge c.\mathcal{W} = \emptyset \wedge x = \langle h,o\rangle = a.\mathit{loc} \wedge\\
&\quad \left(\begin{array}{l}\left(\begin{array}{l} b.\mathit{off} = \langle h',0\rangle \wedge c.\mathit{off} = \langle h',1\rangle \wedge\\ d' = d \uplus h' \mapsto \{\mathit{uhdr} = h; \mathit{uoff} = o; \mathit{loff} = 1; \mathit{flags} = \lambda\_.\ 0\}\end{array}\right)\\ \vee \left(\begin{array}{l} o = d(h).\mathit{loff} \wedge b.\mathit{loc} = a.\mathit{loc} \wedge c.\mathit{loc} = \langle h, o+1\rangle \wedge\\ d' = (d \blacktriangleleft_h \blacktriangleleft_{\mathit{loff}} (o+1)) \blacktriangleleft_h \blacktriangleleft_{\mathit{flags}[o+1]} (0)\end{array}\right)\end{array}\right)\\
\mathsf{split}_s(x,r,(rs,d),(rs',d')) &\triangleq d = d' \wedge \mathsf{split}_c(\mathsf{finalleaf}_d(x), r, rs, rs')\\
\mathsf{sat}_s((rs,d),(rs',d')) &\triangleq d = d' \wedge \mathsf{sat}_c(rs, rs')\\
\mathsf{get}_s(x,r,(rs,d),(rs',d')) &\triangleq d = d' \wedge \mathsf{get}_c(\mathsf{finalleaf}_d(x), r, rs, rs')
\end{aligned}
$$

$$
\begin{aligned}
T_{\mathsf{s}}(\mathsf{send}(x)) &\triangleq \{(a,b) \mid \mathsf{wf}(b) \wedge (\mathsf{renun}_s(x,a,b) \vee \mathsf{set}_s(x,a,b) \vee \mathsf{ext}_s(x,a,b))\}\\
T_{\mathsf{s}}(\mathsf{change}(x,r)) &\triangleq \{(a,b) \mid \mathsf{wf}(b) \wedge (\mathsf{split}_s(x,r,a,b) \vee \mathsf{sat}_s(a,b) \vee \mathsf{get}_s(x,r,a,b))\}\\
T_{\mathsf{s}}(\mathsf{mark}) &\triangleq \left\{((rs,d),(rs,d')) \;\middle|\; \begin{array}{r}\mathsf{wf}(rs,d') \wedge d' = d \blacktriangleleft_h \blacktriangleleft_{flags[o]} (1)\\ \wedge\ \neg\mathsf{leaf}_d(\langle h,o\rangle))\end{array}\right\}
\end{aligned}
$$

Fig. 26. Definition of the transition relation T_{s} for the summarizing implementation.

We can now define $\mathsf{wf}(rs,d)$, which requires that rs and d are independently well formed and that they are correctly tied together. As abstract chains are unchanged from Section 6, we can reuse the prior definition of $\mathsf{wf}(rs)$ (§6.2). For heap-maps, well formedness is defined as follows:

$$
\begin{aligned}
\mathsf{wf}(d) \triangleq\ & \exists r : \mathsf{Addr}.\ \exists \tau : \mathrm{dom}(d) \to \mathbb{N}.\\
& \forall x,i,y,z.\ (\langle x,i\rangle \in d \Rightarrow \exists j.\ \mathsf{descend}(\langle r,j\rangle, \langle x,i\rangle)) \wedge\\
& \qquad ((\mathsf{child}_d(x,i,y) \wedge \mathsf{child}_d(x,i,z)) \Rightarrow y = z) \wedge\\
& \qquad ((\mathsf{child}_d(x,i,y) \wedge \mathsf{isset}_d(x,i)) \Rightarrow \mathsf{allset}_d(y)) \wedge\\
& \qquad (d(x).\mathit{uhdr} = y \wedge y \neq \mathsf{NULL} \Rightarrow\\
& \qquad\quad d(y)\ \mathrm{defined} \wedge d(x).\mathit{uoff} \leq d(y).\mathit{loff} \wedge \tau(y) < \tau(x))
\end{aligned}
$$

Here the address r is the location of the tree root, while the function τ records the distance from the current node to the root—this enforces the absence of cycles. The first clause of the definition ensures all elements in the tree share a common root. The second ensures that children are uniquely identified by address and offset. The third ensures that setting a flag summarizes all descendants. The final clause guarantees the existence of non-NULL parents to a node and enforces the distance function τ.

$$
\begin{aligned}
\mathsf{wf}(rs,d) \triangleq\ & \mathsf{wf}(rs) \wedge \mathsf{wf}(d) \wedge\\
& \forall r \in rs.\ r.\mathit{loc} = \langle l,o\rangle \Rightarrow r.\mathit{flg} = d(l).\mathit{flags}[o] \wedge\\
& \forall r \in rs.\ \mathsf{leaf}_d(r.\mathit{loc}) \wedge\\
& \forall r_1, r_2.\ (rs = \_ \cdot r_1 \cdot \_ \cdot r_2 \cdot \_) \Rightarrow r_2.\mathit{loc} <_d r_1.\mathit{loc}
\end{aligned}
$$

As well as requiring that rs and d are well formed on their own, this requires (1) that flags in the chain are correctly set in the heap map, (2) locations in the chain are leaves in the heap map, and (3) order in the chain is reflected in the heap-map order $<_d$.

Transition map. We define a new transition map T_s to capture changes to the heap map (Figure 26). Many of the transitions are inherited from T_c, the transition relation for the chained implementation (Section 6.2).

set and ext alter the underlying inverted-tree data structure, and so are defined to allow this. set just updates the tree-map flag appropriately using our lens notation. ext

has two cases, reflecting the conditional in the implementation. Either there is enough space in the array to fit another flag or another array must be added to the heap map. In the latter case, note that the address of the existing channel a is shifted into the new array.

Furthermore, `signal` can mark summary flags, that is, flags that are not leaves in the tree; this is allowed by the transition mark. Finally, note finalleaf in the definition of get: this is needed because the real flag may shift its position due to `extend`, with `wait` left reading from a summary. Here x is the original flag that has been superseded, and $\mathsf{finalleaf}_d(x)$ is the current position of the flag. It is safe to pass an address to `wait` that may have been converted into a summary because extension ensures the original flag must be one of those summarized by x.

Interpretation function. For a given map d, chainds maps down to the corresponding data structure definition. Heap maps are intentionally close to the underlying heap: each element in the domain maps to a distinct `chan_addr` object in memory.

$$\begin{aligned}\mathsf{chainds}(d) \triangleq\ & \circledast_{x\in\mathrm{dom}(d)}\,. \\ & x.\mathrm{up} \Mapsto \{\mathrm{hdr} = d(x).uhdr;\ \mathrm{off} = d(x).uoff\} * x.\mathrm{loff} \mapsto (d(x).loff) \\ & \qquad * \circledast_{i\in\{0..\mathsf{MAX}\}}\,.\ x.\mathrm{flags}[i] \mapsto (d(x).flags(i))\end{aligned}$$

Once a `chan_addr` object has been allocated, its up address cannot be modified. To represent this in the definition, we write $x \Mapsto v$ to indicate that x is immutable—shorthand for $\exists f.\ x \overset{f}{\mapsto} v$. This is useful as immutable locations can be freely shared between threads: $x \Mapsto v \Rightarrow x \Mapsto v * x \Mapsto v$.

The interpretation function I_s converts an abstract state $\mathsf{Chain}(rs, d)$ into a data structure:

$$I_s(r)(\mathsf{Chain}(rs,d)) \triangleq \mathsf{chainds}(d) * \mathsf{chainres}(rs) * \mathsf{unused}(r, rs, d)$$

As chainds defines the concrete heap structure, it only requires a heap map d. Conversely, chainres defines the pattern of splits, promises, and renunciations, and so only requires an abstract chain rs. Indeed, chainres is defined identically to Section 6.

The predicate unused, representing the "library" of unused permissions, requires both d and rs. This is because the position of a flag may move as a result of extension. As a result, `wait` may be passed a summary flag x—before extension, the passed node would have been a leaf. The flag x will be represented in d but not represented in the chain rs, and thus unused requires both in order to keep track of permissions.

To define unused, the set $\mathsf{C}(rs, d)$ represents possible targets of `wait`; a permission is missing from the set of unused change permissions, $\mathsf{uC}(rs, d)$, only if it targets one of these nodes.

$$\begin{aligned}\mathsf{C}(rs,d) &\triangleq \{\langle y,i\rangle \mid r \in rs \wedge \mathsf{finalleaf}_d(\langle y,i\rangle, r.loc) \\ &\qquad \wedge \neg\mathsf{finalleaf}_d(\langle d(y).uhdr, d(y).uoff\rangle, r.loc)\} \\ \mathsf{uC}(rs,d) &\triangleq \{(x,r) \mid r \in rs(\mathsf{finalleaf}_d(x)).\mathcal{I} \wedge x \in \mathsf{C}(rs,d) \wedge \neg\exists y.\ r \in rs(y).\mathcal{W}\} \\ \mathsf{unused}(r,rs,d) &\triangleq (\circledast x \notin \mathsf{uS}(rs).\ [\mathsf{send}(x)]^r_1) * (\circledast (x,r') \notin \mathsf{uC}(rs,d).\ [\mathsf{change}(x,r')]^r_1)\end{aligned}$$

Predicate definitions. We can now define the send and recv predicates. These largely follow the definitions in Section 6: the differences in underlying data structures are

abstracted by the interpretation function and transition relation.

$$\begin{aligned}
\mathsf{oreg}(r, S) &\triangleq \mathsf{region}(S, T_{\mathsf{s}}, I_{\mathsf{s}}(r), r)\\
\mathsf{send}(x, P) &\triangleq \exists r_1, r_2.\ r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{send}(x)]^{r_1}_1 * [\mathsf{mark}]^{r_1}_{_} *\\
&\qquad \mathsf{oreg}(r_1, \{\mathsf{Chain}(rs, d) \mid \mathsf{wf}(rs, d) \wedge rs(x) = (r_2, _, 0, _)\})\\
\mathsf{recv}(x, P) &\triangleq \exists r_1, r_2.\ r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(x, r_2)]^{r_1}_1 *\\
&\qquad \mathsf{oreg}(r_1, \{\mathsf{Chain}(rs, d) \mid \mathsf{wf}(rs, d) \wedge r_2 \in rs(\mathsf{finalleaf}_d(x)).\mathcal{I}\})\\
x \prec y &\triangleq \exists r.\ \mathsf{oreg}(r, \{\mathsf{Chain}(rs, d) \mid \mathsf{wf}(rs, d) \wedge (\mathsf{finalleaf}_d(y), \mathsf{finalleaf}_d(x)) \in \mathsf{ord}(rs)\})
\end{aligned}$$

We use x to stand for $\langle x.\mathtt{hdr}, x.\mathtt{off}\rangle$ if x is a `chan_addr` struct, and we use $[\mathsf{mark}]^r_{\_}$ as notation for $\exists\pi.\ [\mathsf{mark}]^r_\pi$ to represent nonexclusive ownership of the mark action.

The stability of most of the predicates is obvious; however, the fact that finalleaf can change means we prove stability explicitly for recv.

LEMMA 7.1. $\mathsf{recv}(x, P)$ *is stable.*

PROOF. Assume the initial abstract state of the chain is $\mathsf{Chain}(rs, d)$ and that ext takes the step $(x', (rs, d), (rs', d'))$. The case where $x \neq x'$ is trivial, so assume $x = x'$. We now need to show $\mathsf{wf}(rs', d') \wedge r_2 \in \mathsf{rs}'(\mathsf{finalleaf}_{d'}(x)).\mathcal{I}$. Assume a, b, and c are chain nodes as used in the definition of ext.

The requirement $\mathsf{wf}(rs', d')$ holds as a constraint on the transition relation. It remains to show the second clause. By the definition of ext, $\mathsf{finalleaf}_d(x) = a.loc$ and $a.\mathcal{I} = b.\mathcal{I}$. There are now two cases: either ext generates a new array or it adds an element to the existing array. In the latter case, d' only changes by adding an element at a higher index in the array. Thus, $\mathsf{finalleaf}_{d'}(x) = b.loc$. In the former case, ext adds a new array that must also descend from x. If $\mathsf{finalleaf}_{d'}(x) \neq b.loc$, there must be leaf z such that $b.loc <_d z$, but the only new leaf c is at the next index in the new array, meaning $c.loc <_d b.loc$. Thus, the result follows by contradiction. □

7.3. Verifying `Wait`, `Signal`, `Extend`

Proving `signal`. A sketch proof for `signal` is given in Figure 27. The algorithm begins by setting the flag at the appropriate address (line 9). Abstractly, the reasoning here is the same as when setting a flag in the nonsummarizing implementation (Section 6.3), so we omit it. The algorithm then climbs up the tree. If all the flags have been set in a given array, the summary flag is also set (line 9). Well formedness allows summary nodes to be set if all their children are set. If a flag is discovered that is not set or the loop climbs to the top of the tree, the algorithm exits.

The assignment on line 9 applies the transition relation step set or mark, depending on whether the node is a leaf or a summary. The following lemma ensures that the library of unused permissions is preserved after each such transition relation step.

LEMMA 7.2.

$$\begin{aligned}
&\mathsf{set}_s(x, (rs, d), (rs', d')) \wedge \mathsf{unused}(r, rs, d) * [\mathsf{send}(x)]^r_1 \Rightarrow \mathsf{unused}(r, rs', d')\\
&((rs, d), (rs', d')) \in T_s(\mathsf{mark}) \wedge \mathsf{unused}(r, rs, d) \Rightarrow \mathsf{unused}(r, rs', d')
\end{aligned}$$

PROOF. Trivial from the definition of T_s and unused. □

Proving `wait`. A sketch proof for `wait` is given in Figures 28 and 29. This proof just deals with the part of the code establishing that all the flags in the chain have been set. In the proof, we use $\mathsf{last}_d(\mathtt{a})$ to stand for the last address in the array associated with `a`, that is, $\langle \mathtt{a.hdr}, d(\mathtt{a.hdr}).loff\rangle$.

The loop starting at line 5 checks the flags in the current array. In line 8, the algorithm waits for the current node's flag. This may not be a leaf; it may be a summary node

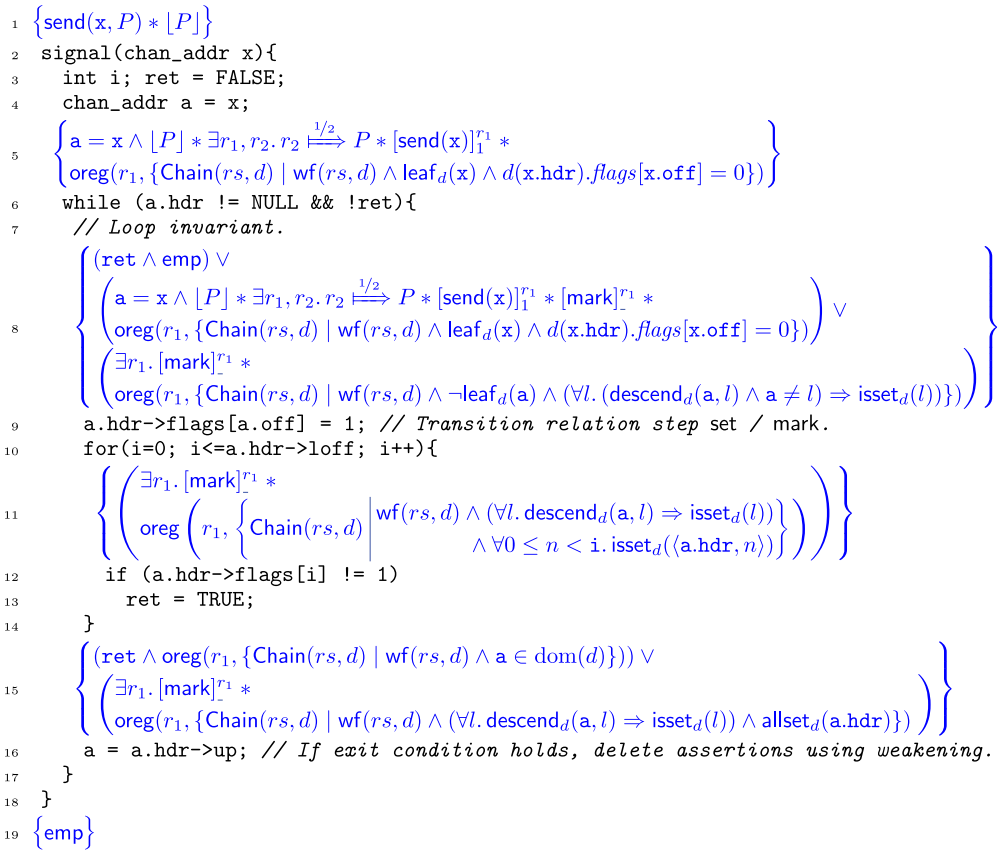


Fig. 27. Sketch proof of `signal` with summarization.

somewhere inside the tree. Once this passes, by the second clause of well formedness (page 52), we can conclude that all the flags in the subsequence rs_3 have been set. Then the algorithm increments the offset—as a is not at the last offset for the array, there must exist an adjacent channel address at this position.

To prove this algorithm correct, we need several sublemmas. The first states that once the algorithm reaches the root of the tree, there are no locations in d that are earlier according to $<_d$. This ensures that searching the tree covers all the preceding channels in the chain.

LEMMA 7.3. $\mathsf{wf}(d) \wedge d(a).loff = o \wedge d(a).uhdr = \mathsf{NULL} \Rightarrow \neg\exists x.\, x <_d \langle a, o\rangle$

PROOF. Assume such an x exists. Then, by the definition of $<_d$, there must exist an address y such that x and $\langle a, o\rangle$ are both descended from y. As the $uhdr$ field is NULL, the only possibility is that both addresses are in the object at a. By the definition of $<_d$, x must be further right in the flag array, but o is the right-most address. This contradicts the assumption and completes the proof. □

The second lemma states that examining the elements reachable through the heap map suffices to show that the corresponding elements in the abstract chain have been set. This lemma justifies our splitting of the invariant into a separate heap map and abstract chain structure.

Fig. 28. Sketch proof of `wait` with summarization (completed in Figure 29).

LEMMA 7.4. $\mathsf{wf}(rs, d) \wedge z \in \mathsf{dom}(rs) \wedge \mathsf{descend}_d(x, z) \wedge \mathsf{leaf}_d(z) \wedge (\forall l.\, l <_d^{\Updownarrow} x \wedge \mathsf{leaf}_d(l) \Rightarrow \mathsf{isset}_d(l)) \Rightarrow \exists rs_1, rs_2.\, rs = rs_1 \cdot rs(z) \cdot rs_2 \wedge \mathsf{ctrue}(rs(z) \cdot rs_2)$

PROOF. As $z \in \mathsf{dom}(rs)$, we can easily divide up rs into $rs_1 \cdot rs(z) \cdot rs_2$. Now pick an arbitrary element y in $dom(rs(z) \cdot rs_2)$ and suppose that $rs(y).flg$ is not set. By well formedness, it must be true that $y <_d^{\Updownarrow} z$. Now we show that $y <_d^{\Updownarrow} x$. The contrary, $x <_d y$, would imply that $z <_d y$, contradicting our assumption. Therefore, by the premise, the associated flag must be set. However, well formedness requires that flags are mirrored in rs and d, contradicting our assumption and completing the proof. □

The final lemma shows that shifting left from the current maximal node reaches a node earlier in the order. Note that the existence of the new node is shown on the left of the implication because this lemma is applied in a negative position in the proof.

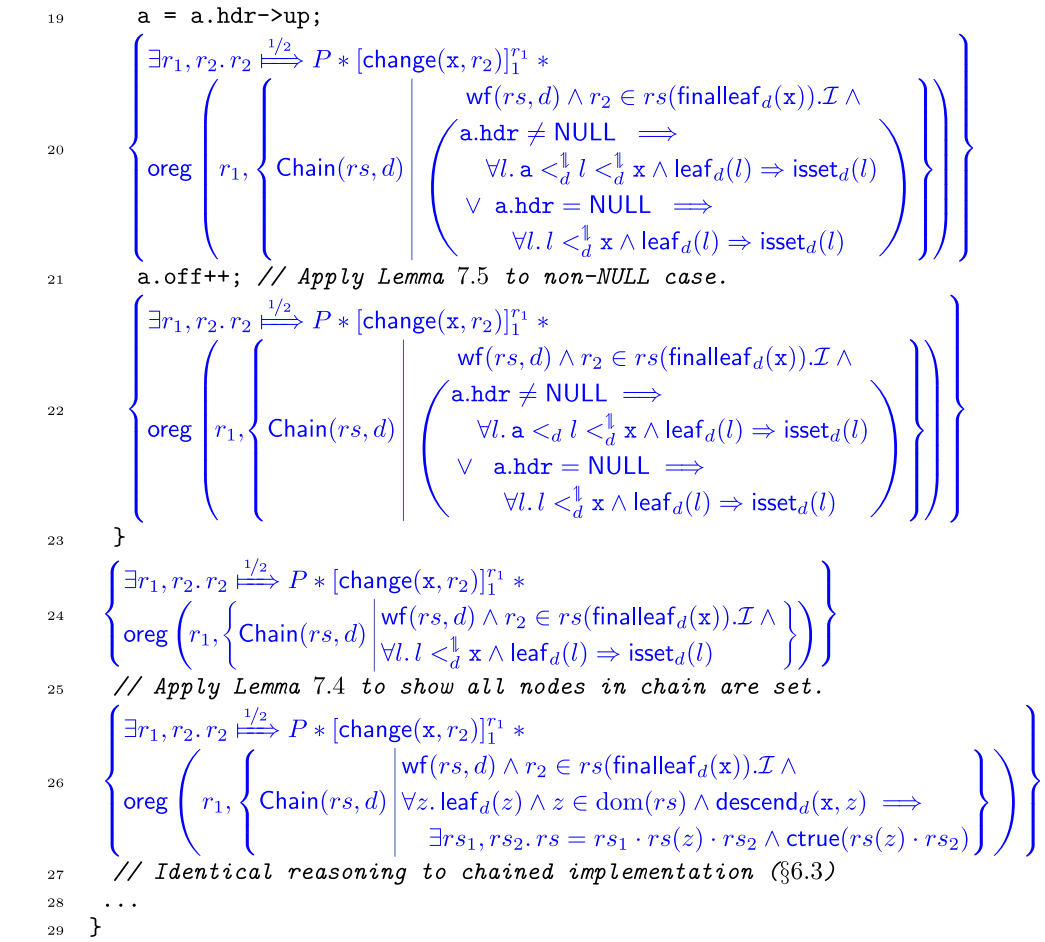


Fig. 29. Sketch proof of `wait` with summarization (continued from Figure 28).

LEMMA 7.5. $0 \leq d(a).off < d(a).loff \wedge \mathsf{wf}(d) \wedge a \blacktriangleleft_{off} (\bullet + 1) <_d k <_d b \Rightarrow a <_d^{\Updownarrow} k <_d b$

PROOF. The result follows from the structure of the heap map and the definition of $<_d$. □

Proving `extend`. A sketch proof of `extend` is given in Figure 30. There are two cases for extending the chain: either the node is the last element in the current array and there is space to add an extra node, or there is no space and the algorithm allocates a fresh array. This choice is made by the conditional in line 6.

The proof needs the following lemmas to show that the unused predicate representing unused permissions is preserved by extending the chain.

LEMMA 7.6.

$$\left(\begin{array}{l}\mathsf{ext}_s(\langle x, i\rangle, (rs, d), (rs', d')) \wedge \mathrm{dom}(d') = \mathrm{dom}(d) \wedge r' \notin rs \wedge \\ rs'(\langle x, i+1\rangle).\mathcal{I} = \{r'\} \wedge \mathsf{wf}(rs, d) \wedge \mathsf{wf}(rs', d') \wedge \mathsf{unused}(r, rs, d)\end{array}\right) \Rightarrow$$
$$\mathsf{unused}(r, rs', d') * [\mathsf{send}(\langle x, i+1\rangle)]_1^r * [\mathsf{change}(\langle x, i+1\rangle, r')]_1^r$$

Fig. 30. Sketch proof of `extend` with summarization.

Proof. Begin by observing that, by the definition of ext_s, $\langle x, i+1\rangle \notin rs$, and thus that

$$\mathsf{unused}(r, rs, d) \Rightarrow [\mathsf{send}(\langle x, i+1\rangle)]^r_1 * [\mathsf{change}(\langle x, i+1\rangle, r')]^r_1 * \mathsf{true}$$

As $\langle x, i+1\rangle \in rs'$, it holds immediately that $\langle x, i+1\rangle \in \mathsf{uS}(rs')$. As $\langle x, i+1\rangle$ is a leaf, $\mathsf{finalleaf}_{d'}(\langle x, i+1\rangle) = \langle x, i+1\rangle$. As it is not the first leaf in the array x, it cannot have a finalleaf parent, meaning it must be in $\mathsf{C}(rs', d')$. Thus, $\langle x, i+1\rangle \in \mathsf{uC}(rs', d')$. This suffices to show that $[\mathsf{send}(\langle x, i+1\rangle)]$ and $[\mathsf{change}(\langle x, i+1\rangle, r')]$ can be safely removed from unused. □

LEMMA 7.7.

$$\begin{pmatrix}\mathsf{ext}_s(\langle x,i\rangle,(rs,d),(rs',d')) \wedge \mathrm{dom}(d') = \mathrm{dom}(d) \uplus \{l\} \wedge r' \notin rs \wedge \\ rs'(\langle l,1\rangle).\mathcal{I} = \{r'\} \wedge \mathsf{wf}(rs,d) \wedge \mathsf{wf}(rs',d') \wedge \mathsf{unused}(r,rs,d) * [\mathsf{send}(\langle x,i\rangle)]^r_1\end{pmatrix} \Rightarrow \\ \mathsf{unused}(r,rs',d') * [\mathsf{send}(\langle l,0\rangle)]^r_1 * [\mathsf{send}(\langle l,1\rangle)]^r_1 * [\mathsf{change}(\langle l,1\rangle,r')]^r_1$$

PROOF. By the structure of ext, $\langle l,0\rangle$ and $\langle l,1\rangle$ are not in rs, but are in rs'. The ability to retrieve $[\mathsf{send}(\langle l,0\rangle)]^r_1 * [\mathsf{send}(\langle l,1\rangle)]^r_1$ follows immediately. As $\langle l,0\rangle$ is leftmost in the array l, its parent $\langle x,i\rangle$ is the maximal final leaf in $\mathsf{C}(rs',d')$. However, $\langle l,1\rangle$ is not leftmost, and thus is in $\mathsf{C}(rs',d')$. By the same argument used in the previous lemma, $[\mathsf{change}(\langle l,1\rangle,r')]^r_1$ can be removed from the unused. □

LEMMA 7.8. $\mathsf{ext}(x,(rs,d),(rs',d')) \wedge \mathsf{wf}(rs,d) \wedge \mathsf{wf}(rs',d') \Rightarrow \mathsf{uC}(rs,d) \subseteq \mathsf{uC}(rs',d')$

PROOF. There are two cases for extension: in-place extension in the array and creation of a new array. In the former case, finalleaf is preserved for existing nodes because the only new node is less than all existing nodes in the array. In the latter case, the parent of the new array is a finalleaf to the new array, and all other finalleaf relationships are preserved.

Now pick a pair $(x,i) \in \mathsf{uC}(rs,d)$, the set of used send permissions. Extending the chain can't stop x from satisfying finalleaf or make any node higher than x satisfy finalleaf. Therefore, $x \in \mathsf{C}(rs',d')$ after extension. The only alteration to renounced sets $\mathcal{W}$ in rs' is to add a new empty set. Thus, $\neg\exists y.\, r \in rs'(y).\mathcal{W}$. Finally, both cases of extension preserve the promise sets $\mathcal{I}$, ensuring that $r \in rs'(\mathsf{finalleaf}_{d'}(x)).\mathcal{I}$. □

7.4. Verifying Splitting and Renunciation Axioms

The splitting and renunciation axioms do not depend on the underlying data structure representation and therefore are largely identical to the ones given in Section 6.4. The main difference is the new definition of unused. The renunciation case is straightforward, but for splitting, we need to show that we can pull the appropriate change permissions out of the "library" predicate unused. This is captured by the following lemma:

LEMMA 7.9.

$$\begin{array}{l}\mathsf{unused}(r_1,rs,d) * [\mathsf{change}(x,r_2)]^{r_1}_1 \wedge \mathsf{split}(x,(rs,d),(rs',d')) \wedge \\ rs(\mathsf{finalleaf}_d(x)) = (r,\mathcal{I} \uplus \{r_2\},f,\mathcal{W}) \wedge rs' = rs \blacktriangleleft_{\mathsf{finalleaf}_{d'}(x)} \blacktriangleleft_{\mathcal{I}} ((\bullet \setminus \{r_2\}) \uplus \{r_3,r_4\}) \\ \qquad \Rightarrow \quad \mathsf{unused}(r_1,rs',d') * [\mathsf{change}(x,r_3)]^{r_1}_1 * [\mathsf{change}(x,r_4)]^{r_1}_1\end{array}$$

PROOF. By the definition of split, $d = d'$, and element locations in rs are unchanged in rs'. Thus, it holds that $\mathsf{C}(rs,d) = \mathsf{C}(rs',d')$ and $\mathsf{finalleaf}_d(x) = \mathsf{finalleaf}_{d'}(x)$. From the definition of uC, the available change permissions are controlled by the set $rs(\mathsf{finalleaf}_d(x)).\mathcal{I}$. This set is correctly updated by the transition, which completes the proof. □

8. COMPARISON TO CONFERENCE PAPER

This article substantially expands and revises the proofs of correctness given in our conference paper [Dodds et al. 2011]. All the proofs have been restructured, and the proof of the summarizing implementation (Section 7) is entirely new. This article also fixes a subtle logical error that rendered some of the reasoning in our conference paper unsound. In this section, we describe how this problem arose, and how we have fixed it.

Our specifications rely crucially on higher-order quantification to abstract over the resources transferred through channels. To support this, in Dodds et al. [2011], we extended the original concurrent abstract predicates logic [Dinsdale-Young et al. 2010] with higher-order assertions and quantification.

In concurrent abstract predicates, resources describe not only the current state of shared regions but also the protocols that govern these shared regions. In the case of higher-order shared resources, these protocols are themselves expressed in terms of assertion variables that might be instantiated with shared resources. Support for such higher-order shared resources thus requires a semantic domain of protocols that include assertions over (among other things) protocols. This results in a circularity, and the resulting equation ($\mathit{protocol} \cong \mathcal{P}(\ldots \times \mathit{protocol})$) has no solution in set theory by a simple cardinality argument.

The logic and model presented in Dodds et al. [2011] broke this circularity by ignoring protocol assertions when interpreting protocols. As a consequence, many of the properties we relied on when reasoning about the higher-order resources $\mathsf{box}(i, P, \pi)$ and $\mathsf{fut}(i, P)$ are unsound. (In that paper, fut played a similar role to recv in this article, while box was used in verifying the splitting axiom.) For instance, $\mathsf{fut}(i, P)$ is generally not stable when P is instantiated with an assertion that includes a protocol assertion, because $\mathsf{fut}(i, P)$ asserts the existence of a shared region whose protocol is defined in terms of P. A more detailed discussion of this class of problem is given in Svendsen et al. [2013].

The program *logic* itself presented in Dodds et al. [2011] still appears sound. However, many steps in the *proofs of programs* depend on unsound auxiliary entailment steps. These steps are common in separation logic proofs, but in most earlier work, entailments generally capture comparatively simple properties. We failed to appreciate how deeply the proofs in Dodds et al. [2011] relied on very subtle entailments between shared regions that were broken in our modified model. The problem came to light a year later when Svendsen attempted to use our logic to verify the Joins library [Svendsen et al. 2013]. Resolving this kind of problem motivated the development of iCAP, which is the proof technique we use in this article.

iCAP uses step-indexing to stratify the construction of the semantic domain of protocols. The resulting logic does support higher-order shared resources but requires $\triangleright$ operators to ensure that protocols are properly stratified. Thus, the problematic circularity in Dodds et al. [2011] is appropriately resolved in the rules of the logic. At the level of human process, we have been much more meticulous in this article in identifying and checking entailment steps used in program proofs.

APPENDIXES

A. SAVED PROPOSITIONS

LEMMA A.1. *Property* (3) *of saved propositions implies Property* (4).

PROOF.

$$
\begin{array}{ll}
& r \overset{\pi_1}{\Longmapsto} P * r \overset{\pi_2}{\Longmapsto} Q * (X \mathrel{-\!\!*} \triangleright(Q * Y)) * (P \mathrel{-\!\!*} Z) \\
& \quad \textit{Property (3), frame off saved propositions.} \\
\Longrightarrow & (X \mathrel{-\!\!*} \triangleright(Q * Y)) * (P \mathrel{-\!\!*} Z) * (\triangleright Q \Rightarrow \triangleright P) \\
& \quad \textit{Apply SMono, } \triangleright(P \mathrel{-\!\!*} Q) \implies (\triangleright P \mathrel{-\!\!*} \triangleright Q) \textit{, and } (P \Rightarrow Q) \implies (P \mathrel{-\!\!*} Q). \\
\Longrightarrow & (X \mathrel{-\!\!*} \triangleright(Q * Y)) * (\triangleright P \mathrel{-\!\!*} \triangleright Z) * (\triangleright Q \mathrel{-\!\!*} \triangleright P) \\
& \quad \textit{Merge } \mathrel{-\!\!*}. \\
\Longrightarrow & (X \mathrel{-\!\!*} \triangleright(Q * Y)) * (\triangleright Q \mathrel{-\!\!*} \triangleright Z) \\
& \quad \textit{Add frame } \triangleright Y \textit{ to } \mathrel{-\!\!*}\textit{, rearrange } \triangleright. \\
\Longrightarrow & (X \mathrel{-\!\!*} \triangleright(Q * Y)) * (\triangleright(Q * Y) \mathrel{-\!\!*} \triangleright(Z * Y)) \\
& \quad \textit{Merge } \mathrel{-\!\!*}\textit{, frame saved propositions back on.} \\
\Longrightarrow & r \overset{\pi_1}{\Longmapsto} P * r \overset{\pi_2}{\Longmapsto} Q * (X \mathrel{-\!\!*} \triangleright(Z * Y)) \quad \square
\end{array}
$$

A.1. Encoding in iCAP

A saved proposition $r \overset{\pi}{\Mapsto} P$ is encoded as a normal iCAP predicate with a structure guaranteeing the properties we want. Intuitively, this predicate consists of a shared region with identifier r, with the proposition P encoded into its transition relation. Linearity comes from a permission with fractional argument π.

More formally, we assume two transition-system states $\{\mathsf{1st}, \mathsf{2nd}\}$ and a single token tok, and invariant map I_{prp} and transition relation T_{prp} defined as follows:

$$\begin{aligned} I_{\mathsf{prp}}(Q)(\mathsf{1st}) &\triangleq \mathsf{emp} \\ I_{\mathsf{prp}}(Q)(\mathsf{2nd}) &\triangleq Q \\ T_{\mathsf{prp}}(\mathsf{tok}) &\triangleq \{(\mathsf{1st}, \mathsf{2nd})\} \end{aligned}$$

We then define the saved proposition $r \overset{\pi}{\Mapsto} Q$ as follows:

$$r \overset{\pi}{\Mapsto} Q \quad \triangleq \quad \mathsf{region}(\{\mathsf{1st}, \mathsf{2nd}\}, T_{\mathsf{prp}}, I_{\mathsf{prp}}(Q), r) * [\mathsf{tok}]^{\pi}_{r}$$

The fact that the representation transition state $\mathsf{1st}$ is emp means that we are not obliged to supply P when creating the saved proposition. The second state encodes the value of the saved proposition.

The linearity property (Property (2)) holds trivially from the linearity of permissions. Two saved propositions with arguments π_1 and π_2 must contain tok permissions with fractional arguments π_1 and π_2. Combining these gives the required result.

For the unification property (Property (3)), we need to reason more deeply about the iCAP model. The following facts about regions and invariant maps hold in iCAP—for proofs see Svendsen and Birkedal [2014b].

$$\mathsf{region}(S, T, I, r) * \mathsf{region}(S', T', J, r) \implies (\triangleright I(s) \Rightarrow \triangleright J(s)) \tag{5}$$

$$\mathsf{region}(S, T, I, r) * \mathsf{region}(S', T', J, r) \implies (\triangleright I(s) \mathrel{-\!\!*} \triangleright J(s)) \tag{6}$$

The later modality, $\triangleright$, is needed in these properties because we are reasoning about the contents of a shared region—albeit one that will not contain any resource. We can then prove the unification property as follows:

Proof (Property (3)).

$$\begin{aligned} & r \overset{\pi_1}{\Mapsto} P * r \overset{\pi_2}{\Mapsto} Q \\ \Rightarrow\ & r \overset{\pi_1}{\Mapsto} P * \mathsf{region}(\{\mathsf{1st}, \mathsf{2nd}\}, T_{\mathsf{prp}}, I_{\mathsf{prp}}(P), r) && \mathsf{region}\ \textit{is duplicable.} \\ & \quad * r \overset{\pi_2}{\Mapsto} Q * \mathsf{region}(\{\mathsf{1st}, \mathsf{2nd}\}, T_{\mathsf{prp}}, I_{\mathsf{prp}}(Q), r) \\ \Rightarrow\ & r \overset{\pi_1}{\Mapsto} P * r \overset{\pi_2}{\Mapsto} Q * (\triangleright(I_{\mathsf{prp}}(P)(\mathsf{2nd})) \Rightarrow \triangleright(I_{\mathsf{prp}}(Q)(\mathsf{2nd}))) && \textit{Property (5).} \\ \Rightarrow\ & r \overset{\pi_1}{\Mapsto} P * r \overset{\pi_2}{\Mapsto} Q * (\triangleright P \Rightarrow \triangleright Q) && \textit{defn of } I_{\mathsf{prp}} \quad \square \end{aligned}$$

For unification inside the separating implication, we reason as follows:

Proof (Property (4)). Using Property (6) and the same proof technique as earlier, we can derive a slightly different version of the unification property:

$$r \overset{\pi_1}{\Mapsto} P * r \overset{\pi_2}{\Mapsto} Q \quad \implies \quad (\triangleright P) \mathrel{-\!\!*} (\triangleright Q) \tag{7}$$

The proof of Property (4) then goes as follows:

$$
\begin{array}{ll}
& r \overset{\pi_1}{\Longrightarrow} P * r \overset{\pi_2}{\Longrightarrow} Q * (X \mathrel{-\!\!*} \triangleright(Q * Y)) * (P \mathrel{-\!\!*} Z) \\
& \quad \textit{SMono} \\
\Rightarrow & r \overset{\pi_1}{\Longrightarrow} P * r \overset{\pi_2}{\Longrightarrow} Q * (X \mathrel{-\!\!*} \triangleright(Q * Y)) * \triangleright(P \mathrel{-\!\!*} Z) \\
& \quad \textit{LBin, assume } \triangleright \textit{ distributes over } \mathrel{-\!\!*} \\
\Rightarrow & r \overset{\pi_1}{\Longrightarrow} P * r \overset{\pi_2}{\Longrightarrow} Q * (X \mathrel{-\!\!*} (\triangleright Q * \triangleright Y)) * ((\triangleright P) \mathrel{-\!\!*} \triangleright Z) \\
& \quad \textit{Assume property 7} \\
\Rightarrow & r \overset{\pi_1}{\Longrightarrow} P * r \overset{\pi_2}{\Longrightarrow} Q * (X \mathrel{-\!\!*} (\triangleright Q * \triangleright Y)) * ((\triangleright P) \mathrel{-\!\!*} \triangleright Z) * ((\triangleright Q) \mathrel{-\!\!*} (\triangleright P)) \\
& \quad \textit{Transitivity of } \mathrel{-\!\!*} \\
\Rightarrow & r \overset{\pi_1}{\Longrightarrow} P * r \overset{\pi_2}{\Longrightarrow} Q * (X \mathrel{-\!\!*} (\triangleright Q * \triangleright Y)) * ((\triangleright Q) \mathrel{-\!\!*} \triangleright Z) \\
& \quad \textit{Framing of } \mathrel{-\!\!*} \\
\Rightarrow & r \overset{\pi_1}{\Longrightarrow} P * r \overset{\pi_2}{\Longrightarrow} Q * (X \mathrel{-\!\!*} (\triangleright Q * \triangleright Y)) * ((\triangleright Q) * (\triangleright Y)) \mathrel{-\!\!*} ((\triangleright Z) * (\triangleright Y)) \\
& \quad \textit{Transitivity of } \mathrel{-\!\!*} \\
\Rightarrow & r \overset{\pi_1}{\Longrightarrow} P * r \overset{\pi_2}{\Longrightarrow} Q * (X \mathrel{-\!\!*} ((\triangleright Z) * (\triangleright Y))) \quad \square
\end{array}
$$

B. PROOFS FOR OUT-OF-ORDER SIGNALING

This appendix gives proofs for some of the lemmas stated in Section 6.

LEMMA 6.1.

$$\mathsf{resource}(\mathcal{I}, \emptyset) \sqsubseteq \circledast_{i \in \mathcal{I}} . \exists Q : \mathsf{Prop}.\ i \overset{1/2}{\Longrightarrow} Q(i) * \lceil \triangleright Q(i) \rceil$$

PROOF.

$$
\begin{array}{ll}
& \mathcal{W} = \emptyset \wedge \exists Q : \mathcal{I} \to \mathsf{Prop},\ R : \mathcal{W} \to \mathsf{Prop}. \\
& \qquad \circledast_{i \in \mathcal{I}} . i \overset{1/2}{\Longrightarrow} Q(i) * \circledast_{w \in \mathcal{W}} . w \overset{1/2}{\Longrightarrow} R(w) \\
& \qquad * \lfloor (\triangleright \circledast_{w \in \mathcal{W}} . R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}} . \lceil Q(i) \rceil \rfloor \\
& \quad \textit{Simplify using } \mathcal{W} = \emptyset, \textit{ weakening.} \\
\sqsubseteq & \exists Q : \mathcal{I} \to \mathsf{Prop}.\ \circledast_{i \in \mathcal{I}} . i \overset{1/2}{\Longrightarrow} Q(i) * \lfloor \triangleright \circledast_{i \in \mathcal{I}} . Q(i) \rfloor \\
& \quad \textit{Switch from } \lfloor - \rfloor \textit{ to } \lceil - \rceil\textit{, pull out } \circledast. \\
\sqsubseteq & \exists Q : \mathcal{I} \to \mathsf{Prop}.\ \circledast_{i \in \mathcal{I}} . i \overset{1/2}{\Longrightarrow} Q(i) * \circledast_{i \in \mathcal{I}} . \lceil \triangleright Q(i) \rceil \\
& \quad \textit{Push in the existential.} \\
\sqsubseteq & \circledast_{i \in \mathcal{I}} . \exists Q : \mathsf{Prop}.\ i \overset{1/2}{\Longrightarrow} Q(i) * \lceil \triangleright Q(i) \rceil \quad \square
\end{array}
$$

LEMMA 6.2. $r \overset{1/2}{\Longrightarrow} P * \lfloor P \rfloor * \mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r\}) \sqsubseteq \mathsf{resource}(\mathcal{I}, \mathcal{W})$

PROOF.

$$
\begin{array}{ll}
& r \overset{1/2}{\Longrightarrow} R * \lfloor R \rfloor * \mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r\}) \\
& \quad \textit{Definition of } \mathsf{resource}. \\
\sqsubseteq & r \overset{1/2}{\Longrightarrow} R * \lfloor R \rfloor * \\
& \left(\begin{array}{l} \exists Q : \mathcal{I} \to \mathsf{Prop},\ R : \mathcal{W} \uplus \{r\} \to \mathsf{Prop}. \\ \circledast_{i \in \mathcal{I}} . i \overset{1/2}{\Longrightarrow} Q(i) * \circledast_{w \in \mathcal{W} \uplus \{r\}} . w \overset{1/2}{\Longrightarrow} R(w) \\ * \lfloor (\triangleright \circledast_{w \in \mathcal{W} \uplus \{r\}} . R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}} . \lceil Q(i) \rceil \rfloor \end{array} \right) \\
& \quad \textit{Property } (3)\textit{, monotonicity of } \triangleright\textit{, monotonicity of } \lfloor - \rfloor.
\end{array}
$$

$$
\begin{array}{ll}
\sqsubseteq & \exists P : \mathsf{Prop}.\ r \overset{1/2}{\Longrightarrow} R * \lfloor \triangleright R \rfloor * (\lfloor \triangleright R \rfloor \Rightarrow \lfloor \triangleright P \rfloor) \\
& \left(\begin{array}{c} \exists Q : \mathcal{I} \to \mathsf{Prop},\ R : \mathcal{W} \to \mathsf{Prop}. \\ r \overset{1/2}{\Longrightarrow} P * \circledast_{i \in \mathcal{I}}.\ i \overset{1/2}{\Longrightarrow} Q(i) * \circledast_{w \in \mathcal{W}}.\ w \overset{1/2}{\Longrightarrow} R(w) \\ * \lfloor (\triangleright P * \triangleright \circledast_{w \in \mathcal{W}}.\ R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}}.\ \lceil Q(i) \rceil \rfloor \end{array} \right) \\
& \textit{Modus ponens.} \\
\sqsubseteq & \exists P : \mathsf{Prop}.\ r \overset{1/2}{\Longrightarrow} R * \lfloor \triangleright P \rfloor * \\
& \left(\begin{array}{c} \exists Q : \mathcal{I} \to \mathsf{Prop},\ R : \mathcal{W} \to \mathsf{Prop}. \\ r \overset{1/2}{\Longrightarrow} P * \circledast_{i \in \mathcal{I}}.\ i \overset{1/2}{\Longrightarrow} Q(i) * \circledast_{w \in \mathcal{W}}.\ w \overset{1/2}{\Longrightarrow} R(w) \\ * \lfloor (\triangleright P * \triangleright \circledast_{w \in \mathcal{W}}.\ R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}}.\ \lceil Q(i) \rceil \rfloor \end{array} \right) \\
& \textit{Combine } \lfloor - \rfloor \textit{, modus ponens for } -\!\!*\textit{, weakening.} \\
\sqsubseteq & \exists Q : \mathcal{I} \to \mathsf{Prop},\ R : \mathcal{W} \to \mathsf{Prop}. \\
& \circledast_{i \in \mathcal{I}}.\ i \overset{1/2}{\Longrightarrow} Q(i) * \circledast_{w \in \mathcal{W}}.\ w \overset{1/2}{\Longrightarrow} R(w) \\
& * \lfloor (\triangleright \circledast_{w \in \mathcal{W}}.\ R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}}.\ \lceil Q(i) \rceil \rfloor \\
\sqsubseteq & \mathsf{resource}(\mathcal{I}, \mathcal{W}) \quad \square
\end{array}
$$

LEMMA 6.3. $\mathsf{chainres}(rs) \wedge \mathsf{wf}(rs) \wedge \mathsf{ctrue}(rs)$
$\sqsubseteq\ \exists rs'.\ \mathsf{chainres}(rs') \wedge \mathsf{cconf}(rs') \wedge rs \overset{pr}{\to}{}^{*}\ rs' \wedge \mathsf{wf}(rs')$

PROOF. We perform a sequence of smaller view shifts corresponding to converting each cnode in turn, starting with the earliest element in the chain:

$$P_0 \sqsubseteq P_1 \sqsubseteq P_2 \sqsubseteq \ldots \sqsubseteq P_n$$

Here n is the length of rs and the subscript $1, 2, 3 \ldots$ denotes the length of suffix of rs that has been checked. We write $rs[a, b]$ for the subsequence of rs from element a to element b, inclusive of both. Thus, the inductive invariant is

$$
\begin{array}{lll}
P_i & \triangleq & \exists rs'.\ \mathsf{chainres}(rs[0, n-i] \cdot rs') \wedge \mathsf{cconf}(rs') \\
& & \wedge\ rs[(n-i)+1, n] \overset{pr}{\to}{}^{*}\ rs' \wedge \mathsf{wf}(rs[0, (n-i)] \cdot rs')
\end{array}
$$

The base case of the proof is simple. If $i = 0$, take rs' to be empty and the invariant follows trivially from the premise. Let us assume that $i > 0$. We reason as follows to pull out the intermediate chain node:

$$
\begin{array}{l}
\exists rs'.\ \mathsf{chainres}(rs[0, n-i] \cdot rs') \wedge \mathsf{cconf}(rs') \wedge rs[(n-i)+1, n] \overset{pr}{\to}{}^{*}\ rs' \wedge \mathsf{wf}(rs[0, n-i] \cdot rs') \\
\sqsubseteq \\
\exists rs', s.\ \mathsf{chainres}(rs[0, n-(i+1)]) * \mathsf{resource}(s.\mathcal{I}, s.\mathcal{W}) * \mathsf{chainres}(rs') \\
\wedge\ \mathsf{cconf}(rs') \wedge rs[n-i, n] \overset{pr}{\to}{}^{*}\ s \cdot rs' \wedge \mathsf{wf}(rs[0, n-(i+1))] \cdot s \cdot rs')
\end{array}
$$

If $s.\mathcal{W} = \emptyset$, then we are done. Otherwise, we induct on the size of $\mathcal{W}$, showing that it can be reduced to $\emptyset$ by classical entailment. (Recall that by definition, $\mathcal{W}$ is finite.)

Pick an element $w \in \mathcal{W}$. Since the chain is well formed, the region identifier w must also be a member of some set $s'.\mathcal{I}$ for an earlier element $s' \in rs'$. If w is a member of $s'.\mathcal{I}$ for multiple s', we pick the first such $s' \in rs'$. Since cconf holds for rs', it follows that $s'.\mathcal{W} = \emptyset$. Thus, there exists rs'_1, rs'_2, and s' such that $w \in s'.\mathcal{I}$, $s'.\mathcal{W} = \emptyset$, $rs' = rs'_1 \cdot s' \cdot rs'_2$, and $\forall x \in rs'_1.\ w \notin x.\mathcal{I}$. By the definition of resource, there exists a saved proposition $w \overset{1/2}{\Longrightarrow} R$ inside $\mathsf{resource}(s.\mathcal{I}, s.\mathcal{W})$. By the same definition, the saved proposition $w \overset{1/2}{\Longrightarrow} P$ and resource $\triangleright P$ must be included in $\mathsf{resource}(s'.\mathcal{I}, \emptyset)$. We move the resource from one

node to the other and delete the saved proposition using weakening:

$$\begin{array}{l}\mathsf{resource}(s.\mathcal{I}, s.\mathcal{W}) * \mathsf{chainres}(rs'_1) * \mathsf{resource}(s'.\mathcal{I}, \emptyset) * \mathsf{chainres}(rs'_2)\\ \quad\sqsubseteq\\ \mathsf{resource}(s.\mathcal{I}, s.\mathcal{W} \setminus \{w\}) * \mathsf{chainres}(rs'_1) * \mathsf{resource}(s'.\mathcal{I} \setminus \{w\}, \emptyset) * \mathsf{chainres}(rs'_2)\end{array}$$

Let rs'' denote $rs'_1 \cdot s'[\mathcal{I} \mapsto s'.\mathcal{I} \setminus \{w\}] \cdot rs'_2$. By definition of $\xrightarrow{\mathcal{W}}$, it thus follows that

$$rs[n-i, n] \xrightarrow{pr}{}^{*} s \cdot rs' \xrightarrow{\mathcal{W}} s[\mathcal{W} \mapsto s.\mathcal{W} \setminus \{w\}] \cdot rs''$$

Hence, by Lemma B.1 it follows that

$$\mathsf{wf}(rs[0, n-(i+1))] \cdot s[\mathcal{W} \mapsto s.\mathcal{W} \setminus \{w\}] \cdot rs'')$$

The result is that the assertion is rewritten as follows:

$$\begin{array}{ll}\sqsubseteq & \exists rs', s.\ \mathsf{chainres}(rs[0, n-(i+1)]) * \mathsf{resource}(s.\mathcal{I}, s.\mathcal{W} \setminus \{w\}) * \mathsf{chainres}(rs')\\ & \quad \wedge\ \mathsf{cconf}(rs') \wedge rs[n-i, n] \xrightarrow{pr}{}^{*} s \cdot rs' \wedge \mathsf{wf}(rs[0, n-(i+1))] \cdot s \cdot rs')\end{array}$$

Thus, we have rewritten $s.\mathcal{W}$ into a smaller set. By inducting on the size of this set, we can get to the point where $\mathcal{W}' = \emptyset$. This allows us to complete one step of the outer induction, which completes the inductive proof. □

LEMMA B.1.

$$\mathsf{wf}(rs) \wedge rs \xrightarrow{\mathcal{W}} rs' \Rightarrow (\mathsf{available}(rs) = \mathsf{available}(rs') \wedge \mathsf{wf}(rs'))$$

PROOF. By definition of $rs \xrightarrow{\mathcal{W}} rs'$, there exists $rs_1, rs_2, rs_3, s_1, s_2$, and w such that

$$\begin{array}{c}rs = rs_1 \cdot s_1 \cdot rs_2 \cdot s_2 \cdot rs_3, \qquad w \in s_2.\mathcal{I}, \qquad w \in s_1.\mathcal{W},\\ rs' = rs_1 \cdot (s_1 \blacktriangleleft_{\mathcal{W}} (\bullet \setminus w)) \cdot rs_2 \cdot (s_2 \blacktriangleleft_{\mathcal{I}} (\bullet \setminus w)) \cdot rs_3\end{array}$$

Since $s_2.\mathcal{W} \cap s_2.\mathcal{I} = \emptyset$ and $\mathsf{wf}(s_2 \cdot rs_3)$, it follows that $w \notin (s_2.\mathcal{W} \cup \mathsf{available}(rs_3))$ and $\forall s \in rs_2.\ w \notin s.\mathcal{I}$. Thus,

$$\begin{array}{rl}\mathsf{available}((s_2 \blacktriangleleft_{\mathcal{I}} (\bullet \setminus w)) \cdot rs_3) &= (\mathsf{available}(rs_3) \setminus s_2.\mathcal{W}) \uplus (s_2.\mathcal{I} \setminus \{w\})\\ &= \mathsf{available}(s_2 \cdot rs_3) \setminus \{w\}\end{array}$$

Since $w \in s_1.\mathcal{W}$ and $\mathsf{wf}(s_1 \cdot rs_2 \cdot s_2 \cdot rs_3)$, it follows that $w \in \mathsf{available}(rs_2 \cdot s_2 \cdot rs_3)$. Hence, since $\forall s \in rs_2.\ w \notin s.\mathcal{I}$, it follows that $\forall s \in rs_2.\ w \notin s.\mathcal{W}$. Thus,

$$\begin{array}{rl}& \mathsf{available}((s_1 \blacktriangleleft_{\mathcal{W}} (\bullet \setminus w)) \cdot rs_2 \cdot (s_2 \blacktriangleleft_{\mathcal{I}} (\bullet \setminus w)) \cdot rs_3)\\ =\ & (\mathsf{available}(rs_2 \cdot (s_2 \blacktriangleleft_{\mathcal{I}} (\bullet \setminus w)) \cdot rs_3) \setminus (s_1.\mathcal{W} \setminus \{w\})) \uplus s_1.\mathcal{I}\\ =\ & ((\mathsf{available}(rs_2 \cdot s_2 \cdot rs_3) \setminus \{w\}) \setminus (s_1.\mathcal{W} \setminus \{w\})) \uplus s_1.\mathcal{I}\\ =\ & (\mathsf{available}(rs_2 \cdot s_2 \cdot rs_3) \setminus s_1.\mathcal{W}) \uplus s_1.\mathcal{I}\\ =\ & \mathsf{available}(s_1 \cdot rs_2 \cdot s_2 \cdot rs_3)\end{array}$$

from which it follows easily that $\mathsf{available}(rs) = \mathsf{available}(rs')$. To show that $\mathsf{wf}(rs')$, we must also show that

$$\begin{array}{rl}s_1.\mathcal{W} \setminus \{w\} &\subseteq \mathsf{available}(rs_2 \cdot (s_2 \blacktriangleleft_{\mathcal{I}} (\bullet \setminus w)) \cdot rs_3)\\ &= \mathsf{available}(rs_2 \cdot s_2 \cdot rs_3) \setminus \{w\}\end{array}$$

and $s_2.\mathcal{W} \subseteq \mathsf{available}(rs_3)$, both of which follow easily from the assumption that $\mathsf{wf}(rs)$. It remains to show that all sets $\mathcal{I}$ for the chain are pairwise disjoint, and likewise for all sets $\mathcal{W}$. As we have only removed identifiers, this is satisfied trivially. □

LEMMA 6.4. $\mathsf{resource}(\mathcal{I} \uplus r_2, \emptyset) * r_2 \overset{1/2}{\Longmapsto} P \quad \sqsubseteq \quad \mathsf{resource}(\mathcal{I}, \emptyset) * \lceil \triangleright P \rceil$

PROOF.

$\mathsf{resource}(\mathcal{I} \uplus r_2, \emptyset) * r_2 \overset{1/2}{\Longmapsto} P$
Lemma 6.1.
$\sqsubseteq \quad r_2 \overset{1/2}{\Longmapsto} P * \circledast_{i \in \mathcal{I} \uplus r_2} . \exists R.\, i \overset{1/2}{\Longmapsto} R * \lceil \triangleright R \rceil$
Pull out resource for identifier r_2.
$\sqsubseteq \quad r_2 \overset{1/2}{\Longmapsto} P * \exists R'.\, r_2 \overset{1/2}{\Longmapsto} R' * \lceil \triangleright R' \rceil * \circledast_{i \in \mathcal{I}} . \exists R.\, i \overset{1/2}{\Longmapsto} R * \lceil \triangleright R \rceil$
Property (3)*, monotonicity of* $\lceil - \rceil$.
$\sqsubseteq \quad r_2 \overset{1/2}{\Longmapsto} P * \exists R'.\, (\lceil \triangleright R' \rceil \Rightarrow \lceil \triangleright P \rceil) * r_2 \overset{1/2}{\Longmapsto} R' * \lceil \triangleright R' \rceil * \circledast_{i \in \mathcal{I}} . \exists R.\, i \overset{1/2}{\Longmapsto} R * \lceil \triangleright R \rceil$
Modus ponens, weakening.
$\sqsubseteq \quad \lceil \triangleright P \rceil * \circledast_{i \in \mathcal{I}} . \exists R.\, i \overset{1/2}{\Longmapsto} R * \lceil \triangleright R \rceil$
Definition of resource.
$\sqsubseteq \quad \lceil \triangleright P \rceil * \mathsf{resource}(\mathcal{I}, \emptyset) \quad \square$

LEMMA 6.7.

$$\{\mathsf{oreg}(r, \{\mathsf{Chain}(rs) \mid x \in rs\}) * \mathsf{oreg}(r', \{\mathsf{Chain}(rs') \mid x \in rs'\})\} \quad \langle \mathtt{skip} \rangle \quad \{r = r'\}$$

PROOF. Prove this by case-splitting on whether the two regions are equal. Suppose the two are equal—then the specification is proved. If they are unequal, we prove this leads to a contradiction by opening both regions and examining their contents. Each region asserts exclusive ownership of heap cell $x.loc$, which leads to a contradiction. Therefore, the postcondition is false, allowing us to prove any postcondition. $\square$

LEMMA 6.8. $\mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r\}) * r \overset{1/2}{\Longmapsto} S * r' \overset{1/2}{\Longmapsto} T_1$
$\sqsubseteq \quad \exists r''.\, \mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r', r''\}) * r'' \overset{1/2}{\Longmapsto} (T_1 \mathrel{-\!\!*} S)$

PROOF. First, we construct a new saved proposition r'' such that $r'' \mapsto (T_1 \mathrel{-\!\!*} S)$. Now it suffices to prove

$\mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r\}) * r \overset{1/2}{\Longmapsto} S * r' \overset{1/2}{\Longmapsto} T_1 * r'' \overset{1/2}{\Longmapsto} T_2 \wedge \mathsf{valid}(T_1 * T_2 \Rightarrow S)$
$\sqsubseteq \quad \mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r', r''\})$

$$r \overset{1/2}{\Longmapsto} S * r' \overset{1/2}{\Longmapsto} T_1 * r'' \overset{1/2}{\Longmapsto} T_2 \wedge \mathsf{valid}(T_1 * T_2 \Rightarrow S) * \left(\begin{array}{l} \exists Q : \mathcal{I} \to \mathsf{Prop},\ R : \mathcal{W} \uplus \{r\} \to \mathsf{Prop}. \\ \circledast_{i \in \mathcal{I}} . \, i \overset{1/2}{\Longmapsto} Q(i) * \circledast_{w \in \mathcal{W} \uplus \{r\}} . \, w \overset{1/2}{\Longmapsto} R(w) \\ * \lfloor (\triangleright \circledast_{w \in \mathcal{W} \uplus \{r\}} . \, R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}} . \, \lceil Q(i) \rceil \rfloor \end{array} \right)$$

Rearrange

$$\begin{array}{ll} \sqsubseteq & \exists P : \mathsf{Prop},\ Q : \mathcal{I} \to \mathsf{Prop},\ R : \mathcal{W} \uplus \{r', r''\} \to \mathsf{Prop}. \\ & \quad \mathsf{valid}(R(r') * R(r'') \Rightarrow S) \wedge r \overset{1/2}{\Longmapsto} S * r \overset{1/2}{\Longmapsto} P \\ & \quad * \circledast_{i \in \mathcal{I}} . \, i \overset{1/2}{\Longmapsto} Q(i) * \circledast_{w \in \mathcal{W} \uplus \{r', r''\}} . \, w \overset{1/2}{\Longmapsto} R(w) \\ & \qquad * \lfloor \triangleright P * \triangleright \circledast_{w \in \mathcal{W}} . \, R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}} . \, \lceil Q(i) \rceil \rfloor \end{array}$$

Property (3)*, SMono and distributing* $\triangleright$ *over* $\Rightarrow$.

$$\begin{array}{ll} \sqsubseteq & \exists P : \mathsf{Prop},\ Q : \mathcal{I} \to \mathsf{Prop},\ R : \mathcal{W} \uplus \{r', r''\} \to \mathsf{Prop}. \\ & \quad \mathsf{valid}((\triangleright R(r')) * (\triangleright R(r'')) \Rightarrow \triangleright P) \wedge \circledast_{i \in \mathcal{I}} . \, i \overset{1/2}{\Longmapsto} Q(i) * \circledast_{w \in \mathcal{W} \uplus \{r', r''\}} . \, w \overset{1/2}{\Longmapsto} R(w) \\ & \qquad * \lfloor (\triangleright P * \triangleright \circledast_{w \in \mathcal{W}} . \, R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}} . \, \lceil Q(i) \rceil \rfloor \end{array}$$

By mono of $\lfloor - \rfloor$, $\mathrel{-\!\!*}$ *and* $*$.

$$\sqsubseteq \quad \exists Q : \mathcal{I} \to \mathsf{Prop},\ R : \mathcal{W} \uplus \{r', r''\} \to \mathsf{Prop}.$$
$$\circledast_{i\in\mathcal{I}} . i \overset{1/2}{\Longmapsto} Q(i) * \circledast_{w\in\mathcal{W}\uplus\{r',r''\}} . w \overset{1/2}{\Longmapsto} R(w)$$
$$* \lfloor((\triangleright R(r')) * (\triangleright R(r'')) * \triangleright \circledast_{w\in\mathcal{W}} . R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i\in\mathcal{I}} . \lceil Q(i)\rceil\rfloor$$

Rearrange

$$\sqsubseteq \quad \exists P : \mathsf{Prop},\ Q : \mathcal{I} \to \mathsf{Prop},\ R : \mathcal{W} \uplus \{r', r''\} \to \mathsf{Prop}.$$
$$\circledast_{i\in\mathcal{I}} . i \overset{1/2}{\Longmapsto} Q(i) * \circledast_{w\in\mathcal{W}\uplus\{r',r''\}} . w \overset{1/2}{\Longmapsto} R(w)$$
$$* \lfloor \triangleright \circledast_{w\in\mathcal{W}\uplus\{r',r''\}} . R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i\in\mathcal{I}} . \lceil Q(i)\rceil\rfloor \quad \square$$

LEMMA B.2.

$$r_2 \overset{1/2}{\Longmapsto} P * \lfloor\lceil P\rceil \mathrel{-\!\!*} (P_1 * P_2)\rfloor * r_3 \overset{1/2}{\Longmapsto} P_1 * r_4 \overset{1/2}{\Longmapsto} P_2 * \mathsf{resource}(\mathcal{I} \uplus \{r_2\}, \mathcal{W})$$
$$\sqsubseteq \quad \mathsf{resource}(\mathcal{I} \uplus \{r_3, r_4\}, \mathcal{W})$$

PROOF.

$$r_2 \overset{1/2}{\Longmapsto} P * \lfloor\lceil P\rceil \mathrel{-\!\!*} (P_1 * P_2)\rfloor * r_3 \overset{1/2}{\Longmapsto} P_1 * r_4 \overset{1/2}{\Longmapsto} P_2 * \mathsf{resource}(\mathcal{I} \uplus \{r_2\}, \mathcal{W})$$

Definition of resource.

$$\sqsubseteq \quad r_2 \overset{1/2}{\Longmapsto} P * \lfloor\lceil P\rceil \mathrel{-\!\!*} (P_1 * P_2)\rfloor * r_3 \overset{1/2}{\Longmapsto} P_1 * r_4 \overset{1/2}{\Longmapsto} P_2$$
$$\left(\begin{array}{c} \exists Q : \mathcal{I} \uplus \{r_2\} \to \mathsf{Prop},\ R : \mathcal{W} \to \mathsf{Prop}. \\ \circledast_{i\in\mathcal{I}\uplus\{r_2\}} . i \overset{1/2}{\Longmapsto} Q(i) * \circledast_{w\in\mathcal{W}} . w \overset{1/2}{\Longmapsto} R(w) \\ * (\lfloor(\triangleright \circledast_{w\in\mathcal{W}} . R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i\in\mathcal{I}\uplus\{r_2\}} . \lceil Q(i)\rceil\rfloor) \end{array}\right)$$

Pull out r_2, $\triangleright$ *mono w.r.t.* $-\!\!*$, *Property* (4).

$$\sqsubseteq \quad r_2 \overset{1/2}{\Longmapsto} P * r_3 \overset{1/2}{\Longmapsto} P_1 * r_4 \overset{1/2}{\Longmapsto} P_2$$
$$\left(\begin{array}{c} \exists Q : \mathcal{I} \uplus \{r_2\} \to \mathsf{Prop},\ R : \mathcal{W} \to \mathsf{Prop}. \\ r_2 \overset{1/2}{\Longmapsto} Q(r_2) * \circledast_{i\in\mathcal{I}} . i \overset{1/2}{\Longmapsto} Q(i) * \circledast_{w\in\mathcal{W}} . w \overset{1/2}{\Longmapsto} R(w) \\ * \lfloor(\triangleright \circledast_{w\in\mathcal{W}} . R(w)) \mathrel{-\!\!*} \triangleright (P_1 * P_2 * \circledast_{i\in\mathcal{I}} . \lceil Q(i)\rceil)\rfloor) \end{array}\right)$$

Fold r_3, r_4 *into* $\mathcal{I}$, *weaken with* $\lceil\_\rceil$, *weakening.*

$$\sqsubseteq \quad \exists Q : \mathcal{I} \uplus r_2 \to \mathsf{Prop},\ R : \mathcal{W} \to \mathsf{Prop}.$$
$$\circledast_{i\in\mathcal{I}\uplus\{r_3,r_4\}} . i \overset{1/2}{\Longmapsto} Q(i) * \circledast_{w\in\mathcal{W}} . w \overset{1/2}{\Longmapsto} R(w)$$
$$* \lfloor(\triangleright \circledast_{w\in\mathcal{W}} . R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i\in\mathcal{I}\uplus\{r_3,r_4\}} . \lceil Q(i)\rceil\rfloor)$$

Definition of resource.

$$\sqsubseteq \quad \mathsf{resource}(\mathcal{I} \uplus \{r_3, r_4\}, \mathcal{W}) \quad \square$$

LEMMA 6.9. $r_2 \in rs(x).\mathcal{I} \wedge r_2 \overset{1/2}{\Longmapsto} P * \lfloor\lceil P\rceil \mathrel{-\!\!*} (P_1 * P_2)\rfloor * \mathsf{chainres}(rs)$
$$\sqsubseteq \quad \exists rs', r_3, r_4.\ r_3, r_4 \notin rs \wedge rs' = rs \blacktriangleleft_x \blacktriangleleft_{\mathcal{I}} (\bullet \setminus r_2) \uplus \{r_3, r_4\} \wedge$$
$$r_3 \overset{1/2}{\Longmapsto} P_1 * r_4 \overset{1/2}{\Longmapsto} P_2 * \mathsf{chainres}(rs')$$

PROOF.

$$r_2 \in rs(x).\mathcal{I} \wedge r_2 \overset{1/2}{\Longmapsto} P * \lfloor\lceil P\rceil \mathrel{-\!\!*} (P_1 * P_2)\rfloor * \mathsf{chainres}(rs)$$

Make saved propositions, fresh by construction.

$$\sqsubseteq \quad r_2 \in rs(x).\mathcal{I} \wedge r_2 \overset{1/2}{\Longmapsto} P * \lfloor\lceil P\rceil \mathrel{-\!\!*} (P_1 * P_2)\rfloor * \mathsf{chainres}(rs) *$$
$$\exists r_3, r_4.\ r_3 \overset{1}{\Longmapsto} P_1 * r_4 \overset{1}{\Longmapsto} P_2 \wedge r_3, r_4 \notin rs$$

Pull out resource *predicate for* x.

$$\sqsubseteq \quad \exists rs_1, rs_2.\, r_2 \in rs(x).\mathcal{I} \wedge r_2 \overset{1/2}{\Longrightarrow} P * \lfloor \ulcorner P \urcorner \mathrel{-\!\!*} (P_1 * P_2) \rfloor * \mathsf{chainres}(rs_1) * \mathsf{resource}(rs(x).\mathcal{I}, rs(x).\mathcal{W} \uplus \{rs(x).res \mid rs(x).flg = 0\}) * \mathsf{chainres}(rs_2) * \exists r_3, r_4.\, r_3 \overset{1}{\Longrightarrow} P_1 * r_4 \overset{1}{\Longrightarrow} P_2 \wedge r_3, r_4 \notin rs \wedge rs = rs_1 \cdot rs(x) \cdot rs_2$$

Apply Lemma B.8.

$$\sqsubseteq \quad \exists rs_1, rs_2, r_3, r_4.\, r_2 \in rs(x).\mathcal{I} \wedge r_2 \overset{1/2}{\Longrightarrow} P * \lfloor \ulcorner P \urcorner \mathrel{-\!\!*} (P_1 * P_2) \rfloor * \mathsf{chainres}(rs_1) * \mathsf{resource}((rs(x).\mathcal{I} \setminus r_2) \uplus \{r_3, r_4\}, rs(x).\mathcal{W} \uplus \{rs(x).res \mid rs(x).flg = 0\}) * \mathsf{chainres}(rs_2) * r_3 \overset{1/2}{\Longrightarrow} P_1 * r_4 \overset{1/2}{\Longrightarrow} P_2 \wedge r_3, r_4 \notin rs \wedge rs = rs_1 \cdot rs(x) \cdot rs_2$$

Definition of chainres.

$$\sqsubseteq \quad \exists rs', r_3, r_4.\, r_3, r_4 \notin rs \wedge rs' = rs \blacktriangleleft_x \blacktriangleleft_{\mathcal{I}} (\bullet \setminus r_2) \uplus \{r_3, r_4\} \wedge r_3 \overset{1/2}{\Longrightarrow} P_1 * r_4 \overset{1/2}{\Longrightarrow} P_2 * \mathsf{chainres}(rs') \quad \square$$

C. ICAP PROOF SYSTEM

In this appendix, we introduce the formal iCAP proof system. The introduction is self-contained but does not cover the full iCAP proof system. In particular, certain iCAP features, such as guarded recursive predicates and phantom state, are not necessary for the present article and have been dropped from the proof system.

C.1. Syntax

The proof system consists of two logics, an assertion logic and a specification logic, over a common simply typed term language generated by the following grammar:

$$\begin{aligned}
&\mathsf{M}, \mathsf{N}, \mathsf{P}, \mathsf{Q}, \mathsf{S}, \mathsf{T}, \mathsf{F}, \mathsf{R} ::= \\
&\quad \mid \lambda \mathsf{x} : \tau.\, \mathsf{M} \mid \mathsf{M}\,\mathsf{N} \mid \mathsf{x} \\
&\quad \mid \bot \mid \top \mid \mathsf{M} \vee \mathsf{N} \mid \mathsf{M} \wedge \mathsf{M} \mid \mathsf{M} \Rightarrow \mathsf{N} \mid \forall \mathsf{x} : \tau.\, \mathsf{P} \mid \exists \mathsf{x} : \tau.\, \mathsf{P} \mid \mathsf{M} =_\tau \mathsf{N} \\
&\quad \mid \mathsf{P} * \mathsf{Q} \mid \mathsf{P} \mathrel{-\!\!*} \mathsf{Q} \mid \mathsf{emp} \mid \mathsf{M}.\mathsf{F} \mapsto \mathsf{N} \mid \mathsf{M}{:}\mathsf{N} \mid \mathsf{region}(\mathsf{R}, \mathsf{M}, \mathsf{N}) \mid [\mathsf{M}]^{\mathsf{R}}_{\mathsf{N}} \mid \mathsf{stable}(\mathsf{P}) \\
&\quad \mid \mathsf{P} \sqsubseteq^{\mathsf{R}} \mathsf{Q} \mid (\Delta).\{\mathsf{P}\}\bar{\mathsf{s}}\{\mathsf{Q}\} \mid (\Delta).\langle \mathsf{P}\rangle \bar{\mathsf{s}} \langle \mathsf{Q}\rangle^{\mathsf{R}} \mid \mathsf{M}.\mathsf{N} : (\Delta).\{\mathsf{P}\}\{\mathsf{x}.\mathsf{Q}\} \mid \mathsf{M} : (\Delta).\{\mathsf{P}\}\{\mathsf{x}.\mathsf{Q}\} \\
&\quad \mid \triangleright \mathsf{M} \mid \mathsf{valid}(\mathsf{P}) \mid \mathsf{spec}(\mathsf{S}) \mid \Delta_X(x)
\end{aligned}$$

Here X is an arbitrary set and x an arbitrary element of X. The $\Delta_X(x)$ gives a shallow embedding of the meta-theory into iCAP. Correspondingly, the grammar of types (given next) features a type constructor $\Delta(X)$ for injecting arbitrary sets into iCAP.

$$\text{Types } \tau, \sigma ::= 1 \mid \tau \to \sigma \mid \tau \times \sigma \mid \tau + \sigma \mid \mathcal{P}(\tau) \mid \Delta(X) \mid \mathsf{Prop} \mid \mathsf{Spec}$$

In addition to the usual type constructors, iCAP includes two proposition types, one for each logic—the Prop type for the assertion logic and the Spec type for the specification logic. Base types for values, Val; state identifiers, SId; action identifiers, AId; region identifiers, RId; class names, Class; and field names, Field, are just syntactic sugar for injections of the corresponding set (e.g., Val is syntactic sugar for $\Delta(Val)$). We will usually leave out the explicit injection for elements when reasoning about elements of these injected types.

Well-Formed Terms $\boxed{\Gamma; \Delta \vdash \mathsf{M} : \tau}$

The typing rules of the logic are given later. The rules have been split into standard higher-order logic typing rules, followed by iCAP specific typing rules. Terms are typed in a logical variable context, Γ, and program variable context, Δ. Logical variables are used purely for specification purposes and may not appear free in the code of Hoare

triples. Program variables may appear free in both the pre- and postcondition of Hoare triples and in the code snippet. The logical variable context, Γ, maps variables to types, while all variables in the program variable context, Δ, have the type Val.

$$\frac{(x : \tau) \in \Gamma}{\Gamma; \Delta \vdash x : \tau} \qquad \frac{(x : \mathsf{Val}) \in \Delta}{\Gamma; \Delta \vdash x : \mathsf{Val}} \qquad \frac{\Gamma, x : \tau; \Delta \vdash \mathsf{M} : \sigma}{\Gamma; \Delta \vdash \lambda x : \tau.\, \mathsf{M} : \tau \to \sigma} \qquad \frac{\Xi \vdash \mathsf{M} : \tau \to \sigma \qquad \Xi \vdash \mathsf{N} : \tau}{\Xi \vdash \mathsf{M}\,\mathsf{N} : \sigma}$$

$$\frac{p \in \{\bot, \top, \mathsf{emp}\}}{\Xi \vdash p : \mathsf{Prop}} \qquad \frac{\Xi \vdash \mathsf{P} : \mathsf{Prop} \qquad \Xi \vdash \mathsf{Q} : \mathsf{Prop} \qquad op \in \{\vee, \wedge, \Rightarrow, *, -\!\!*\}}{\Xi \vdash \mathsf{P}\, op\, \mathsf{Q} : \mathsf{Prop}}$$

$$\frac{\Xi \vdash \mathsf{M} : \tau \qquad \Xi \vdash \mathsf{N} : \tau}{\Xi \vdash \mathsf{M} =_\tau \mathsf{N} : \mathsf{Prop}} \qquad \frac{\Gamma, x : \tau; \Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad Q \in \{\exists, \forall\}}{\Gamma; \Delta \vdash Qx : \tau.\, \mathsf{P} : \mathsf{Prop}} \qquad \frac{p \in \{\bot, \top\}}{\Gamma; - \vdash p : \mathsf{Spec}}$$

$$\frac{\Gamma; - \vdash \mathsf{S} : \mathsf{Spec} \qquad \Gamma; - \vdash \mathsf{T} : \mathsf{Spec} \qquad op \in \{\vee, \wedge, \Rightarrow\}}{\Gamma; - \vdash \mathsf{S}\, op\, \mathsf{T} : \mathsf{Spec}} \qquad \frac{\Gamma; - \vdash \mathsf{M} : \tau \qquad \Gamma; - \vdash \mathsf{N} : \tau}{\Gamma; - \vdash \mathsf{M} =_\tau \mathsf{N} : \mathsf{Spec}}$$

$$\frac{\Gamma, x : \tau; - \vdash \mathsf{S} : \mathsf{Spec} \qquad Q \in \{\forall, \exists\}}{\Gamma; - \vdash Qx : \tau.\, \mathsf{S} : \mathsf{Spec}}$$

$$\frac{\Xi \vdash \mathsf{M} : \mathcal{P}(\mathsf{SId}) \qquad \Xi \vdash \mathsf{I} : \mathsf{SId} \to \mathsf{Prop} \qquad \Xi \vdash \mathsf{T} : \mathsf{AId} \to \mathcal{P}(\mathsf{SId} \times \mathsf{SId}) \qquad \Xi \vdash \mathsf{R} : \mathsf{RId}}{\Xi \vdash \mathsf{region}(\mathsf{M}, \mathsf{I}, \mathsf{T}, \mathsf{R}) : \mathsf{Prop}}$$

$$\frac{\Xi \vdash \mathsf{A} : \mathsf{AId} \qquad \Xi \vdash \mathsf{R} : \mathsf{RId} \qquad \Xi \vdash \mathsf{P} : \mathsf{Perm}}{\Xi \vdash [\mathsf{A}]^{\mathsf{R}}_{\mathsf{P}} : \mathsf{Prop}} \qquad \frac{x \in X \qquad X \in obj(Sets)}{\Xi \vdash \Delta_X(x) : \Delta(X)}$$

$$\frac{\Xi \vdash \mathsf{M} : \mathsf{Val} \qquad \Xi \vdash \mathsf{C} : \mathsf{Class}}{\Xi \vdash \mathsf{M} : \mathsf{C} : \mathsf{Prop}} \qquad \frac{\Xi \vdash \mathsf{M} : \mathsf{Val} \qquad \Xi \vdash \mathsf{F} : \mathsf{Field} \qquad \Xi \vdash \mathsf{N} : \mathsf{Val}}{\Xi \vdash \mathsf{M}.\mathsf{F} \mapsto \mathsf{N} : \mathsf{Prop}}$$

$$\frac{\Gamma; - \vdash \mathsf{S} : \mathsf{Spec}}{\Gamma; \Delta \vdash \mathsf{spec}(\mathsf{S}) : \mathsf{Prop}} \qquad \frac{\Gamma; - \vdash \mathsf{P} : \mathsf{Prop}}{\Gamma; - \vdash \mathsf{valid}(\mathsf{P}) : \mathsf{Spec}} \qquad \frac{\Xi \vdash \mathsf{P} : \mathsf{Prop}}{\Xi \vdash \triangleright \mathsf{P} : \mathsf{Prop}} \qquad \frac{\Gamma; - \vdash \mathsf{P} : \mathsf{Prop}}{\Gamma; - \vdash \mathsf{stable}(\mathsf{P}) : \mathsf{Spec}}$$

$$\frac{\Gamma; - \vdash \mathsf{R} : \mathcal{P}(\mathsf{RId}) \qquad \Gamma; - \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; - \vdash \mathsf{Q} : \mathsf{Prop}}{\Gamma; - \vdash \mathsf{P} \sqsubseteq^{\mathsf{R}} \mathsf{Q} : \mathsf{Spec}} \qquad \frac{\Gamma; - \vdash \mathsf{S} : \mathsf{Spec}}{\Gamma; - \vdash \triangleright \mathsf{S} : \mathsf{Spec}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop} \qquad \mathsf{FV}(\mathsf{s}) \subseteq \mathsf{vars}(\Delta)}{\Gamma; - \vdash (\Delta).\{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} : \mathsf{Spec}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop} \qquad \Gamma; - \vdash \mathsf{R} : \mathcal{P}(\mathsf{RId}) \qquad \mathsf{FV}(\mathsf{s}) \subseteq \mathsf{vars}(\Delta)}{\Gamma; - \vdash (\Delta).\langle \mathsf{P} \rangle \mathsf{s} \langle \mathsf{Q} \rangle^{\mathsf{R}} : \mathsf{Spec}}$$

C.2. Logics

The iCAP proof system consists of two logics: an assertion logic for reasoning about the current state and a specification logic for reasoning about the behavior of programs. The specification logic is given by the specification entailment judgment $\Gamma \mid \Phi \vdash \mathsf{S}$, where S is a specification and Φ is a specification context. The assertion logic is given by the assertion entailment judgment $\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q}$, where P and Q are assertions and Φ is a specification context. The assertion entailment includes the specification context Φ, to allow the use of assertion assumptions embedded in specifications.

C.2.1. Specification and Assertion Embeddings. The valid embedding is used to export axioms about abstract resources in library specifications to clients. For instance, the axioms about the channel order (duplication and transitivity) are implicitly expressed as validities about the channel order resource. The introduction and elimination rule

for valid specifications are given here:

$$\frac{\Gamma; - \mid \Phi \mid \top \vdash \mathsf{P}}{\Gamma \mid \Phi \vdash \mathsf{valid}(\mathsf{P})} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{valid}(\mathsf{P})}{\Gamma; - \mid \Phi \mid \top \vdash \mathsf{P}}$$

C.2.2. Later Operator. The SLob rule internalizes induction on step indices in the logic and is implicitly used when verifying mutually recursive methods. In particular, to verify a method call, it suffices to know that the body of the called method satisfies a given specification one step later, to know that the call satisfies the given specification now. Intuitively, calling a method uses one step in the operational semantics before the method body starts executing.

$$\textsc{SLob}\ \frac{\Gamma \mid \Phi, \triangleright \mathsf{S} \vdash \mathsf{S}}{\Gamma \mid \Phi \vdash \mathsf{S}} \qquad \textsc{SMono}\ \frac{}{\Gamma \mid \Phi \vdash \mathsf{S} \Rightarrow \triangleright \mathsf{S}} \qquad \textsc{LPoints}\ \frac{\Gamma \mid \Phi \vdash \langle x.f \mapsto v \rangle\, c\, \langle \mathsf{Q} \rangle^{\mathcal{E}}}{\Gamma \mid \Phi \vdash \langle \triangleright (x.f \mapsto v) \rangle\, c\, \langle \mathsf{Q} \rangle^{\mathcal{E}}}$$

$$\textsc{LBin}\ \frac{op \in \{\wedge, \vee, *\}}{\Gamma; \Delta \mid \Phi \mid \triangleright(\mathsf{P}\, op\, \mathsf{Q}) \dashv\vdash (\triangleright \mathsf{P})\, op\, (\triangleright \mathsf{Q})} \qquad \textsc{LImpl}\ \frac{}{\Gamma; \Delta \mid \Phi \mid \triangleright(\mathsf{P} \Rightarrow \mathsf{Q}) \vdash (\triangleright \mathsf{P}) \Rightarrow (\triangleright \mathsf{Q})}$$

$$\textsc{LWand}\ \frac{}{\Gamma; \Delta \mid \Phi \mid \triangleright(\mathsf{P} \mathrel{-\!\!*} \mathsf{Q}) \vdash (\triangleright \mathsf{P}) \mathrel{-\!\!*} (\triangleright \mathsf{Q})} \qquad \textsc{LCeil}\ \frac{}{\Gamma; \Delta \mid \Phi \mid \triangleright \lceil \mathsf{P} \rceil \dashv\vdash \lceil \triangleright \mathsf{P} \rceil}$$

$$\textsc{LFloor}\ \frac{}{\Gamma; \Delta \mid \Phi \mid \triangleright \lfloor \mathsf{P} \rfloor \dashv\vdash \lfloor \triangleright \mathsf{P} \rfloor} \qquad \textsc{LQuant}\ \frac{Q \in \{\forall, \exists\}}{\Gamma; \Delta \mid \Phi \mid \triangleright(Qx : \tau.\, \mathsf{P}(x)) \dashv\vdash Qx : \tau. \triangleright \mathsf{P}(x)}$$

$$\frac{}{\Gamma; \Delta \mid \Phi \mid \mathsf{region}(X, T, I_1, r) * \mathsf{region}(Y, T, I_2, r) \vdash I_1(s) \Rightarrow I_2(s)}$$

C.2.3. View Shifts. The view-shift relation includes the standard assertion implication. In addition, it is transitive and supports framing of stable frames. There is no implicit assumption that the pre- and postcondition of view shifts are stable.

$$\textsc{VTrans}\ \frac{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{\mathcal{E}} \mathsf{Q} \qquad \Gamma \mid \Phi \vdash \mathsf{Q} \sqsubseteq^{\mathcal{E}} \mathsf{R}}{\Gamma \mid \Phi \vdash \mathsf{Q} \sqsubseteq^{\mathcal{E}} \mathsf{R}} \qquad \textsc{VImpl}\ \frac{\Gamma \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q}}{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{\mathcal{E}} \mathsf{Q}}$$

$$\textsc{VAlloc}\ \frac{\begin{array}{c}\Gamma \mid \Phi \vdash \Gamma \mid \Phi \vdash \forall \alpha \in \mathsf{AId}.\, \forall x \in \mathsf{SId} \times \mathsf{SId}.\, (\triangleright T(\alpha)(x)) \Rightarrow T(\alpha)(x) \vee \triangleright \bot \\ \Gamma \mid \Phi \vdash \mathcal{E} \text{ is infinite} \qquad \Gamma \mid \Phi \vdash \forall n \in \mathcal{E}.\, \mathsf{P} * \circledast_{\alpha \in A} [\alpha]_1^n \Rightarrow \triangleright I(n)(x) \\ \Gamma \mid \Phi \vdash \forall n \in \mathcal{E}.\, \forall s.\, \mathsf{stable}(I(n)(s)) \qquad \Gamma \mid \Phi \vdash A \cap B = \emptyset\end{array}}{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{\mathcal{E}} \exists n \in \mathcal{E}.\, \mathsf{region}(\{x\}, T, I(n), n) * \circledast_{\alpha \in B} [\alpha]_1^n}$$

$$\textsc{VOpen}\ \frac{\begin{array}{c}\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{P}) \qquad \Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{Q}) \\ \Gamma \mid \Phi \vdash \forall x \in X.\, f(x) \in Y \\ \Gamma \mid \Phi \vdash \forall x \in X.\, (x,\, f(x)) \in T(\alpha) \vee f(x) = x \\ \Gamma \mid \Phi \vdash \forall x \in X.\, \mathsf{P} * \triangleright I(x) * [\alpha]_\pi^n \sqsubseteq^{\mathcal{E}} \mathsf{Q} * \triangleright I(f(x))\end{array}}{\Gamma \mid \Phi \vdash \mathsf{region}(X, T, I, n) * \mathsf{P} * [\alpha]_\pi^n \sqsubseteq^{\mathcal{E} \uplus \{n\}} \mathsf{region}(Y, T, I, n) * \mathsf{Q}}$$

The VAlloc rule presented previously generalizes the region allocation rule presented in Section 4, by allowing the newly allocated region to immediately take ownership of action permissions on the newly allocated region ($\circledast_{\alpha\in A}[\alpha]^n_1$). In addition, it allows the allocator to pick an infinite set $\mathcal{E}$ of region identifiers, from which the region identifier of the newly allocated region will be chosen. This is used to reason about the inequality of region identifiers, which is necessary when reasoning about nested region openings.

C.2.4. Atomic Commands. All pre- and postconditions that appear in nonatomic Hoare triples are implicitly required to be stable.

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash \mathsf{P}, \mathsf{Q} : \mathsf{Prop} \qquad \mathsf{atomic}(\mathsf{s}) \\ \Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{P}) \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{Q}) \\ \Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P}\rangle \mathsf{s} \langle \mathsf{Q}\rangle^{\mathcal{E}}\end{array}}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\}}$$

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash \mathsf{P}, \mathsf{Q} : \mathsf{Prop} \qquad \Gamma, \Delta \vdash \mathcal{E}_1, \mathcal{E}_2 : \mathcal{P}(\mathsf{RId}) \\ \Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P}\rangle \mathsf{s} \langle \mathsf{Q}\rangle^{\mathcal{E}_1 \setminus \mathcal{E}_2}\end{array}}{\Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P}\rangle \mathsf{s} \langle \mathsf{Q}\rangle^{\mathcal{E}_1}}$$

Atomic

$$\frac{\begin{array}{c}\Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{P}) \qquad \Gamma, \Delta \mid \Phi \vdash \forall y.\, \mathsf{stable}(\mathsf{Q}(y)) \\ \Gamma, \Delta \mid \Phi \vdash \forall x \in X.\, (x,\, f(x)) \in \overline{T(A)} \vee f(x) = x \\ \Gamma \mid \Phi \vdash \forall x \in X.\, (\Delta).\langle \mathsf{P} * \circledast_{\alpha\in A}[\alpha]^n_{g(\alpha)} * \triangleright I(x)\rangle\ c\ \langle \mathsf{Q}(x) * \triangleright I(f(x))\rangle^{\mathcal{E}}\end{array}}{\Gamma \mid \Phi \vdash (\Delta).\, \langle \mathsf{P} * \circledast_{\alpha\in A}[\alpha]^n_{g(\alpha)} * \mathsf{region}(X, T, I, n)\rangle}$$

$$c$$

$$\langle \exists x.\, \mathsf{Q}(x) * \mathsf{region}(\{f(x)\}, T, I, n)\rangle^{\mathcal{E}\uplus\{n\}}$$

C.2.5. Stability.

$$\frac{\Gamma \mid \Phi \vdash \forall \alpha \notin A.\, \forall x \in X.\, T(\alpha)(x) \subseteq X}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{region}(X, T, I, n) * \circledast_{\alpha\in A}[\alpha]^n_1)} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\top)} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\bot)}$$

$$\frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{emp})} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{M.F} \mapsto \mathsf{N})} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{M} : \mathsf{N})}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{P}) \qquad \Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{Q}) \qquad op \in \{\vee, \wedge, *\}}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{P}\, op\, \mathsf{Q})} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{M} =_\tau \mathsf{N})}$$

$$\frac{\Gamma \mid \Phi \vdash \forall x : \tau.\, \mathsf{stable}(\mathsf{P}(x)) \qquad Q \in \{\forall, \exists\}}{\Gamma \mid \Phi \vdash \mathsf{stable}(Qx : \tau.\, \mathsf{P}(x))} \qquad \frac{\Gamma \vdash \mathsf{S} : \mathsf{Spec}}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{spec}(\mathsf{S}))} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{P})}{\Gamma \mid \Phi \vdash \mathsf{stable}(\triangleright \mathsf{P})}$$

C.2.6. Structural Rules.

$$\textsc{VFrame}\ \frac{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{\mathcal{E}} \mathsf{Q} \qquad \Gamma \mid \Phi \mid \mathsf{stable}(\mathsf{R})}{\Gamma \mid \Phi \vdash \mathsf{P} * \mathsf{R} \sqsubseteq^{\mathcal{E}} \mathsf{Q} * \mathsf{R}} \qquad \textsc{AFrame}\ \frac{\Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P} \rangle\, c\, \langle \mathsf{Q} \rangle^{\mathcal{E}} \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{R})}{\Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P} * \triangleright \mathsf{R} \rangle\, c\, \langle \mathsf{Q} * \mathsf{R} \rangle^{\mathcal{E}}}$$

$$\textsc{Frame}\ \frac{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\, c\, \{\mathsf{Q}\} \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{R})}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P} * \mathsf{R}\}\, c\, \{\mathsf{Q} * \mathsf{R}\}}$$

$$\textsc{Conseq}\ \frac{\Gamma, \Delta \mid \Phi \vdash \mathsf{P}_1 \sqsubseteq^{\mathcal{E}} \mathsf{P}_2 \qquad \Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}_2\}\, c\, \{\mathsf{Q}_2\} \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{Q}_2 \sqsubseteq^{\mathcal{E}} \mathsf{Q}_1}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}_1\}\, c\, \{\mathsf{Q}_1\}}$$

$$\textsc{AConseq}\ \frac{\Gamma, \Delta \mid \Phi \vdash \mathsf{P}_1 \sqsubseteq^{\mathcal{E}} \mathsf{P}_2 \qquad \Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P}_2 \rangle\, c\, \langle \mathsf{Q}_2 \rangle^{\mathcal{E}} \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{Q}_2 \sqsubseteq^{\mathcal{E}} \mathsf{Q}_1}{\Gamma \mid \Phi \vdash (\Delta).\langle \mathsf{P}_1 \rangle\, c\, \langle \mathsf{Q}_1 \rangle^{\mathcal{E}}}$$

$$\textsc{Seq}\ \frac{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\mathsf{s}_1\{\mathsf{Q}\} \qquad \Gamma \mid \Phi \vdash (\Delta).\{\mathsf{Q}\}\mathsf{s}_2\{\mathsf{R}\}}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\mathsf{s}_1; \mathsf{s}_2\{\mathsf{R}\}}$$

ACKNOWLEDGMENTS

Thanks to the anonymous referees, Richard Bornat, Matko Botinčan, Thomas Dinsdale-Young, Philippa Gardner, Ralf Jung, Robbert Krebbers, Neel Krishnaswami, Daiva Naudžiūnienė, Viktor Vafeiadis, and John Wickerson.

REFERENCES

C. J. Bell, A. Appel, and D. Walker. 2009. Concurrent separation logic for pipelined parallelization. In *SAS*.

E. D. Berger, T. Yang, T. Liu, and G. Novark. 2010. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*.

L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. 2012. First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. *Logical Methods in Computer Science* 8, 4 (2012).

R. L. Bocchino Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. 2009. A type and effect system for deterministic parallel Java. In *OOPSLA*.

R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. 2005. Permission accounting in separation logic. In *POPL*.

M. Botinčan, M. Dodds, and S. Jagannathan. 2013. Resource-sensitive synchronization inference by abduction. *TOPLAS* 32, 2 (2013).

P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, and M. Wheelhouse. 2011. A simple abstraction for complex concurrent indexes. In *OOPSLA*.

P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. 2014. TaDA: A logic for time and data abstraction. In *ECOOP*.

T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. 2010. Concurrent abstract predicates. In *ECOOP*.

M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis. 2009. Deny-guarantee reasoning. In *ESOP*.

M. Dodds, S. Jagannathan, and M. J. Parkinson. 2011. Modular reasoning for deterministic parallelism. In *POPL*.

X. Feng, R. Ferreira, and Z. Shao. 2007. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*.

A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. 2007. Local reasoning for storable locks and threads. In *APLAS*.

C. Haack, M. Huisman, and C. Hurlin. 2008. Reasoning about Java's reentrant locks. In *APLAS*.

C. A. R. Hoare and P. W. O'Hearn. 2008. Separation logic semantics for communicating processes. *ENTCS* 212 (2008), 3–25.

A. Hobor, A. W. Appel, and F. Z. Nardelli. 2008. Oracle semantics for concurrent separation logic. In *ESOP*.

B. Jacobs and F. Piessens. 2009. *Modular Full Functional Specification and Verification of Lock-Free Data Structures*. Technical Report CW 551. Katholieke Universiteit Leuven, Dept. of Computer Science.

C. B. Jones. 1983. Tentative steps toward a development method for interfering programs. *TOPLAS* 5, 4 (1983).

N. R. Krishnaswami, L. Birkedal, and J. Aldrich. 2010. Verifying event-driven programs using ramified frame properties. In *TLDI*.

K. R. M. Leino, P. Müller, and J. Smans. 2010. Deadlock-free channels and locks. In *ESOP*.

A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. 2014. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*.

A. Navabi, X. Zhang, and S. Jagannathan. 2008. Quasi-static scheduling for safe futures. In *PPoPP*.

P. W. O'Hearn. 2007. Resources, concurrency and local reasoning. *TCS* 375, 1–3 (2007).

M. J. Parkinson and G. M. Bierman. 2005. Separation logic and abstraction. In *POPL*.

M. C. Rinard and M. S. Lam. 1992. Semantic foundations of Jade. In *POPL*.

K. Svendsen and L. Birkedal. 2014a. Impredicative concurrent abstract predicates. In *ESOP*.

K. Svendsen and L. Birkedal. 2014b. *Impredicative Concurrent Abstract Predicates*. Technical Report. Aarhus University. Retrieved from https://bitbucket.org/logsem/public/src/master/icap/esop2014-tr.pdf.

K. Svendsen, L. Birkedal, and M. J. Parkinson. 2013. Joins: A case study in modular specification of a concurrent reentrant higher-order library. In *ECOOP*.

A. Turon, D. Dreyer, and L. Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*.

V. Vafeiadis. 2007. *Modular Fine-Grained Concurrency Verification*. Ph.D. Dissertation. University of Cambridge.

V. Vafeiadis and M. J. Parkinson. 2007. A marriage of rely/guarantee and separation logic. In *CONCUR*.

J. Villard, É. Lozes, and C. Calcagno. 2010. Tracking heaps that hop with heap-hop. In *TACAS*.

A. Welc, S. Jagannathan, and A. Hosking. 2005. Safe futures for Java. In *OOPSLA*.

J. Wickerson, M. Dodds, and M. Parkinson. 2010. Explicit stabilisation for modular rely-guarantee reasoning. In *ESOP*.

Received August 2014; revised August 2015; accepted August 2015