A

Verifying Custom Synchronisation 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, Aarhus University, Denmark

Synchronisation 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 synchronisation constructs with special-purpose behaviour. These constructs present a significant challenge for verification. Like standard constructs, they rely on subtle racy behaviour, 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 paper examines the formal specification and verification of custom synchronisation constructs. Our target is a library of channels used in automated parallelization to enforce sequential behaviour between program statements. Our high-level specification captures the conditions necessary for correct execution; these conditions reflect program dependencies necessary to ensure sequential behaviour. 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

1. INTRODUCTION

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

This work is supported TODO.
Author's addresses: TODO

DOI:http://dx.doi.org/10.1145/0000000.0000000

For standard synchronisation constructs – e.g. locks, queues, stacks – abstract specifications are well-understood. However, many concurrent applications depend on non-standard, custom synchronisation constructs. These may have poorly-defined abstract interfaces, while at the same time depending on complex racy implementation behaviour. 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 paper.

Our target is to verify one such custom concurrency construct: barriers used for automated parallelisation. In *deterministic* parallelisation, 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 behaviour as its sequential counterpart. Automatic parallelisation 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 enforcing the original sequential program dependencies on shared resources. A resource could be any program variable, data structure, memory region, lock, etc. for which 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 behaviour 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 summarise signals.
— *Non-local signalling.* The patterns of signalling in a barrier implementation are highly non-local. 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 signalling.* The parallelisation 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 signalling 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 behaviour. 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 paper, 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 behaviour, meaning that each thread can be permitted exactly the behaviour 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 emphasising 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 paper we focus on just the verification problem for barriers, but in a companion paper, we define a parallelising program analysis which 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 synchronisation constructs, and our work in this paper demonstrates how to reason soundly about such bespoke concurrency constructs.

Contribution

This paper 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 synchronisation barriers used in deterministic parallelism. Our new specification is cleaner and more general.
— New proofs of this specification for simplified, out-of-order and summarising 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 summarising version is entirely novel.
— 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 *explicit stabilization* [Wickerson et al. 2010] to reason about the stability of complex separation logic assertions (see §3.5).

This paper also corrects a subtle logical problem in [Dodds et al. 2011], discovered by Svendsen a year after publication. As is often true in logic, this problem arose as a result of self-reference – in this case, a circularity in the model of higher-order propositions rendered several important reasoning steps unsound. The details are discussed in §8, but we emphasise 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 paper, we show that iCAP can be used to verify tricky practical algorithms.

Paper Structure

§2 discusses related work. §3 introduces the behaviour of barriers informally, and defines our abstract specification. §4 gives a very simple barrier implementation, and shows how it can be verified with respect to the core of the specification. §5 discusses how the specification can be extended to cover the splitting of resources offered by a channel. §6 gives a more complicated implementation where channels are arranged into chains, and verifies the full abstract specification. §7 gives an optimised implementation where signals between channels are summarised, and verifies it. §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 paper 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 synchronisation 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 synchronisation 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 paper 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.

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, e.g. 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 papers 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. We share two co-authors with [Svendsen et al. 2013], and iCAP was largely developed as a improvement on the HOCAP logic it uses.

The most 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 paper 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.

3. A SPECIFICATION FOR DETERMINISTIC PARALLELISM

In this section, we describe the intuitive behaviour of a library of barriers for enforcing deterministic parallelism that forms our case study. Based on this, we define a high-level specifications for barriers – the full abstract specification is given in §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 behaviour.

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 signalled that they will no longer use the resource (i.e., have issued signals).

We assume barriers are injected by an analysis which ensure 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 below. 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 behaviour 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 signalling machinery as part of a parallelization program analysis. Clients would then reason about program behaviour 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 behaviour 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': $\texttt{x} \mapsto v * \texttt{y} \mapsto v'$ 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 $\texttt{x} \mapsto v * \texttt{y} \mapsto v'$ 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:

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

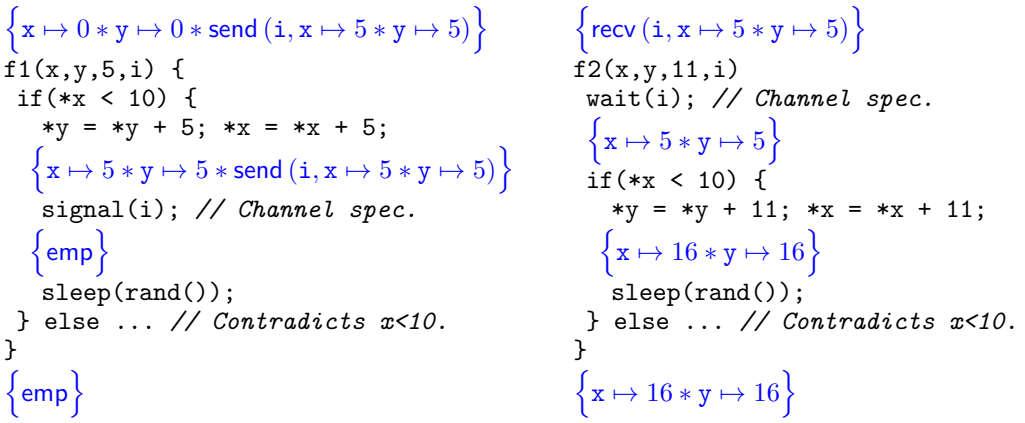


Fig. 1. Proofs for `f1` (left) and `f2` (right).

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:

$$\begin{array}{rcl} \{\mathsf{emp}\} & \texttt{i = newchan()} & \{\mathsf{send}(\texttt{i}, P) * \mathsf{recv}(\texttt{i}, P)\} \\ \{\mathsf{send}(\texttt{i}, P) * \lfloor P \rfloor\} & \texttt{signal(i)} & \{\mathsf{emp}\} \\ \{\mathsf{recv}(\texttt{i}, P)\} & \texttt{wait(i)} & \{\lceil P \rceil\} \end{array}$$

The specification of `newchan` is implicitly universally quantified for all assertions P, meaning that we can construct a channel for any assertion.[1] The assigned variable `i` stands for `newchan`'s return value – i.e. the address of the new channel (we also use this notation in the specification of `extend`, below).

New recv and send predicates can be constructed at run-time 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 specialise to the particular parameter values of 5 / 11; it would be easy to generalise).

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

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. Fig. 1 gives sketch-proofs for these two specifications.

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

[1] In iCAP, assertions can be shared between multiple threads. In this case we need to establish that each assertion is *stable*, i.e. invariant under changes performed by other threads. The *explicit stabilisation* operators $\lfloor - \rfloor$ and $\lceil - \rceil$ in the specification are needed because P might contain assertions about shared state. These are discussed further at the end of the section, in §3.5. For the moment, note that if P is a thread-local assertion such as $x \mapsto v$, then $\lfloor P \rfloor \iff P \iff \lceil P \rceil$.

$\{x \mapsto \_ * y \mapsto \_\}$

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

$\{x \mapsto 0 * y \mapsto 0 * \mathsf{send}(\mathtt{i}, x \mapsto 5 * y \mapsto 5) * \mathsf{recv}(\mathtt{i}, x \mapsto 5 * y \mapsto 5)\}$

$\{x \mapsto 0 * y \mapsto 0 * \mathsf{send}(\mathtt{i}, x \mapsto 5 * y \mapsto 5)\}$ `f1(x,y,5,i)` $\{\mathsf{emp}\}$ $\parallel$ $\{\mathsf{recv}(\mathtt{i}, x \mapsto 5 * y \mapsto 5)\}$ `f2(x,y,11,i)` $\{x \mapsto 16 * y \mapsto 16\}$ — PAR rule application.

$\{x \mapsto 16 * y \mapsto 16\}$

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 use a different aspect of the promised resource.

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

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

This axiom states that when a thread has been promised a resource P, access can be split between two threads, potentially before the resource is available. The specification uses separation logic's separating implication operator $\mathrel{-\!\!*}$ (separating implication and separating conjuction have a similar relationship shared by classical implication, $\Rightarrow$, and classical conjunction, $\wedge$, i.e. they are adjoint). The assertion $P \mathrel{-\!\!*} (P_1 * P_2)$ asserts ownership of a resource that when combined with P can be split into P_1 and P_2. As the resource P is promised by the recv predicate, the result of applying this property is two new recv predicates, one of which promises P_1, and the other P_2.

Note that this 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 §4.1. To use this property, we have to assume that every application in a proof is associated with a `skip`. We discuss whether this assumption is justified in §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 above, a thread can only call `signal` if it has acquired the required resource from its predecessors. However, this is often unnecessary. For example, if we take the second branch of the conditional in `f`, we do not need the heap location `y`. It is safe to notify future threads that `y` is available, conditional on all logically prior threads releasing it, even though the thread itself never acquired access to the resource. Without renunciation the thread would have to `wait` for all the logically earlier threads to `signal` before it could proceed.

Chains. To support chains, 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:

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

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. It also creates recv, send, and order predicates representing the new channel.

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

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

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

Using this specification, we can partially discharge a send using the preceding recv predicate. The assertion $P \mathrel{-\!\!*} Q$ stands for the resource which gives Q if combined with a resource satisfying P. Thus, combining it with the predicate $\mathsf{recv}(x, P)$ provides Q. In other words, the thread gives up the ability to ever acquire the resource, and instead forwards this ability to logically later threads. When the resource becomes available from prior threads, the *next* thread in the order will receive it (unless it, too, renounces the resource).

3.4. Full abstract specification

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

Example parallelisation. Suppose we want to run many copies of the function `f` in sequence, for example over an array of values `vs`.

```
for(j = 0; j < max; j++){
  f(x,y,vs[j]);
}
```

To parallelize this program, we define `fp` (left of Fig. 3) a version of `f` which is safe to run in parallel with many copies of itself. To do this, `f` is modified to call both `signal` and `wait`. Each call to `f` receives the resource from logically earlier threads (those invoked in earlier loop iterations) with `wait`, then releases it to logically later threads (those invoked in later loop iterations) using `signal`.

In the original transformation involving `f1` and `f2`, we did not distinguish between the resources `x` and `y`. However, we need to gain access to `y` only if we take the first branch

SPECIFICATIONS:

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

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

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

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

AXIOMS:

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

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

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

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

Fig. 2. Full abstract specification for deterministic parallelism.

FUNCTION DEFINITION:

```
fp(x,y,v,ix,iy,ixp,iyp) {
  wait(ixp);
  if (*x < 10) {
    wait(iyp);
    *y = *y + v;
    signal(iy);
    *x = *x + v;
    signal(ix);
  } else {
    signal(ix);
    signal(iy);
    sleep(rand());
  }
}
```

PROGRAM BODY:

```
ixf = newchan();
(ixn,ixf) = extend(ixf);
signal(ixn);
iyf = newchan();
(iyn,iyf) = extend(iyf);
signal(iyn);
for(j = 0; j < max; j++){
  v = vs[j];
  ixl = ixn;
  (ixn,ixf) = extend(ixf);
  iyl = iyn;
  (iyn,iyf) = extend(iyf);
  fork( fp(x,y,v,ixn,iyn,ixl,iyl) );
}
wait(ixn);
wait(iyn);
```

Fig. 3. Example parallelization of `f` and a client.

```
1  {recv(ixp, x ↦ _) ∗ send(ix, x ↦ _) ∗ recv(iyp, y ↦ _) ∗ send(iy, y ↦ _) ∗ iyp ≺ iy}
2  fp(x,y,v,ix,iy,ixp,iyp) {
3    wait(ixp);
4    {x ↦ _ ∗ send(ix, x ↦ _) ∗ recv(iyp, y ↦ _) ∗ send(iy, y ↦ _) ∗ iyp ≺ iy}
5    if (*x < 10) {
6      wait(iyp);
7      {x ↦ _ ∗ y ↦ _ ∗ send(ix, x ↦ _) ∗ send(iy, y ↦ _)}
8      *y = *y + v;
9      signal(iy);
10     {x ↦ _ ∗ send(ix, x ↦ _)}
11     *x = *x + v;
12     signal(ix);
13   } else {
14     signal(ix);
15     {recv(iyp, y ↦ _) ∗ send(iy, y ↦ _) ∗ iyp ≺ iy}
16     signal(iy);
17     {emp}
18     sleep(rand());
19   }
20 }
21 {emp}
```

Fig. 4. Proof for parallelized program `fp`.

of the conditional. Otherwise we can release `y` to logically future threads. To realise this parallelism in the new version of `f`, we use two chains of channels: one for `x`, and one for `y`. The function takes arguments `ix` and `iy` representing the next points in the two chains, and `ixp` and `iyp` representing the previous points.

In Fig. 4, we verify `fp` against the following specification:

$$\left\{\begin{array}{c}\mathsf{recv}(\mathtt{ixp}, \mathtt{x} \mapsto \_) * \mathsf{send}(\mathtt{ix}, \mathtt{x} \mapsto \_) * \\ \mathsf{recv}(\mathtt{iyp}, \mathtt{y} \mapsto \_) * \mathsf{send}(\mathtt{iy}, \mathtt{y} \mapsto \_) * \mathtt{iyp} \prec \mathtt{iy}\end{array}\right\} \quad \mathtt{fp(...)} \quad \{\mathsf{emp}\}$$

Note that we only prove basic memory safety, but we could easily verify stronger invariants.

Line 16 in Fig. 4 is noteworthy. There, the precondition *does not* assert that the thread has access to $\mathtt{y} \mapsto \_$; rather, it asserts it can acquire access by calling `wait`. Instead of doing this, the thread renounces access to the resource without ever holding it.

The parallelized version of the client program is given on the right of Fig. 3; a proof of correctness is given in Fig. 5. The predicates $\mathsf{send}(\mathtt{ixf}, \mathsf{true})$ and $\mathsf{send}(\mathtt{iyf}, \mathsf{true})$ represent the logically last element of the chain, while $\mathsf{array}()$ stands for the array holding processed values.

Thus, using our abstract specification, we have shown that our parallelized version of the program is memory-safe. With a little more effort, we could verify the behaviour of the program. 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 the implementation of the barriers. While the example

```
1  {x ↦ _ * y ↦ _ * array(vs, max)}
2  ixf = newchan();
3  (ixn,ixf) = extend(ixf);
4  signal(ixn);
5  {y ↦ _ * array(vs, max) * recv(ixn, x ↦ _) * send(ixf, true) * ixn ≺ ixf}
6  iyf = newchan();
7  (iyn,iyf) = extend(iyf);
8  signal(iyn);
9  {array(vs, max) * recv(ixn, x ↦ _) * send(ixf, true) * ixn ≺ ixf
                   * recv(iyn, y ↦ _) * send(iyf, true) * iyn ≺ iyf}
10 for(j = 0; j < max; j++){
11   ixl = ixn;
12   (ixn,ixf) = extend(ixf);
     {array(vs, max) * recv(ixl, x ↦ _) * send(ixn, x ↦ _)
13   * recv(ixn, x ↦ _) * send(ixf, true) * ixn ≺ ixf
     * recv(iyn, y ↦ _) * send(iyf, true) * iyn ≺ iyf}
14   iyl = iyn;
15   (iyn,iyf) = extend(iyf);
     {array(vs, max) * recv(ixl, x ↦ _) * send(ixn, x ↦ _)
16   * recv(ixn, x ↦ _) * send(ixf, true) * ixn ≺ ixf
     * recv(iyl, y ↦ _) * send(iyn, y ↦ _) * iyl ≺ iyn
     * recv(iyn, y ↦ _) * send(iyf, true) * iyn ≺ iyf}
17   fork( fp(x,y,vs[j],ixn,iyn,ixl,iyl) ); // apply function spec.
18 }
19 wait(ixn);
20 wait(iyn);
21 {array(vs, max) * x ↦ _ * y ↦ _}
```

Fig. 5. Proof of `fp` client program.

given here is trivial, in [Botinčan et al. 2013] we use the same mechanism as the basis for a general parallelisation analysis.

3.5. Explicit Stabilisation

Our abstract specification includes the assertion $\lfloor P \rfloor$ in the precondition to `signal()`, and $\lceil P \rceil$ in the postcondition to `wait()`, as well as similar assertions in the axiom for splitting. $\lfloor - \rfloor$ and $\lceil - \rceil$ are *explicit stabilisation* operators [Wickerson et al. 2010].

In iCAP assertions can refer to resources shared between threads. Any such assertion in the proof of one thread could potentially be invalidated by interference from other threads. Therefore, any assertion in iCAP must be *stable*, meaning invariant under the worst-case interference from other threads (we inherit this terminology from Jones's rely-guarantee logic). In our abstract specification, higher-order arguments such as P could include assertions about shared state. Thus, P and similar assertions must be provably stable.

In our conference paper [Dodds et al. 2011] we used a predicate $\mathsf{stable}(P)$ to explicitly assert stability of P the specification. However, this predicate is hard to manipulate in proofs

```
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. 6. Implementation of the barrier library.

because it satisfies few distribution properties. This is especially problematic for assertions containing separating implication, $-\!\!*$. In this paper, we instead use explicit stabilization operators, which strengthen or weaken an assertion until it is stable:

— $\lfloor P \rfloor$ stands for the weakest assertion stronger than P that is stable.
— $\lceil P \rceil$ stands for the strongest assertion weaker than P that is stable.

Two properties of explicit stabilization operators make them easy to remove if not needed:

— If P is already stable, then $\lfloor P \rfloor \iff P \iff \lceil P \rceil$. This holds by default if P contains no shared assertions nor predicate variables.
— $\lfloor P \rfloor \implies P \implies \lceil P \rceil$.

Furthermore, explicit stabilisation operators are semi-distributive over separating conjunction. As a result, they are easy to move around in proofs.

— $\lfloor P \rfloor * \lfloor Q \rfloor \implies \lfloor P * Q \rfloor$.
— $\lceil P * Q \rceil \implies \lceil P \rceil * \lceil Q \rceil$.

The predicate stable is much harder to use in proofs, because the only one of these properties that holds is the first one, elimination if the assertion contains no shared assertions or predicate variables. The properties are especially useful when combined with separating implication. We use $-\!\!*$ extensively, but are aware of no other approach that would allow reasoning about the stability of complex assertions that include separating implication. The following Lemma provides the common pattern of our usage:

LEMMA 3.1. *If* P stable *and* $P * Q \Rightarrow R$, *then* $P \Rightarrow \lfloor Q -\!\!* R \rfloor$.

4. PROOF STRATEGY FOR THE SIMPLIFIED SPECIFICATION

This section provides an intuitive introduction to our general proof approach, and to iCAP, the reasoning system our proofs are based on. To motivate this, we define a very simple channel implementation and verify it with respect to just the first three axioms of our abstract specification:

$$\begin{array}{rcl} \{\mathsf{emp}\} & \texttt{i = newchan()} & \{\mathsf{recv}(\texttt{i}, P) * \mathsf{send}(\texttt{i}, P)\} \\ \{\mathsf{send}(\texttt{i}, P) * \lfloor P \rfloor\} & \texttt{signal(i)} & \{\mathsf{emp}\} \\ \{\mathsf{recv}(\texttt{i}, P)\} & \texttt{wait(i)} & \{\lceil P \rceil\} \end{array}$$

By avoiding splitting, chain extension, and renunciation, we can illustrate the basic features of iCAP in a straightforward manner. In §5 and §6 we reintroduce the necessary extra reasoning to verify our full abstract specification.

Figure 6 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` simple sets the flag, while the `wait` loops until the flag is set.

4.1. Introduction to iCAP

Before we can present our verification, we need to sketch the key details of the logic we use: Impredicative Concurrent Abstract Predicates (iCAP). iCAP is a separation logic variant intended for verifying concurrent higher-order programs. Full details are given in [Svendsen and Birkedal 2014a].

To handle concurrency, iCAP extends separation logic with *regions* containing resources shared between threads. Resources stored in a region must be modified atomically in a way that satisfies the expectations of the other threads. 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 which 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 the shared region owns in the 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 labelled 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 labelled transition system T labelled with actions owned by the given thread. Ownership of actions is tracked using tokens. For instance, the following assertion asserts ownership of a set and change token: $[\mathsf{set}]^{r_1}_i * [\mathsf{change}]^{r_2}_j$. 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.

Tokens are linear objects created at the same time as a region – by issuing threads with different tokens, we grant them different abilities over the shared region. 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. Tokens can be split arbitrarily: $[\alpha]^r_{i+j} \Leftrightarrow [\alpha]^r_i * [\alpha]^r_j$. In contrast, region assertions can be duplicated arbitrarily:

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

Thus, each region assertion should be interpreted as one view on a resource shared between many threads. Such a region assertion, $\mathsf{region}(R, T, I, r)$ is *stable* (i.e., closed under possible inference from the environment) if the set R is closed under all transitions in T labelled with actions potentially owned by the environment.

As well as information about the underlying memory cells, the abstract state of region captures auxiliary state representing the way threads can interact. To manipulate this auxiliary state, iCAP uses the *view-shift* operator $\sqsubseteq$. View-shift generalises standard implication to allow creation, destruction and manipulation of regions.

As a convenience, we assume that we can garbage collect unwanted portions of assertions. As we mostly apply this to logical constructs, this assumption loses little generality, and in any case could easily be lifted at the cost of larger proofs.

Higher-order assertions and 'later'. The features discussed above could equally be expressed in prior logics such as RGSep [Vafeiadis and Parkinson 2007]. The distinction with iCAP is that it is based on a higher-order separation logic and supports shared higher-order resources – i.e., shared regions containing shared resources.

It is well-known that reasoning about shared higher-order resources is difficult (for example, see the problems with our previous paper discussed in §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. To capture this stratification in the logic, iCAP introduces a 'later' modality, written $\triangleright$. Intuitively, $\triangleright P$ expresses that the assertion P holds after one step of execution. To ensure that protocols defined in iCAP are well-defined they are implicitly interpreted one step later. The region assertion $\mathsf{region}(\{x\}, T, I, r)$ thus expresses that the shared region r currently owns the resources described by $\triangleright I(x)$.

If an assertion P holds now, then it also holds after one step of execution. This is expressed by the (SMono) rule given below. In general, if $\triangleright P$ holds now, it is not the case that P also holds now. Instead, $\triangleright$ operators can be eliminated by taking an execution step, as expressed by the frame rule for atomic commands (AFrame). Using (SMono) and the rule of consequence, one can derive the standard CAP frame rule.

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

$$\mathsf{atomic}(C) \wedge \mathsf{stable}(R) \wedge \{P\}\ C\ \{Q\} \implies \{P * \triangleright R\}\ C\ \{Q * R\} \qquad \text{(AFrame)}$$

The effect of this 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] 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.

Properties of 'later'. The later operator commutes over conjunction, disjunction and separating conjunction and semi-commutes over implication and separating implication:

$$\triangleright(P \wedge Q) \iff \triangleright P \wedge \triangleright Q, \quad \triangleright(P \vee Q) \iff \triangleright P \vee \triangleright Q, \quad \triangleright(P * Q) \iff \triangleright P * \triangleright Q \qquad \text{(LBin)}$$

$$\triangleright(P \Rightarrow Q) \implies \triangleright P \Rightarrow \triangleright Q, \qquad \triangleright(P \mathrel{-\!\!*} Q) \implies \triangleright P \mathrel{-\!\!*} \triangleright Q \qquad \text{(LImpl/LWand)}$$

Later also commutes over existential and universal quantification over non-empty types τ:

$$\triangleright \exists x : \tau.\ P(x) \iff \exists x : \tau.\ \triangleright P(x), \qquad \triangleright \forall x : \tau.\ P(x) \iff \forall x : \tau.\ \triangleright P(x) \qquad \text{(LQuant)}$$

In proof outlines we often apply these properties silently. Lastly, later can be moved freely in and out of stabilization brackets.

$$\triangleright \lceil P \rceil \iff \lceil \triangleright P \rceil, \qquad \triangleright \lfloor P \rfloor \iff \lfloor \triangleright P \rfloor \qquad \text{(LCeil/LFloor)}$$

Reasoning about shared regions. All statements that access resources owned by shared regions must be atomic and obey the protocols of the regions in question. To illustrate how this is expressed formally, we consider a simplified proof rule for reasoning about shared regions. We refer to the iCAP technical report [Svendsen and Birkedal 2014b] for the general proof rules.

We reason about shared regions using rules that allow us to treat the shared resources as local reasources for the duration of an atomic statement. We refer to these rules as "region opening" rules and as entering and exiting a shared region in proof outlines. To illustrate, consider the verification of an atomic statement C that changes the abstract state of a

shared region r from x to y:

$$\{\mathsf{region}(\{x\}, T, I, r) * [\alpha]^r_\pi * P\}\ C\ \{\mathsf{region}(\{y\}, T, I, r) * Q\}$$

To prove that this triple holds, we must first prove that we own sufficient permissions to change the abstract state from x to y. In this case we own non-exclusive permission to the α action. We thus have to prove that there is an α-labelled path from x to y in the transition system T. Furthermore, we have to prove that C does indeed transform the resources associated with abstract state x to those associated with abstract state y, according to I. Since protocols are implicitly interpreted one step later, we have to prove the following triple:

$$\{\triangleright I(x) * [\alpha]^r_\pi * P\}\ C\ \{\triangleright I(y) * Q\}$$

Here P and Q can be used to transfer local resources in and out of the shared region as appropriate.

Imagine that $I = \lambda v.\ l \mapsto v$ and C is the atomic statement $l := y$. Then we would be left with the following proof obligation (after instantiating P and Q with emp).

$$\{\triangleright(l \mapsto x) * [\alpha]^r_\pi\}\ l := y\ \{\triangleright l \mapsto y\}$$

Conceptually, this is provable because primitive points-to assertions, such as $l \mapsto x$, are independent of the step-indexing. This is capture by the structural (LPoints) rule given below, which allows us to remove a later from the pre-condition of a points-to assertion.

$$\mathsf{atomic}(C) \wedge \{x \mapsto y\}\ C\ \{Q\} \implies \{\triangleright x \mapsto y\}\ C\ \{Q\} \qquad \text{(LPoints)}$$

4.2. Verifying the Simple Implementation

Verifying the implementation shown in Figure 6 with respect to the abstract specification amounts to giving concrete definitions to the abstract predicates send and recv, and then using iCAP to prove that the resulting concrete specification is satisfied.

We begin by defining the structure of the shared region. There are three possible abstract states: either the flag is low, or the flag is high and the sent resource is available, or the flag is high and the resource has been taken (we call this 'done'). We define three corresponding states: $\{\mathsf{Low}, \mathsf{High}, \mathsf{Done}\}$. Each abstract state is associated with an invariant by I_b, defined as follows:

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

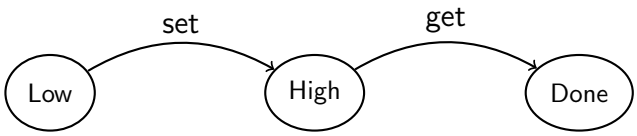
Here x is the location of the channel, while P is the resource controlled by the channel. Note that moving from Low to High requires the resource $\lceil P \rceil$ and setting the flag, while moving from High to Done removes $\lceil P \rceil$.

To ensure that these regions are invariant in concurrent contexts, we need to show that the interpretation of each abstract state, High, Low and Done, is stable. For these definitions, this is simple. Stability distributes over separating conjunction, so we can consider each conjunct separately. Points-to assertions are automatically stable. Furthermore, the higher-order parameter P is wrapped in explicit stabilization operators $\lceil - \rceil$, ensuring that it is also stable.

The transition relation T_b has the following form. Each transition corresponds to an operation that can be performed on the channel.

```
1  {send(x, P) ∗ ⌊P⌋}
2  signal(chan *x) {
3    // definition of send.
4    {region({Low}, T_b, I_b(x, P), r) ∗ [set]_1^r ∗ ⌊P⌋}
5    // enter the region.
6    {▷(x.flag ↦ 0) ∗ [set]_1^r ∗ ⌊P⌋}
7    // drop ▷ using LPoints
8    {x.flag ↦ 0 ∗ [set]_1^r ∗ ⌊P⌋}
9    x->flag := 1;
10   {x.flag ↦ 1 ∗ [set]_1^r ∗ ⌊P⌋}
11   // flip explicit stabilization with ⌊P⌋ ⟹ ⌈P⌉.
12   {x.flag ↦ 1 ∗ [set]_1^r ∗ ⌈P⌉}
13   // close region and delete set token.
14   {region({High}, T_b, I_b(x, P), r)}
15   // stabilise the region's abstract state.
16   {region({High, Done}, T_b, I_b(x, P), r)}
17 }
18 {emp}
```

Fig. 7. Proof of `signal()` using simple predicate definitions.



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 shared region is Low – otherwise we could not send the resource. On the other hand, recv asserts it is either Low or High, but not Done. $[\mathsf{set}]_1^r$ and $[\mathsf{get}]_1^r$ are tokens allowing the thread to take particular transitions in T_b. The send predicate allows the thread to set the flag and supply the resource, while recv allows the thread to retrieve the supplied resource.

Figures 7 and 8 show sketch proofs for `signal()` and `wait()` respectively. Both proofs involve accessing resources stored in the shared region – for example, `signal` enters the region between lines 5–13 of Fig. 7. This whole section corresponds to one atomic action. On entering the region at line 6, note the region invariant is wrapped in a ▷ operator, but this can be dropped immediately using the LPoints rule.

When closing the region (Fig. 7, line 13), the resulting invariant must satisfy one of the abstract states. By examining the definition of I_b above, observe that High is satisfied.

PROOF BODY:

1 $\left\{\mathsf{recv}(\mathtt{x}, P)\right\}$
2 `wait(chan *x) {`
3 `  int b;`
4 `  do {`
5 $\left\{\mathsf{region}(\{\mathsf{Low}, \mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * [\mathsf{get}]_1^r\right\}$
6 `  // case-split on` Low `/` High`.`
7 $\left\{(\mathsf{region}(\{\mathsf{Low}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) \lor \mathsf{region}(\{\mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r)) * [\mathsf{get}]_1^r\right\}$
8 `  b = x->flag; // Low &` High `cases given below.`
9 $\left\{\begin{array}{l}(\mathsf{region}(\{\mathsf{Low}, \mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * [\mathsf{get}]_1^r \land \mathtt{b} = 0) \ \lor \\ (\mathsf{region}(\{\mathsf{Done}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * [\mathsf{get}]_1^r * \lceil P \rceil \land \mathtt{b} = 1)\end{array}\right\}$
10 `  } while (b == 0)`
11 `} // Garbage collect the shared region assertions.`
12 $\left\{\lceil P \rceil\right\}$

LOW CASE:

1 $\left\{\mathsf{region}(\{\mathsf{Low}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * [\mathsf{get}]_1^r\right\}$
2 `// open region`
3 $\left\{\triangleright(\mathtt{x}.\mathsf{flag} \mapsto 0) * [\mathsf{get}]_1^r\right\}$
4 `// drop ▷ and read value.`
5 `b = x->flag;`
6 $\left\{\mathtt{x}.\mathsf{flag} \mapsto 0 * [\mathsf{get}]_1^r \land \mathtt{b} = 0\right\}$
7 `// close region, and stabilise`
8 $\left\{\begin{array}{c}\mathsf{region}(\{\mathsf{Low}, \mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) \\ * [\mathsf{get}]_1^r \land \mathtt{b} = 0\end{array}\right\}$

HIGH CASE:

1 $\left\{\mathsf{region}(\{\mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) * [\mathsf{get}]_1^r\right\}$
2 `// open region`
3 $\left\{\triangleright(\mathtt{x}.\mathsf{flag} \mapsto 1 * \lceil P \rceil) * [\mathsf{get}]_1^r\right\}$
4 `// drop ▷ and read value.`
5 `b = x->flag;`
6 $\left\{\mathtt{x}.\mathsf{flag} \mapsto 1 * \lceil P \rceil * [\mathsf{get}]_1^r \land \mathtt{b} = 1\right\}$
7 `// close region.`
8 $\left\{\begin{array}{c}\mathsf{region}(\{\mathsf{Done}\}, T_\mathsf{b}, I_\mathsf{b}(\mathtt{x}, P), r) \\ * \lceil P \rceil * [\mathsf{get}]_1^r \land \mathtt{b} = 1\end{array}\right\}$

Fig. 8. Proof of `wait()` using simple predicate definitions.

Furthermore, because the thread holds the set token, according to the transition relation T_b it is allowed to move from the Low (line 4) to the High (line 14) abstract state.

The final step of the proof is to ensure that all assertions are stable, i.e. invariant under interference from other threads. By itself, the assertion $\mathsf{region}(\{\mathsf{High}\}, T_\mathsf{b}, I_\mathsf{b}(x, P), r)$ is unstable, because according to T_b, some thread could move the region from High to Done. On line 15 we weaken the set of abstract states to add Done. This assertion is stable.

The proof of `wait` (Fig. 8) uses similar reasoning. 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 bottom left and right of Fig. 8. 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 $\lceil P \rceil$ and close the region in the Done state.

5. SPLITTING CHANNELS

In this section, we extend our proof to cover *splitting*, expressed by the following axiom:

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

Verifying this axiom requires us to introduce some extra logical machinery. (As the simple implementation sequentialises signalling, we leave extension and renunciation to §6.)

5.1. Saved Propositions

The splitting axiom requires the proof to manipulate offered resource: the resource parameter to recv has to be be split into two separate offered resources. For example, in the axiom, the offered resource P is split into resources P_1 and P_2. As the splitting axiom is exposed to the client, P, P_1 and P_2 cannot be fixed in the proof; they are chosen by the client.

In iCAP, modifications to a shared region must be represented in the transition system for the region. An obvious (but wrong) way to represent splitting is to parameterise transition system states with sets of propositions representing the splitting. Then we would have transition system states $\mathsf{Low}(\mathcal{I})$ and $\mathsf{High}(\mathcal{I})$, with $\mathcal{I} \in \mathcal{P}_{fin}(\mathsf{Prop})$ representing the splittings. In the case of the axiom, we would then have the transition:

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

However, iCAP does not support states and actions indexed by step-indexed sets, such as Prop (the type of assertions). Our solution is to introduce a new logical construct we call a *saved proposition*. One can see saved propositions as capturing some the features of higher-order propositions stored in the heap. However, unlike true HO stored propositions, we can encode and verify saved propositions as predicates in iCAP, avoiding the need to develop a new logic. In our proofs, saved propositions have all the properties we need, and we are uncertain whether there would be any benefit in extending iCAP with true HO stored propositions. Our encoding of saved propositions is given in Appendix A.

A saved proposition, written $r \overset{\pi}{\Longmapsto} P$, associates an identifier r with a proposition P. As r comes from a non step-indexed set (in fact RID, the set of region names) we can parameterise transition system states by them. We thus represent each splitting by a set of identifiers combined with a collection of saved propositions. In the case of the axiom, we would assume saved propositions $r \overset{\pi}{\Longmapsto} P * r_1 \overset{\pi_1}{\Longmapsto} P_1 * r_2 \overset{\pi_2}{\Longmapsto} P_2$, and give the transition as:

$$\begin{aligned} \mathsf{Low}(\mathcal{I} \uplus \{r\}) &\rightsquigarrow \mathsf{Low}(\mathcal{I} \uplus \{r_1, r_2\}) \\ \mathsf{High}(\mathcal{I} \uplus \{r\}) &\rightsquigarrow \mathsf{High}(\mathcal{I} \uplus \{r_1, r_2\}) \end{aligned}$$

The difference is subtle but important. 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 four properties:

$$\mathsf{emp} \implies \exists r.\ r \overset{1}{\Longrightarrow\!\!\!|}\ P \tag{1}$$

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

$$r \overset{\pi_1}{\Longrightarrow\!\!\!|}\ P * r \overset{\pi_2}{\Longrightarrow\!\!\!|}\ Q \implies r \overset{\pi_1}{\Longrightarrow\!\!\!|}\ P * r \overset{\pi_2}{\Longrightarrow\!\!\!|}\ Q * (\triangleright P \Rightarrow \triangleright Q) \tag{3}$$

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

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 on the same region allow us to convert from one to another. This is a fixed version of the unification property discussed above. The iCAP later operator $\triangleright$ is needed because we use shared regions internally in the definition of saved propositions. *Property* (4) says that we can apply property (3) inside separating implications. This is useful in the proof when modifying a resource embedded into a larger assertion.

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 region 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, i.e. the resources that other threads expect to be supplied. As described above, 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 thread which expects to receive a resource. To find out *what* resource P is expected, we examine the associated saved proposition $i \overset{\pi}{\Longrightarrow\!\!\!|}\ P$.

Actions in the transition relation T_{m} are of the form send and $\mathsf{change}(r)$, where r is a region identifier. The first action, send, sets the flag, and simply moves from Low to High. The second, $\mathsf{change}(r)$, is both the action of taking the resource associated to region r when the flag is high, and splitting the resource from region r to the resource required by r_1 and r_2.

$$T_{\mathsf{m}}(\mathsf{send}) \triangleq \{(\mathsf{Low}(\mathcal{I}), \mathsf{High}(\mathcal{I}))\}$$

$$\begin{aligned} T_{\mathsf{m}}(\mathsf{change}(r)) \triangleq\ & \{(\mathsf{High}(\mathcal{I} \uplus \{r\}), \mathsf{High}(\mathcal{I}))\} \cup \{(\mathsf{Low}(\mathcal{I} \uplus \{r\}), \mathsf{Low}(\mathcal{I} \uplus \{r_1, r_2\}))\} \\ & \cup \{(\mathsf{High}(\mathcal{I} \uplus \{r\}), \mathsf{High}(\mathcal{I} \uplus \{r_1, r_2\}))\} \end{aligned}$$

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

$$I_{\mathsf{m}}(x,r,P)(\mathsf{Low}(\mathcal{I})) \triangleq x.\mathrm{flag} \mapsto 0 * \mathsf{waiting}(P,\mathcal{I}) * \mathsf{changes}_r(\mathcal{I})$$

$$I_{\mathsf{m}}(x,r,P)(\mathsf{High}(\mathcal{I})) \triangleq x.\mathrm{flag} \mapsto 1 * \mathsf{ress}(\mathcal{I}) * \mathsf{changes}_r(\mathcal{I})$$

where

$$\mathsf{waiting}(P,\mathcal{I}) \triangleq \exists Q\colon \mathcal{I} \to \mathsf{Prop}.\ \lfloor (\triangleright P) \mathrel{-\!\!*} \triangleright \circledast_{i\in\mathcal{I}}. Q(i)\rfloor * \circledast_{i\in\mathcal{I}}. i \overset{1/2}{\Longmapsto} Q(i)$$

$$\mathsf{ress}(\mathcal{I}) \triangleq \circledast_{i\in\mathcal{I}} \exists R.\, i \overset{1/2}{\Longmapsto} R * \lceil \triangleright R \rceil$$

$$\mathsf{changes}_r(\mathcal{I}) \triangleq \circledast_{r'\notin\mathcal{I}}. [\mathsf{change}(r')]^r_1$$

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 permissions on unused regions – this allows new shared propositions to be added when splitting.

The representation of Low consists of the flag, change tokens, and the $\mathsf{waiting}$ 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 shared 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 $Q(i)$.

The representation of the High state consists of the flag and change tokens, and the ress predicate. $\mathsf{ress}(\mathcal{I})$ pairs together fractional saved propositions, $i \overset{1/2}{\Longmapsto} R$ 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 below). 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 $\mathsf{recv}(x,Q)$ predicate is more complex. It includes $r' \overset{1/2}{\Longmapsto} Q$, half the permission on the saved proposition Q. It also asserts that r' is one of identifiers recorded in the region. This ensures that the resource retrieved from the shared region is the correct one, i.e. the one that was promised (see the next section for the reasoning steps involved).

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

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

Proving `newchan`. The proof of `newchan` (Fig. 9) works by allocating an abstract region containing the concrete state for the channel. Most steps in the proof are are straightforward. The challenging ones are line 6 – creating the stabilised assertion – and line 10 – the view-shift which creates the region itself. In line 6, we need the following implication:

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

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

1 $\{\mathsf{emp}\}$
2 `chan *newchan() {`
3 `  chan *x := new(chan);`
4 `  x->flag := 0;`
5 $\{\mathtt{x}.\mathrm{flag} \mapsto 0\}$
6 `  // Create floor assertion`
7 $\{\mathtt{x}.\mathrm{flag} \mapsto 0 * \lfloor(\triangleright P) \mathrel{-\!\!*} (\triangleright P)\rfloor\}$
8 `  skip; // Create saved proposition`
9 $\{\exists r_1.\, r_1 \overset{1}{\Longmapsto} P * \mathtt{x}.\mathrm{flag} \mapsto 0 * \lfloor(\triangleright P) \mathrel{-\!\!*} (\triangleright P)\rfloor\}$
10 `  // Create channel region`
11 $\{\exists r_1, r_2.\, r_1 \overset{1/2}{\Longmapsto} P * \mathsf{creg}(\mathtt{x}, r_2, P, \{\mathsf{Low}(\{r_1\})\}) * [\mathsf{send}]_1^{r_2} * [\mathsf{change}(r_1)]_1^{r_2}\}$
12 `  // Satisfy the` send `predicate`
13 $\{\exists r_1, r_2.\, \mathsf{send}(\mathtt{x}, P) * r_1 \overset{1/2}{\Longmapsto} P * \mathsf{creg}(\mathtt{x}, r_2, P, \{\mathsf{Low}(\{r_1\})\}) * [\mathsf{change}(r_1)]_1^{r_2}\}$
14 `} // Satisfy the` recv `predicate`
15 $\{\mathsf{send}(\mathtt{x}, P) * \mathsf{recv}(\mathtt{x}, P)\}$

Fig. 9. Proof of `newchan()` w.r.t. the full specification.

Line 10 requires us to prove the following view-shift:

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

To prove this, we appeal to iCAP's VAlloc rule, which controls construction of new regions.

Proving `signal`. The proof of `signal` (Fig. 10) 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 $\mathsf{High}(\mathcal{I})$ to 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.

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

To prove the second step, we appeal to the fact that $\lfloor - \rfloor$ is semi-distributitive over separating conjunction, $\lfloor A \rfloor * \lfloor B \rfloor \implies \lfloor A * B \rfloor$, and *modus ponens* for separating implication, $A * (A \mathrel{-\!\!*} B) \implies B$. The third step follows from the fact that $\lceil - \rceil$ is weaker than $\lfloor - \rfloor$, and $\lceil - \rceil$ is semi-distributitive over separating conjunction, $\lceil A * B \rceil \implies \lceil A \rceil * \lceil B \rceil$. □

1 $\{\mathsf{send}(\mathtt{x}, P) * \lfloor P \rfloor\}$
2 `signal(chan *x) {`
3 $\{\mathsf{creg}(\mathtt{x}, r, P, \{\mathsf{Low}(\mathcal{I}') \mid \mathsf{true}\}) * [\mathsf{send}]^r_1 * \lfloor P \rfloor\}$
4 *// open the region.* $\mathsf{Low}(\mathcal{I})$ *is an arbitrary member of* $\{\mathsf{Low}(\mathcal{I}') \mid \mathsf{true}\}$*.*
5 $\{\triangleright I_{\mathsf{m}}(\mathtt{x}, r, P)(\mathsf{Low}(\mathcal{I})) * [\mathsf{send}]^r_1 * \lfloor P \rfloor\}$
6 *// invariant definition.*
7 $\{[\mathsf{send}]^r_1 * \lfloor P \rfloor * \triangleright(\mathtt{x}.\mathsf{flag} \mapsto 0 * \mathsf{waiting}(P, \mathcal{I}) * \mathsf{changes}_r(\mathcal{I}))\}$
8 *// pull out points-to using* $LBin$ *and* $LPoints$*.*
9 $\{[\mathsf{send}]^r_1 * \lfloor P \rfloor * \mathtt{x}.\mathsf{flag} \mapsto 0 * \triangleright(\mathsf{waiting}(P, \mathcal{I}) * \mathsf{changes}_r(\mathcal{I}))\}$
10 `x->flag := 1;` *// rewrite flag, drop* $\triangleright$ *using* $AFrame$
11 $\{[\mathsf{send}]^r_1 * \lfloor P \rfloor * \mathtt{x}.\mathsf{flag} \mapsto 1 * \mathsf{waiting}(P, \mathcal{I}) * \mathsf{changes}_r(\mathcal{I})\}$
12 *// Lemma* 5.1*, add a* $\triangleright$ *using* $SMono$*.*
13 $\{[\mathsf{send}]^r_1 * \triangleright(\mathtt{x}.\mathsf{flag} \mapsto 1 * \mathsf{ress}(\mathcal{I}) * \mathsf{changes}_r(\mathcal{I}))\}$
14 *// Invariant definition.*
15 $\{[\mathsf{send}]^r_1 * \triangleright I_{\mathsf{m}}(\mathtt{x}, r, P)(\mathsf{High}(\mathcal{I}))\}$
16 *// close the region.*
17 $\{[\mathsf{send}]^r_1 * \mathsf{creg}(\mathtt{x}, r, P, \{\mathsf{High}(\mathcal{I}') \mid \mathsf{true}\})\}$
18 `}` *// delete unneeded predicates.*
19 $\{\mathsf{emp}\}$

Fig. 10. Proof of `signal()` w.r.t. the full specification.

Proving `wait`. In the proof of `wait` (Fig. 11) 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 modelled 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. $r \overset{1/2}{\Longmapsto} P * \mathsf{ress}(\mathcal{I} \uplus \{r\}) \sqsubseteq \mathsf{ress}(\mathcal{I}) * \triangleright\lceil P \rceil$

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

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 enter the shared region and get rid of $\triangleright$. We present the proof outline in Figure 12. The core of the proof is two lemmas which express splitting in the Low and High cases.

$\left\{\mathsf{recv}(\mathsf{x}, P)\right\}$
`wait(chan *x) {`
$\left\{\exists R, r, r'.\, r' \overset{1/2}{\Longmapsto} P * [\mathsf{change}(r')]^r_1 * \mathsf{creg}(\mathsf{x}, r, R, \{\mathsf{LoHi}(\mathcal{I}') \mid r' \in \mathcal{I}'\})\right\}$
// Open the region. Only consider the High *case.*
$\left\{r' \in \mathcal{I} \wedge r' \overset{1/2}{\Longmapsto} P * [\mathsf{change}(r')]^r_1 * \triangleright I_{\mathsf{m}}(\mathsf{x}, r, R)(\mathsf{High}(\mathcal{I}))\right\}$
// Apply invariant definition
// Pull out points-to using $LBin$ *and* $LPoints$
$\left\{r' \in \mathcal{I} \wedge r' \overset{1/2}{\Longmapsto} P * [\mathsf{change}(r')]^r_1 * \mathsf{x.flag} \mapsto 1 * \triangleright(\mathsf{ress}(\mathcal{I}) * \mathsf{changes}_r(\mathcal{I}))\right\}$
`assume(x->flag == 1)` *// drop* $\triangleright$ *using* $AFrame$*, push in* $[\mathsf{change}(r')]$ *perm.*
$\left\{r' \in \mathcal{I} \wedge r' \overset{1/2}{\Longmapsto} P * \mathsf{x.flag} \mapsto 1 * \mathsf{ress}(\mathcal{I}) * \mathsf{changes}_r(\mathcal{I} \uplus \{r'\})\right\}$
// Lemma 5.2*, add* $\triangleright$ *using* $SMono$
$\left\{\triangleright\lceil P\rceil * \triangleright(\mathsf{x.flag} \mapsto 1 * \mathsf{ress}(\mathcal{I} \setminus \{r'\}) * \mathsf{changes}_r(\mathcal{I} \setminus \{r'\}))\right\}$
// Invariant definition
$\left\{\triangleright\lceil P\rceil * \triangleright I_{\mathsf{m}}(\mathsf{x}, r, R)(\mathsf{High}(\mathcal{I} \setminus \{r'\}))\right\}$
// close the region
$\left\{\triangleright\lceil P\rceil * \exists R, r.\, \mathsf{creg}(\mathsf{x}, r, R, \{\mathsf{High}(\mathcal{I}) \mid \mathsf{true}\})\right\}$
// abstract garbage collect
$\left\{\triangleright\lceil P\rceil\right\}$
`}`

Fig. 11. Proof of `wait()` w.r.t. the full specification.

$\left\{\mathsf{recv}(\mathsf{x}, P) * \lfloor\lceil P\rceil \mathrel{-\!\!*} P_1 * P_2\rfloor\right\}$
$\left\{\exists R, r, r'.\, r' \overset{1/2}{\Longmapsto} P * [\mathsf{change}(r')]^r_1 * \mathsf{creg}(\mathsf{x}, r, R, \{\mathsf{LoHi}(\mathcal{I}') \mid r' \in \mathcal{I}'\}) * \lfloor\lceil P\rceil \mathrel{-\!\!*} P_1 * P_2\rfloor\right\}$
// Open the region using an arbitrary state containing r'
$\left\{\exists R, r, r'.\, r' \overset{1/2}{\Longmapsto} P * [\mathsf{change}(r')]^r_1 * \triangleright I_{\mathsf{m}}(\mathsf{x}, r, R)(\mathsf{LoHi}(\mathcal{I} \uplus \{r'\})) * \lfloor\lceil P\rceil \mathrel{-\!\!*} P_1 * P_2\rfloor\right\}$
`skip` *// Use AFrame to remove* $\triangleright$
$\left\{\exists R, r, r'.\, r' \overset{1/2}{\Longmapsto} P * [\mathsf{change}(r')]^r_1 * I_{\mathsf{m}}(\mathsf{x}, r, R)(\mathsf{LoHi}(\mathcal{I} \uplus \{r'\})) * \lfloor\lceil P\rceil \mathrel{-\!\!*} P_1 * P_2\rfloor\right\}$
// Lemma 5.5
$\left\{\begin{array}{l}\exists R, r, r_1, r_2.\, r_1 \overset{1/2}{\Longmapsto} P_1 * r_1 \overset{1/2}{\Longmapsto} P_2 * [\mathsf{change}(r_1)]^r_1 * [\mathsf{change}(r_2)]^r_1 \\ * I_{\mathsf{m}}(\mathsf{x}, r, R)(\mathsf{LoHi}(\mathcal{I} \uplus \{r_1, r_2\}))\end{array}\right\}$
// Close the region
$\left\{\begin{array}{l}\exists R, r, r_1, r_2.\, r_1 \overset{1/2}{\Longmapsto} P_1 * r_1 \overset{1/2}{\Longmapsto} P_2 * [\mathsf{change}(r_1)]^r_1 * [\mathsf{change}(r_2)]^r_1 \\ * \mathsf{creg}(\mathsf{x}, r, R, \{\mathsf{LoHi}(\mathcal{I}') \mid r_1, r_2 \in \mathcal{I}'\})\end{array}\right\}$
$\left\{\mathsf{recv}(\mathsf{x}, P_1) * \mathsf{recv}(\mathsf{x}, P_2)\right\}$

Fig. 12. Proof outline for splitting axiom.

LEMMA 5.3. $r \overset{1/2}{\Longrightarrow} P * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor * \mathsf{waiting}(R, \mathcal{I} \uplus r)$
$\sqsubseteq \ \exists r_1, r_2.\, r_1 \overset{1/2}{\Longrightarrow} P_1 * r_2 \overset{1/2}{\Longrightarrow} P_2 * \mathsf{waiting}(R, \mathcal{I} \uplus \{r_1, r_2\})$

PROOF.

$r \overset{1/2}{\Longrightarrow} P * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor * \mathsf{waiting}(R, \mathcal{I} \uplus r)$
$\sqsubseteq\ r \overset{1/2}{\Longrightarrow} P * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor *$
$\exists Q \colon \mathcal{I} \uplus \{r\} \to \mathsf{Prop}.\, \lfloor (\triangleright R) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I} \uplus r}.\, Q(i) \rfloor * (\circledast_{i \in \mathcal{I} \uplus r}.\, i \overset{1/2}{\Longrightarrow} Q(i))$
$\sqsubseteq\ \exists Q'.\ r \overset{1/2}{\Longrightarrow} P * r \overset{1/2}{\Longrightarrow} Q' * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor *$
$\exists Q \colon \mathcal{I} \to \mathsf{Prop}.\, \lfloor (\triangleright R) \mathrel{-\!\!*} \triangleright Q' * \triangleright \circledast_{i \in \mathcal{I}}.\, Q(i) \rfloor * (\circledast_{i \in \mathcal{I}}.\, i \overset{1/2}{\Longrightarrow} Q(i))$
$\sqsubseteq\ \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor *$
$\exists Q \colon \mathcal{I} \to \mathsf{Prop}.\, \lfloor (\triangleright R) \mathrel{-\!\!*} \triangleright P * \triangleright \circledast_{i \in \mathcal{I}}.\, Q(i) \rfloor * (\circledast_{i \in \mathcal{I}}.\, i \overset{1/2}{\Longrightarrow} Q(i))$
$\sqsubseteq\ \lfloor \lceil \triangleright P \rceil \mathrel{-\!\!*} (\triangleright P_1) * (\triangleright P_2) \rfloor *$
$\exists Q \colon \mathcal{I} \to \mathsf{Prop}.\, \lfloor (\triangleright R) \mathrel{-\!\!*} \lceil \triangleright P \rceil * \triangleright \circledast_{i \in \mathcal{I}}.\, Q(i) \rfloor * (\circledast_{i \in \mathcal{I}}.\, i \overset{1/2}{\Longrightarrow} Q(i))$
$\sqsubseteq\ \exists Q \colon \mathcal{I} \to \mathsf{Prop}.\, \lfloor (\triangleright R) \mathrel{-\!\!*} (\triangleright P_1) * (\triangleright P_2) * \triangleright \circledast_{i \in \mathcal{I}}.\, Q(i) \rfloor * (\circledast_{i \in \mathcal{I}}.\, i \overset{1/2}{\Longrightarrow} Q(i))$
$\sqsubseteq\ \exists r_1, r_2.\, r_1 \overset{1}{\Longrightarrow} P_1 * r_2 \overset{1}{\Longrightarrow} P_2 \wedge r_1, r_2 \notin \mathcal{I} *$
$\exists Q \colon \mathcal{I} \to \mathsf{Prop}.\, \lfloor (\triangleright R) \mathrel{-\!\!*} (\triangleright P_1) * (\triangleright P_2) * \triangleright \circledast_{i \in \mathcal{I}}.\, Q(i) \rfloor * (\circledast_{i \in \mathcal{I}}.\, i \overset{1/2}{\Longrightarrow} Q(i))$
$\sqsubseteq\ \exists r_1, r_2.\, r_1 \overset{1/2}{\Longrightarrow} P_1 * r_2 \overset{\frac{1}{2}}{\Longrightarrow} P_2 *$
$\exists Q \colon \mathcal{I} \uplus \{r_1, r_2\} \to \mathsf{Prop}.\, \lfloor (\triangleright R) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I} \uplus \{r_1, r_2\}}.\, Q(i) \rfloor * (\circledast_{i \in \mathcal{I} \uplus \{r_1, r_2\}}.\, i \overset{1/2}{\Longrightarrow} Q(i))$
$\sqsubseteq\ \exists r_1, r_2.\, r_1 \overset{1/2}{\Longrightarrow} P_1 * r_2 \overset{\frac{1}{2}}{\Longrightarrow} P_2 * \mathsf{waiting}(R, \mathcal{I} \uplus \{r_1, r_2\})$

□

LEMMA 5.4. $r \overset{1/2}{\Longrightarrow} P * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor * \mathsf{ress}(\mathcal{I} \uplus r)$
$\sqsubseteq\ \exists r_1, r_2.\, r_1 \overset{1/2}{\Longrightarrow} P_1 * r_2 \overset{1/2}{\Longrightarrow} P_2 * \mathsf{ress}(\mathcal{I} \uplus \{r_1, r_2\})$

PROOF.

$r \overset{1/2}{\Longrightarrow} P * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor * \mathsf{ress}(\mathcal{I} \uplus r)$
$\sqsubseteq\ (\lceil P \rceil \mathrel{-\!\!*} P_1 * P_2) * \mathsf{ress}(\mathcal{I}) * \triangleright \lceil P \rceil$ (Lemma 5.2)
$\sqsubseteq\ ((\triangleright \lceil P \rceil) \mathrel{-\!\!*} (\triangleright P_1) * (\triangleright P_2)) * \mathsf{ress}(\mathcal{I}) * \triangleright \lceil P \rceil$ (SMono, dist $\triangleright$ over $\mathrel{-\!\!*}$)
$\sqsubseteq\ \mathsf{ress}(\mathcal{I}) * (\triangleright P_1) * (\triangleright P_2)$ (Modus poens)
$\sqsubseteq\ \mathsf{ress}(\mathcal{I}) * (\triangleright P_1) * (\triangleright P_2) * \exists r_1, r_2.\, r_1 \overset{1}{\Longrightarrow} P_1 * r_2 \overset{1}{\Longrightarrow} P_2$ (Property 1)
$\sqsubseteq\ \mathsf{ress}(\mathcal{I} \uplus \{r_1, r_2\}) * \exists r_1, r_2.\, r_1 \overset{1/2}{\Longrightarrow} P_1 * r_2 \overset{1/2}{\Longrightarrow} P_2$ (Sub-lemma)

The last step consists of two applications of the following sub-lemma:

$\mathsf{ress}(\mathcal{I}) * \triangleright P * r \overset{1}{\Longrightarrow} P$
$\sqsubseteq\ (\circledast_{i \in \mathcal{I}} \exists Q.\, i \overset{1/2}{\Longrightarrow} Q * \lceil \triangleright Q \rceil) * \lceil \triangleright P \rceil * r \overset{1}{\Longrightarrow} P$
$\sqsubseteq\ (\circledast_{i \in \mathcal{I} \uplus \{r\}} \exists Q.\, i \overset{1/2}{\Longrightarrow} Q * \lceil \triangleright Q \rceil) * r \overset{1/2}{\Longrightarrow} P$
$\sqsubseteq\ \mathsf{ress}(\mathcal{I} \uplus \{r\}) * r \overset{1/2}{\Longrightarrow} P$

□

These lemmas are combined as follows.

LEMMA 5.5.
$$r' \overset{1/2}{\Longmapsto} P * I_{\mathsf{m}}(x,r,R)(\mathsf{LoHi}(\mathcal{I} \uplus \{r'\})) * [\mathsf{change}(r')]^1_r * \lfloor \lceil P \rceil \mathrel{-\!\!*} P_1 * P_2 \rfloor$$
$$\sqsubseteq$$
$$\exists r_1, r_2.\, [\mathsf{change}(r_1)]^1_r * r_1 \overset{1/2}{\Longmapsto} P_1 * [\mathsf{change}(r_2)]^1_r * r_2 \overset{1/2}{\Longmapsto} P_2 * \triangleright I_{\mathsf{m}}(x,r,R)(\mathsf{LoHi}(\mathcal{I} \uplus \{r_1, r_2\}))$$

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

6. CHAINS AND RENUNCIATION

The simple barrier implementation verified in §4 and §5 does not consider an order of channels. In this section, we verify an implementation that supports chains of channels and early renuncication. Calls to `signal` can complete in any order consistent with the specification. To ensure renounced resources are available, `wait` checks all predecessors in the chain. We prove that this implementation implements our abstract specification.

This new implementation uses a linked list data-structure. The implementations of `extend`, `newchan`, `signal` and `wait` operations are 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 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.

6.1. Proof Structure

In our proof of the simple barrier implementation, each shared region contained a single channel. In the current implementation, `wait` traverses the whole preceding chain. To allow this, the proof stores the whole chain in a single region, represented by a single invariant. This makes it more complex, but the core approach remains the same:

— send, recv, and ordering predicates are reified as constraints on the shared region.
— Resource obligations are represented by sets of identifiers, which are tied to saved propositions.

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.

Abstract chain type. To represent the state of channels in the chain, we define a type of abstract chain nodes, CNode:

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

Each CNode represents one channel in the chain, while an *abstract chain* is a finite sequence in CNode^+. Given an element $(\mathit{loc}, \mathit{res}, \mathcal{I}, \mathit{flg}, \mathcal{W}) \in \mathsf{CNode}$, *loc* is the physical address, *res* is the identifier used when sending resources, $\mathcal{I}$ is the set of identifiers for resources promised to other predicates, *flg* is the current state of the flag, and $\mathcal{W}$ is the set of identifiers for resources earlier in the chain promised to this channel through renunciation. (Given a CNode s, we sometimes write $s.\mathit{flg}$, $s.\mathcal{I}$ etc. to identify the appropriate components of the tuple. We use 0 and 1 to represent the Low and High flag state, respectively.)

Given an abstract chain $x \cdot xs$, it is important that identifiers in the set $x.\mathcal{W}$ are in the set of resources promised in xs (i.e. in some set $\mathcal{I}$). To ensure this, we define well-formedness of abstract chains as follows:

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

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

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 promised resources are empty.

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

Finally, we use $rs_1 \overset{\mathcal{W}}{\rightarrow}{}^* rs_2$ to denote that the abstract chain rs_2 follows from rs_1 by cancelling out renounced resources with the corresponding promise. Formally, $\overset{\mathcal{W}}{\rightarrow}{}^*$ is defined as the transitive-reflexive closure of the following relation, which cancels a single promise to a later node using a renounced resource from an earlier node.

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

The notation $\blacktriangleleft$ is a *lens* allowing a single field of a chain to be updated without modifying the remainder of the chain.

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

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

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

Fig. 13. Definition of T_{c}, the transition relation for the chained channel implementation.

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

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

$$
\begin{array}{rcl}
\mathsf{uS}(rs) & \triangleq & \{x \mid (x, \_, \_, 0, \_) \in rs\} \\
\mathsf{uC}(rs) & \triangleq & \{(x, i) \mid (x, \_, \mathcal{I}, \_, \_) \in rs \land i \in \mathcal{I} \land \neg\exists (y, \_, \_, \_, \mathcal{W}) \in rs.\, i \in \mathcal{W}\} \\
\mathsf{unused}(r, rs) & \triangleq & (\circledast\, x \notin \mathsf{uS}(rs).\, [\mathsf{send}(x)]_1^r) * (\circledast (x, r') \notin \mathsf{uC}(rs).\, [\mathsf{change}(x, r')]_1^r)
\end{array}
$$

Fig. 14. Predicates used in defining the state of a region.

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 Fig. 13. 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 Fig. 13. 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 Fig. 14). The predicate chainds represents the list

data-structure underpinning the implementation. Each link in the chain has the appropriate prev and flag values set.

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 renounced resources that are being waited for $\mathcal{W}$. When there are no waiting resources the predicate can be drastically simplified to just the promised resources:

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

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:

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

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

$$\mathsf{oreg}(r, S) \triangleq \mathsf{region}(S, T_{\mathsf{c}}, I_{\mathsf{c}}(r), r)$$

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

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

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

$$x \prec y \triangleq \exists r.\, \mathsf{oreg}(r, \{\mathsf{Chain}(rs) \mid (y, x) \in \mathsf{ord}(rs)\})$$

6.2. Proving `signal`, `wait` and `extend`

Proving `signal`. The sketch-proof is shown in Fig. 15 – it is similar in structure to the one in §5.3. The main additional challenge is to show that resources 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 is sufficient to allow the flag to be set.

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

PROOF. Given in Appendix B. □

In the proof of `signal`, this lemma is used to show that the appropriate resource has been supplied (Fig. 15, line 12). By factoring logical resource transfer away from the physical signalling, 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 Fig. 16. The three most important steps are checking that all preceding bits are set (lines 5–11), checking that renounced resources have been supplied (line 14), and retrieving the resource from the chain (line 18). The last two of these require helper lemmas, given below.

1 $\left\{\mathsf{send}(\mathtt{x}, P) * \lfloor P \rfloor\right\}$

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 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \lfloor P \rfloor * \\ \triangleright \left(\mathsf{chainds}(rs) * \mathsf{chainres}(rs) * \mathsf{unused}(r_1, rs) \wedge \mathsf{wf}(rs) \wedge rs(\mathtt{x}) = (r_2, \mathcal{I}, 0, \mathcal{W})\right)\end{array}\right\}$

6 *// Split up using chain axioms.*

7 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \lfloor P \rfloor * \\ \triangleright \left(\begin{array}{l}\exists rs_1, rs_2, a.\, \mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 0\} * \mathsf{chainds}(rs_2) * \\ \mathsf{chainres}(rs) * \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge a = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W})\end{array}\right)\end{array}\right\}$

8 `x->flag = 1;` *// Set the flag, drop the ▷ using Aframe.*

9 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \lfloor P \rfloor * \\ \exists rs_1, rs_2, a.\, \mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 1\} * \mathsf{chainds}(rs_2) * \\ \mathsf{chainres}(rs) * \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge a = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W})\end{array}\right\}$

10 *// Pull out* resource *predicate.*

11 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \lfloor P \rfloor * \\ \exists rs_1, rs_2, a.\, \mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 1\} * \mathsf{chainds}(rs_2) \\ \qquad * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r_2\}) * \mathsf{chainres}(rs_2) \\ \qquad * \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge a = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W})\end{array}\right\}$

12 *// Apply Lemma* 6.2.

13 $\left\{\begin{array}{l}\exists r_1, r_2.\, [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \\ \exists rs_1, rs_2, a.\, \mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 1\} * \mathsf{chainds}(rs_2) \\ \qquad * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}, \mathcal{W}) * \mathsf{chainres}(rs_2) \\ \qquad * \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot a \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge a = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W})\end{array}\right\}$

14 *// Collapse the chain, add ▷.*

15 $\left\{\begin{array}{l}\exists r_1, r_2. \\ \triangleright\, (\mathsf{chainds}(rs') * \mathsf{chainres}(rs') * \mathsf{unused}(r_1, rs') \wedge rs' = rs[\mathtt{x} \mapsto (r_2, \mathcal{I}, 1, \mathcal{W})])\end{array}\right\}$

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 *// Garbage collect.*

20 $\left\{\mathsf{emp}\right\}$

Fig. 15. Sketch-proof for `signal` with out-of-order signalling.

1 $\left\{\mathsf{recv}(\mathtt{x}, P)\right\}$

2 `wait(x){`

3 `chan *c = x;`

4 *// Definition of* recv*.*

5 $\left\{\begin{array}{l}\exists r_1, r_2.\, \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid \exists \mathcal{I}'.\, rs(\mathtt{x}) = (\_, \mathcal{I}', \_, \_) \land r_2 \in \mathcal{I}' \land \mathsf{wf}(rs)\}) \\ \qquad * \, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]_1^{r_1} \land \mathsf{c} = \mathtt{x}\end{array}\right\}$

6 `while(c != NULL) {`

7 `while(!c->flag) skip;`

8 `c = c->prev;`

9 `}`

10 *// Flags up to x are now set. Stable because flags cannot get unset.*

11 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]_1^{r_1} * \\ \mathsf{oreg}\left(r_1, \left\{\mathsf{Chain}(rs) \;\middle|\; \begin{array}{l}\exists \mathcal{I}'.\, \mathsf{wf}(rs) \land \exists i.\, rs[i] = (\mathtt{x}, \_, \mathcal{I}' \uplus r_2, \_, \_) \land \\ \forall j.\, i \le j < \mathrm{len}(rs) \implies rs[j] = (\_, \_, \_, 1, \_)\end{array}\right\}\right)\end{array}\right\}$

12 $\langle$`skip`$\rangle$; *// Enter the region and drop* $\triangleright$ *using* $AFrame$*.*

13 $\left\{\begin{array}{l}\exists rs_1, rs_2, a.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]_1^{r_1} * \mathsf{chainds}(rs) * \mathsf{chainres}(rs) * \mathsf{unused}(r_1, rs) \land \\ \mathsf{wf}(rs) \land rs = rs_1 \cdot a \cdot rs_2 \land a = (\mathtt{x}, r, \mathcal{I}' \uplus r_2, \_, \mathcal{W}) \land \mathsf{ctrue}(a \cdot rs_2)\end{array}\right\}$

14 *// Apply Lemma* 6.3 *to convert from* ctrue *to* cconf*..*

15 $\left\{\begin{array}{l}\exists rs_1, rs_2, rs_2', a.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]_1^{r_1} * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1 \cdot rs_2') * \mathsf{unused}(r_1, rs) \land \mathsf{wf}(rs) \land rs = rs_1 \cdot a \cdot rs_2 \land \\ a = (\mathtt{x}, r, \mathcal{I}' \uplus r_2, \_, \mathcal{W}) \land \mathsf{cconf}(rs_2') \land a \cdot rs_2 \overset{\mathcal{W}_*}{\to} rs_2'\end{array}\right\}$

16 *// Pull out the* resource *predicate for x.*

17 $\left\{\begin{array}{l}\exists rs_1, rs_2, rs_2', a.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]_1^{r_1} * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}' \uplus r_2, \emptyset) * \mathsf{chainres}(rs_2') * \mathsf{unused}(r_1, rs) \land \\ \mathsf{wf}(rs) \land rs = rs_1 \cdot a \cdot rs_2 \land a = (\mathtt{x}, r, \mathcal{I}' \uplus r_2, \_, \mathcal{W}) \land \mathsf{cconf}(rs_2') \land a \cdot rs_2 \overset{\mathcal{W}_*}{\to} rs_2'\end{array}\right\}$

18 *// Apply Lemma* 6.4*.*

19 $\left\{\begin{array}{l}\lceil \triangleright P \rceil * \exists rs_1, rs_2, rs_2', a.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(\mathtt{x}, r_2)]_1^{r_1} * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}', \emptyset) * \mathsf{chainres}(rs_2') * \mathsf{unused}(r_1, rs) \land \\ \mathsf{wf}(rs) \land rs = rs_1 \cdot a \cdot rs_2 \land a = (\mathtt{x}, r, \mathcal{I}' \uplus r_2, \_, \mathcal{W}) \land \mathsf{cconf}(rs_2') \land a \cdot rs_2 \overset{\mathcal{W}_*}{\to} rs_2'\end{array}\right\}$

20 *// Close the region, use transitions* sat *and* get*.*

21 $\left\{\begin{array}{l}\exists r_1, r_2.\, \lceil \triangleright P \rceil * \\ \mathsf{oreg}\left(r_1, \left\{\mathsf{Chain}(rs') \;\middle|\; \begin{array}{l}\mathsf{wf}(rs') \land rs = rs_1 \cdot a \cdot rs_2 \land a \cdot rs_2 \overset{\mathcal{W}_*}{\to} rs_2' \land \\ \mathsf{cconf}(rs_2') \land rs' = (rs_1 \cdot rs_2') \blacktriangleleft_x \blacktriangleleft_{\mathcal{I}} (\bullet \setminus \{r_2\})\end{array}\right\}\right)\end{array}\right\}$

22 `}`*// Use* $\lceil \triangleright P \rceil \implies \triangleright \lceil P \rceil$ *and clean up garbage.*

23 $\left\{\triangleright \lceil P \rceil\right\}$

Fig. 16. Sketch-proof for `wait` with out-of-order signalling.

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 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. $\mathsf{chainres}(rs) \land \mathsf{wf}(rs) \land \mathsf{ctrue}(rs)$
$\sqsubseteq \quad \exists rs'.\, \mathsf{chainres}(rs') \land \mathsf{cconf}(rs') \land rs \overset{\mathcal{W}}{\rightarrow}{}^{*}\ rs' \land \mathsf{wf}(rs')$

PROOF. Given in Appendix B. □

We apply this lemma on line 14 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. $\mathsf{resource}(\mathcal{I} \uplus r_2, \emptyset) * r_2 \overset{1/2}{\Longmapsto} P \quad \sqsubseteq \quad \mathsf{resource}(\mathcal{I}, \emptyset) * \lceil \triangleright P \rceil$

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 18 of the sketch-proof.

Proving `extend`. The sketch-proof is given in Fig. 17. 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 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 following lemma uses this freshness property, along with the freshness of allocated locations to show that we can we can retrieve the required permissions from unused. We use this lemma on line 16 of the sketch-proof.

LEMMA 6.6.

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

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

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 $\left\{\begin{array}{l}\exists r_1, r_2, x'.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \mathtt{z} \mapsto \{\mathrm{prev} = x'; \mathrm{flag} = 0\} * \\ \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(\mathtt{x}) = (\mathtt{x}, r_2, \_, 0, \_) \wedge (\mathtt{x}, x') \in \mathsf{seq}(rs) \wedge \mathtt{z} \notin rs\})\end{array}\right\}$

6 `  skip; skip;` *// Make a pair of stored propositions, appeal to Lemma* 6.5.

7 $\left\{\begin{array}{l}\exists r_1, r_2, x', r', r''.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \mathtt{z} \mapsto \{\mathrm{prev} = x'; \mathrm{flag} = 0\} * r' \overset{1}{\Longrightarrow} Q * r'' \overset{1}{\Longrightarrow} Q * \\ \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(\mathtt{x}) = (\mathtt{x}, r_2, \_, 0, \_) \wedge (\mathtt{x}, x') \in \mathsf{seq}(rs) \wedge \mathtt{z}, r', r'' \notin rs\})\end{array}\right\}$

8 `  // Create a node in the chain.`

9 $\left\{\begin{array}{l}\exists r_1, r_2, x', r', r''.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \\ \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(\mathtt{x}) = (\mathtt{x}, r_2, \_, 0, \_) \wedge (\mathtt{x}, x') \in \mathsf{seq}(rs) \wedge \mathtt{z}, r', r'' \notin rs\}) * \\ \qquad r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q * \mathtt{z} \mapsto \{\mathrm{prev} = x'; \mathrm{flag} = 0\} * \mathsf{resource}(\{r''\}, \{r'\})\end{array}\right\}$

10 `  // Enter the region, split the chain up.`

11 $\left\{\begin{array}{l}\exists r_1, r_2, x', r', r'', h, rs_1, rs_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \\ \triangleright \left(\begin{array}{l}\mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = x'; \mathrm{flag} = 0\} * \mathsf{chainds}(rs_2) * \mathsf{chainres}(rs) * \\ \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot e \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge e = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W}) \wedge \mathtt{z}, r', r'' \notin rs\end{array}\right) * \\ \quad r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q * \mathtt{z} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 0\} * \mathsf{resource}(\{r''\}, \{r'\})\end{array}\right\}$

12 `  x->prev = z;` *// Stitch the cnode into the chain using LPoints / AFrame.*

13 $\left\{\begin{array}{l}\exists r_1, r_2, x', r', r'', h, rs_1, rs_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * \\ \mathsf{chainds}(rs_1) * \mathtt{x} \mapsto \{\mathrm{prev} = \mathtt{z}; \mathrm{flag} = 0\} * \mathsf{chainds}(rs_2) * \mathsf{chainres}(rs) * \\ \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot e \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge e = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W}) \wedge \mathtt{z}, r', r'' \notin rs * \\ r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q * \mathtt{z} \mapsto \{\mathrm{prev} = \mathsf{hd}(rs_2).loc; \mathrm{flag} = 0\} * \mathsf{resource}(\{r''\}, \{r'\})\end{array}\right\}$

14 `  // Fold the` chainds *predicate back up.*

15 $\left\{\begin{array}{l}\exists r_1, r_2, x', r', r'', h, rs_1, rs_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q \\ \mathsf{chainds}(rs_1 \cdot e \cdot e' \cdot rs_2) * \mathsf{chainres}(rs_1 \cdot e \cdot e' \cdot rs_2) * \mathsf{unused}(r_1, rs) \wedge \\ rs = rs_1 \cdot e \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge e = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W}) \wedge e' = (\mathtt{z}, r', \{r''\}, 0, \emptyset) \wedge \mathtt{z}, r', r'' \notin rs\end{array}\right\}$

16 `  // Get` $[\mathsf{send}(z)] * [\mathsf{change}(z, r'')]$ *from* $\mathsf{unused}(r_1, rs)$ *using Lemma* 6.6.

17 $\left\{\begin{array}{l}\exists r_1, r_2, r', r'', h, rs_1, rs_2. \\ \mathsf{chainds}(rs_1 \cdot e \cdot e' \cdot rs_2) * \mathsf{chainres}(rs_1 \cdot e \cdot e' \cdot rs_2) * \mathsf{unused}(r_1, rs_1 \cdot e \cdot e' \cdot rs_2) \\ \quad \wedge rs = rs_1 \cdot e \cdot rs_2 \wedge \mathsf{wf}(rs) \wedge e = (\mathtt{x}, r_2, \mathcal{I}, 0, \mathcal{W}) \wedge e' = (\mathtt{z}, r', \{r''\}, 0, \emptyset) \\ \wedge\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q * [\mathsf{send}(\mathtt{z})]^{r_1}_1 * [\mathsf{change}(\mathtt{z}, r'')]^{r_1}_1\end{array}\right\}$

18 `  // Close the region using the` ext *transition.*

19 $\left\{\begin{array}{l}\exists r_1, r_2, r', r'', h. \\ \mathsf{oreg}\left(r_1, \left\{\mathsf{Chain}(rs') \;\middle|\; rs' = rs_1 \cdot (\mathtt{x}, r_2, \_, 0, \_) \cdot (\mathtt{z}, r', \mathcal{I}', 0, \_) \cdot rs_2 \wedge r'' \in \mathcal{I}' \wedge \mathsf{wf}(rs')\right\}\right) \\ \quad \wedge\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{send}(\mathtt{x})]^{r_1}_1 * r' \overset{1/2}{\Longmapsto} Q * r'' \overset{1/2}{\Longmapsto} Q * [\mathsf{send}(\mathtt{z})]^{r_1}_1 * [\mathsf{change}(\mathtt{z}, r'')]^{r_1}_1\end{array}\right\}$

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. 17. Sketch-proof of `extend` with out-of-order signalling.

is stable because we hold the send tokens on x and z, meaning these channels cannot be extended or renounced.

Proving `newchan`. Omitted: this proof is similar to `extend`, but simpler.

6.3. 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 Fig. 18. The important steps are conjoining the three predicates to give a single stable view on the shared structure (line 2), supplying the renounced resource to the shared region (line 12), and closing the region to give a new send predicate (line 14).

In order to conjoin the regions arising from the send, recv and the 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 2, Fig. 18. The conjoined region that arises from this lemma (line 5) 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 12) we use the following lemma to show that the renunciation set $\mathcal{W}$ is updated appropriately:

Lemma 6.8.
$$\begin{aligned} &\mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r\}) * r \overset{1/2}{\Longmapsto} Q * r' \overset{1/2}{\Longmapsto} P \\ \sqsubseteq\ &\exists r''.\, \mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r', r''\}) * r'' \overset{1/2}{\Longmapsto} (P \mathrel{-\!\!*} Q) \end{aligned}$$

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 garbage collecting one saved proposition and creating another.

On line 14 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] token is held.

Splitting. The axiom is defined as follows:

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

A sketch-proof is given in Fig. 19. The key step is splitting the promised resource set $\mathcal{I}$ for a node (line 6). To do this, we use the following lemma.

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

1 $\left\{\mathsf{recv}(x,P) * \mathsf{send}(y,Q) * x \prec y\right\}$

2 $\langle \texttt{skip} \rangle$; *// Definition of* recv, send, *Lemma* 6.7.

3 $\left\{\begin{array}{l} \exists r, r_2, r_3.\, \mathsf{oreg}(r, \{\mathsf{Chain}(rs) \mid (y,x) \in \mathsf{ord}(rs)\}) * \\ \left(\begin{array}{c}\mathsf{oreg}(r, \{\mathsf{Chain}(rs) \mid rs(x) = (\_, \mathcal{I}, \_, \_) \wedge r_2 \in \mathcal{I} \wedge \mathsf{wf}(rs)\}) \\ * \, r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(x, r_2)]^r_1\end{array}\right) * \\ \left(\begin{array}{c}\mathsf{oreg}\,(r, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(y) = (r_3, \_, 0, \_)\}) \\ * \, r_3 \overset{1/2}{\Longrightarrow} Q * [\mathsf{send}(y)]^r_1\end{array}\right) \end{array}\right\}$

4 *// Conjunction on regions. Stable because of the permissions held.*

5 $\left\{\begin{array}{l} \exists r, r_2, r_3.\, r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(x, r_2)]^r_1 * r_3 \overset{1/2}{\Longrightarrow} Q * [\mathsf{send}(y)]^r_1 * \\ \mathsf{oreg}\left(r, \left\{\mathsf{Chain}(rs) \;\middle|\; \begin{array}{l} rs = rs_1 \cdot e \cdot rs_2 \cdot e' \cdot rs_3 \wedge \mathsf{wf}(rs) \wedge \\ e = (y, r_3, \_, 0, \_) \wedge e' = (x, \_, \mathcal{I}, \_, \_) \wedge r_2 \in \mathcal{I} \end{array}\right\}\right) \end{array}\right\}$

6 $\langle \texttt{skip} \rangle$; *// Open the region, drop* $\triangleright$ *using AFrame.*

7 $\left\{\begin{array}{l} \exists r, r_2, r_3.\, r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(x, r_2)]^r_1 * r_3 \overset{1/2}{\Longrightarrow} Q * [\mathsf{send}(y)]^r_1 * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs) * \mathsf{unused}(r, rs) \wedge rs = rs_1 \cdot e \cdot rs_2 \cdot e' \cdot rs_3 \wedge \\ \mathsf{wf}(rs) \wedge e = (y, r_3, \mathcal{I}', 0, \mathcal{W}') \wedge e' = (x, r', \mathcal{I}, f, \mathcal{W}) \wedge r_2 \in \mathcal{I} \end{array}\right\}$

8 *// Pull out node.*

9 $\left\{\begin{array}{l} \exists r, r_2, r_3.\, r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{change}(x, r_2)]^r_1 * r_3 \overset{1/2}{\Longrightarrow} Q * [\mathsf{send}(y)]^r_1 * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}', \mathcal{W}' \uplus \{r_3\}) * \mathsf{chainres}(rs_2 \cdot e' \cdot rs_3) * \mathsf{unused}(r, rs) \wedge \\ rs = rs_1 \cdot e \cdot rs_2 \cdot e' \cdot rs_3 \wedge \mathsf{wf}(rs) \wedge e = (y, r_3, \mathcal{I}', 0, \mathcal{W}') \wedge e' = (x, r', \mathcal{I}, f, \mathcal{W}) \wedge r_2 \in \mathcal{I} \end{array}\right\}$

10 *// Apply Lemma* 6.8 *to supply the resource*

11 $\left\{\begin{array}{l} \exists r, r_2, r_3.\, [\mathsf{change}(x, r_2)]^r_1 * r_4 \overset{1/2}{\Longrightarrow} (P \mathrel{-\!\!*} Q) * [\mathsf{send}(y)]^r_1 * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I}', \mathcal{W}' \uplus \{r_2, r_4\}) * \mathsf{chainres}(rs_2 \cdot e' \cdot rs_3) * \mathsf{unused}(r, rs) \wedge \\ rs = rs_1 \cdot e \cdot rs_2 \cdot e' \cdot rs_3 \wedge \mathsf{wf}(rs) \wedge e = (y, r_3, \mathcal{I}', 0, \mathcal{W}') \wedge e' = (x, r', \mathcal{I}, f, \mathcal{W}) \wedge r_2 \in \mathcal{I} \end{array}\right\}$

12 *// Close the chain.*

13 $\left\{\begin{array}{l} \exists r, r_2, r_4.\, r_4 \overset{1/2}{\Longrightarrow} (P \mathrel{-\!\!*} Q) * [\mathsf{send}(y)]^r_1 * \mathsf{chainds}(rs') * \mathsf{chainres}(rs') * \mathsf{unused}(r, rs') \wedge \\ rs = rs_1 \cdot e \cdot rs_2 \cdot e' \cdot rs_3 \wedge rs' = rs_1 \cdot (y, r_4, \mathcal{I}', 0, \mathcal{W}' \uplus \{r_2\}) \cdot rs_2 \cdot e' \cdot rs_3 \wedge \\ \mathsf{wf}(rs) \wedge e = (y, \_, \mathcal{I}', 0, \mathcal{W}') \wedge e' = (x, r', \mathcal{I}, f, \mathcal{W}) \wedge r_2 \in \mathcal{I} \end{array}\right\}$

14 *// Close region using the* renun *transition.*

15 $\left\{\exists r, r_4.\, r_4 \overset{1/2}{\Longrightarrow} (P \mathrel{-\!\!*} Q) * [\mathsf{send}(y)]^r_1 * \mathsf{oreg}\,(r, \{\mathsf{Chain}(rs) \mid \mathsf{wf}(rs) \wedge rs(y) = (r_4, \_, 0, \_)\})\right\}$

16 *// Definition of* send.

17 $\left\{\mathsf{send}(y, P \mathrel{-\!\!*} Q)\right\}$

Fig. 18. Proof of the renunciation axiom for out-of-order implementation.

1 $\left\{\mathsf{recv}(x,P) * \lfloor \lceil P \rceil \mathrel{-\!\!*} (P_1 * P_2)\rfloor\right\}$

2 *// Definition of* recv*.*

3 $\left\{\begin{array}{l}\exists r_1, r_2.\, \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid rs(x) = (\_, \mathcal{I}, \_, \_) \wedge r_2 \in \mathcal{I} \wedge \mathsf{wf}(rs)\}) \\ \qquad * \, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(x, r_2)]_1^{r_1} * \lfloor \lceil P \rceil \mathrel{-\!\!*} (P_1 * P_2)\rfloor\end{array}\right\}$

4 *// Enter the region.*

5 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longmapsto} P * [\mathsf{change}(x, r_2)]_1^{r_1} * \lfloor \lceil P \rceil \mathrel{-\!\!*} (P_1 * P_2)\rfloor * \\ \triangleright \left(\begin{array}{l}\mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I} \uplus \{r_2\}, \mathcal{W}) * \mathsf{chainres}(rs_2) \wedge \\ \qquad rs = rs_1 \cdot (x, r, \mathcal{I} \uplus \{r_2\}, f, \mathcal{W}) \cdot rs_2 \wedge \mathsf{unused}(r_1, rs)\end{array}\right)\end{array}\right\}$

6 $\langle \mathtt{skip} \rangle$ *// AFrame to drop* $\triangleright$*, Lemma* 6.9 *to split the region.*

7 $\left\{\begin{array}{l}\exists r_1, r_2, r_3, r_4.\, r_3, r_4 \notin rs \wedge r_3 \overset{1/2}{\Longmapsto} P_1 * r_4 \overset{1/2}{\Longmapsto} P_2 * [\mathsf{change}(x, r_2)]_1^{r_1} * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I} \uplus \{r_3, r_4\}, \mathcal{W}) * \mathsf{chainres}(rs_2) * \\ \qquad \mathsf{unused}(r_1, rs) \wedge rs = rs_1 \cdot (x, r, \mathcal{I} \uplus \{r_2\}, f, \mathcal{W}) \cdot rs_2\end{array}\right\}$

8 *// Close chain, pull* [change] *perms out of* unused*.*

9 $\left\{\begin{array}{l}\exists r_1, r_2, r_3, r_4.\, r_3 \overset{1/2}{\Longmapsto} P_1 * r_4 \overset{1/2}{\Longmapsto} P_2 * [\mathsf{change}(x, r_3)]_1^{r_1} * [\mathsf{change}(x, r_4)]_1^{r_1} * \\ \mathsf{chainds}(rs) * \mathsf{chainres}(rs_1) * \mathsf{resource}(\mathcal{I} \uplus \{r_3, r_4\}, \mathcal{W}) * \mathsf{chainres}(rs_2) * \mathsf{unused}(r_1, rs') \\ \qquad \wedge rs = rs_1 \cdot (x, r, \mathcal{I} \uplus \{r_2\}, f, \mathcal{W}) \cdot rs_2 \wedge rs' = rs_1 \cdot (x, r, \mathcal{I} \uplus \{r_3, r_4\}, f, \mathcal{W}) \cdot rs_2\end{array}\right\}$

10 *// Close region using the* split *transition.*

11 $\left\{\begin{array}{l}\exists r_1, r_2, r_3, r_4.\, r_3 \overset{1/2}{\Longmapsto} P_1 * r_4 \overset{1/2}{\Longmapsto} P_2 * [\mathsf{change}(x, r_3)]_1^{r_1} * [\mathsf{change}(x, r_4)]_1^{r_1} * \\ \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs) \mid rs(x) = (\_, \mathcal{I}, \_, \_) \wedge r_3, r_4 \in \mathcal{I} \wedge \mathsf{wf}(rs)\})\end{array}\right\}$

12 *// Definition of* recv*.*

13 $\left\{\mathsf{recv}(x, P_1) * \mathsf{recv}(x, P_2)\right\}$

Fig. 19. 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 §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 towards a shared root rather than along a linear chain. This barrier implementation corresponds closely to the one in [Navabi et al. 2008] – the summarisation process is how it achieves its efficiency. Verifying this implementation demonstrates that our approach scales to custom synchronisation constructs developed for performance-sensitive concurrency applications.

7.1. Implementation approach

Figure 20 shows the implementation source-code. The data-structure is organised as follows. Flags are stored in fixed-size arrays of boolean values. These arrays are organised into a tree, with arrays closer to the root summarising arrays closer to the leaves. The header of each non-root array stores the address of a parent flag up the tree (in the `up` field). If one of these non-leaf flags is set, it denotes that all the leaves below it are also set. Thus `wait`

```
1  typedef struct chan_hdr chan_hdr;
2
3  typedef struct chan_addr {
4    chan_hdr *hdr;
5    int off;
6  } chan_addr;
7
8  typedef struct chan_hdr {
9    chan_addr up;
10   int loff;
11   bool flags[MAX];
12 } chan_hdr;
13
14 (chan_addr, chan_addr)
15 extend(chan_addr x){
16   chan_addr nx,r;
17   if(x.off == x.hdr->loff
18              && x.off < MAX){
19     x.hdr->flags[x.loff+1] = 0;
20     x.hdr->loff++;
21     r.hdr = x.hdr;
22     r.off = x.off + 1;
23     nx = x;
24   } else {
25     nh = malloc(chan_hdr);
26     nh->up = x;
27     nh->loff = 1;
28     nx.hdr = nh;
29     nx.off = 0;
30     r.hdr = nh;
31     r.off = 1;
32   }
33   return (r,nx);
34 }
```

```
35 signal(chan_addr x){
36   int i;
37   bool ret = FALSE;
38   chan_addr a = x;
39   while (a.hdr != NULL && !ret){
40     a.hdr->flags[a.off] = 1;
41     for(i=0; i<=a.hdr->loff; i++){
42       if (a.hdr->flags[i] != 1)
43         ret = TRUE;
44     }
45     a = a.hdr->up;
46   }
47 }
48
49
50
51
52
53
54
55
56
57 wait(chan_addr x){
58   chan_addr a = x;
59   while (a.hdr != NULL) {
60     for(skip; a.off<=a.hdr->loff;
61                          a.off++){
62       while(a.hdr->flags[a.off] != 1)
63         skip;
64     }
65     a = a.hdr->up;
66     a.off++;
67   }
68 }
```

Fig. 20. Channel implementation with out-of-order signalling and summarization.

can check these summary flags rather than the entire chain. An instance of the structure is illustrated in Fig. 21.

Signals can be sent out-of-order, as with our previous implementation. To `signal` a channel, the function reads the array location from the array header, and writes 1 into the flag at the appropriate offset (Fig. 20, 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, a call to `signal` summarises its siblings. If a flag's siblings are 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. If a summary flag is set, the algorithm can conclude that all the summarised leaf flags are also set.

The `wait` function exploits summaries to reduce the number of flags it must test. Rather than examining all the preceding elements in the whole chain, `wait` just examines summary flags that precede the current node. Because it only ever climbs the tree, the function avoids the cost of iterating over the entire chain. The function spin-waits on each preceding flag in

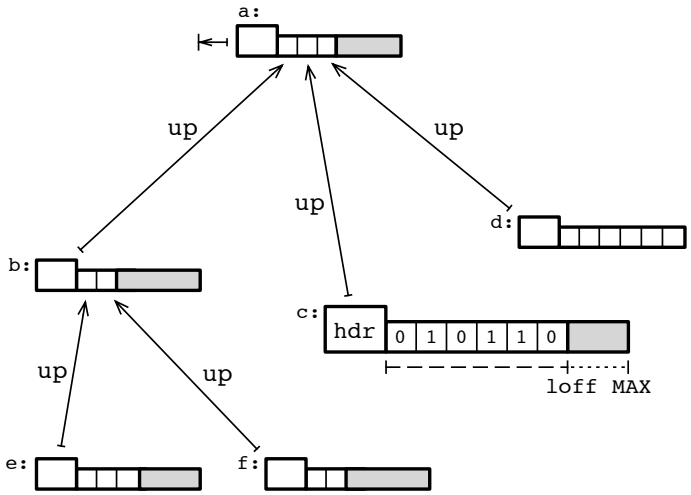
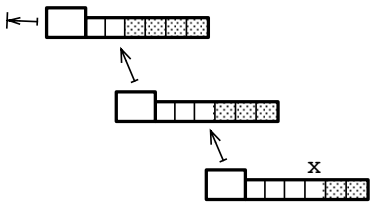


Fig. 21. Example of the summarising channel structure.

the current array (Fig. 20, lines 60–64 – note that increasing the index goes 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. The following diagram shows the nodes accessed when traversing up from `x`:



Extending the chain to insert a new link in a channel is a subtle process – array elements begin as leaves, but may become summary nodes. Thus `extend` has two cases depending 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 existing and new channels are moved to the start of this array (lines 25–31).

7.2. Proof Strategy

At the abstract level, the behaviour of the implementation is unchanged from §6. Channels are arranged into a logical chain, with later channels waiting on earlier ones. Before acquiring resources using `wait`, all the preceding flags in the chain must be set using `signal`. The added summarization mechanism just changes the flags that must be examined to determine all flags in an interval are set. As a result, we can reuse much of the reasoning from §6.

Abstract state and basic predicates. The abstract state of a chain is once again an element of CNode^+, identical to the one used in the previous section, except a 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 in the implementation). To represent the contents of the heap, we define a *heap map* d, consisting of a finite, partial function from physical addresses:

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

A heap map records the `chan_hdr`-typed objects forming the tree. To simplify the representation, we flatten the `chan_adr`-typed field `up` into the two fields *uhdr* and *uoff*. A location $\langle x, i\rangle$ is in d (written $\langle x, i\rangle \in d$) if $x \in \mathrm{dom}(d) \wedge i \leq d.\mathit{loff}$.

For a given map d, $\mathsf{chainds}$ defines the corresponding data-structure. In the following definition, we write $x \rightarrowtail v$ to indicate that x is immutable – shorthand for $\exists f.\, x \overset{f}{\mapsto} v$.

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

In order to ensure that rs and d are well-formed and correspond correctly, we require several auxiliary notions:

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

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

$$\begin{aligned}\langle x,i\rangle <_d \langle y,j\rangle &\triangleq \exists z, ix, iy.\, \mathsf{descend}_d(\langle z, ix\rangle, \langle x,i\rangle) \wedge \mathsf{descend}_d(\langle z, iy\rangle, \langle y,j\rangle) \wedge ix > iy \\ a <_d^{\mathbb{1}} b &\triangleq a <_d b \vee \mathsf{descend}_d(a,b) \vee \mathsf{descend}_d(b,a) \\ \mathsf{finalleaf}_d(x,y) &\triangleq \mathsf{descend}_d(x,y) \wedge \mathsf{leaf}_d(y) \wedge (\forall z.\, \mathsf{descend}_d(x,z) \wedge \mathsf{leaf}_d(z) \Rightarrow z <_d^{\mathbb{1}} y)\end{aligned}$$

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

Well-formedness ensures that rs and d are independently well-formed, and that they are correctly tied together into an inverted tree structure.

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

The address r is the location of the common tree root. The function τ records the distance from the current node to the tree root: this ensures that all paths up through the tree are finite. The first clause 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 summarises all descendants. The final clause guarantees the existence of non-NULL parents to a node, and enforces the distance function τ.

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

The region state $\mathsf{Chain}(rs, d)$ now takes as its argument both an abstract chain rs and a heap map d. Note that the $\mathsf{chainds}$ predicate now takes only a heap map – this captures all the structure it requires.

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

Because of this, the predicate unused takes a heap map d as an argument in addition to a chain rs. uS, the set of unused $[\mathsf{send}(x)]$ permissions, remains as in the previous section.

Well-formedness ensures several properties of $<_d$. However, note that $<_d$ is not truly an order, as it is only transitive over leaf nodes.

LEMMA 7.1. $\mathsf{wf}(d) \Rightarrow \forall a, b \in d.\ (a <_d^{\mathbb{1}} b \vee b < a) \wedge \neg(a <_d b <_d^{\mathbb{1}} a) \wedge$
$(a <_d^{\mathbb{1}} b \wedge \mathsf{leaf}_d(b) \implies a <_d b \vee \mathsf{descend}_d(a, b))$

PROOF. The first property follows from the existance of a single root node. For the second, as $a <_d b$, there must exist a common array where the ancestors of a and b are ordered. This contradicts both $\mathsf{descend}_d(a, b)$ and $\mathsf{descend}_d(b, a)$, and irreflexivity precludes $b <_d a$. For the third, either $a <_d b$, $\mathsf{descend}_d(b, a)$ or $\mathsf{descend}_d(a, b)$. All three cases trivially satisfy the conclusion. □

Predicate definitions. As noted above, chain extension may result in calls to `wait` targeting summary nodes – before extension, these nodes would have been leaves. This is sound because these nodes will be set as part of summarization. For a given leaf l, the node that `wait` will be watching is the maximal node x such that $\mathsf{finalleaf}(x, l)$ holds. Intuitively, this is because new arrays created through extension use the first element in the new array to stand for the extended chain element. The set $\mathsf{C}(rs, d)$ represents these maximal nodes; a permission is missing from the set of unused change permissions, $\mathsf{uC}(rs, d)$, only if it targets one of these nodes.

$$\begin{aligned}\mathsf{C}(rs, d) &\triangleq \{\langle y, i\rangle \mid r \in rs \wedge \mathsf{finalleaf}_d(\langle y, i\rangle, r.loc) \wedge \neg\mathsf{finalleaf}_d(\langle d(y).\mathit{uhdr}, d(y).\mathit{uoff}\rangle, r.loc)\} \\ \mathsf{uC}(rs, d) &\triangleq \{(x, r) \mid r \in rs(\mathsf{finalleaf}_d(x)).\mathcal{I} \wedge x \in \mathsf{C}(rs, d) \wedge \neg\exists y.\, r \in rs(y).\mathcal{W}\} \\ \mathsf{unused}(r, rs, d) &\triangleq (\circledast x \notin \mathsf{uS}(rs).\, [\mathsf{send}(x)]_1^r) * (\circledast (x, r') \notin \mathsf{uC}(rs, d).\, [\mathsf{change}(x, r')]_1^r)\end{aligned}$$

We can now define the send and recv predicates (we also use x to refer to the pair $\langle x.\mathtt{hdr}, x.\mathtt{off}\rangle$ when x is a `chan_addr` struct):

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

$$
\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 d' = d \blacktriangleleft_h \blacktriangleleft_{\mathit{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 a.\mathit{loc} = x \wedge \\
&\quad \exists y.\, b = a[\mathit{loc} \mapsto y] \wedge c.\mathit{flg} = 0 \wedge c.\mathcal{W} = \emptyset \wedge \langle h, o\rangle = a.\mathit{loc} \wedge \\
&\quad \left(\begin{pmatrix} 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{pmatrix} \right. \\
&\quad \left. \vee \begin{pmatrix} 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{pmatrix} \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(x,(rs,d),(rs',d')) &\triangleq d = d' \wedge \mathsf{sat}_c(\mathsf{finalleaf}_d(x), 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') \\
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(x,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_{\mathit{flags}[o]} (1) \\ \wedge\, \neg \mathsf{leaf}_d(\langle h, o\rangle)) \end{array} \right\}
\end{aligned}
$$

Fig. 22. Definition of the transition relation T_{s} for the summarising implementation.

We define a new transition map T_s to capture changes to the heap map. The definition is given in Figure 22. Most of the transitions are inherited from T_c, the transition relation for the chained implementation. (We modify the type of T_c slightly by assuming it is defined on address-offset pairs, rather than just addresses). However, `set` and `extend` correspond to changes to the underlying heap, and they thus have to be extensively altered. Furthermore, `signal` can mark summary bits that are not leaves in the tree; this is allowed by the transition mark. We use $[\mathsf{mark}]^r_-$ as notation for $\exists \pi.\ [\mathsf{mark}]^r_\pi$ to represent non-exclusive ownership of the mark action. Finally, note finalleaf in the definition of change: this is needed because a leaf x may be converted by `extend` into a summary node representing $\mathsf{finalleaf}_d(x)$.

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

LEMMA 7.2. $\mathsf{recv}(x, P)$ *is stable.*

PROOF. Assume the inital 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 simple, so assume $x = x'$. We now need to show $\mathsf{wf}(rs', d') \wedge r_2 \in \mathsf{rs}'(\mathsf{finalleaf}_{d'}(x)).\mathcal{I}$.

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, $\mathsf{finalleaf}_d(x) = a.\mathit{loc}$ 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 $\mathsf{finalleaf}_{d'}(x) = b.\mathit{loc}$. In the former case, ext adds a new array which must also descend from x. If $\mathsf{finalleaf}_{d'}(x) \neq b.\mathit{loc}$, there must be leaf z such that $b.\mathit{loc} <_d z$, but the only new leaf c is at the next index in the new array, meaning $c.\mathit{loc} <_d b.\mathit{loc}$. Thus the result follows by contradiction. □

7.3. Verifying `wait`, `signal`, `extend`

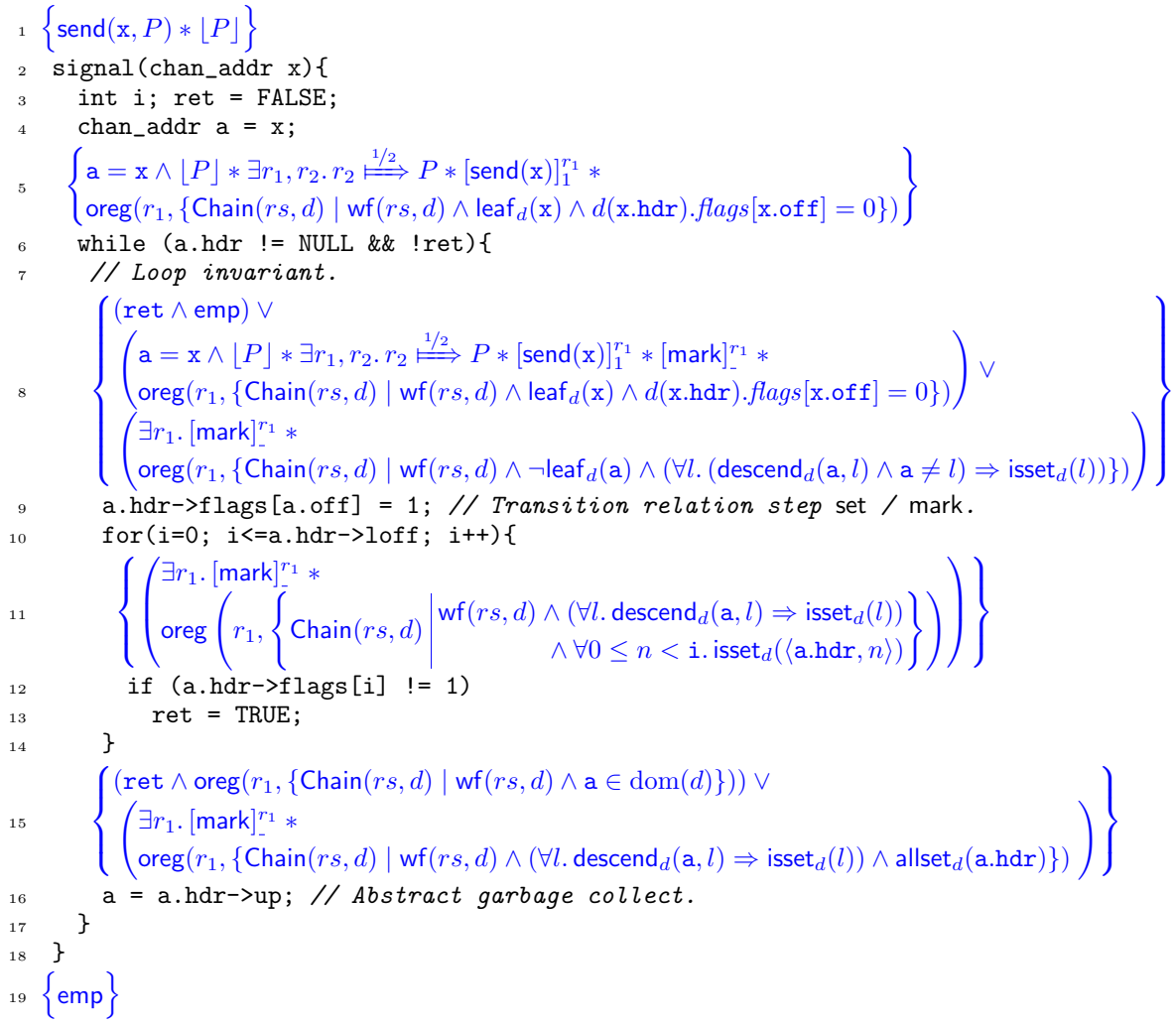


Fig. 23. Sketch-proof of `signal` with summarization.

Proving `signal`. A sketch-proof for `signal` is given in Fig. 23. 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 non-summarising implementation (§6.2) 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, so doing this does not change the abstract representation. If a flag is discovered which 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 7.3.

$$\mathsf{set}_s(x,(rs,d),(rs',d')) \wedge \mathsf{unused}(r,rs,d) * [\mathsf{send}(x)]_1^\tau \implies \mathsf{unused}(r,rs',d')$$
$$((rs,d),(rs',d')) \in T_s(\mathsf{mark}) \wedge \mathsf{unused}(r,rs,d) \implies \mathsf{unused}(r,rs',d')$$

Proving `wait`. A sketch-proof for `wait` is given in Fig. 24 / 25. 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 $\mathsf{last}_d(\mathsf{a})$ to stand for the last address in the array associated with a, i.e. $\langle \mathsf{a.hdr}, d(\mathsf{a.hdr}).\mathit{loff} \rangle$.

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 somewhere inside the tree. Once this passes, by the second clause of well-formedness (page 39) 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 sub-lemmas. The first states that once the algorithm reaches the root of the tree, it has examined all the addresses greater than the starting address.

LEMMA 7.4. $\mathsf{wf}(d) \wedge d(a).\mathit{loff} = o \wedge d(a).\mathit{uhdr} = \mathsf{NULL} \implies \neg\exists x.\, x <_d \langle a, o \rangle$

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.

LEMMA 7.5.
$$\mathsf{wf}(rs,d) \wedge z \in \mathsf{dom}(rs) \wedge \mathsf{descend}_d(x,z) \wedge \mathsf{leaf}_d(z) \wedge (\forall l.\, l <_d^{\Uparrow} x \wedge \mathsf{leaf}_d(l) \Rightarrow \mathsf{isset}_d(l)) \implies \exists rs_1, rs_2.\, rs = rs_1 \cdot rs(z) \cdot rs_2 \wedge \mathsf{ctrue}(rs(z) \cdot rs_2)$$

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 $\mathit{dom}(rs(z) \cdot rs_2)$ and suppose that $rs(y).\mathit{flg}$ is not set. By well-formedness, it must be true that $y <_d^{\Uparrow} z$. Now we show that $y <_d^{\Uparrow} x$. 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. □

Fig. 24. Sketch-proof of `wait` with summarization (completed in Fig. 25).

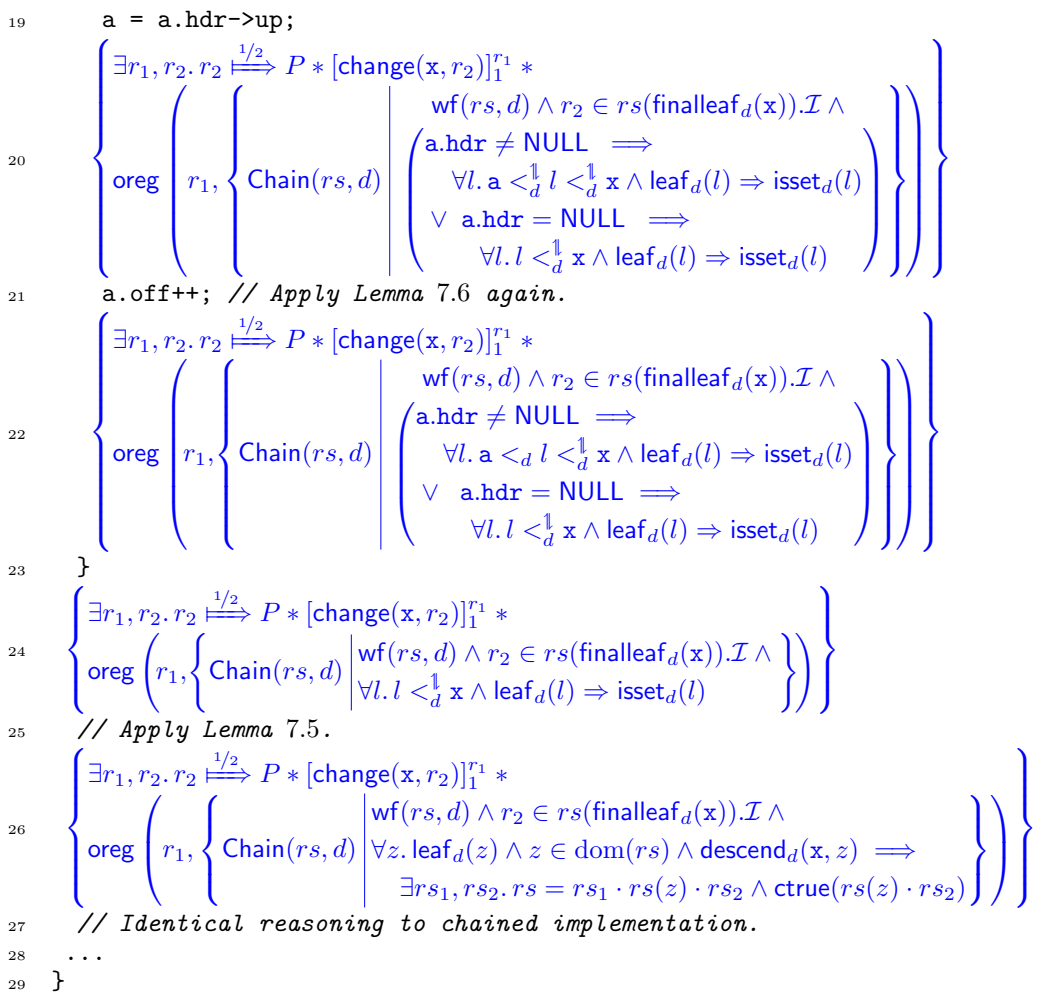


Fig. 25. Sketch-proof of `wait` with summarization (continued from Fig. 24).

The final lemma shows that shifting left from the current maximal node reaches a node earlier in the order.

LEMMA 7.6. $0 \leq d(a).off < d(a).loff \wedge \mathsf{wf}(d) \wedge a \blacktriangleleft_{off} (\bullet{+}1) <_d k <_d b \implies a <_d^{\Downarrow} k <_d b$

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 Fig. 26. 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 pair of lemmas to show that the unused predicate representing unused permissions is preserved by extending the chain.

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(chan_addr x){
3    // Frame off order predicates and unfold send.
```

4 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{send}(\mathtt{x})]_1^{r_1} * [\mathsf{mark}]_-^{r_1} * \\ \mathsf{oreg}(r_1, \{\mathsf{Chain}(rs, d) \mid \mathsf{wf}(rs, d) \wedge \mathsf{leaf}_d(\mathtt{x}) \wedge rs(\mathtt{x}).\mathit{flg} = 0\})\end{array}\right\}$

```
5    chan_addr nx,r;
6    if(x.off == x.hdr->loff && x.off < MAX){
```

7 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longrightarrow} P * [\mathsf{mark}]_-^{r_1} * [\mathsf{send}(\mathtt{x})]_1^{r_1} * r' \overset{1}{\Longrightarrow} Q * r'' \overset{1}{\Longrightarrow} Q * \\ \mathsf{oreg}\left(r_1, \left\{\mathsf{Chain}(rs, d) \,\middle|\, \begin{array}{l}\mathsf{wf}(rs, d) \wedge \mathsf{leaf}_d(\mathtt{x}) \wedge rs(\mathtt{x}).\mathit{flg} = 0 \\ \wedge\, d(\mathtt{x.hdr}).\mathit{loff} = \mathtt{x.off} \wedge \mathtt{x.off} < \mathsf{MAX}\end{array}\right\}\right)\end{array}\right\}$

```
8      x.hdr->flags[x.loff+1] = 0;
9      x.hdr->loff++; // Transition relation step ext.
```

10 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longrightarrow} P * r' \overset{1/2}{\Longrightarrow} Q * r'' \overset{1/2}{\Longrightarrow} Q * \\ [\mathsf{mark}]_-^{r_1} * [\mathsf{send}(\mathtt{x})]_1^{r_1} * [\mathsf{send}(\langle \mathtt{x.hdr}, \mathtt{x.off}+1\rangle)]_1^{r_1} * [\mathsf{change}(\langle \mathtt{x.hdr}, \mathtt{x.off}+1\rangle, r'')]_1^{r_1} * \\ \mathsf{oreg}\left(r_1, \left\{\mathsf{Chain}(rs, d) \,\middle|\, \begin{array}{l}rs = rs_1 \cdot rs(\mathtt{x}) \cdot (\langle \mathtt{x.hdr}, \mathtt{x.off}+1\rangle, r', \{r''\}, 0, \emptyset) \cdot rs_2 \wedge \\ \mathsf{wf}(rs, d) \wedge \mathsf{leaf}_d(\mathtt{x}) \wedge rs(\mathtt{x}).\mathit{flg} = 0 \wedge \mathsf{leaf}_d(\langle \mathtt{x.hdr}, \mathtt{x.off}+1\rangle) \\ \wedge\, d(\mathtt{x.hdr}).\mathit{loff} = \mathtt{x.off}+1 \wedge \mathtt{x.off} < \mathsf{MAX}\end{array}\right\}\right)\end{array}\right\}$

```
11     r.hdr = x.hdr; r.off = x.off + 1;
12     nx = x;
```

13 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longrightarrow} P * r' \overset{1/2}{\Longrightarrow} Q * r'' \overset{1/2}{\Longrightarrow} Q * \\ [\mathsf{mark}]_-^{r_1} * [\mathsf{send}(\mathtt{nx})]_1^{r_1} * [\mathsf{send}(\mathtt{r})]_1^{r_1} * [\mathsf{change}(\mathtt{r}, r'')]_1^{r_1} * \\ \mathsf{oreg}\left(r_1, \left\{\mathsf{Chain}(rs, d) \,\middle|\, \begin{array}{l}rs = rs_1 \cdot rs(\mathtt{nx}) \cdot (\mathtt{r}, r', \{r''\}, 0, \emptyset) \cdot rs_2 \wedge \mathsf{wf}(rs, d) \wedge \\ \mathsf{leaf}_d(\mathtt{nx}) \wedge rs(\mathtt{nx}).\mathit{flg} = 0 \wedge \mathsf{leaf}_d(\mathtt{r}) \wedge rs(\mathtt{r}).\mathit{flg} = 0\end{array}\right\}\right)\end{array}\right\}$

```
14   } else {
```

15 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longrightarrow} P * r' \overset{1}{\Longrightarrow} Q * r'' \overset{1}{\Longrightarrow} Q * [\mathsf{mark}]_-^{r_1} * [\mathsf{send}(\mathtt{x})]_1^{r_1} * \\ \mathsf{oreg}\,(r_1, \{\mathsf{Chain}(rs, d) \mid \mathsf{wf}(rs, d) \wedge \mathsf{leaf}_d(\mathtt{x}) \wedge rs(\mathtt{x}).\mathit{flg} = 0\})\end{array}\right\}$

```
16     nh = malloc(chan_hdr);
17     nh->up = x; nh->loff = 1;
18     // Transition relation step ext.
```

19 $\left\{\begin{array}{l}\exists r_1, r_2.\, r_2 \overset{1/2}{\Longrightarrow} P * r' \overset{1/2}{\Longrightarrow} Q * r'' \overset{1/2}{\Longrightarrow} Q * \\ [\mathsf{mark}]_-^{r_1} * [\mathsf{send}(\langle \mathtt{nh}, 0\rangle)]_1^{r_1} * [\mathsf{send}(\langle \mathtt{nh}, 1\rangle)]_1^{r_1} * [\mathsf{change}(\langle \mathtt{nh}, 1\rangle, r'')]_1^{r_1} * \\ \mathsf{oreg}\left(r_1, \left\{\mathsf{Chain}(rs, d) \,\middle|\, \begin{array}{l}rs = rs_1 \cdot (rs(\mathtt{x}) \blacktriangleleft_{loc} \langle \mathtt{nh}, 0\rangle) \cdot (\langle \mathtt{nh}, 1\rangle, r', \{r''\}, 0, \emptyset) \cdot rs_2 \wedge \\ \mathsf{wf}(rs, d) \wedge \mathsf{leaf}_d(\mathtt{x}) \wedge rs(\mathtt{x}).\mathit{flg} = 0 \wedge \mathsf{leaf}_d(\langle \mathtt{nh}, 0\rangle) \wedge \mathsf{leaf}_d(\langle \mathtt{nh}, 1\rangle) \\ \wedge\, rs(\langle \mathtt{nh}, 0\rangle).\mathit{flg} = 0 \wedge rs(\langle \mathtt{nh}, 1\rangle).\mathit{flg} = 0\end{array}\right\}\right)\end{array}\right\}$

```
20     nx.hdr = nh; nx.off = 0;
21     r.hdr = nh; r.off = 1;
22   }
23   return (r,nx);
24 }
```

25 $\left\{\mathsf{send}(\mathtt{r}, Q) * \mathsf{recv}(\mathtt{r}, Q) * \mathsf{send}(\mathtt{nx}, P) * \mathtt{r} \prec \mathtt{nx} * \circledast_{e\in E}\, e \prec \mathtt{r} * \circledast_{l\in L}\, \mathtt{nx} \prec l\right\}$

Fig. 26. Sketch-proof of `extend` with summarization.

LEMMA 7.7.

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

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

$$\mathsf{unused}(r,rs,d)\implies[\mathsf{send}(\langle x,i+1\rangle)]_1^r*[\mathsf{change}(\langle x,i+1\rangle,r')]_1^r*\mathsf{true}$$

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 7.8.

$$\begin{pmatrix}\mathsf{ext}_s(\langle x,i\rangle,(rs,d),(rs',d'))\wedge \mathrm{dom}(d')=\mathrm{dom}(d)\uplus\{l\}\wedge r'\notin rs\wedge\\ rs'(\langle l,1\rangle).\mathcal{I}=\{r'\}\wedge \mathsf{wf}(rs,d)\wedge\mathsf{wf}(rs',d')\wedge\mathsf{unused}(r,rs,d)*[\mathsf{send}(\langle x,i\rangle)]_1^r\end{pmatrix}\implies$$
$$\mathsf{unused}(r,rs',d')*[\mathsf{send}(\langle l,0\rangle)]_1^r*[\mathsf{send}(\langle l,1\rangle)]_1^r*[\mathsf{change}(\langle l,1\rangle,r')]_1^r$$

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 $[\mathsf{send}(\langle l,0\rangle)]_1^r*[\mathsf{send}(\langle l,1\rangle)]_1^r$ 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, $[\mathsf{change}(\langle l,1\rangle,r')]_1^r$ can be removed from the unused. □

LEMMA 7.9. $\mathsf{ext}(x,(rs,d),(rs',d'))\wedge\mathsf{wf}(rs,d)\wedge\mathsf{wf}(rs',d')\implies\mathsf{uC}(rs,d)\subseteq\mathsf{uC}(rs',d')$

PROOF. There are two cases for extension: in-place extension in the array, or 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 $(x,i)\in\mathsf{uC}(rs,d)$. 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 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 §6.3. The main difference is the new definition of unused. The renunciation case is straightforward, but we need to show that we can pull the appropriate change permissions out of the region. This is captured by the following lemma:

LEMMA 7.10.

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

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 paper 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 summarising implementation (§7) is entirely new. This paper also fixes a subtle logical error which 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 require 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 paper, 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.

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 paper.

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 paper in identifying and checking entailment steps used in program proofs.

REFERENCES

Christian J. Bell, Andrew Appel, and David Walker. 2009. Concurrent Separation Logic for Pipelined Parallelization. In *SAS*.

Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2010. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*.

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science* 8, 4 (2012).

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *OOPSLA '09*. ACM, 97–116.

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *POPL*. 259–270.

M. Botinčan, M. Dodds, and S. Jagannathan. 2013. Resource-Sensitive Synchronization Inference by Abduction. In *ACM Trans. on Programming Languages and Systems*.

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*. To appear.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP*.

Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *ESOP*.

M. Dodds, S. Jagannathan, and M. J. Parkinson. 2011. Modular Reasoning for Deterministic Parallelism. In *POPL*.

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *ESOP*.

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *APLAS*.

Christian Haack, Marieke Huisman, and Clément Hurlin. 2008. Reasoning about Java's Reentrant Locks. In *APLAS*. 171–187.

C. A. R. Hoare and Peter W. O'Hearn. 2008. Separation Logic Semantics for Communicating Processes. *ENTCS* 212 (2008), 3–25.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP*.

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

Cliff B. Jones. 1983. Tentative steps toward a development method for interfering programs. *TOPLAS* 5, 4 (1983), 596–619.

Neel R. Krishnaswami, Lars Birkedal, and Jonathan 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. Navabi, X. Zhang, and S. Jagannathan. 2008. Quasi-static Scheduling for Safe Futures. In *PPoPP*. ACM, 23–32.

Peter W. O'Hearn. 2007. Resources, Concurrency and Local Reasoning. *TCS* (2007).

M. J. Parkinson and G. M. Bierman. 2005. Separation logic and abstraction. In *POPL*. 247–258.

Martin C. Rinard and Monica S. Lam. 1992. Semantic Foundations of Jade. In *POPL*. ACM, 105–118.

Kasper Svendsen and Lars Birkedal. 2014a. Impredicative Concurrent Abstract Predicates. In *ESOP*.

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

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Joins: A Case Study in Modular Specification of a Concurrent Reentrant Higher-Order Library. In *ECOOP*.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*.

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

Viktor Vafeiadis and Matthew J Parkinson. 2007. A marriage of Rely/Guarantee and Separation Logic. In *CONCUR*. 256–271.

Jules Villard, Étienne Lozes, and Cristiano Calcagno. 2010. Tracking Heaps That Hop with Heap-Hop. In *TACAS*. 275–279.

A. Welc, S. Jagannathan, and A. Hosking. 2005. Safe Futures for Java. In *OOPSLA*. 439–435.

John Wickerson, Mike Dodds, and Matthew Parkinson. 2010. Explicit Stabilisation for Modular Rely-Guarantee Reasoning. In *ESOP*.

A. SAVED PROPOSITIONS

A saved proposition $r \overset{\pi}{\Longmapsto} P$ 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 state 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 $r \overset{\pi}{\Longmapsto} Q$ as follows:

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

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).

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

□

For unification inside separating implication we reason as follows:

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

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

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

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

□

B. PROOFS FOR OUT-OF-ORDER SIGNALLING

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

LEMMA 6.1.

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

PROOF.

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

□

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.

$$
\begin{array}{ll}
& r \overset{1/2}{\Longmapsto} R * \lfloor R \rfloor * \mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r\}) \\
& \quad \textit{Definition of } \mathsf{resource}. \\
\sqsubseteq & \quad r \overset{1/2}{\Longmapsto} R * \lfloor R \rfloor * \\
& \quad \left(\begin{array}{c} \exists Q \colon \mathcal{I} \to \mathsf{Prop},\ R \colon \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) \\
& \quad \textit{Property (3), monotonicity of } \triangleright, \textit{ monotonicity of } \lfloor - \rfloor. \\
\sqsubseteq & \quad \exists P \colon \mathsf{Prop}.\, r \overset{1/2}{\Longmapsto} R * \lfloor \triangleright R \rfloor * (\lfloor \triangleright R \rfloor \Rightarrow \lfloor \triangleright P \rfloor) \\
& \quad \left(\begin{array}{c} \exists Q \colon \mathcal{I} \to \mathsf{Prop},\ R \colon \mathcal{W} \to \mathsf{Prop}. \\ r \overset{1/2}{\Longmapsto} P * \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 P * \triangleright \circledast_{w \in \mathcal{W}}.\, R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}}.\, \lceil Q(i) \rceil \rfloor \end{array} \right) \\
& \quad \textit{Modus ponens.} \\
\sqsubseteq & \quad \exists P \colon \mathsf{Prop}.\, r \overset{1/2}{\Longmapsto} R * \lfloor \triangleright P \rfloor * \\
& \quad \left(\begin{array}{c} \exists Q \colon \mathcal{I} \to \mathsf{Prop},\ R \colon \mathcal{W} \to \mathsf{Prop}. \\ r \overset{1/2}{\Longmapsto} P * \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 P * \triangleright \circledast_{w \in \mathcal{W}}.\, R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}}.\, \lceil Q(i) \rceil \rfloor \end{array} \right) \\
& \quad \textit{Combine } \lfloor - \rfloor, \textit{ modus ponens for } \mathrel{-\!\!*}, \textit{ garbage collect.} \\
\sqsubseteq & \quad \begin{array}{c} \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 \end{array} \\
\sqsubseteq & \quad \mathsf{resource}(\mathcal{I}, \mathcal{W})
\end{array}
$$

□

LEMMA 6.3. $\mathsf{chainres}(rs) \land \mathsf{wf}(rs) \land \mathsf{ctrue}(rs)$
$\quad \sqsubseteq \quad \exists rs'.\, \mathsf{chainres}(rs') \land \mathsf{cconf}(rs') \land rs \overset{\mathcal{W}_*}{\to} rs' \land \mathsf{wf}(rs')$

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 which 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:

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

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:

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

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 $\mathsf{resource}$, there exists a saved proposition $w \overset{1/2}{\Longmapsto} R$ inside $\mathsf{resource}(s.\mathcal{I}, s.\mathcal{W})$. By the same definition, the saved proposition $w \overset{1/2}{\Longmapsto} P$ 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 garbage collect the saved proposition:

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

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

$$rs[n-i, n] \overset{\mathcal{W}_*}{\to} s \cdot rs' \overset{\mathcal{W}}{\to} s[\mathcal{W} \mapsto s.\mathcal{W} \setminus \{w\}] \cdot rs''$$

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:

$$\sqsubseteq \quad \exists rs', s.\, \mathsf{chainres}(rs[0, n-(i+1)]) * \mathsf{resource}(s.\mathcal{I}, s.\mathcal{W} \setminus \{w\}) * \mathsf{chainres}(rs')$$
$$\wedge\, \mathsf{cconf}(rs') \wedge rs[n-i, n] \overset{\mathcal{W}_*}{\to} s \cdot rs' \wedge \mathsf{wf}(rs[0, n-(i+1))] \cdot s \cdot rs')$$

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.

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

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

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

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,

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

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,

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

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

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

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

PROOF.

$\mathsf{resource}(\mathcal{I} \uplus r_2, \emptyset) * r_2 \overset{1/2}{\Longrightarrow} P$

Lemma 6.1.

$\sqsubseteq \quad r_2 \overset{1/2}{\Longrightarrow} P * \circledast_{i \in \mathcal{I} \uplus r_2}. \exists R.\, i \overset{1/2}{\Longrightarrow} R * \lceil \triangleright R \rceil$

Pull out resource for identifier r_2.

$\sqsubseteq \quad r_2 \overset{1/2}{\Longrightarrow} P * \exists R'.\, r_2 \overset{1/2}{\Longrightarrow} R' * \lceil \triangleright R' \rceil * \circledast_{i \in \mathcal{I}}. \exists R.\, i \overset{1/2}{\Longrightarrow} R * \lceil \triangleright R \rceil$

Property (3), monotonicity of $\lceil - \rceil$.

$\sqsubseteq \quad r_2 \overset{1/2}{\Longrightarrow} P * \exists R'.\, (\lceil \triangleright R' \rceil \Rightarrow \lceil \triangleright P \rceil) * r_2 \overset{1/2}{\Longrightarrow} R' * \lceil \triangleright R' \rceil * \circledast_{i \in \mathcal{I}}. \exists R.\, i \overset{1/2}{\Longrightarrow} R * \lceil \triangleright R \rceil$

Modus ponens, garbage collect.

$\sqsubseteq \quad \lceil \triangleright P \rceil * \circledast_{i \in \mathcal{I}}. \exists R.\, i \overset{1/2}{\Longrightarrow} R * \lceil \triangleright R \rceil$

Definition of resource*.*

$\sqsubseteq \quad \lceil \triangleright P \rceil * \mathsf{resource}(\mathcal{I}, \emptyset)$

□

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 post-condition is false, allowing us to prove any post-condition. □

LEMMA 6.8. $\mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r\}) * r \overset{1/2}{\Longrightarrow} S * r' \overset{1/2}{\Longrightarrow} T_1$

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

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

$$\begin{aligned}&\mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r\}) * r \overset{1/2}{\Longrightarrow} S * r' \overset{1/2}{\Longrightarrow} T_1 * r'' \overset{1/2}{\Longrightarrow} T_2 \wedge \mathsf{valid}(T_1 * T_2 \Rightarrow S)\\ \sqsubseteq\ &\mathsf{resource}(\mathcal{I}, \mathcal{W} \uplus \{r', r''\})\end{aligned}$$

$$\begin{aligned}&r \overset{1/2}{\Longrightarrow} S * r' \overset{1/2}{\Longrightarrow} T_1 * r'' \overset{1/2}{\Longrightarrow} T_2 \wedge \mathsf{valid}(T_1 * T_2 \Rightarrow S) *\\ &\left(\begin{array}{c}\exists Q\colon \mathcal{I} \to \mathsf{Prop},\ R\colon \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{Rearrange}\\ \sqsubseteq\ &\exists P\colon \mathsf{Prop},\ Q\colon \mathcal{I} \to \mathsf{Prop},\ R\colon \mathcal{W} \uplus \{r', r''\} \to \mathsf{Prop}.\\ &\qquad \mathsf{valid}(R(r') * R(r'') \Rightarrow S) \wedge r \overset{1/2}{\Longrightarrow} S * r \overset{1/2}{\Longrightarrow} P\\ &\qquad * \circledast_{i \in \mathcal{I}}.\, i \overset{1/2}{\Longrightarrow} Q(i) * \circledast_{w \in \mathcal{W} \uplus \{r', r''\}}.\, w \overset{1/2}{\Longrightarrow} R(w)\\ &\qquad\quad * \lfloor \triangleright P * \triangleright \circledast_{w \in \mathcal{W}}.\, R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}}.\, \lceil Q(i) \rceil \rfloor\\ &\quad \textit{Property (3), SMono and distributing } \triangleright \textit{ over } \Longrightarrow.\\ \sqsubseteq\ &\exists P\colon \mathsf{Prop},\ Q\colon \mathcal{I} \to \mathsf{Prop},\ R\colon \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}{\Longrightarrow} Q(i) * \circledast_{w \in \mathcal{W} \uplus \{r', r''\}}.\, w \overset{1/2}{\Longrightarrow} R(w)\\ &\qquad\qquad * \lfloor (\triangleright P * \triangleright \circledast_{w \in \mathcal{W}}.\, R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}}.\, \lceil Q(i) \rceil \rfloor\\ &\quad \textit{By mono of } \lfloor \_ \rfloor, \mathrel{-\!\!*} \textit{ and } *.\\ \sqsubseteq\ &\exists Q\colon \mathcal{I} \to \mathsf{Prop},\ R\colon \mathcal{W} \uplus \{r', r''\} \to \mathsf{Prop}.\\ &\qquad \circledast_{i \in \mathcal{I}}.\, i \overset{1/2}{\Longrightarrow} Q(i) * \circledast_{w \in \mathcal{W} \uplus \{r', r''\}}.\, w \overset{1/2}{\Longrightarrow} R(w)\\ &\quad * \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\\ &\quad \textit{Rearrange}\\ \sqsubseteq\ &\exists P\colon \mathsf{Prop},\ Q\colon \mathcal{I} \to \mathsf{Prop},\ R\colon \mathcal{W} \uplus \{r', r''\} \to \mathsf{Prop}.\\ &\qquad \circledast_{i \in \mathcal{I}}.\, i \overset{1/2}{\Longrightarrow} Q(i) * \circledast_{w \in \mathcal{W} \uplus \{r', r''\}}.\, w \overset{1/2}{\Longrightarrow} R(w)\\ &\qquad\quad * \lfloor \triangleright \circledast_{w \in \mathcal{W} \uplus \{r', r''\}}.\, R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I}}.\, \lceil Q(i) \rceil \rfloor\end{aligned}$$

□

LEMMA B.2.

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

PROOF.

$$
\begin{aligned}
& 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}) \\
& \quad \textit{Definition of } \mathsf{resource}. \\
\sqsubseteq \;& 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 \\
& \qquad \left(\begin{array}{c} \exists Q \colon \mathcal{I} \uplus \{r_2\} \to \mathsf{Prop},\ R \colon \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) \\ * \left(\lfloor (\triangleright \circledast_{w \in \mathcal{W}}. R(w)) \mathrel{-\!\!*} \triangleright \circledast_{i \in \mathcal{I} \uplus \{r_2\}}. \lceil Q(i) \rceil \rfloor \right) \end{array} \right) \\
& \quad \textit{Pull out } r_2, \triangleright \textit{ mono w.r.t. } -\!\!*, \textit{ property } (4). \\
\sqsubseteq \;& 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 \colon \mathcal{I} \uplus \{r_2\} \to \mathsf{Prop},\ R \colon \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) \\
& \quad \textit{Fold } r_3, r_4 \textit{ into } \mathcal{I}, \textit{ weaken with } \lceil \_ \rceil, \textit{ garbage collect.} \\
\sqsubseteq \;& \exists Q \colon \mathcal{I} \uplus r_2 \to \mathsf{Prop},\ R \colon \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) \\
& \quad \textit{Definition of } \mathsf{resource}. \\
\sqsubseteq \;& \mathsf{resource}(\mathcal{I} \uplus \{r_3, r_4\}, \mathcal{W})
\end{aligned}
$$

□

LEMMA 6.9. $r_2 \in rs(x).\mathcal{I} \wedge r_2 \overset{1/2}{\Longmapsto} P * \lfloor \lceil P \rceil \mathrel{-\!\!*} (P_1 * P_2) \rfloor * \mathsf{chainres}(rs)$

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

Proof.

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

Make saved propositions, fresh by construction.

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

Pull out resource *predicate for* x.

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

Apply Lemma B.2.

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

Definition of chainres.

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

□