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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© YYYY ACM 0000-0000/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

ACM Journal Name, Vol. V, No. N, Article A, Publication date: January YYYY.
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 an 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 \texttt{f} in sequence (here \texttt{sleep(rand())} waits for an unknown period of time).

```c
void f(int *x, int *y, int v) {
    if(*x < 10) {
        *y = *y + v;
        *x = *x + v;
        sleep(rand());
        f(x, y, 5);
    } else {
        sleep(rand());
    }
}
```

When this program terminates, location \texttt{x} and \texttt{y} will both hold 16.

The second call to \texttt{f} will wait for the first call to finish its arbitrarily long \texttt{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 \texttt{f1} and \texttt{f2} are given below. We run both concurrently, but require that \texttt{f1} passes control of \texttt{x} and \texttt{y} to \texttt{f2} before sleeping, allowing \texttt{f2} to continue executing.

**Function definitions:**

```c
f1(x, y, v, i) {
    if(*x < 10) {
        *y = *y + v;
        *x = *x + v;
        signal(i);
        sleep(rand());
        f1(x, y, 5, i) || f2(x, y, 11, i);
    } else {
        signal(i);
        sleep(rand());
    }
}
```

**Program body:**

```c
f2(x, y, v, i) {
    *x = 0; *y = 0;
    if(*x < 10) {
        wait(i);
        *y = *y + v;
        *x = *x + v;
        f1(x, y, 5, i) || f2(x, y, 11, i);
    } else {
        sleep(rand());
    }
}
```
The barriers in $f_1$ and $f_2$ 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, $\text{newchan}$ 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 $f_1$ and $f_2$ 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 $\text{signal}$, $\text{wait}$ and $\text{newchan}$. 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 $f_1$ and $f_2$ 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 $v'$: $x \mapsto v * y \mapsto v'$. 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 $x \mapsto v * y \mapsto v'$ 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]:

$$\frac{\{P_1\} C_1 \{Q_1\} \quad \{P_2\} C_2 \{Q_2\}}{\{P_1 * P_2\} C_1 \parallel C_2 \{Q_1 * Q_2\}} \text{PAR}$$

To verify $f_1$ and $f_2$, we must encode the fact that $f_1$ gives up access to $x$ and $y$ by calling $\text{signal(i)}$, while $f_2$ retrieves access to them by calling $\text{wait(i)}$. We encode these two facts with two predicates, $\text{recv}$ and $\text{send}$, 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:

$\text{recv}(i, P)$ – By calling $\text{wait}$ on $i$, the thread will acquire a resource satisfying the assertion $P$.

$\text{send}(i, P)$ – By calling $\text{signal}$ on $i$ when holding a resource satisfying $P$, the thread will lose the resource $P$. 

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

\[
\begin{align*}
\{ \text{emp} \} & \quad i = \text{newchan}() & \{ \text{send}(i, P) \ast \text{recv}(i, P) \} \\
\{ \text{send}(i, P) \ast \lfloor P \rfloor \} & \quad \text{signal}(i) & \{ \text{emp} \} \\
\{ \text{recv}(i, P) \} & \quad \text{wait}(i) & \{ \lceil P \rceil \}
\end{align*}
\]

The specification of \text{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 \text{newchan}'s return value – i.e. the address of the new channel (we also use this notation in the specification of \text{extend}, below).

New \text{recv} and \text{send} predicates can be constructed at run-time using \text{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 \( f_1 \) and \( f_2 \). (Here we specialise to the particular parameter values of 5 / 11; it would be easy to generalise).

\[
\begin{align*}
\{ x \mapsto 0 \ast y \mapsto 0 \ast \text{send}(i, x \mapsto 5 \ast y \mapsto 5) \} & \quad f_1(x, y, 5, i) & \{ \text{emp} \} \\
\{ \text{recv}(i, x \mapsto 5 \ast y \mapsto 5) \} & \quad f_2(x, y, 11, i) & \{ x \mapsto 5 \ast y \mapsto 5 \}
\end{align*}
\]

The \text{send} predicate in the specification for \( f_1 \) says that the thread must supply the resources \( x \) and \( y \) such that they both contain the value 5. The specification for \( f_2 \) 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 \( \lfloor P \rfloor \iff P \iff \lceil P \rceil \).
\[
\begin{align*}
\{x \mapsto y \mapsto \} \\
* x = 0; * y = 0; \text{ chan } *i = \text{ newchan}(); \\
\{x \mapsto 0 \text{ } y \mapsto 0 * \text{ send}(i, x \mapsto 5 * y \mapsto 5) * \text{ recv}(i, x \mapsto 5 * y \mapsto 5)\} \\
\{x \mapsto 0 \text{ } y \mapsto 0 * \text{ send}(i, x \mapsto 5 * y \mapsto 5)\} \quad \{\text{recv}(i, x \mapsto 5 * y \mapsto 5)\} \\
f1(x, y, 5, i) \quad f2(x, y, 11, i) \\
\{\text{emp}\} \\
\{x \mapsto 16 \text{ } y \mapsto 16\} \\
\end{align*}
\]

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:

\[
\begin{align*}
* x = * y + 1; & \quad \text{wait}(i); \\
\text{signal}(i) & \quad \text{wait}(i); \\
z := * x & \quad * y := 4 \\
\end{align*}
\]

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:

\[
\{\text{recv}(a, P) * [[P] \implies (P_1 * P_2)]\} \quad \langle \text{skip} \rangle \quad \{\text{recv}(a, P_1) * \text{recv}(a, P_2)\}
\]

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 \( \implies \) (separating implication and separating conjunction have a similar relationship shared by classical implication, \( \implies \), and classical conjunction, \( \land \), i.e. they are adjoint). The assertion \( P \implies (P_1 * P_2) \) 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 \text{recv} predicate, the result of applying this property is two new \text{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, \text{skip}. This is because the property manipulates a shared higher-order resource: \text{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 \text{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 \text{wait} barrier called on a channel waits for \text{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 \text{newchan}. We introduce an operation \text{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:

\[
\begin{align*}
    x \prec y & \implies x \prec y \& x \prec y \tag{duplication} \\
    x \prec y \& y \prec z & \implies x \prec z \tag{transitivity}
\end{align*}
\]

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.

\[
\begin{align*}
    \{ \text{send}(x,P) \} & \times \{ \text{recv}(a,Q) \times \text{send}(b,P) \} \\
    \{ \times e \in E e \prec x \& \times l \in L x \prec l \} & = \{ \text{extend}(x) \}
\end{align*}
\]

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

\[
\{ \text{recv}(x,P) \times \text{send}(y,Q) \times x \prec y \} \langle \text{skip} \rangle \{ \text{send}(y,P \rightarrow Q) \}
\]

Using this specification, we can partially discharge a send using the preceding recv predicate. The assertion \( P \rightarrow Q \) stands for the resource which gives \( Q \) if combined with a resource satisfying \( P \). Thus, combining it with the predicate \( \text{recv}(x,P) \) 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.

\[
\begin{align*}
    \text{for}(j = 0; j < \text{max}; j++)\{} \\
    \text{f}(x,y,\text{vs}[j]) \\
    \text{\{} \\
\end{align*}
\]

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 \( f_1 \) and \( f_2 \), 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:

\[
\begin{align*}
\{\text{emp}\} & \quad i = \text{newchan()} \quad \{\text{recv}(i,P) \ast \text{send}(i,P)\} \\
\{\text{send}(i,P) \ast [P]\} & \quad \text{signal}(i) \quad \{\text{emp}\} \\
\{\text{recv}(i,P)\} & \quad \text{wait}(i) \quad \{[[P]]\} \\
\left\{\begin{array}{c}
\text{send}(x,P) \ast \\
\bigotimes_{e \in E} e \prec x \ast \bigotimes_{l \in L} x \prec l
\end{array} \right\} & \quad (a,b) = \text{extend}(x) \left\{\begin{array}{c}
\text{send}(a,Q) \ast \text{recv}(a,Q) \ast \text{send}(b,P) \\
* a \prec b \star \bigotimes_{e \in E} e \prec a \star \bigotimes_{l \in L} b \prec l
\end{array} \right\}
\end{align*}
\]

Axioms:

\[
\begin{align*}
x \prec y & \implies x \prec y \ast x \prec y \\
x \prec y \ast y \prec z & \implies x \prec z \\
\{\text{recv}(x,P) \ast \text{send}(y,Q) \ast x \prec y\} & \langle \text{skip} \rangle \{\text{send}(y,P \ast Q)\} \\
\{\text{recv}(a,P) \ast [[P] \ast (P_1 \ast P_2)]\} & \langle \text{skip} \rangle \{\text{recv}(a,P_1) \ast \text{recv}(a,P_2)\}
\end{align*}
\]

Fig. 2. Full abstract specification for deterministic parallelism.

Function definition:

\[
\text{fp}(x,y,v,ix,iy,ixp,iyp) \{ \\
\text{wait}(ixp); \\
\text{if} (*x < 10) \{ \\
\text{wait}(iyp); \\
\text{signal}(ixn); \\
\text{signal}(iyn); \\
\text{for(j = 0; j < max; j++)} \{ \\
\text{v} = \text{vs}[j]; \\
\text{fork( fp}(x,y,v,ixn,iyn,ixl,iyl) ); \\
\} \\
\} \text{else} \{ \\
\text{signal}(ix); \\
\text{signal}(iy); \\
\text{sleep}(\text{rand}()); \\
\} \\
\}
\]

Program body:

\[
\begin{align*}
\text{ixf} & = \text{newchan}(); \\
(ixn,ixf) & = \text{extend}(ixf); \\
\text{iyf} & = \text{newchan}(); \\
(iyn,iyf) & = \text{extend}(iyf); \\
\text{v} & = \text{vs}[j]; \\
\text{fork( fp}(x,y,v,ixn,iyn,ixl,iyl) ); \\
\}
\]

wait(ixn); \\
wait(iyn);
\]

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

$$\{ \text{recv}(ixp,x \mapsto _) \ast \text{send}(ix,x \mapsto _) \ast \text{recv}(iyp,y \mapsto _) \ast \text{send}(iy,y \mapsto _) \ast iyp \prec iy \}$$

$fp(x,y,v,ix,iy,ixp,iyp) \{$

wait(ixp);

$\{ x \mapsto _ \ast \text{send}(ix,x \mapsto _) \ast \text{recv}(iyp,y \mapsto _) \ast \text{send}(iy,y \mapsto _) \ast iyp \prec iy \}$

if ($*x < 10$) {

wait(iyp);

$\{ x \mapsto _ \ast y \mapsto _ \ast \text{send}(ix,x \mapsto _) \ast \text{send}(iy,y \mapsto _) \}$

$*y = *y + v;$

signal(iy);

$\{ x \mapsto _ \ast \text{send}(ix,x \mapsto _) \}$

$*x = *x + v;$

signal(ix);

} else {

signal(ix);

$\{ \text{recv}(iyp,y \mapsto _) \ast \text{send}(iy,y \mapsto _) \ast iyp \prec iy \}$

signal(iy);

$\{ \text{emp} \}$

sleep(rand());

}$

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 $y \mapsto _$; 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 $\text{send}(ixf,\text{true})$ and $\text{send}(iyf,\text{true})$ represent the logically last element of the chain, while 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
\[
\{ x \mapsto \_ \ast y \mapsto \_ \ast \text{array}(\text{vs}, \text{max}) \}
\]

\[\text{ixf} = \text{newchan}();\]
\[\text{(ixn,ixf)} = \text{extend}(\text{ixf});\]
\[\text{signal}(\text{ixn});\]
\[\{ y \mapsto \_ \ast \text{array}(\text{vs}, \text{max}) \ast \text{recv}(\text{ixn}, x \mapsto \_) \ast \text{send}(\text{ixf}, \text{true}) \ast \text{ixn} < \text{ixf} \}\]
\[\text{iyf} = \text{newchan}();\]
\[\text{(iyn,iyf)} = \text{extend}(\text{iyf});\]
\[\text{signal}(\text{iyn});\]
\[\{ \text{array}(\text{vs}, \text{max}) \ast \text{recv}(\text{ixn}, x \mapsto \_) \ast \text{send}(\text{ixf}, \text{true}) \ast \text{ixn} < \text{ixf} \}\]
\[\text{fork}((\text{fp}(x,y,\text{vs}[j]),\text{ixn},\text{iyn},\text{ixl},\text{iyl}) ); \] // apply function spec.

\[\text{for}(j = 0; j < \text{max}; j++){\]
\[\text{ixl} = \text{ixn};\]
\[\{ \text{array}(\text{vs}, \text{max}) \ast \text{recv}(\text{ixn}, x \mapsto \_) \ast \text{send}(\text{ixn}, x \mapsto \_) \ast \text{recv}(\text{ixn}, x \mapsto \_) \ast \text{send}(\text{ixf}, \text{true}) \ast \text{ixn} < \text{ixf} \}\]
\[\text{iyl} = \text{iyn};\]
\[\{ \text{array}(\text{vs}, \text{max}) \ast \text{recv}(\text{ixn}, x \mapsto \_) \ast \text{send}(\text{ixn}, x \mapsto \_) \ast \text{recv}(\text{ixn}, x \mapsto \_) \ast \text{send}(\text{ixf}, \text{true}) \ast \text{ixn} < \text{ixf} \}\]
\[\text{fork}(\text{fp}(x,y,\text{vs}[j]),\text{ixn},\text{iyn},\text{ixn},\text{iyn});\]
\[\text{wait}(\text{ixn});\]
\[\text{wait}(\text{iyn});\]
\[\{ \text{array}(\text{vs}, \text{max}) \ast x \mapsto \_ \ast y \mapsto \_ \}\]

Fig. 5. Proof of fp client program.

given here is trivial, in [Botinˇcan 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 \([P]\) in the precondition to signal(), and \([\neg P]\) in the postcondition to wait(), as well as similar assertions in the axiom for splitting. \([\_]\) and \([\_\_]\) 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 \(\text{stable}(P)\) to explicitly assert stability of \(P\) the specification. However, this predicate is hard to manipulate in proofs.
because it satisfies few distribution properties. This is especially problematic for assertions containing separating implication, $\rightarrow$. In this paper, we instead use explicit stabilization operators, which strengthen or weaken an assertion until it is stable:

- $[P]$ 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 $[P] \iff P \iff \lceil P \rceil$. This holds by default if $P$ contains no shared assertions nor predicate variables.
- $\lceil P \rceil \implies P \implies [P]$.

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

- $[P] \ast [Q] \implies [P \ast Q]$.
- $[P \ast Q] \implies [P] \ast [Q]$.

The predicate $\text{stable}$ 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 $\rightarrow$ 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 $P \text{ stable}$ and $P \ast Q \implies R$, then $P \implies [Q \rightarrow R]$.

### 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:

```c
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;
}
```

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:

\[ \text{region}(R, T, I, r) \]

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: \( [\text{set}]_{i1}^{r1} \ast [\text{change}]_{j2}^{r2} \). 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: \( [\alpha]_{i+j}^{r} \Leftrightarrow [\alpha]_{i}^{r} \ast [\alpha]_{j}^{r} \). In contrast, region assertions can be duplicated
arbitrarily:

\[ \text{region}(R, T, I, r) \Longrightarrow \text{region}(R, T, I, r) \ast \text{region}(R, T, I, r) \]

Thus, each region assertion should be interpreted as one view on a resource shared between
many threads. Such a region assertion, \( \text{region}(R, T, I, r) \) 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 ex-
pressed 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 $\triangledown$. Intuitively, $\triangledown P$ 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 $\text{region}(\{x\}, T, I, r)$ thus expresses that the shared region $r$ currently owns the resources described by $\triangledown I(x)$.

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 $\triangledown P$ holds now, it is not the case that $P$ also holds now. Instead, $\triangledown$ 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.

$$P \implies \triangledown P \quad \text{(SMono)}$$

$$\text{atomic}(C) \land \text{stable}(R) \land \{P\} \ C \ \{Q\} \implies \{P \ast \triangledown R\} \ C \ \{Q \ast R\} \quad \text{(AFrame)}$$

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 $\triangledown$ instruction to justify the transfer of shared resources in and out of regions. While these $\triangledown$ 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 $\triangledown$ instructions can freely be eliminated. Hence, if $P$ and $Q$ are expressible in first-order separation logic and $\vdash_{iCAP} \{P\} \ C \ \{Q\}$, then $\vdash_{SL} \{P\} \overline{C} \ \{Q\}$, where $\overline{C}$ is $C$ stripped of $\triangledown$ instructions.

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

$$\triangledown (P \land Q) \iff \triangledown P \land \triangledown Q, \quad \triangledown (P \lor Q) \iff \triangledown P \lor \triangledown Q, \quad \triangledown (P \ast Q) \iff \triangledown P \ast \triangledown Q \quad \text{(LBin)}$$

$$\triangledown (P \Rightarrow Q) \iff \triangledown P \Rightarrow \triangledown Q, \quad \triangledown (P \Leftarrow Q) \iff \triangledown P \Leftarrow \triangledown Q \quad \text{(LImpl/LWand)}$$

Later also commutes over existential and universal quantification over non-empty types $\tau$:

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

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

$$\triangledown [P] \iff [\triangledown P], \quad \triangledown [P] \iff [\triangledown P] \quad \text{(LCeil/LFloor)}$$

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 $y$:

\[ \{ \text{region}(\{x\}, T, I, r) \} \{ \text{region}(\{y\}, T, I, r) \} \]

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 $\alpha$ action. We thus have to prove that there is an $\alpha$-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:

\[ \{ \triangledown I(x) \} \{ \triangledown \alpha \} \{ \triangledown P \} \]

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 $\text{emp}$).

\[ \{ \triangledown (l \mapsto x) \} \{ \triangledown \alpha \} \{ \triangledown l \mapsto y \} \]

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

\[
\text{atomic}(C) \land \{ x \mapsto y \} \ C \{ Q \} \Rightarrow \{ \triangledown x \mapsto y \} \ C \{ Q \} \quad \text{(LPoints)}
\]

### 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 $\text{send}$ and $\text{recv}$, 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: \{\text{Low, High, Done}\}. Each abstract state is associated with an invariant by $I_b$, defined as follows:

\[
\begin{align*}
I_b(x, P)(\text{Low}) & \triangleq x.\text{flag} \mapsto 0 \\
I_b(x, P)(\text{High}) & \triangleq x.\text{flag} \mapsto 1 \ast [P] \\
I_b(x, P)(\text{Done}) & \triangleq x.\text{flag} \mapsto 1
\end{align*}
\]

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

To ensure that these regions are invariant in concurrent contexts, we need to show that the interpretation of each abstract state, $\text{High}$, $\text{Low}$ and $\text{Done}$, 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 $[-]$, 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.
Fig. 7. Proof of `signal()` using simple predicate definitions.

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

\[
\begin{align*}
\text{send}(x, P) &\triangleq \exists r. \text{region}([\text{Low}], T_b, I_b(x, P), r) \ast [\text{set}]_r^x \ast [P] \\
\text{recv}(x, P) &\triangleq \exists r. \text{region}([\text{Low}, \text{High}], T_b, I_b(x, P), r) \ast [\text{get}]_r^x
\end{align*}
\]

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`. `[set]^x_r` and `[get]^x_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:

```c
{recv(x, P)}
wait(chan *x) {
  int b;
  do {
    region({Low, High}, T_b, I_b(x, P), r) * [get]_r
  }  // case-split on Low / High.
  \{ (region({Low}, T_b, I_b(x, P), r) ∨ region({High}, T_b, I_b(x, P), r))) * [get]_r \}
  b = x->flag;  // Low & High cases given below.
  \{ (region({Low, High}, T_b, I_b(x, P), r) * [get]_r ∧ b = 0) ∨ \}
  \{ (region({Done}, T_b, I_b(x, P), r) * [get]_r * [P] ∧ b = 1) \}
}  // while (b == 0)
\{ [P] \}
```

Low case:

```c
\{ region({Low}, T_b, I_b(x, P), r) * [get]_r \}
// open region
\{ ▷(x.flag -> 0) * [get]_r \}
// drop ▷ and read value.
\{ x.flag -> 0 * [get]_r ∧ b = 0 \}
// close region, and stabilise
\{ region({Low, High}, T_b, I_b(x, P), r) * [get]_r ∧ b = 0 \}
```

High case:

```c
\{ region({High}, T_b, I_b(x, P), r) * [get]_r \}
// open region
\{ ▷(x.flag -> 1 * [P]) * [get]_r \}
// drop ▷ and read value.
\{ x.flag -> 1 * [P] * [get]_r ∧ b = 1 \}
// close region.
\{ region({Done}, T_b, I_b(x, P), r) * [P] * [get]_r ∧ b = 1 \}
```

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 region({High}, T_b, I_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 [P] 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:

\[
\{\text{recv}(a, P) * ([P] \rightarrow (P_1 * P_2))\} \quad \langle \text{skip} \rangle \quad \{\text{recv}(a, P_1) * \text{recv}(a, P_2)\}
\]

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 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 \(\text{Low}(I)\) and \(\text{High}(I)\), with \(I \in \mathcal{P}_{\text{fin}}(\text{Prop})\) representing the splittings. In the case of the axiom, we would then have the transition:

\[
\text{Low}(I \uplus \{P\}) \leadsto \text{Low}(I \uplus \{P_1, P_2\})
\]

\[
\text{High}(I \uplus \{P\}) \leadsto \text{High}(I \uplus \{P_1, P_2\})
\]

However, iCAP does not support states and actions indexed by step-indexed sets, such as \(\text{Prop}\) (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 of 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 \(r \overset{\pi}{\mapsto} P\), associates an identifier \(r\) with a proposition \(P\). As \(r\) comes from a non step-indexed set (in fact \(\text{RID}\), 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 \(r \overset{\pi}{\mapsto} P \ast r_1 \overset{\pi_1}{\mapsto} P_1 \ast r_2 \overset{\pi_2}{\mapsto} P_2\), and give the transition as:

\[
\text{Low}(I \uplus \{r\}) \leadsto \text{Low}(I \uplus \{r_1, r_2\})
\]

\[
\text{High}(I \uplus \{r\}) \leadsto \text{High}(I \uplus \{r_1, r_2\})
\]

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 \(r \overset{\pi}{\mapsto} P\) and \(r \overset{\pi}{\mapsto} Q\), 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:

\[
\begin{align*}
\text{emp} & \quad \Rightarrow \quad \exists r. \ P \tag{1} \\
\pi_1 P' \pi_2 P & \quad \iff \quad \begin{cases} 
\pi_1 P + \pi_2 P & \text{if } (\pi_1 + \pi_2) \leq 1 \\
\text{false} & \text{otherwise}
\end{cases} \tag{2} \\
\pi_1 P' \pi_2 Q & \quad \Rightarrow \quad \pi_1 P + \pi_2 Q \tag{3} \\
\pi_1 P' \pi_2 Q & \quad \Rightarrow \quad \pi_1 P' + \pi_2 Q' \tag{4}
\end{align*}
\]

\(\text{Property (1)}\) allows us to create a saved proposition for an arbitrary proposition \(P\). \(\text{Property (2)}\) says that saved propositions are linear, meaning we can split and join them without worrying about unwanted duplication. Observe that the fractions \(\pi_1\) and \(\pi_2\) are used to track splitting. \(\text{Property (3)}\) 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 \(\text{iCAP later operator} \triangleright\) is needed because we use shared regions internally in the definition of saved propositions. \(\text{Property (4)}\) 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 \(\text{Low} (I)\) and \(\text{High} (I)\), where \(I\) is a finite set of region identifiers in \(P_{\text{fin}} (\text{RID})\). We use \(\text{LoHi}\) to stand for either \(\text{Low}\) or \(\text{High}\). The \(I\) 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 \(\text{Low} (I)\) or \(\text{High} (I)\), then each \(i \in I\) corresponds to a thread which expects to receive a resource. To find out what resource \(P\) is expected, we examine the associated saved proposition \(i \Rightarrow P\).

Actions in the transition relation \(T_m\) are of the form \(\text{send}\) and \(\text{change}(r)\), where \(r\) is a region identifier. The first action, \(\text{send}\), sets the flag, and simply moves from \(\text{Low}\) to \(\text{High}\). The second, \(\text{change}(r)\), 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 \(r_2\).

\[
T_m(\text{send}) \triangleq \{(\text{Low} (I), \text{High} (I))\}
\]

\[
T_m(\text{change}(r)) \triangleq \{(\text{High} (I \cup \{r\}), \text{High} (I))\} \cup \{(\text{Low} (I \cup \{r\}), \text{Low} (I \cup \{r_1, r_2\}))\} \\
\cup \{(\text{High} (I \cup \{r\}), \text{High} (I \cup \{r_1, r_2\}))\}
\]
The invariants associated with Low and High are defined as follows:

\[ I_m(x, r, P)(\text{Low}(I)) \triangleq x.\text{flag} \Rightarrow 0 * \text{waiting}(P, I) * \text{changes}_r(I) \]

\[ I_m(x, r, P)(\text{High}(I)) \triangleq x.\text{flag} \Rightarrow 1 * \text{ress}(I) * \text{changes}_r(I) \]

where

\[ \text{waiting}(P, I) \triangleq \exists Q : I \rightarrow \text{Prop.} \left( [(\triangleright P) \rightarrow \bigotimes_{i \in I} Q(i)] * \bigotimes_{i \in I} i \frac{1}{2} \rightarrow Q(i) \right) \]

\[ \text{ress}(I) \triangleq \bigotimes_{i \in I} \exists R. i \frac{1}{2} \rightarrow R * [\triangleright R] \]

\[ \text{changes}_r(I) \triangleq \bigotimes_{i \in I} [\text{change}(r')] \]

The definitions here use three auxiliary predicates: waiting, standing for resources that have been promised but not supplied; ress, standing for resources once they have been supplied; and 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 waiting predicate. \( \text{waiting}(P, I) \) 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 \( i \frac{1}{2} \rightarrow Q(i) \). The assertion \( [(\triangleright P) \rightarrow \bigotimes_{i \in I} Q(i)] \) 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. \( \text{ress}(I) \) pairs together fractional saved propositions, \( i \frac{1}{2} \rightarrow R \) with resources \([\triangleright R]\). 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:

\[ \text{creg}(x, r, P, S) \triangleq \text{region}(S, T_m, I_m(x, r, P), r) \]

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.

\[ \text{send}(x, P) \triangleq \exists r. \text{creg}(x, r, P, \{\text{Low}(I) | \text{true}\}) * [\text{send}]_r \]

The definition of recv\( (x, Q) \) predicate is more complex. It includes \( r' \frac{1}{2} \rightarrow Q \), 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).

\[ \text{recv}(x, Q) \triangleq \exists R, r, r'. \text{creg}(x, r, R, \{\text{LoHi}(I) | r' \in I\}) * r' \frac{1}{2} \rightarrow Q * [\text{change}(r')]_r \]

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 straightforward. The challenging ones are line 6 – creating the stabilised assertion – and line 10 – the viewpoint which creates the region itself. In line 6, we need the following implication:

\[ \text{emp} \implies [(\triangleright P) \rightarrow (\triangleright P)] \]

To show this holds, we observe that for any \( X, \text{emp} \implies X \rightarrow X \), and that \( \text{emp} \) is always stable. The implication then follows by monotonicity of explicit stabilisation brackets, \((A \Rightarrow B) \implies ([A] \Rightarrow [B])\).
Fig. 9. Proof of $\text{newchan}()$ w.r.t. the full specification.

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

$$r_1 \xrightarrow{1/2} P * x \xrightarrow{0} 0 * [(\perp P) \rightarrow (\perp P)]$$

$$\subseteq \exists r_2. \text{creg}(x, r_2, P, \{\text{Low}(\{r_1\})\}) * |\text{send}|^{r_2} * |\text{change}(r_1)|^{r_2}$$

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

**Proving signal.** The proof of $\text{signal}$ (Fig. 10) works by opening the channel region (line 4), merging in the supplied resource $[P]$ 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 Low$(I)$ to High$(I)$ to is allowed, but this is simple: it’s the only transition associated to $\text{send}$ 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.** $[P] * \text{waiting}(P, I) \subseteq \text{ress}(I)$

**Proof.** $[P] * \text{waiting}(P, I)$

$$\subseteq [\triangleright P] * \exists Q: I \rightarrow \text{Prop.} [\triangleright P \rightarrow [\odot_{i \in I}. Q(i)]] * ([\odot_{i \in I}. i \xrightarrow{1/2} Q(i)]]$$

$$\subseteq \exists Q: I \rightarrow \text{Prop.} [\odot_{i \in I}. Q(i)] * ([\odot_{i \in I}. i \xrightarrow{1/2} Q(i)]$$

$$\subseteq \exists Q: I \rightarrow \text{Prop.} (\odot_{i \in I}. [\triangleright Q(i)]) * ([\odot_{i \in I}. i \xrightarrow{1/2} Q(i)]$$

$$\subseteq (\odot_{i \in I}. \exists R. i \xrightarrow{1/2} R * [\triangleright R])$$

$$\subseteq \text{ress}(I)$$

To prove the second step, we appeal to the fact that $[\_]$ is semi-distributive over separating conjunction, $[A * [B] \equiv [A * B]$, and *modus ponens* for separating implication, $A * (A \rightarrow B) \equiv B$. The third step follows from the fact that $[\_]$ is weaker than $[\_]$, and $[\_]$ is semi-distributive over separating conjunction, $[A * B] \equiv [A] * [B]$. 

\[ \square \]
\begin{verbatim}
\{send(x, P) \& \{P\}\}
signal(chan *x) {  
  \{creg(x, r, P; \{Low(I') \& true\}) \& \{send\} [\{P\}]\}
  // open the region. Low(I) is an arbitrary member of \{Low(I') \& true\}.  
  \{\triangledown I_m(x, r, P; \{Low(I)\}) \& \{send\} [\{P\}]\}
  // invariant definition.  
  \{\{send\} [\{P\}] \& \triangledown (x.\text{flag} \mapsto 0 \& \text{waiting}(P, I) \& \text{changes}(I))\}
  // pull out points-to using LBin and LPoints.
  \{\{send\} [\{P\}] \& x.\text{flag} \mapsto 0 \& \triangledown (\text{waiting}(P, I) \& \text{changes}(I))\}
  x->\text{flag} := 1; // rewrite flag, drop \triangledown using AFrame
  \{\{send\} [\{P\}] \& x.\text{flag} \mapsto 1 \& \text{waiting}(P, I) \& \text{changes}(I)\}
  // Lemma 5.1, add a \triangledown using SMono.
  \{\{send\} [\{P\}] \& \triangledown (x.\text{flag} \mapsto 1 \& \text{ress}(I) \& \text{changes}(I))\}
  // Invariant definition.
  \{\{send\} [\{P\}] \& \triangledown I_m(x, r, P; \{High(I)\})\}
  // close the region.
  \{\{send\} [\{P\}] \& \text{creg}(x, r, P; \{High(I') \& true\})\}
  } // delete unneeded predicates.
\{emp\}
\end{verbatim}

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

\textit{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 $I$; 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.** $r \overset{1/2}{\longrightarrow} P \& \text{ress}(I \uplus \{r\}) \preceq \text{ress}(I) \& \triangledown [P]$

**Proof.** $r \overset{1/2}{\longrightarrow} P \& \text{ress}(I \uplus \{r\})$

\begin{align*}
  & \preceq r \overset{1/2}{\longrightarrow} P \& \text{ress}(I) \& \exists R. r \overset{1/2}{\longrightarrow} R \& \triangledown [R] \quad \text{(Rearrange)} \\
  & \preceq r \overset{1/2}{\longrightarrow} P \& \text{ress}(I) \& \exists R. r \overset{1/2}{\longrightarrow} R \& \triangledown [P] \quad \text{(Property 3, mono of $[-]$)} \\
  & \preceq \text{ress}(I) \& \triangledown [P] \quad \text{(LCEIL/LFLOOR)} \\
\end{align*}

**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 $\triangledown$. 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.
1 \{ \text{recv}(x, P) \} 

2 \text{wait(chan *x) \{ }

3 \exists R, r, r`, r` \xrightarrow{1/2} P \star [\text{change}(r')]_I \star \text{creg}(x, r, R, \{\text{LoHi}(I') \mid r' \in I'\}) \}

4 // Open the region. Only consider the High case.

5 \{ r' \in I \land r` \xrightarrow{1/2} P \star [\text{change}(r')]_I \star \text{Im}(x, r, R)(\text{High}(I)) \}

6 // Apply invariant definition

7 // Pull out points-to using LBin and LPoints

8 \{ r' \in I \land \rightarrow I r` \xrightarrow{1/2} P \star [\text{change}(r')]_I \star x.\text{flag} \rightarrow 1 \star \text{ress}(I) \star \text{changes}_I(I) \}

9 \text{assume}(x->\text{flag} == 1) // drop \triangleright \text{using AFrame, push in } [\text{change}(r')] \text{ perm.}

10 \{ r' \in I \land \rightarrow I r` \xrightarrow{1/2} P \star x.\text{flag} \rightarrow 1 \star \text{ress}(I) \star \text{changes}_I(I \cup \{r'\}) \}

11 // Lemma 5.2, add \triangleright \text{using SMono}

12 \{ \triangleright P \star \triangleright (x.\text{flag} \rightarrow 1 \star \text{ress}(I \{r'}) \star \text{changes}_I(I \{r'})) \}

13 // Invariant definition

14 \{ \triangleright P \star \triangleright \text{Im}(x, r, R)(\text{High}(I \{r'})) \}

15 // close the region

16 \{ \triangleright P \star \exists R, r. \text{creg}(x, r, R, \{\text{High}(I) \mid \text{true}\}) \}

17 // abstract garbage collect

18 \{ \triangleright [P] \}

19 \}

Fig. 11. Proof of \text{wait()} w.r.t. the full specification.

1 \{ \text{recv}(x, P) \star [P] \rightarrow P_1 \star P_2 \} 

2 \exists R, r, r`, r` \xrightarrow{1/2} P \star [\text{change}(r')]_I \star \text{creg}(x, r, R, \{\text{LoHi}(I') \mid r' \in I'\}) \star [P] \rightarrow P_1 \star P_2 \}

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

4 \exists R, r, r`, r` \xrightarrow{1/2} P \star [\text{change}(r')]_I \star \text{Im}(x, r, R)(\text{LoHi}(I \cup \{r'\})) \star [P] \rightarrow P_1 \star P_2 \}

5 \text{skip} // Use AFrame to remove \triangleright

6 \exists R, r, r`, r` \xrightarrow{1/2} P \star [\text{change}(r')]_I \star \text{Im}(x, r, R)(\text{LoHi}(I \cup \{r'\})) \star [P] \rightarrow P_1 \star P_2 \}

7 // Lemma 5.5

8 \exists R, r, r_1, r_2, r_1 \xrightarrow{1/2} P_1 \star r_1 \xrightarrow{1/2} P_2 \star [\text{change}(r_1)]_I \star [\text{change}(r_2)]_I \}

9 // Close the region

10 \exists R, r, r_1, r_2, r_1 \xrightarrow{1/2} P_1 \star r_1 \xrightarrow{1/2} P_2 \star [\text{change}(r_1)]_I \star [\text{change}(r_2)]_I \}

11 \text{recv}(x, P_1) \star \text{recv}(x, P_2) \}

Fig. 12. Proof outline for splitting axiom.
Lemma 5.3. \( r \xrightarrow{1/2} P \cdot \lceil P \rceil \to P_1 \cdot P_2 \cdot \text{waiting}(R, \mathcal{I} \cup r) \)
\[
\begin{array}{l}
\exists r_1, r_2, r_1 \xrightarrow{1/2} P_1 \cdot r_2 \xrightarrow{1/2} P_2 \cdot \text{waiting}(R, \mathcal{I} \cup \{r_1, r_2\})
\end{array}
\]

Proof.
\[
\begin{array}{l}
r \xrightarrow{1/2} P \cdot \lceil P \rceil \to P_1 \cdot P_2 \cdot \text{waiting}(R, \mathcal{I} \cup r)
\end{array}
\]
\[
\begin{array}{l}
\exists Q: \mathcal{I} \cup \{r\} \to \text{Prop}. \lceil (\mathcal{O}R) \to \mathcal{O}_{i \in \mathcal{I} \cup r}. Q(i) \rceil \cdot (\mathcal{O}_{i \in \mathcal{I} \cup r}. i \xrightarrow{1/2} Q(i))
\end{array}
\]
\[
\begin{array}{l}
\exists Q'. r \xrightarrow{1/2} P \cdot r \xrightarrow{1/2} Q' \cdot \lceil P \rceil \to P_1 \cdot P_2 \cdot \text{waiting}(R, \mathcal{I} \cup r)
\end{array}
\]
\[
\begin{array}{l}
\exists Q: \mathcal{I} \to \text{Prop}. \lceil (\mathcal{O}R) \to \mathcal{O}_{i \in \mathcal{I}}. Q(i) \rceil \cdot (\mathcal{O}_{i \in \mathcal{I}}. i \xrightarrow{1/2} Q(i))
\end{array}
\]
\[
\begin{array}{l}
\lceil P \rceil \to P_1 \cdot P_2 \cdot \text{waiting}(R, \mathcal{I} \cup r)
\end{array}
\]

Lemma 5.4. \( r \xrightarrow{1/2} P \cdot \lceil P \rceil \to P_1 \cdot P_2 \cdot \text{ress}(\mathcal{I} \cup r) \)
\[
\begin{array}{l}
\exists r_1, r_2, r_1 \xrightarrow{1/2} P_1 \cdot r_2 \xrightarrow{1/2} P_2 \cdot \text{ress}(\mathcal{I} \cup \{r_1, r_2\})
\end{array}
\]

Proof.
\[
\begin{array}{l}
r \xrightarrow{1/2} P \cdot \lceil P \rceil \to P_1 \cdot P_2 \cdot \text{ress}(\mathcal{I} \cup r)
\end{array}
\]
\[
\begin{array}{l}
\exists r_1, r_2, r_1 \xrightarrow{1/2} P_1 \cdot r_2 \xrightarrow{1/2} P_2 \cdot \text{ress}(\mathcal{I} \cup \{r_1, r_2\})
\end{array}
\]

The last step consists of two applications of the following sub-lemma:
\[
\begin{array}{l}
\text{ress}(\mathcal{I}) \cdot \mathcal{O}P \cdot r \xrightarrow{1} P
\end{array}
\]
\[
\begin{array}{l}
\exists Q. i \xrightarrow{1/2} Q \cdot \lceil \mathcal{O}P \rceil \cdot \lceil \mathcal{O}P \rceil \to P
\end{array}
\]
\[
\begin{array}{l}
\exists Q. i \xrightarrow{1/2} Q \cdot \lceil \mathcal{O}P \rceil \cdot \r \xrightarrow{1} P
\end{array}
\]
\[
\begin{array}{l}
\exists \mathcal{I} \cup \{r\} \cdot \r \xrightarrow{1/2} P
\end{array}
\]
These lemmas are combined as follows.

**Lemma 5.5.** \( r' \xrightarrow{1/2} P \ast I_m(x, r, R)(\text{LoHi}(I \cup \{r'\})) \ast \frac{[\text{change}(r')]_r}{1} \ast \frac{[P]}{[P]} \ast P_1 \ast P_2 \)

\[ \exists r_1, r_2. \quad \frac{[\text{change}(r_1)]_r}{1} \ast r_1 \xrightarrow{1/2} P_1 \ast \frac{[\text{change}(r_2)]_r}{1} \ast r_2 \xrightarrow{1/2} P_2 \ast I_m(x, r, R)(\text{LoHi}(I \cup \{r_1, r_2\})) \]

**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. \(\square\)

### 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 renunciation. 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;
    }
}

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, \(\text{CNode}^\dagger\):

\[
\text{CNode} \triangleq \langle \text{loc} \in \text{Addr}, \text{(physical address)} \rangle \times \langle \text{res} \in \text{RId}, \text{(region ID for sent resource)} \rangle \times \langle \mathcal{I} \in \mathcal{P}\text{fin}(\text{RId}), \text{(region IDs for promised resources)} \rangle \times \langle \text{flg} \in \{\text{High}, \text{Low}\}, \text{(flag status)} \rangle \times \langle \mathcal{W} \in \mathcal{P}\text{fin}(\text{RId}), \text{(region IDs for earlier renounced resources)} \rangle
\]

Each \(\text{CNode}\) represents one channel in the chain, while an abstract chain is a finite sequence in \(\text{CNode}^*\). Given an element \((\text{loc}, \text{res}, \mathcal{I}, \text{flg}, \mathcal{W}) \in \text{CNode}\), \(\text{loc}\) is the physical address, \(\text{res}\) is the identifier used when sending resources, \(\mathcal{I}\) is the set of identifiers for resources promised to other predicates, \(\text{flg}\) 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 \(\text{CNode} s\), we sometimes write \(s.\text{flg}\), \(s.\mathcal{I}\) 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:

\[
\begin{align*}
\text{available}([]) & \triangleq \emptyset, \quad \text{available}(s \cdot xs) \triangleq (\text{available}(xs) \setminus s.\mathcal{W}) \uplus s.\mathcal{I} \\
\text{wf}([]) & \triangleq \text{true}, \quad \text{wf}(s \cdot xs) \triangleq \text{wf}(xs) \land s.\mathcal{W} \subseteq \text{available}(xs) \land s.\mathcal{I} \cap s.\mathcal{W} = \emptyset \land \\
\text{for all } s' \in xs. s.\mathcal{I} \cap s'.\mathcal{I} = \emptyset \land s.\mathcal{W} \cap s'.\mathcal{W} = \emptyset
\end{align*}
\]

The predicate \(\text{available}\) 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, \(\text{wf}\), 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, \(\text{ctru}\) and \(\text{cconf}\). 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.

\[
\begin{align*}
\text{ctru}(rs) & \triangleq \forall e \in rs. e.\text{flg} = 1 \\
\text{cconf}(rs) & \triangleq \forall e \in rs. e.\text{flg} = 1 \land e.\mathcal{W} = \emptyset
\end{align*}
\]

Finally, we use \(rs_1 \xrightarrow{\mathcal{W}} rs_2\) to denote that the abstract chain \(rs_2\) follows from \(rs_1\) by cancelling out renounced resources with the corresponding promise. Formally, \(\xrightarrow{\mathcal{W}}\) 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.

\[
rs \xrightarrow{\mathcal{W}} rs' \triangleq \exists x, y, r. \\
r \in rs(x).\mathcal{W} \land r \in rs(y).\mathcal{I} \land (x, y) \in \text{ord}(rs) \land \\
r s' = (rs \downarrow_i \downarrow_j \langle \mathcal{W} \setminus \{r\} \rangle) \downarrow_i \downarrow_j \langle \mathcal{I} \setminus \{r\} \rangle
\]

The notation \(\downarrow_i\) is a lens allowing a single field of a chain to be updated without modifying the remainder of the chain.

\[
x \downarrow_i f \triangleq x[i \mapsto (f[x(i)/\bullet])] \quad x \downarrow_i \downarrow_j f \triangleq x[i \mapsto (x(i) \downarrow_j f)]
\]
where convenient we use chains as finite functions of type $T$
relation send form a new $W$
ation, which adds an element to $W$
chainds The transition relation defines six kinds of transitions in Fig. 13. For renun, which adds an element to $W$
chainds
get c change chainres $x,r,rs,rs' x,r,rs,rs'$
x.r,rs,rs' x.r,rs,rs' $w$ to $W$
chainds

$$\begin{align*}
\text{renun}_c(x,rs,rs') & \triangleq rs(x).flg = 0 \land rs' = (rs \downarrow x,rs,rs') \downarrow W (\cup \cup w) \\
\text{set}_c(x,rs,rs') & \triangleq rs(x).flg = 0 \land rs' = rs \downarrow x,flg (1) \\
\text{ext}_c(x,rs,rs') & \triangleq rs = (rs_1 \cdot a \cdot rs_2) \land rs' = (rs_1 \cdot a \cdot b \cdot rs_2) \\
& \quad \land a.loc = x \land a.flg = 0 \land b.flg = 0 \land b.W = \emptyset \\
\text{split}_c(x,r,rs,rs') & \triangleq r \in rs(x).I \land rs' = rs \downarrow x,\downarrow (\{t\} \cup \{r_2,r_3\}) \\
\text{sat}_c(x,rs,rs') & \triangleq rs = (rs_1 \cdot rs_2) \land rs' = (rs_1 \cdot rs_2') \land cconf(rs_2') \land rs_2 \leftarrow rs' \\
\text{get}_c(x,r,rs,rs') & \triangleq rs = (rs_1 \cdot rs(x) \cdot rs_2) \land cconf(rs(x) \cdot rs_2) \land rs' = rs \downarrow x,rs,rs' \leftarrow \{t\}
\end{align*}$$

$$T_c(\text{send}(x)) \triangleq \{(a,b) | \text{wf}(b) \land (\text{renun}_c(x,a,b) \lor \text{set}_c(x,a,b) \lor \text{ext}_c(x,a,b))\}$$

$$T_c(\text{change}(x,r)) \triangleq \{(a,b) | \text{wf}(b) \land (\text{split}_c(x,r,a,b) \lor \text{sat}_c(x,a,b) \lor \text{get}_c(x,r,a,b))\}$$

Fig. 13. Definition of $T_c$, the transition relation for the chained channel implementation.

chainsds($x \cdot y \cdot rs$) \triangleq x.loc \leftrightarrow \{\text{prev} = y.loc; \text{flag} = x.flg\} \ast \text{chainsds}(y \cdot rs) 
chainsds($x \cdot null$) \triangleq x.loc \leftrightarrow \{\text{prev} = \text{NULL}; \text{flag} = x.flg\} 

resource($\mathcal{I},W$) \triangleq \exists Q: I \rightarrow \text{Prop}, R: W \rightarrow \text{Prop.}$

$$\begin{align*}
\bigotimes_{i \in I} i \xrightarrow{1/2} Q(i) \ast \bigotimes_{w \in W} w \xrightarrow{1/2} R(w) \\
\ast \left[\left(\bigotimes_{i \in I} R(w)\right) \rightarrow \bigotimes_{i \in I} \left[Q(i)\right]\right]
\end{align*}$$

chainres($x \cdot rs$) \triangleq \text{resource}($x.I,x.W \cup \{x.rs | \neg x.flg\}$) \ast \text{chainres}(rs) 
chainres($null$) \triangleq \text{emp} 

$$\begin{align*}
\text{uS}(rs) & \triangleq \{x | (x,rs,0,rs) \in rs\} \\
\text{uC}(rs) & \triangleq \{(x,i) | (x,rs,i,rs) \in rs \land i \in I \land \neg \exists(y,rs,w) \in rs. i \in W\} \\
\text{unused}(r,rs) & \triangleq \bigotimes x \notin \text{uS}(rs). \left[\text{send}(x)\right]_r \ast \left(\bigotimes x \notin \text{uC}(rs). \left[\text{change}(x,r')\right]_r\right)
\end{align*}$$

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 $\text{Chain}(rs)$, where $rs$ is an abstract chain. Actions have the form $\text{send}(x)$ and $\text{change}(x,r)$, 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

$$\text{Addr} \xrightarrow{\triangleq} (\text{Rld} \times \mathcal{P}(\text{Rld}) \times \{\text{High,Low}\} \times \mathcal{P}(\text{Rld}))$$

The transition relation defines six kinds of transitions in Fig. 13. For $\text{send}$ we have renunciation, which adds an element to $W$; setting the flag; and extending the chain, which creates a new $\text{CNode} b$. For $\text{change}$ we have splitting; satisfying the renounced resource set, which sets $W$ 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: $\text{chains}$, $\text{chainres}$, and $\text{unused}$ (defined in Fig. 14). The predicate $\text{chains}$ 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 $I$ and a set of renounced resources that are being waited for $W$. When there are no waiting resources the predicate can be drastically simplified to just the promised resources:

**Lemma 6.1.** \( \text{resource}(I, \emptyset) \sqsubseteq \bigotimes_{i \in I} \exists Q : \text{Prop}. i \overset{1/2}{\Rightarrow} Q(i) \ast [\forall Q(i)] \)

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

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

\[ I_c(r)(\text{Chain}(rs)) \triangleq \text{chains}(rs) \ast \text{chainres}(rs) \ast \text{unused}(r, rs) \]

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

\[ \text{oreg}(r, S) \triangleq \text{region}(S, T_c, I_c(r), r) \]

We can now define the send, recv, and ordering predicates. We write \((x, y) \in \text{seq}(rs)\) to say that the two addresses \(x\) and \(y\) appear adjacent in the sequence \(rs\), and \((x, y) \in \text{ord}(rs)\) to say just that they are ordered in \(rs\).

**send** \((x, P) \triangleq \exists r_1, r_2. \text{oreg}(r_1, \{ \text{Chain}(rs) \mid \text{wf}(rs) \land rs(x) = (r_2, \_ , 0, \_ ) \}) \]

\[ \ast r_2 \overset{1/2}{\Rightarrow} P \ast [\text{send}(x)]_1^{r_1} \]

**recv** \((x, P) \triangleq \exists r_1, r_2. \text{oreg}(r_1, \{ \text{Chain}(rs) \mid rs(x) = (\_ , I, \_ , \_ ) \land r_2 \in I \land \text{wf}(rs) \}) \]

\[ \ast r_2 \overset{1/2}{\Rightarrow} P \ast [\text{change}(x, r_2)]_1^{r_1} \]

**x < y \triangleq \exists r. \text{oreg}(r, \{ \text{Chain}(rs) \mid (y, x) \in \text{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 \([P]\) and an associated saved proposition is sufficient to allow the flag to be set.

**Lemma 6.2.** \( r \overset{1/2}{\Rightarrow} P \ast [P] \ast \text{resource}(I, W \cup \{r\}) \sqsubseteq \text{resource}(I, W) \)

**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.
\[
\{ \text{send}(x, P) \mid [P] \} \\
\]

// Definition of send.

\[
\exists r_1, r_2, r_2 \frac{\frac{1}{2}}{P \cdot [send(x)]^\ast_1 \cdot [P] \ast}
\]

\[
\text{oreg} (r_1, \{ \text{Chain}(rs) \mid \text{wf}(rs) \wedge rs(x) = (r_2, I, 0, W) \})
\]

// Enter the region.

\[
\exists r_1, r_2, r_2 \frac{\frac{1}{2}}{P \cdot [send(x)]^\ast_1 \cdot [P] \ast}
\]

\[
\triangleright (\text{chain}(rs) \cdot \text{chain}(rs) \cdot \text{unused}(r_1, rs) \wedge rs(x) = (r_2, I, 0, W))
\]

// Split up using chain axioms.

\[
\exists r_1, r_2, r_2 \frac{\frac{1}{2}}{P \cdot [send(x)]^\ast_1 \cdot [P] \ast}
\]

\[
\triangleright (\exists s_1, s_2, a. \text{chain}(rs_1) \cdot x \mapsto \{ \text{prev} = \text{hd}(rs_2), \text{loc}: \text{flag} = 0 \} \ast \text{chain}(rs_2) \ast \\
\text{chain}(rs) \ast \text{unused}(r_1, rs) \wedge rs = r_1 \cdot a \cdot rs_2 \wedge \text{wf}(rs) \wedge a = (x, r_2, I, 0, W))
\]

x->flag = 1; // Set the flag, drop the \triangleright using Aframe.

\[
\exists r_1, r_2, r_2 \frac{\frac{1}{2}}{P \cdot [send(x)]^\ast_1 \cdot [P] \ast}
\]

\[
\exists s_1, s_2, a. \text{chain}(rs_1) \cdot x \mapsto \{ \text{prev} = \text{hd}(rs_2), \text{loc}: \text{flag} = 1 \} \ast \text{chain}(rs_2) \ast \\
\text{chain}(rs) \ast \text{unused}(r_1, rs) \wedge rs = r_1 \cdot a \cdot rs_2 \wedge \text{wf}(rs) \wedge a = (x, r_2, I, 0, W)
\]

// Pull out resource predicate.

\[
\exists r_1, r_2, r_2 \frac{\frac{1}{2}}{P \cdot [send(x)]^\ast_1 \cdot [P] \ast}
\]

\[
\exists s_1, s_2, a. \text{chain}(rs_1) \cdot x \mapsto \{ \text{prev} = \text{hd}(rs_2), \text{loc}: \text{flag} = 1 \} \ast \text{chain}(rs_2) \ast \\
\ast \text{chain}(rs_1) \ast \text{resource}(I, W \uplus \{ r_2 \}) \ast \text{chain}(rs_2) \ast \\
\ast \text{unused}(r_1, rs) \wedge rs = r_1 \cdot a \cdot rs_2 \wedge \text{wf}(rs) \wedge a = (x, r_2, I, 0, W)
\]

// Apply Lemma 6.2.

\[
\exists r_1, r_2, [send(x)]^\ast_1
\]

\[
\exists s_1, s_2, a. \text{chain}(rs_1) \cdot x \mapsto \{ \text{prev} = \text{hd}(rs_2), \text{loc}: \text{flag} = 1 \} \ast \text{chain}(rs_2) \ast \\
\ast \text{chain}(rs_1) \ast \text{resource}(I, W) \ast \text{chain}(rs_2) \ast \\
\ast \text{unused}(r_1, rs) \wedge rs = r_1 \cdot a \cdot rs_2 \wedge \text{wf}(rs) \wedge a = (x, r_2, I, 0, W)
\]

// Collapse the chain, add \triangleright.

\[
\exists r_1, r_2
\]

\[
\triangleright (\text{chain}(rs') \ast \text{chain}(rs') \ast \text{unused}(r_1, rs') \wedge rs' = rs[x \mapsto (r_2, I, 1, W)])
\]

// Close the region using transition set.

// Well-formed structure of chain hasn't changed.

\[
\exists r_1, r_2, \text{oreg} (r_1, \{ \text{Chain}(rs) \mid \text{wf}(rs) \wedge rs(x) = (r_2, I, 1, W) \})
\]

// Garbage collect.

\{emp\}

Fig. 15. Sketch-proof for signal with out-of-order signalling.
\{ \texttt{recv}(x, P) \}

wait(x)
ch = x;

// Definition of \texttt{recv}.
\{ \exists r_1, r_2. \texttt{oreg}(r_1, \{ \texttt{Chain}(rs) \} \mid \exists I'. \texttt{rs}(x) = (\_ , I', \_ ) \land r_2 \in I' \land \texttt{wf}(rs)) \}
\begin{align*}
* r_2 \xrightarrow{1/2} P * [\text{change}(x, r_2)]_{\text{r}_1}^1 \land c = x
\end{align*}

while(c != NULL) {
while(!c->flag) skip;
c = c->prev;
}

// Flags up to x are now set. Stable because flags cannot get unset.
\{ \exists r_1, r_2. \texttt{r}_2 \xrightarrow{1/2} P * [\text{change}(x, r_2)]_{\text{r}_1}^1 * \}
\{ \texttt{oreg}(r_1, \{ \texttt{Chain}(rs) \} \mid \exists I'. \texttt{wf}(rs) \land \exists i. \texttt{rs}[i] = (x, r, I' \cup r_2, \_ ) \land \forall j. i \leq j < \text{len}(rs) \implies rs[j] = (\_ , \_ , 1, \_ ) \}) \}
\{ \texttt{skip}; // Enter the region and drop \triangleright \texttt{ using AFrame.} \}
\{ \exists rs_1, rs_2, a, r_2 \xrightarrow{1/2} P * [\text{change}(x, r_2)]_{\text{r}_1}^1 * \texttt{chains}(rs) * \texttt{chainres}(rs) * \texttt{unused}(r_1, rs) \land \texttt{wf}(rs) \land rs = rs_1 \cdot a \cdot rs_2 \land a = (x, r, I' \cup r_2, \_ , W) \land \texttt{ctrue}(a \cdot rs_2) \}
\{ \texttt{cf}(rs_1) \land \texttt{wf}(rs) \land rs = rs_1 \cdot a \cdot rs_2 \land a = (x, r, I' \cup r_2, \_ , W) \land \texttt{ctrue}(a \cdot rs_2) \}
\{ \texttt{cf}(rs_1) \land \texttt{chainres}(rs_1) \land \texttt{resource}(I' \cup r_2, \emptyset ) \land \texttt{chainres}(rs_2) \land \texttt{unused}(r_1, rs) \land \texttt{wf}(rs) \land rs = rs_1 \cdot a \cdot rs_2 \land a = (x, r, I' \cup r_2, \_ , W) \land \texttt{cf}(rs_2) \land a \cdot rs_2 \xrightarrow{W_e} rs_2 \}
\begin{align*}
\texttt{change}(rs_1) \land \texttt{resource}(I', \emptyset ) \land \texttt{chainres}(rs_2) \land \texttt{unused}(r_1, rs) \land \\
\texttt{wf}(rs) \land rs = rs_1 \cdot a \cdot rs_2 \land a = (x, r, I' \cup r_2, \_ , W) \land \texttt{cf}(rs_2) \land a \cdot rs_2 \xrightarrow{W_e} rs_2 \}
\end{align*}

// Pull out the resource predicate for x.
\{ \exists rs_1, rs_2, rs_1', a, r_2 \xrightarrow{1/2} P * [\text{change}(x, r_2)]_{\text{r}_1}^1 * \}
\{ \texttt{chains}(rs) * \texttt{chainres}(rs_1 \cdot rs_2') \land \texttt{unused}(r_1, rs) \land \texttt{wf}(rs) \land rs = rs_1 \cdot a \cdot rs_2 \land \texttt{cf}(rs_2') \land \texttt{a} \cdot rs_2 \xrightarrow{W_e} rs_2' \}
\begin{align*}
\texttt{change}(rs_1) \land \texttt{resource}(I', \emptyset ) \land \texttt{chainres}(rs_2') \land \texttt{unused}(r_1, rs) \land \\
\texttt{wf}(rs) \land rs = rs_1 \cdot a \cdot rs_2 \land a = (x, r, I' \cup r_2, \_ , W) \land \texttt{cf}(rs_2') \land \texttt{a} \cdot rs_2 \xrightarrow{W_e} rs_2' \}
\end{align*}

// Close the region, use transitions \texttt{sat} and get.
\{ \exists r_1, r_2. \triangleright [P] \}
\{ \texttt{oreg}(r_1, \{ \texttt{Chain}(rs') \}) \mid \texttt{wf}(rs') \land rs = rs_1 \cdot a \cdot rs_2 \land \texttt{a} \cdot rs_2 \xrightarrow{W_e} rs' \land \texttt{cf}(rs_2') \land rs' = (rs_1 \cdot rs_2') \triangleright [P] \}
\begin{align*}
\texttt{cf}(rs_2') \land rs' = (rs_1 \cdot rs_2') \triangleright [P] \}
\end{align*}

// Use \texttt{[P] \implies \triangleright [P] and clean up garbage.}

Fig. 16. Sketch-proof for \texttt{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.**

\[
\text{chainres}(rs) \land \text{wf}(rs) \land \text{cttrue}(rs) \sqsubseteq \exists rs'. \text{chainres}(rs') \land \text{cconf}(rs') \land rs \xrightarrow{\mathcal{W}} rs' \land \text{wf}(rs')
\]

**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.**

\[
\text{resource}(I \cup r_2, \emptyset) \ast r_2 \xrightarrow{\frac{1}{2}} P \sqsubseteq \text{resource}(I, \emptyset) \ast [\triangleright P]
\]

**Proof.** Given in Appendix B

This lemma says that, given resource and an identifier $r_2$ in $I$ 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.**

\[
\{ \text{oreg}(r, T) \ast r' \xrightarrow{\frac{1}{2}} P \} \langle \text{skip} \rangle \{ \text{oreg}(r, T \cap \{ \text{Chain}(rs) \mid r' \notin rs \}) \ast r' \xrightarrow{\frac{1}{2}} P \}
\]

**Proof.** Each identifier $r''$ used in $rs$ is associated with a fractional saved proposition $r'' \xrightarrow{\frac{1}{2}} P$. 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 retrieve the required permissions from unused. We use this lemma on line 16 of the sketch-proof.

**Lemma 6.6.**

\[
rs = rs_1 \cdot (x, r_2, I, 0, W) \cdot rs_2 \land \text{unused}(r_1, rs) \land z, r', r'' \notin rs \\
\sqsubseteq \text{unused}(r_1, rs_1 \cdot (x, r_2, I, 0, W) \cdot (z, r', \{r''\}, 0, \emptyset) \cdot rs_2) \ast [\text{send}(z)]_1^{\tau_1} * [\text{change}(z, r'')]_1^{\tau_1}
\]

**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
\[ \{ \text{send}(x, P) \ast \bigotimes_{e \in E} e \prec x \ast \bigotimes_{l \in L} x \prec l \} \]

\[ \text{extend}(x) \{ \text{ // Frame off order predicates} \]  

\text{chan} \ast x = \text{new}(\text{chan}); \ z->\text{flag} = 0;

\text{z->prev} = x->\text{prev}; \ // \text{ Stable as thread holds exclusive} [\text{send}(x)] \text{ permission.}

\[ \exists r_1, r_2, x'. r_2 \xrightarrow{1/2} P \ast [\text{send}(x)]^r_1 \ast \{ \text{prev} = x'; \text{flag} = 0 \} \]

\[ \text{oreg}(r_1, \{ \text{Chain}(r_2) \mid \text{wf}(r_2) \land \text{rs}(x) = (x, r_2, _, 0, _) \land (x, x') \in \text{seq}(r_2) \land z \notin r_2) \} \]

\text{skip; \ // Make a pair of stored propositions, appeal to Lemma 6.5.}

\[ \exists r_1, r_2, x', r', r'', r_2 \xrightarrow{1/2} P \ast [\text{send}(x)]^r_1 \ast z \rightarrow \{ \text{prev} = x'; \text{flag} = 0 \} \ast r' \xrightarrow{1/2} Q \ast r'' \xrightarrow{1/2} Q \ast \{ \text{prev} = x'; \text{flag} = 0 \} \ast \text{resource}\{r''\}, \{r'\} \]

\text{Create a node in the chain.}

\[ \exists r_1, r_2, x', r', r'', r_2 \xrightarrow{1/2} P \ast [\text{send}(x)]^r_1 \ast \{ \text{prev} = x'; \text{flag} = 0 \} \ast \text{resource}\{r''\}, \{r'\} \]

\[ \text{Enter the region, split the chain up.} \]

\[ \exists r_1, r_2, x', r', r'', r_2, h, r_1, r_2, r_2 \xrightarrow{1/2} P \ast [\text{send}(x)]^r_1 \ast \text{chains}(r_2) \ast x \rightarrow \{ \text{prev} = x'; \text{flag} = 0 \} \ast \text{resource}\{r''\}, \{r'\} \]

\[ \text{Fold the chains predicate back up.} \]

\[ \exists r_1, r_2, x', r', r'', h, r_1, r_2, r_2 \xrightarrow{1/2} P \ast [\text{send}(x)]^r_1 \ast \text{chains}(r_2) \ast x \rightarrow \{ \text{prev} = x'; \text{flag} = 0 \} \ast \text{resource}\{r''\}, \{r'\} \]

\[ \text{Get} [\text{send}(z)] \ast [\text{change}(z, r'')] \text{ from unused}(r_1, r_2) \text{ using Lemma 6.6.} \]

\[ \{ \text{send}(z, Q) \ast \text{recv}(z, Q) \ast \text{send}(x, P) \ast z \prec x \ast \bigotimes_{e \in E} e \prec z \ast \bigotimes_{l \in L} x \prec l \} \]

\[ \text{Fig. 17. Sketch-proof of extend with out-of-order signalling.} \]
is stable because we hold the \texttt{send} tokens on \( x \) and \( z \), meaning these channels cannot be extended or renounced.

\textit{Proving newchan.} Omitted: this proof is similar to \texttt{extend}, but simpler.

6.3. Proving Renunciation and Splitting Axioms

\textbf{Renunciation.} The axiom is defined as follows:

\[
\{ \text{recv}(x, P) \} \times \{ \text{send}(y, Q) \} \times x < y \quad \langle \text{skip} \rangle \quad \{ \text{send}(y, P \to Q) \}
\]

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 \texttt{send} predicate (line 14).

In order to conjoin the regions arising from the \texttt{send}, \texttt{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:

\textbf{Lemma 6.7.}

\[
\{ \text{oreg}(r, \{ \text{Chain}(rs) | x \in rs \}) \} \times \{ \text{oreg}(r', \{ \text{Chain}(rs') | x \in rs' \}) \} \quad \langle \text{skip} \rangle \quad \{ r = r' \}
\]

\textbf{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 [\texttt{chain}] and [\texttt{send}] permissions are held for \( x \) and \( y \) respectively.

When we push the renounced resource into the \texttt{resource} predicate (line 12) we use the following lemma to show that the renunciation set \( \mathcal{W} \) is updated appropriately:

\textbf{Lemma 6.8.} \quad \text{resource}(\mathcal{I}, \mathcal{W} \cup \{ r \}) \times r \xrightarrow{1/2} Q \times r' \xrightarrow{1/2} P

\[\exists r''. \text{resource}(\mathcal{I}, \mathcal{W} \cup \{ r', r'' \}) \times r'' \xrightarrow{1/2} (P \to Q)\]

\textbf{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 \( P \to Q \). 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 \texttt{unused}. Furthermore, the remainder of the chain stays the same. The resulting chain is trivially stable because the exclusive \texttt{send} token is held.

\textbf{Splitting.} The axiom is defined as follows:

\[
\{ \text{recv}(a, P) \} \times [P] \to (P_1 \times P_2) \} \quad \langle \text{skip} \rangle \quad \{ \text{recv}(a, P_1) \times \text{recv}(a, P_2) \}
\]

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.

\textbf{Lemma 6.9.} \quad r_2 \in rs(x) \mathcal{I} \wedge r_2 \xrightarrow{1/2} P \times [P] \to (P_1 \times P_2) \times \text{chainres}(rs)

\[\exists rs', r_3, r_4, r_3 \notin rs \wedge rs' = rs \triangleleft_I (r_2) \cup \{ r_3, r_4 \} \wedge r_3 \xrightarrow{1/2} P_1 \times r_4 \xrightarrow{1/2} P_2 \times \text{chainres}(rs')\]
\begin{align*}
&\{\text{recv}(x, P) * \text{send}(y, Q) * x < y\} \\
&(\text{skip}); \quad \text{// Definition of \text{recv}, \text{send}, Lemma 6.7.} \\
&\exists r, r_2, r_3. (r) \triangleright \left(\{\text{Chain}(rs) \mid (y, x) \in \text{ord}(rs)\}\right) * \\
&\quad \left(\{\text{recv}(r, \{\text{Chain}(rs) \mid rs(x) = (\_, \_x) \land r_2 \in I \land \text{wf}(rs)\})\} * \\
&\quad \left(\{\text{skip} \quad \left(\{\text{Chain}(rs) \mid \text{wf}(rs) \land \text{rs}(y) = (r_3, \_, \_, \_0)\}\} * \\
&\quad \left(\{\text{send}(y)\}\right)\right)\right)\right) \right) \\
&\text{// Conjunction on regions. Stable because of the permissions held.} \\
&\exists r, r_2, r_3, r_2 \triangleright (\{\text{send}(y)\}) * \\
&\quad \left(\{\text{Chain}(rs) \mid rs(x) = (\_, \_x) \land r_2 \in I \land \text{wf}(rs)\}\right) \\
&\quad \left(\{\text{skip} \quad \left(\{\text{Chain}(rs) \mid \text{wf}(rs) \land \text{rs}(y) = (r_3, \_, \_, \_0)\}\} * \\
&\quad \left(\{\text{send}(y)\}\right)\right)\right)\right) \\
&\text{// Open the region, drop \triangleright using AFrame.} \\
&\exists r, r_2, r_3, r_2 \triangleright (\{\text{send}(y)\}) * \\
&\quad \left(\{\text{Chain}(rs) \mid rs(x) = (\_, \_x) \land r_2 \in I \land \text{wf}(rs)\}\right) \\
&\quad \left(\{\text{skip} \quad \left(\{\text{Chain}(rs) \mid \text{wf}(rs) \land \text{rs}(y) = (r_3, \_, \_, \_0)\}\} * \\
&\quad \left(\{\text{send}(y)\}\right)\right)\right)\right) \\
&\text{// Pull out node.} \\
&\exists r, r_2, r_3, r_2 \triangleright (\{\text{send}(y)\}) * \\
&\quad \left(\{\text{Chain}(rs) \mid rs(x) = (\_, \_x) \land r_2 \in I \land \text{wf}(rs)\}\right) \\
&\quad \left(\{\text{skip} \quad \left(\{\text{Chain}(rs) \mid \text{wf}(rs) \land \text{rs}(y) = (r_3, \_, \_, \_0)\}\} * \\
&\quad \left(\{\text{send}(y)\}\right)\right)\right)\right) \\
&\text{// Apply Lemma 6.8 to supply the resource} \\
&\exists r, r_2, r_3, r_2 \triangleright (\{\text{send}(y)\}) * \\
&\quad \left(\{\text{Chain}(rs) \mid rs(x) = (\_, \_x) \land r_2 \in I \land \text{wf}(rs)\}\right) \\
&\quad \left(\{\text{skip} \quad \left(\{\text{Chain}(rs) \mid \text{wf}(rs) \land \text{rs}(y) = (r_3, \_, \_, \_0)\}\} * \\
&\quad \left(\{\text{send}(y)\}\right)\right)\right)\right) \\
&\text{// Close the chain.} \\
&\exists r, r_2, r_3, r_2 \triangleright (\{\text{send}(y)\}) * \\
&\quad \left(\{\text{Chain}(rs) \mid \text{wf}(rs) \land \text{rs}(y) = (r_3, \_, \_, \_0)\}\right) \\
&\text{// Close region using the renun transition.} \\
&\exists r, r_2 \triangleright (\{\text{send}(y)\}) * \\
&\quad \left(\{\text{Chain}(rs) \mid \text{wf}(rs) \land \text{rs}(y) = (r_3, \_, \_, \_0)\}\right) \\
&\text{// Definition of \text{send}.} \\
&\{\text{send}(y, P) * Q\} \\
\end{align*}

Fig. 18. Proof of the renunciation axiom for out-of-order implementation.
\[
\left\{ \text{recv}(x, P) \ast \left[ [ P ] \rightarrow (P_1 \ast P_2) \right] \right\}
\]

// Definition of recv.
\[\exists r_1, r_2. \text{oreg}(r_1, \{ \text{Chain}(rs) \mid rs(x) = (\_ , I , \_ , \_ , \_ \_ \_ ) \wedge r_2 \in I \wedge \text{wf}(rs) \})\]
\[\exists r_2 \frac{1}{2} \Rightarrow P \ast \left[ \text{change}(x, r_2) \right]_{I_1} \ast \left[ [ P ] \rightarrow (P_1 \ast P_2) \right] \]

// Enter the region.
\[\exists r_1, r_2. \frac{1}{2} \Rightarrow P \ast \left[ \text{change}(x, r_2) \right]_{I_1} \ast \left[ [ P ] \rightarrow (P_1 \ast P_2) \right] * \]
\[\text{chainds}(rs) \ast \text{chainres}(rs_1) \ast \text{resource}(I \cup \{r_2\}, W) \ast \text{chainres}(rs_2) \wedge \]
\[rs = rs_1 \cdot (x, r, I \cup \{r_2\}, f, W) \cdot rs_2 \wedge \text{unused}(r_1, rs) \]

(skip) // AFrame to drop \( \triangleright \), Lemma 6.9 to split the region.
\[\exists r_1, r_2, r_3, r_4, r_3, r_4 \notin rs \wedge r_3 \frac{1}{2} \Rightarrow P_1 \ast r_4 \frac{1}{2} \Rightarrow P_2 \ast \left[ \text{change}(x, r_2) \right]_{I_1} \ast \]
\[\left( \text{chainds}(rs) \ast \text{chainres}(rs_1) \ast \text{resource}(I \cup \{r_3, r_4\}, W) \ast \text{chainres}(rs_2) \right) \ast \]
\[\text{unused}(r_1, rs) \wedge rs = rs_1 \cdot (x, r, I \cup \{r_2\}, f, W) \cdot rs_2 \]

// Close chain, pull \( \text{change} \) perms out of unused.
\[\exists r_1, r_2, r_3, r_4, r_3 \Rightarrow P_1 \ast r_4 \Rightarrow P_2 \ast \left[ \text{change}(x, r_4) \right]_{I_1} \ast \left[ \text{change}(x, r_4) \right]_{I_1} \ast \]
\[\left( \text{chainds}(rs) \ast \text{chainres}(rs_1) \ast \text{resource}(I \cup \{r_3, r_4\}, W) \ast \text{chainres}(rs_2) \right) \ast \text{unused}(r_1, rs') \wedge \]
\[rs = rs_1 \cdot (x, r, I \cup \{r_2\}, f, W) \cdot rs_2 \wedge rs' = rs_1 \cdot (x, r, I \cup \{r_3, r_4\}, f, W) \cdot rs_2 \]

// Close region using the split transition.
\[\exists r_1, r_2, r_3, r_4, r_3 \Rightarrow P_1 \ast r_4 \Rightarrow P_2 \ast \left[ \text{change}(x, r_3) \right]_{I_1} \ast \left[ \text{change}(x, r_4) \right]_{I_1} \ast \]
\[\left( \text{oreg}(r_1, \{ \text{Chain}(rs) \mid rs(x) = (\_ , I , \_ , \_ , \_ \_ \_ ) \wedge r_3, r_4 \in I \wedge \text{wf}(rs) \}) \right) \]

// Definition of recv.
\[\{ \text{recv}(x, P_1) \ast \text{recv}(x, P_2) \} \]

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
typedef struct chan_hdr chan_hdr;

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

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

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

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

wait(chan_addr x){
  chan_addr a = x;
  while (a.hdr != NULL) {
    nh->up = x;
    nh->loff = 1;
    nh->hdr = nh;
    nh->off = 0;
    a = a.hdr->up;
    a.off++;
  }
}

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
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:

![Diagram showing 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 $\text{CNode}$, identical to the one used in the previous section, except a location $\text{loc}$ is now a pair $\langle h, o \rangle$ consisting of a physical address and an offset (this corresponds to the $\text{chan_addr}$ 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:

$$d: \text{Addr} \xrightarrow{\text{fin}} \{ \text{uhdr}: \text{Addr}; \text{uoff}: \text{Int}; \text{loff}: \text{Int}; \text{flags}: \{0..\text{MAX}\} \rightarrow \text{Bool} \}$$
A heap map records the chan hdr-typed objects forming the tree. To simplify the representation, we flatten the chan hdr-typed field up into the two fields uhdr and uoff. A location (x, i) is in d (written (x, i) ∈ d) if x ∈ dom(d) ∧ i ≤ d.loff.

For a given map d, chains defines the corresponding data-structure. In the following definition, we write x ↦→ v to indicate that x is immutable – shorthand for ∃f, x f v.

\[
\text{chains}(d) \triangleq \bigotimes_{x \in \text{dom}(d)}. \\
x.\text{up} \mapsto \{\text{chan hdr} = d(x).\text{uhdr}; \text{off} = d(x).\text{uoff} \} \land x.\text{loff} \mapsto (d(x).\text{loff}) \land \bigotimes_{i \in \{0..\text{MAX}\}}. x.\text{flags}[i] \mapsto (d(x).\text{flags}(i)) \\
\]

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

\[
\begin{align*}
\text{child}_d(x, i, y) & \triangleq d(y).\text{uhdr} = x \land d(y).\text{uoff} = i \land 0 \leq i \leq d(x).\text{loff} \\
\text{leaf}_d((x, i)) & \triangleq (x, i) \in d \land \nexists y. d(y).\text{uhdr} = x \land d(y).\text{uoff} = i \\
\text{descend}_d((x, i), (y, j)) & \triangleq (x, i) = (y, j) \lor \text{descend}_d((x, i), (d(y).\text{uhdr}, d(y).\text{uoff})) \\
\text{isset}_d((x, i)) & \triangleq d(x).\text{flags}(i) = 1 \\
\text{allset}_d(x) & \triangleq \forall i. 0 \leq i \leq d(x).\text{loff} \implies \text{isset}_d((x, i)) \\
\end{align*}
\]

Intuitively, the child, 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 descend is well-defined. isset and allset respectively assert that a single address and a whole array have their flags set.

\[
\begin{align*}
\langle x, i \rangle <_d \langle y, j \rangle & \triangleq \exists z, ix, iy. \text{descend}_d((z, ix), (x, i)) \land \text{descend}_d((z, iy), (y, j)) \land ix > iy \\
a <_d b \triangleq a <_d b \lor \text{descend}_d(a, b) \lor \text{descend}_d(b, a) \\
\text{finalleaf}_d(x, y) & \triangleq \text{descend}_d(x, y) \land \text{leaf}_d(y) \land (\forall z. \text{descend}_d(x, z) \land \text{leaf}_d(z) \implies z <_d y) \\
\end{align*}
\]

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. finalleaf_d(x, y) indicates that y is the rightmost 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 superseded by y. For each x there exists at most one y, so we generally use finalleaf_d as a partial function, i.e. finalleaf_d(x, y) stands for the unique y such that 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.

\[
\text{wf}(d) \triangleq \exists r: \text{Addr}. \exists \tau: \text{dom}(d) \to \mathbb{N}. \\
\forall x, i, y, z. (x, i) \in d \implies \exists j. \text{descend}((r, j), (x, i)) \land \\
(\text{child}_d(x, i, y) \land \text{child}_d(x, i, z)) \implies y = z \land \\
(\text{child}_d(x, i, y) \land \text{isset}_d(x, i)) \implies \text{allset}_d(y) \land \\
d(x).\text{uhdr} = y \land y \neq \text{NULL} \implies \\
d(y).\text{defined} \land d(x).\text{uoff} \leq d(y).\text{loff} \land \tau(y) < \tau(x) \\
\]

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
Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.

Because of this, the predicate `the structure it requires.
\[\text{renun}_s(x, (rs, d), (rs', d')) \triangleq d' = d \land \text{renun}_s(x, rs, rs')\]

\[\text{set}_s(x, (rs, d), (rs', d')) \triangleq d' = d \land h \triangleright_{\text{flags}[o]} (1) \land \text{set}_c((h, a), rs, rs')\]

\[\text{ext}_s(x, (rs, d), (rs', d')) \triangleq rs = (rs_1 \cdot a \cdot rs_2) \land rs' = (rs_1 \cdot b \cdot c \cdot rs_2) \land a, flg = 0 \land a, loc = x \land \exists y, b = a[\text{loc} \rightarrow y] \land c, flg = 0 \land c, W = \emptyset \land (h, o) = a, loc \land (b, off = (h', 0) \land c, off = (h', 1) \land (d' = d \lor h' \lor \{ \text{uhdr} = h; woff = o; loff = 1; \text{flags} = \lambda _, 0\}) \lor (o = d(h), loff \land b, loc = a, loc \land c, loc = (h, o + 1) \land (d' = (d \land h, \triangleright_{loff} (a + 1)) \land h, \triangleright_{\text{flags}[o]} (0))\})\]

\[\text{split}_s(x, r, (rs, d), (rs', d')) \triangleq d = d' \land \text{split}_c(\text{finalleaf}_d(x), r, rs, rs')\]

\[\text{sat}_s(x, (rs, d), (rs', d')) \triangleq d = d' \land \text{sat}_c(\text{finalleaf}_d(x), rs, rs')\]

\[\text{get}_s(x, r, (rs, d), (rs', d')) \triangleq d = d' \land \text{get}_c(\text{finalleaf}_d(x), r, rs, rs')\]

\[T_s(\text{send}(x)) \triangleq \{(a, b) | \text{wf}(b) \land (\text{renun}_s(x, a, b) \lor \text{set}_s(x, a, b) \lor \text{ext}_s(x, a, b))\}\]

\[T_s(\text{change}(x, r)) \triangleq \{(a, b) | \text{wf}(b) \land (\text{split}_s(x, r, a, b) \lor \text{sat}_s(x, a, b) \lor \text{get}_s(x, r, a, b))\}\]

\[T_s(\text{mark}) \triangleq \{(rs, d), (rs', d')) | \text{wf}(rs, d') \land d' = d \land h \triangleright_{\text{flags}[a]} (1) \land \neg \text{leaf}_d((h, o))\}\]

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, \text{set} and \text{extend} correspond to changes to the underlying heap, and they thus have to be extensively altered. Furthermore, \text{signal} can mark summary bits that are not leaves in the tree; this is allowed by the transition \text{mark}. We use \([\text{mark}]^*_c\) as notation for \(\exists \pi. [\text{mark}]^*_c\) to represent non-exclusive ownership of the \text{mark} action. Finally, note \text{finalleaf} in the definition of \text{change}; this is needed because a leaf \(x\) may be converted by \text{extend} into a summary node representing \text{finalleaf}_d(x).

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

\textbf{Lemma 7.2.} \text{recv}(x, P) is stable.

\textbf{Proof.} Assume the initial state of the chain is \text{Chain}(rs, d) and that \text{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 \(\text{wf}(rs', d') \land r_2 \in rs'(\text{finalleaf}_d(x))I\).

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

\(\Box\)

7.3. Verifying \text{wait}, \text{signal}, \text{extend}
\[
\{ \text{send}(x, P) \mid [P] \}
\]

```c
signal(chan_addr x){
  int i; ret = FALSE;
  chan_addr a = x;
  \{
    a = x \land [P] \land \exists r_1, r_2. r_2 \equiv P * [\text{send}(x)]_1 \land
    \text{oreg}(r_1, \{ \text{Chain}(rs, d) \mid \text{wf}(rs, d) \land \text{leaf}(x) \land d(x.hdr).flags[x.off] = 0 \})
  \}
  while (a.hdr != NULL && !ret){
    // Loop invariant.
    \{
      \begin{align*}
        \{ \text{ret} \land & \text{emp} \} \lor \\
        \{ a = x \land [P] \land \exists r_1, r_2. r_2 \equiv P * [\text{send}(x)]_1 \land \text{mark}_1 \land \\
        \text{oreg}(r_1, \{ \text{Chain}(rs, d) \mid \text{wf}(rs, d) \land \text{leaf}(x) \land d(x.hdr).flags[x.off] = 0 \})
      \} \lor
      \{ \exists r_1. \text{mark}_1 \} \land \\
      \{ \text{oreg}(r_1, \{ \text{Chain}(rs, d) \mid \text{wf}(rs, d) \land \text{leaf}(x) \land d(x.hdr).flags[x.off] = 0 \})
      \} \lor
      \{ \exists r_1. \text{mark}_1 \} \land \\
      \{ \text{oreg}(r_1, \{ \text{Chain}(rs, d) \mid \text{wf}(rs, d) \land \text{leaf}(x) \land d(x.hdr).flags[x.off] = 0 \})
      \} \lor
      \{ \exists r_1. \text{mark}_1 \} \land \\
      \{ \text{oreg}(r_1, \{ \text{Chain}(rs, d) \mid \text{wf}(rs, d) \land \text{leaf}(x) \land d(x.hdr).flags[x.off] = 0 \})
      \}
    \}
    a = a.hdr->up; // Abstract garbage collect.
  }
  emp
}
```

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-summaring 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.**

\[
\text{set}_{s}(x,(rs,d),(rs',d')) \land \text{unused}(r,rs,d) \Rightarrow \text{unused}(r,rs',d')
\]

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 last\(_d\)(a) to stand for the last address in the array associated with a, i.e.
\((\text{a.hdr},d(\text{a.hdr}).\text{loff}))\).

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.

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.**

\[
\text{wf}(rs,d) \land z \in \text{dom}(rs) \land \text{descend}_d(x,z) \land \text{leaf}_d(z) \land \\
(\forall l. <_{d} x \land \text{leaf}_d(l) \Rightarrow \text{isset}_d(l)) \\
\Rightarrow \exists rs_1,rs_2.rs = rs_1 \cdot rs(z) \cdot rs_2 \land \text{ctrue}(rs(z) \cdot rs_2)
\]
\{ \text{recv}(x, P) \} \\
\text{wait(chan\_addr x) \{ \\
\text{chan\_addr a = x; }\\n\begin{align*}
& \exists r_1, r_2. r_2 \implies P \land \text{change(x, r_2)} \land a = x \\
& \text{oreg} \left( r_1, \left\{ \text{Chain}(rs, d) \mid \text{wf}(rs, d) \land r_2 \in \text{rs(finalleaf}_d(x)).I \right\} \right) \\
\end{align*} \\
\text{while (a.hdr \neq \text{NULL}) \{ }\\n\begin{align*}
& \exists r_1, r_2. r_2 \implies P \land \text{change(x, r_2)} \land a = x \\
& \text{oreg} \left( r_1, \left\{ \text{Chain}(rs, d) \mid \text{wf}(rs, d) \land r_2 \in \text{rs(finalleaf}_d(x)).I \land \\
& \quad \forall l. a <_d l < \frac{1}{d} x \land \text{leaf}_d(l) \Rightarrow \text{isset}_d(l) \right\} \right) \\
\end{align*} \\
\text{for(skip; a.off<=a.hdr->loff; a.off++) \{ }\\n\begin{align*}
& \exists r_1, r_2. r_2 \implies P \land \text{change(x, r_2)} \land a = x \\
& \text{oreg} \left( r_1, \left\{ \text{Chain}(rs, d) \mid \text{wf}(rs, d) \land r_2 \in \text{rs(finalleaf}_d(x)).I \land \\
& \quad \forall l. a <_d l < \frac{1}{d} x \land \text{leaf}_d(l) \Rightarrow \text{isset}_d(l) \right\} \right) \\
\end{align*} \\
\text{// Appeal to well-formedness.} \\
\begin{align*}
& \exists r_1, r_2. r_2 \implies P \land \text{change(x, r_2)} \land a = x \\
& \text{oreg} \left( r_1, \left\{ \text{Chain}(rs, d) \mid \text{wf}(rs, d) \land r_2 \in \text{rs(finalleaf}_d(x)).I \land \\
& \quad \forall l. a <_d l < \frac{1}{d} x \land \text{leaf}_d(l) \Rightarrow \text{isset}_d(l) \right\} \right) \\
\end{align*} \\
\text{// Apply Lemma 7.6 when incrementing a.off.} \\
\text{\} }\\n\text{// Stable because chain cannot be extended once all the flags have been set.} \\
\begin{align*}
& \exists r_1, r_2. r_2 \implies P \land \text{change(x, r_2)} \land a = x \\
& \text{oreg} \left( r_1, \left\{ \text{Chain}(rs, d) \mid \text{wf}(rs, d) \land r_2 \in \text{rs(finalleaf}_d(x)).I \land \\
& \quad \forall l. \text{last}_d(a) <_d l < \frac{1}{d} x \land \text{leaf}_d(l) \Rightarrow \text{isset}_d(l) \right\} \right) \\
\end{align*} \\
\text{// Case-split on whether d(a.hdr).uhdr = NULL, if so, apply Lemma 7.4.} \\
\text{// As } \forall x. \exists \frac{1}{d} x, \text{ well-formedness gives us } \forall x. \text{last}_d(a) <_d x. \\
\begin{align*}
& \exists r_1, r_2. r_2 \implies P \land \text{change(x, r_2)} \land a = x \\
& \text{oreg} \left( r_1, \left\{ \text{Chain}(rs, d) \mid \text{wf}(rs, d) \land r_2 \in \text{rs(finalleaf}_d(x)).I \land \\
& \quad d(a.hdr).uhdr \neq \text{NULL} \implies \forall l. \text{last}_d(a) <_d l < \frac{1}{d} x \land \text{leaf}_d(l) \Rightarrow \text{isset}_d(l) \\
& \quad \forall l. d(a.hdr).uhdr = \text{NULL} \implies \forall l. l < \frac{1}{d} x \land \text{leaf}_d(l) \Rightarrow \text{isset}_d(l) \right\} \right) \\
\end{align*} \\
\} \\
\text{Fig. 24. Sketch-proof of } \text{wait} \text{ with summarization (completed in Fig. 25).}
a = a.hdr->up;

```
\exists r_1, r_2. r_2 \frac{1}{2} \Rightarrow P \ast [\text{change}(x, r_2)]_{\text{ct}}^* \\
  \{ \text{oereg } \} \\
  \{ \text{Chain}(r_1, d) \}
```

```
\text{wf}(r_1, d) \land r_2 \in r_1(\text{finalleaf}_d(x)) \text{.} \mathcal{I} \land
\begin{align*}
\text{a.hdr} \neq \text{NULL} \implies \\
\forall l. a < \frac{1}{d} l \leq \frac{1}{d} x \land \text{leaf}(l) \Rightarrow \text{iset}(l) \\
\lor \\
\text{a.hdr} = \text{NULL} \implies \\
\forall l. l < \frac{1}{d} x \land \text{leaf}(l) \Rightarrow \text{iset}(l)
\end{align*}
```

```
a._off++; // Apply Lemma 7.6 again.

\exists r_1, r_2. r_2 \frac{1}{2} \Rightarrow P \ast [\text{change}(x, r_2)]_{\text{ct}}^* \\
  \{ \text{oereg } \} \\
  \{ \text{Chain}(r_1, d) \}
```

```
\text{wf}(r_1, d) \land r_2 \in r_1(\text{finalleaf}_d(x)) \text{.} \mathcal{I} \land
\begin{align*}
\text{a.hdr} \neq \text{NULL} \implies \\
\forall l. a < \frac{1}{d} l \leq \frac{1}{d} x \land \text{leaf}(l) \Rightarrow \text{iset}(l) \\
\lor \\
\text{a.hdr} = \text{NULL} \implies \\
\forall l. l < \frac{1}{d} x \land \text{leaf}(l) \Rightarrow \text{iset}(l)
\end{align*}
```

```
\text{Apply Lemma 7.5.}

\exists r_1, r_2. r_2 \frac{1}{2} \Rightarrow P \ast [\text{change}(x, r_2)]_{\text{ct}}^* \\
  \{ \text{oereg } \} \\
  \{ \text{Chain}(r_1, d) \}
```

```
\text{wf}(r_1, d) \land r_2 \in r_1(\text{finalleaf}_d(x)) \text{.} \mathcal{I} \land
\begin{align*}
\forall l. l < \frac{1}{d} x \land \text{leaf}(l) \Rightarrow \text{iset}(l) \\
\forall z. \text{leaf}(z) \land z \in \text{dom}(r_1) \land \text{descend}_d(x, z) \implies \\
\exists r_1, r_2. r_2 = r_1 \cdot r_2(z) \cdot r_2 \land \text{ctrue}(r_2(z) \cdot r_2)
\end{align*}
```

```
// Identical reasoning to chained implementation.
```

\text{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.** \(0 \leq d(a).\text{off} < d(a).\text{loff} \land \text{wf}(d) \land a \text{\texttt{\textcircled{~}off}} (\bullet + 1) <_d k <_d b \implies a <_d k <_d b\)

**Proof.** The result follows from the structure of the heap map and the definition of \(<_d\).

\[\square\]

**Proving extend.** A sketch-proof of \emph{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 \emph{unused} predicate representing unused permissions is preserved by extending the chain.
\{send(x,P) \odot e \in E e \prec x \odot l \in L x \prec l\}
extend(chan_addr x){
  // Frame off order predicates and unfold send.
  \{\exists_{r_1, r_2} r_2 \iff P * [send(x)]^{1}_1 * [mark]^{r_1}_r \}
  oreg(r_1, \{Chain(rs, d) \mid wf(rs, d) \land \text{leaf}_d(x) \land rs(x).flg = 0\})
  chan_addr nx, r;
  if(x.off == x.hdr->loff && x.off < MAX){
    \{\exists_{r_1, r_2} r_2 \iff P * [mark]^{r_1}_r * [send(x)]^{r_1}_1 * r' \iff Q * r'' \iff Q * \}
    oreg \left( r_1, \left\{ Chain(rs, d) \mid \begin{align*}
      &\text{wf}(rs, d) \land \text{leaf}_d(x) \land rs(x).flg = 0 \\
      &\land d(x.hdr).loff = x.off \land x.off < MAX
    \end{align*}\right. \right) \}
    x.hdr->flags[x.loff+1] = 0;
    x.hdr->loff++;
    // Transition relation step ext.
    \{\exists_{r_1, r_2} r_2 \iff P * r' \iff Q * r'' \iff Q * \}
    \left. \begin{align*}
      &\text{mark}^{r_1}_r * [send(x)]^{r_1}_1 * [send((x.hdr,x.woff+1))]^{r_1}_1 * [change((x.hdr,x.woff+1),r'')]^{r_1}_r * \\
      &\left\{ \begin{align*}
        &rs = rs_1 \cdot rs(x) \cdot (x.hdr,x.woff+1), r', \{r''\}, 0, 0, 0 \cdot rs_2 \land \\
        &\text{wf}(rs, d) \land \text{leaf}_d(x) \land rs(x).flg = 0 \land \text{leaf}_d(x.hdr,x.woff+1) \\
        &\land d(x.hdr).loff = x.off+1 \land x.off < MAX
      \end{align*} \right. \right) \}
    r_hdr = x.hdr; r.off = x.off + 1;
    nx = x;
    \{\exists_{r_1, r_2} r_2 \iff P * r' \iff Q * r'' \iff Q * \}
    \left. \begin{align*}
      &\text{mark}^{r_1}_r * [send(nx)]^{r_1}_1 * [send(r)]^{r_1}_1 * [change(r,r'')]^{r_1}_r * \\
      &\left\{ \begin{align*}
        &rs = rs_1 \cdot rs(nx) \cdot (r, r', \{r''\}, 0, 0) \cdot rs_2 \land \text{wf}(rs, d) \land \\
        &\text{leaf}_d(nx) \land rs(nx).flg = 0 \land \text{leaf}_d(r) \land rs(r).flg = 0
      \end{align*} \right. \right) \}
    } else {
      \{\exists_{r_1, r_2} r_2 \iff P * r' \iff Q * r'' \iff Q * \}
      oreg \left( r_1, \{Chain(rs, d) \mid \text{send}(x,P) \land rs(x).flg = 0\} \right)
      nh = malloc(chan_hdr);
      nh->up = x; nh->loff = 1;
      // Transition relation step ext.
      \{\exists_{r_1, r_2} r_2 \iff P * r' \iff Q * r'' \iff Q * \}
      \left. \begin{align*}
        &\text{mark}^{r_1}_r * [send((nh,0))]^{r_1}_1 * [send((nh,1))]^{r_1}_1 * [change((nh,1),r'')]^{r_1}_r * \\
        &\left\{ \begin{align*}
          &rs = rs_1 \cdot (rs(x) \downarrow_{loc} (nh,0)) \cdot (nh,1), r', \{r''\}, 0, 0) \cdot rs_2 \land \\
          &\text{wf}(rs, d) \land \text{leaf}_d((nh,0)) \land rs((nh,0)).flg = 0 \land \text{leaf}_d((nh,1)) \land \\
          &\text{rs}((nh,0)).flg = 0 \land \text{rs}((nh,1)).flg = 0
        \end{align*} \right. \right) \}
      nx.hdr = nh; nx.of = 0;
      r.hdr = nh; r.off = 1;
    }
  return \left( r, nx \right); \}
\{send(r,Q) \odot recv(r,Q) \odot send(nx,P) \odot r \prec nx \odot e \odot e \prec r \odot \}\}
Lemma 7.7.
\[
\left(\text{ext}_s(x, i), (r, s, d), (r', s'), d') \land \text{dom}(d') = \text{dom}(d) \land r' \notin r s \land \text{l}
\right)
\Rightarrow
\left(\text{unused}(r, r', s') \equiv \text{send}((x, i + 1)) \equiv \text{change}((x, i + 1), r') \equiv \text{true}
\right)
\]

Proof. Begin by observing that, by the definition of \text{ext}_s, \langle x, i + 1 \rangle \notin r s, and thus that:
\[
\text{unused}(r, s, d) \equiv \text{send}((x, i + 1)) \equiv \text{change}((x, i + 1), r') \equiv \text{true}
\]

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

Lemma 7.8.
\[
\left(\text{ext}_s((x, i), (r, s, d), (r', s'), d') \land \text{dom}(d') = \text{dom}(d) \equiv \{l\} \land r' \notin r s \land \text{l}
\right)
\Rightarrow
\left(\text{unused}(r, r', s') \equiv \text{send}((l, 0)) \equiv \text{change}((l, 1), r') \equiv \text{true}
\right)
\]

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

Lemma 7.9. \text{ext}(x, r s, d), (r s', d') \land \text{wf}(r s, d) \land \text{wf}(r s', d') \Rightarrow uC(r s, d) \subseteq uC(r s', d')

Proof. There are two cases for extension: in-place extension in the array, or creation of a new array. In the former case, \text{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 \text{finalleaf} to the new array, and all other \text{finalleaf} relationships are preserved.

Now pick a pair \langle x, i \rangle \in uC(r s, d). Extending the chain can't stop x from satisfying \text{finalleaf} or make any node higher than x satisfy \text{finalleaf}. Therefore x \in C(r s', d') after extension. The only alteration to renounced sets \mathcal{W} in r s' is to add a new empty set. Thus \exists y. r \in r s'(y) \mathcal{W}. Finally, both cases of extension preserve the promise sets \mathcal{I}, ensuring that r \in r s'(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.
\[
\text{unused}(r_1, r s, d) \equiv \text{change}(x, r_2) \equiv \text{split}(x, r s, d, r s', d') \equiv \text{used}\]
\[
\text{rs}(\text{finalleaf}_{d'}(x)) = (r, \mathcal{I} \equiv \{r_2\}, f, \mathcal{W}) \land r s' = r s \equiv \text{finalleaf}_{d'}(x) \equiv ((\mathcal{I} \equiv \{r_2\}) \equiv \{r_3, r_4\})
\]
\[
\Rightarrow \text{unused}(r_1, r s', d') \equiv \text{change}(x, r_3) \equiv \text{change}(x, r_4)
\]

Proof. By the definition of split, d = d', and element locations in rs are unchanged in rs'. Thus it holds that C(r s, d) = C(r s', d') and finalleaf_{d'}(x) = finalleaf_{d'}(x). From the
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 (protocol \( \cong \mathcal{P}(\ldots \times \text{protocol}) \)) 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 box\((i, P, \pi)\) and 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, fut\((i, P)\) is generally not stable when P is instantiated with an assertion that includes a protocol assertion, because 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.


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.


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 Mooyi Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In APPLAS.


Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. 2010. Verifying event-driven programs using ramified frame properties. In TLDI.


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


John Wickerson, Mike Dodds, and Matthew Parkinson. 2010. Explicit Stabilisation for Modular Rely-Guarantee Reasoning. In ECOOP.
A. SAVED PROPOSITIONS

A saved proposition $r \xRightarrow{π} P$ 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 $\{1st, 2nd\}$ and a single token state $tok$, and invariant map $I_{prp}$ and transition relation $T_{prp}$ defined as follows:

$$
I_{prp}(Q)(1st) \triangleq \text{emp}
$$
$$
I_{prp}(Q)(2nd) \triangleq Q
$$
$$
T_{prp}(tok) \triangleq \{(1st, 2nd)\}
$$

We then define the saved proposition $r \xRightarrow{π} Z \Rightarrow Q$ as follows:

$$
r \xRightarrow{π} Q \triangleq \text{region}(\{1st, 2nd\}, T_{prp}, I_{prp}(Q), r) \ast \text{[tok]}^π
$$

The fact that the representation transition state 1st is $\text{emp}$ 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].

$$
\text{region}(S, T, I, r) \ast \text{region}(S', T', J, r) = \Rightarrow (\triangledown I(s) \Rightarrow \triangledown J(s)) \quad (5)
$$
$$
\text{region}(S, T, I, r) \ast \text{region}(S', T', J, r) = \Rightarrow (\triangledown I(s) \neg \triangledown J(s)) \quad (6)
$$

The later modality, $\triangledown$, 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).**

$$
r \xRightarrow{π_1} P \ast r \xRightarrow{π_2} Q
$$

$$
\Rightarrow r \xRightarrow{π_1} P \ast \text{region}(\{1st, 2nd\}, T_{prp}, I_{prp}(P), r) \quad \text{region is duplicable.}
$$
$$
\ast r \xRightarrow{π_2} Q \ast \text{region}(\{1st, 2nd\}, T_{prp}, I_{prp}(Q), r)
$$

$$
\Rightarrow r \xRightarrow{π_1} P \ast r \xRightarrow{π_2} Q \ast (\triangledown (I_{prp}(P)(2nd)) \Rightarrow \triangledown (I_{prp}(Q)(2nd))) \quad \text{Property 5.}
$$
$$
\Rightarrow r \xRightarrow{π_1} P \ast r \xRightarrow{π_2} Q \ast (\triangledown P \Rightarrow \triangledown Q) \quad \text{defn of } I_{prp}
$$

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:

$$
r \xRightarrow{π_1} P \ast r \xRightarrow{π_2} Q \quad \Rightarrow (\triangledown P) \Rightarrow (\triangledown Q) \quad (7)
$$
The proof of Property (4) then goes as follows:

\[
\begin{align*}
r \xrightarrow{\pi_1} & P \rightarrow^r Q \rightarrow^* (X \rightarrow \triangleright (Q \triangleright Y)) \rightarrow^* (P \rightarrow Z) \\
\text{SMono} & \\
\Rightarrow & r \xrightarrow{\pi_1} P \rightarrow^r Q \rightarrow^* (X \rightarrow \triangleright (Q \triangleright Y)) \rightarrow^* (P \rightarrow Z) \\
\text{LBin, assume } \triangleright \text{ distributes over } \rightarrow & \\
\Rightarrow & r \xrightarrow{\pi_1} P \rightarrow^r Q \rightarrow^* (X \rightarrow ((Q \triangleright Y)) \rightarrow^* (P \rightarrow Z) \\
\text{Assume property 7} & \\
\Rightarrow & r \xrightarrow{\pi_1} P \rightarrow^r Q \rightarrow^* (X \rightarrow ((Q \triangleright Y)) \rightarrow^* (P \rightarrow Z) \\
\text{Transitivity of } \rightarrow & \\
\Rightarrow & r \xrightarrow{\pi_1} P \rightarrow^r Q \rightarrow^* (X \rightarrow ((Q \triangleright Y)) \rightarrow^* (P \rightarrow Z) \\
\text{Framing of } \rightarrow & \\
\Rightarrow & r \xrightarrow{\pi_1} P \rightarrow^r Q \rightarrow^* (X \rightarrow ((Q \triangleright Y)) \rightarrow^* (P \rightarrow Z) \\
\text{Transitivity of } \rightarrow & \\
\Rightarrow & r \xrightarrow{\pi_1} P \rightarrow^r Q \rightarrow^* (X \rightarrow ((Q \triangleright Y)) \rightarrow^* (P \rightarrow Z)) \\
\end{align*}
\]

\[\blacksquare\]

B. PROOFS FOR OUT-OF-ORDER SIGNALLING

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

**Lemma 6.1.**

\[
\text{resource}(\mathcal{I}, \emptyset) \subseteq \bigoplus\limits_{i \in \mathcal{I}} \exists Q : \text{Prop}. i \xleftrightarrow{1/2} Q(i) \rightarrow^* \triangleright Q(i)]
\]

**Proof.**

\[
\begin{align*}
W = \emptyset \land \exists Q : \mathcal{I} \rightarrow \text{Prop}. R : W \rightarrow \text{Prop}. \\
\bigotimes\limits_{i \in \mathcal{I}}. i \xleftrightarrow{1/2} Q(i) \rightarrow^* \bigotimes\limits_{w \in W}. w \xleftrightarrow{1/2} R(w) \\
* \left[ Q(i) \rightarrow^* \bigotimes\limits_{i \in \mathcal{I}}. Q(i) \right] \\
\text{Simplify using } W = \emptyset, \text{ garbage collect.} \\
\subseteq \exists Q : \mathcal{I} \rightarrow \text{Prop}. \bigotimes\limits_{i \in \mathcal{I}}. i \xleftrightarrow{1/2} Q(i) \rightarrow^* \bigotimes\limits_{i \in \mathcal{I}}. Q(i) \\
\text{Switch from } [-] \text{ to } [-], \text{ pull out } \bigotimes. \\
\subseteq \exists Q : \mathcal{I} \rightarrow \text{Prop}. \bigotimes\limits_{i \in \mathcal{I}}. i \xleftrightarrow{1/2} Q(i) \rightarrow^* \bigotimes\limits_{i \in \mathcal{I}}. Q(i) \\
\text{Push in the existential.} \\
\subseteq \bigotimes\limits_{i \in \mathcal{I}}. \exists Q : \text{Prop}. i \xleftrightarrow{1/2} Q(i) \rightarrow^* \bigotimes\limits_{i \in \mathcal{I}}. Q(i) \\
\end{align*}
\]

\[\blacksquare\]

**Lemma 6.2.**

\[
r \xleftrightarrow{1/2} P \rightarrow^* \text{resource}(\mathcal{I}, W \cup \{r\}) \subseteq \text{resource}(\mathcal{I}, W)
\]

\[\blacksquare\]
Proof.

\[
r \xrightarrow{\frac{1}{2}} R \ast [R] \ast \text{resource}(I, W \cup \{r\})
\]

Definition of resource.

\[
\exists Q : I \rightarrow \text{Prop}, R : W \cup \{r\} \rightarrow \text{Prop}.
\]

\[
\bigwedge_{i \in I} i \xrightarrow{\frac{1}{2}} Q(i) \ast \bigwedge_{w \in W \cup \{r\}} w \xrightarrow{\frac{1}{2}} R(w)
\]

\[
\bigwedge_{i \in I} (\triangleright \bigwedge_{w \in W \cup \{r\}} R(w)) \rightarrow \bigwedge_{i \in I} [Q(i)]
\]

Property (3), monotonicity of \(\triangleright\), monotonicity of \([-\).]

\[\exists P : \text{Prop}, r \xrightarrow{\frac{1}{2}} R \ast [\triangleright R] \ast ([\triangleright R] \implies [\triangleright P]) \]

\[
\exists Q : I \rightarrow \text{Prop}, R : W \rightarrow \text{Prop}.
\]

\[
r \xrightarrow{\frac{1}{2}} P \ast \bigwedge_{i \in I} i \xrightarrow{\frac{1}{2}} Q(i) \ast \bigwedge_{w \in W} w \xrightarrow{\frac{1}{2}} R(w)
\]

\[
\bigwedge_{i \in I} (\triangleright P \triangleright \bigwedge_{w \in W} R(w)) \rightarrow \bigwedge_{i \in I} [Q(i)]
\]

Modus ponens.

\[\exists P : \text{Prop}, r \xrightarrow{\frac{1}{2}} R \ast [\triangleright P] \ast (\triangleright P \implies \text{resource}(I, W)) \]

Lemma 6.3. \(\text{chainres}(rs) \wedge \text{wf}(rs) \wedge \text{cttrue}(rs)\)

\[\exists rs', \text{chainres}(rs') \wedge \text{cconf}(rs') \wedge rs \xrightarrow{\mathcal{W}_s} rs' \wedge \text{wf}(rs')\]

Proof. We perform a sequence of smaller view-shifts corresponding to converting each \text{cnode} in turn, starting with the earliest element in the chain:

\[
P_0 \subseteq P_1 \subseteq P_2 \subseteq \ldots \subseteq P_n
\]

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:

\[
P_i \triangleq \exists rs'. \text{chainres}(rs[0, n - i] \cdot rs') \wedge \text{cconf}(rs')
\]

\[
\wedge rs[(n - i) + 1, n] \xrightarrow{\mathcal{W}_s} rs' \wedge \text{wf}(rs[0, (n - i)] \cdot rs')
\]

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:

$$\exists s'. \text{chainres}(rs[0, n - i] \cdot rs') \land \text{cconf}(rs') \land rs[(n - i) + 1, n] \xrightarrow{\mathcal{W}^s} rs' \land \text{wf}(rs[0, n - i] \cdot rs')$$

$$\subseteq$$

$$\exists s', s. \text{chainres}(rs[0, n - (i + 1)]) * \text{resource}(s, I, s, \mathcal{W}) * \text{chainres}(rs')$$

$$\land \text{cconf}(rs') \land rs[n - i, n] \xrightarrow{\mathcal{W}^s} s \cdot rs' \land \text{wf}(rs[0, n - (i + 1)]) \cdot s \cdot rs'$$

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 $s', I$ for an earlier element $s' \in rs'$. If $w$ is a member of $s', I$ for multiple $s'$, we pick the first such $s' \in rs'$. Since $\text{cconf}$ holds for $rs'$ it follows that $s'.I = \emptyset$. Thus, there exists $rs'_1, rs'_2$ and $s'$ such that $w \in s', I$, $s'.I = \emptyset$, $rs' = rs'_1 \cdot s' \cdot rs'_2$ and $\forall x \in rs'_1, w \notin x. I$. By the definition of $\text{resource}$, there exists a saved proposition $w \xrightarrow{1/2} R$ inside $\text{resource}(s, I, s, \mathcal{W})$. By the same definition, the saved proposition $w \xrightarrow{1/2} P$ and resource $\triangleright P$ must be included in $\text{resource}(s', I, \emptyset)$. We move the resource from one node to the other, and garbage collect the saved proposition:

$$\text{resource}(s, I, s, \mathcal{W}) * \text{chainres}(rs'_1) * \text{resource}(s', I, \emptyset) * \text{chainres}(rs'_2)$$

$$\subseteq$$

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

Let $rs''$ denote $rs'_1 \cdot s'[I \mapsto s' [1 \setminus \{w\}] \cdot rs'_2$. By definition of $\xrightarrow{\mathcal{W}}$, it thus follows that

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

Hence, by Lemma B.1 it follows that

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

The result is that the assertion is rewritten as follows:

$$\subseteq$$

$$\exists s', s. \text{chainres}(rs[0, n - (i + 1)]) * \text{resource}(s, I, s, \mathcal{W} \setminus \{w\}) * \text{chainres}(rs')$$

$$\land \text{cconf}(rs') \land rs[n - i, n] \xrightarrow{\mathcal{W}^s} s \cdot rs' \land \text{wf}(rs[0, n - (i + 1)]) \cdot s \cdot rs'$$

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.

\[\square\]

**Lemma B.1.**

$$\text{wf}(rs) \land rs \xrightarrow{\mathcal{W}} rs' \Rightarrow (\text{available}(rs) = \text{available}(rs') \land \text{wf}(rs'))$$

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

$$rs = rs_1 \cdot s_1 \cdot rs_2 \cdot s_2 \cdot rs_3,$$

$$w \in s_2. I,$$

$$s_1, s_2 \cdot rs_3,$$

$$rs' = rs_1 \cdot (s_1 \cup \mathcal{W} (\bullet \setminus w)) \cdot rs_2 \cdot (s_2 \cup \mathcal{W} (\bullet \setminus w)) \cdot rs_3$$

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

$$\text{available}(s_2 \cup \mathcal{W} (\bullet \setminus w)) \cdot rs_3) = (\text{available}(rs_3) \setminus s_2. I) \cup (s_2. I \setminus \{w\})$$

$$= \text{available}(s_2 \cdot rs_3) \setminus \{w\}$$
Since \( w \in s_1.W \) and \( \text{wf}(s_1 \cdot r s_2 \cdot s_2 \cdot r s_3) \) it follows that \( w \in \text{available}(r s_2 \cdot s_2 \cdot r s_3) \). Hence, since \( \forall s \in r s_2, w \notin s.I \) it follows that \( \forall s \in r s_2, w \notin s.W. \) Thus,

\[
\begin{align*}
\text{available}(s_1 \cdot (\bullet \setminus w)) \cdot r s_2 \cdot (s_2 \cdot (\bullet \setminus w)) \cdot r s_3 \\
= (\text{available}(r s_2 \cdot (s_2 \cdot (\bullet \setminus w)) \cdot r s_3) \setminus (s_1.W \setminus \{w\})) \cup s_1.I \\
= ((\text{available}(r s_2 \cdot s_2 \cdot r s_3) \setminus \{w\}) \setminus (s_1.W \setminus \{w\})) \cup s_1.I \\
= (\text{available}(r s_2 \cdot s_2 \cdot r s_3) \setminus s_1.W) \cup s_1.I \\
= \text{available}(s_1 \cdot r s_2 \cdot s_2 \cdot r s_3)
\end{align*}
\]

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

\[
\begin{align*}
s_1.W \setminus \{w\} \subseteq \text{available}(r s_2 \cdot (s_2 \cdot (\bullet \setminus w)) \cdot r s_3) \\
= \text{available}(r s_2 \cdot s_2 \cdot r s_3) \setminus \{w\}
\end{align*}
\]

and \( s_2.W \subseteq \text{available}(r s_3) \) both of which follow easily from the assumption that \( \text{wf}(rs) \). It remains to show that all sets \( I \) for the chain are pairwise disjoint, and likewise for all sets \( W \). As we have only removed identifiers, this is satisfied trivially. \( \square \)

**Lemma 6.4.** \( \text{resource}(I \cup r_2, \emptyset) \ast r_2 \xrightarrow{1/2} P \subseteq \text{resource}(I, \emptyset) \ast [\triangleright P] \)

**Proof.**

\[
\begin{align*}
\text{resource}(I \cup r_2, \emptyset) \ast r_2 \xrightarrow{1/2} P \\
\text{Lemma 6.1.}
\end{align*}
\]

\[
\begin{align*}
\text{Pull out resource for identifier} \ r_2. \\
\text{Property (3), monotonicity of} \ [\cdot]. \\
\text{Modus ponens, garbage collect.} \\
\text{Definition of} \ \text{resource.}
\end{align*}
\]

\( \square \)

**Lemma 6.7.**

\[
\{\text{oreg}(r, \{\text{Chain}(rs) \mid x \in rs\}) \ast \text{oreg}(r', \{\text{Chain}(rs') \mid x \in rs'\})\} \ast (\text{skip}) \ast \{r = r'\}
\]

**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. \( \square \)

**Lemma 6.8.** \( \text{resource}(I, W \cup \{r\}) \ast r \xrightarrow{1/2} S \ast r' \xrightarrow{1/2} T_1 \ ast \ (T_1 \ast S) \)

\[
\begin{align*}
\exists r''. \text{resource}(I, W \cup \{r', r''\}) \ast r'' \xrightarrow{1/2} (T_1 \ast S)
\end{align*}
\]
PROOF. First we construct a new saved proposition \( r'' \) such that \( r'' \Rightarrow (T_1 \Rightarrow S) \). Now it suffices to prove

\[
\text{resource}(\mathcal{I}, W \cup \{r\}) \ast r \xrightarrow{\frac{1}{2}} S \ast r' \xrightarrow{\frac{1}{2}} T_1 \ast r'' \xrightarrow{\frac{1}{2}} T_2 \land \text{valid}(T_1 \ast T_2 \Rightarrow S) \subseteq \text{resource}(\mathcal{I}, W \cup \{r', r''\})
\]

\[
r \xrightarrow{\frac{1}{2}} S \ast r' \xrightarrow{\frac{1}{2}} T_1 \ast r'' \xrightarrow{\frac{1}{2}} T_2 \land \text{valid}(T_1 \ast T_2 \Rightarrow S) \ast
\]

\[
\exists Q : \mathcal{I} \rightarrow \text{Prop}, R : W \cup \{r\} \rightarrow \text{Prop}.
\]

\[
\bigotimes_{i \in \mathcal{I}. i \xrightarrow{\frac{1}{2}} Q(i) \ast \bigotimes_{w \in W} \ast w \xrightarrow{\frac{1}{2}} R(w)
\ast \lfloor [\lfloor Q(i) \ast \bigotimes_{i \in \mathcal{I}. [Q(i)]} \rfloor \rfloor
\]

Rearrange

\[
\exists P : \text{Prop}, Q : \mathcal{I} \rightarrow \text{Prop}, R : W \cup \{r', r''\} \rightarrow \text{Prop}.
\]

\[
\text{valid}(R(r') \ast R(r'') \Rightarrow S) \land r \xrightarrow{\frac{1}{2}} S \ast r \xrightarrow{\frac{1}{2}} P
\ast \bigotimes_{i \in \mathcal{I}. i \xrightarrow{\frac{1}{2}} Q(i) \ast \bigotimes_{w \in W} \ast w \xrightarrow{\frac{1}{2}} R(w)
\ast \lfloor [\lfloor Q(i) \ast \bigotimes_{i \in \mathcal{I}. [Q(i)]} \rfloor \rfloor
\]

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

\[
\exists P : \text{Prop}, Q : \mathcal{I} \rightarrow \text{Prop}, R : W \cup \{r', r''\} \rightarrow \text{Prop}.
\]

\[
\text{valid}((R(r')) \ast (R(r'')) \Rightarrow \triangleright P) \land \bigotimes_{i \in \mathcal{I}. i \xrightarrow{\frac{1}{2}} Q(i) \ast \bigotimes_{w \in W} \ast w \xrightarrow{\frac{1}{2}} R(w)
\ast \lfloor [\lfloor Q(i) \ast \bigotimes_{i \in \mathcal{I}. [Q(i)]} \rfloor \rfloor
\]

By mono of \( [\_] \), \( \ast \) and \( \triangleright \).

\[
\exists Q : \mathcal{I} \rightarrow \text{Prop}, R : W \cup \{r', r''\} \rightarrow \text{Prop}.
\]

\[
\bigotimes_{i \in \mathcal{I}. i \xrightarrow{\frac{1}{2}} Q(i) \ast \bigotimes_{w \in W} \ast w \xrightarrow{\frac{1}{2}} R(w)
\ast \lfloor [\lfloor Q(i) \ast \bigotimes_{i \in \mathcal{I}. [Q(i)]} \rfloor \rfloor
\]

Rearrange

\[
\exists P : \text{Prop}, Q : \mathcal{I} \rightarrow \text{Prop}, R : W \cup \{r', r''\} \rightarrow \text{Prop}.
\]

\[
\bigotimes_{i \in \mathcal{I}. i \xrightarrow{\frac{1}{2}} Q(i) \ast \bigotimes_{w \in W} \ast w \xrightarrow{\frac{1}{2}} R(w)
\ast \lfloor [\lfloor Q(i) \ast \bigotimes_{i \in \mathcal{I}. [Q(i)]} \rfloor \rfloor
\]

Lemma B.2.

\[
r_2 \xrightarrow{\frac{1}{2}} P \ast [P \rightarrow (P_1 \ast P_2)] \ast r_3 \xrightarrow{\frac{1}{2}} P_1 \ast r_4 \xrightarrow{\frac{1}{2}} P_2 \ast \text{resource}(\mathcal{I} \cup \{r_2\}, W)
\subseteq \text{resource}(\mathcal{I} \cup \{r_3, r_4\}, W)
\]
Proof.

\[ r_2 \xrightarrow{1/2} P \star [\{P\} \rightarrow (P_1 \star P_2)] \star r_3 \xrightarrow{1/2} P_1 \star r_4 \xrightarrow{1/2} P_2 \star \text{resource}(\mathcal{I} \cup \{r_2\}, W) \]

Definition of resource.

\[ \exists Q: \mathcal{I} \cup \{r_2\} \rightarrow \text{Prop}, R: W \rightarrow \text{Prop.} \]

\[ \bigoplus_{i \in \mathcal{I}(r_2)} i \xrightarrow{1/2} Q(i) \star \bigoplus_{w \in W} w \xrightarrow{1/2} R(w) \]

* \((\triangleright \bigoplus_{w \in W} R(w) \triangleright \bigoplus_{i \in \mathcal{I}(r_2)} [Q(i)])\)

Pull out \(r_2\), \(\triangleright\) mono w.r.t. \(-\star\), property (4).

\[ r_2 \xrightarrow{1/2} P \star r_3 \xrightarrow{1/2} P_1 \star r_4 \xrightarrow{1/2} P_2 \]

\[ \exists Q: \mathcal{I} \cup \{r_2\} \rightarrow \text{Prop}, R: W \rightarrow \text{Prop.} \]

\[ \bigoplus_{i \in \mathcal{I}(r_3, r_4)} i \xrightarrow{1/2} Q(i) \star \bigoplus_{w \in W} w \xrightarrow{1/2} R(w) \]

* \((\triangleright \bigoplus_{w \in W} R(w) \triangleright \bigoplus_{i \in \mathcal{I}(r_3, r_4)} [Q(i)])\)

Fold \(r_3, r_4\) into \(\mathcal{I}\), weaken with [], garbage collect.

Definition of resource.

\[ \text{resource}(\mathcal{I} \cup \{r_3, r_4\}, W) \]

\[ \square \]

Lemma 6.9. \( r_2 \in rs(x) \mathcal{I} \land r_2 \xrightarrow{1/2} P \star [\{P\} \rightarrow (P_1 \star P_2)] \star \text{chainres}(rs) \)

\[ \exists r_3', r_4, r_3, r_4 \notin rs \land r_3' = rs \triangleright r_2 \triangleright \mathcal{I} (\bullet \setminus r_2) \cup \{r_3, r_4\} \land \]

\[ r_3 \xrightarrow{1/2} P_1 \star r_4 \xrightarrow{1/2} P_2 \star \text{chainres}(r_3') \]
Proof.

\[ r_2 \in rs(x).I \land r_2 \xrightarrow{1/2} P \ast [\neg P \rightarrow (P_1 \ast P_2)] \ast \text{chainres}(rs) \]

Make saved propositions, fresh by construction.

\[ r_2 \in rs(x).I \land r_2 \xrightarrow{1/2} P \ast [\neg P \rightarrow (P_1 \ast P_2)] \ast \text{chainres}(rs) \ast \]
\[ \exists r_3, r_4. r_3 \models P_1 \ast r_4 \models P_2 \land r_3, r_4 \notin rs \]
Pull out resource predicate for \( x \).

\[ \exists r_{s1}, r_{s2}, r_{r2} \in rs(x).I \land r_{s2} \xrightarrow{1/2} P \ast [\neg P \rightarrow (P_1 \ast P_2)] \ast \]
\[ \text{chainres}(r_{s1}) \ast \text{resource}(rs(x).I, rs(x).W \cup \{rs(x).res \mid rs(x).flg = 0\}) \ast \text{chainres}(r_{s2}) \ast \]
\[ \exists r_3, r_4, r_{r3} \models P_1 \ast r_4 \models P_2 \land r_3, r_4 \notin rs \land rs = r_{s1} \cdot rs(x) \cdot r_{s2} \]
Apply Lemma B.2.

\[ \exists r_{s1}, r_{s2}, r_{r3}, r_{r4} \in rs(x).I \land r_{s2} \xrightarrow{1/2} P \ast [\neg P \rightarrow (P_1 \ast P_2)] \ast \]
\[ \text{chainres}(r_{s1}) \ast \text{resource}(rs(x).I \setminus r_{s2} \cup \{r_{r3}, r_{r4}\}, rs(x).W \cup \{rs(x).res \mid rs(x).flg = 0\}) \ast \]
\[ \text{chainres}(r_{s2}) \ast r_{r3} \xrightarrow{1/2} P_1 \ast r_4 \xrightarrow{1/2} P_2 \land r_3, r_4 \notin rs \land rs = r_{s1} \cdot rs(x) \cdot r_{s2} \]
Definition of chainres.

\[ \exists r_{s'}, r_{r3}, r_{r4}, r_{r4} \notin rs \land r_{s'} = rs \Leftarrow I \setminus r_2 \cup \{r_{r3}, r_{r4}\} \land \]
\[ r_{r3} \xrightarrow{1/2} P_1 \ast r_4 \xrightarrow{1/2} P_2 \ast \text{chainres}(r_{s'}) \]

\[ \square \]