Bachelor's Thesis Computing Science

Verifying a Barrier using Iris

Simcha van Collem
s1040283

March 18, 2023

First supervisor/assessor:
dr. Robbert Krebbers

Second assessor:
prof. dr. Herman Geuvers

Radboud University

Abstract

A barrier is a low-level concurrency construct. One thread can *wait* until another thread *signals*. This allows the programmer to add deterministic behavior to their concurrent program, and is a common technique to eliminate data races. To formally reason about a program with barriers, one wants a modular verification of the barrier. This allows the programmer to use the barrier as a black box. Dodds et al., 2016 stated such a specification, and proved it correct for three different barrier implementations, using the iCAP separation logic (Svendsen & Birkedal, 2014). However, they did not have the possibility to mechanize their proofs using a proof assistant. In this thesis we mechanize the verification of the first two barrier implementations, using the Iris separation logic (Jung et al., 2018), in the Coq proof assistant. We present a different verification and to achieve this we develop two new constructs: *authoritative ordering* and *recursively nested wands*. The former keeps track of the order of elements in a list via *ghost state*, while the latter is an essential part of the resource invariant.

Contents

Chapter 1

Introduction

Tony Hoare was one of the first to provide a program logic to reason about imperative programs, in what we now call Hoare logic (Hoare, 1969). In Hoare logic we have Hoare triples $\{P\}\, e\, \{Q\}$, which should intuitively be read as the following: if precondition P holds and program e terminates, then postcondition Q holds. Although this logic worked great for simple imperative languages, for many decades, there did not seem to be an extension to languages with a heap to store values. In the early 2000s, Peter O'Hearn, John Reynolds and Hongseok Yang presented separation logic (O'Hearn et al., 2001; Reynolds, 2002). This extends Hoare logic with the idea of *ownership* of *resources*. For example, a program can *own* the points-to resource $\ell \mapsto v$, which asserts that value v is stored at location ℓ in the heap. Ownership of this resource allows us to verify read and write operations to ℓ, with the guarantee that ℓ does not get updated by another program.

A few years later, separation logic was extended to reason about concurrent programs (Brookes, 2007; O'Hearn, 2007). Ownership of resources is now given to threads, allowing a thread to reason about a piece of memory with the guarantee that no other thread changes its content. This allows for verification of a single thread without one having to worry about interference from other threads. However, in concurrent programs, it is common that there is some form of communication between threads. One way to do this, is via the use of barriers. A programmer can use a barrier to let one thread *wait* until another thread *signals* the first thread to continue. As an example, let us consider the following program.

$$\begin{array}{c} \texttt{let}\,\ell = \texttt{ref}(37)\,\texttt{in} \\ \texttt{let}\,b = \texttt{new\_barrier}\,()\,\texttt{in} \\ \left(\begin{array}{l||l} \texttt{wait}\,b; & \texttt{let}\,n = !\,\ell\,\texttt{in} \\ \texttt{let}\,n = !\,\ell\,\texttt{in} & \ell \leftarrow n+2; \\ \ell \leftarrow n+3 & \texttt{signal}\,b \end{array}\right) \end{array}$$

The program starts by storing 37 on the heap and returning its location (i.e. a pointer), after which a new barrier b is created. It then executes the two programs separated by the double line in separate threads. The left thread first has to wait until the barrier is signalled. After that, it reads the current value from ℓ and increments it by 3. The right thread first reads the current value, then increments it by 2, and finally signals the barrier. The left thread can thus only update ℓ after the right thread has finished executing.

To verify programs like these, we need an abstract specification for the barrier. A client can then use this during verification, without bother thinking about the barrier's implementation details, which typically involve low-level atomic operations. Dodds et al. (2011) presented such a specification which they verified using *Concurrent Abstract Predicates* (CAP) (Dinsdale-Young et al., 2010), a separation logic designed to reason about concurrent modules in a abstract way. They later strengthened the specification in (Dodds et al., 2016). They verified three different barrier implementations using *impredicative Concurrent Abstract Predicates* (iCAP) (Svendsen & Birkedal, 2014), a descendant from CAP which furthermore allows to reason about higher-order code.

However, the verification of their second barrier implementation requires a lot of bookkeeping, and has not been mechanized in a proof assistant. Furthermore, there has been a lot of follow-on work on iCAP. Iris (Jung et al., 2018) is one such separation logic, which is implemented and verified in the Coq proof assistant (Krebbers et al., 2017). This allows a user to verify their programs in Coq, which was not possible for Dodds et al.

In this thesis, we verify the first two barrier implementations discussed in (Dodds et al., 2016). Our verification is fully formalized in Coq using Iris. The verification of the first barrier has already been done in Iris (Jung et al., 2016, §4). However, our verification uses a different technique compared to Dodds et al. and Jung et al., which we elaborate in Chapter 6, and also serves as a stepping stone to the more complex verification of the second barrier. The invariant used in the verification of the second implementation is significantly different from the one presented by Dodds et al. Finally, we add a new `clone` operation to the second barrier implementation of Dodds et al., and extend the specification with a rule for `clone`.

We argue that our verification of the second barrier implementation is cleaner compared to Dodds et al. To achieve this we design a suitable notion of logical state for the barriers. This allows us to define a different resource invariant using a *recursively nested wands* construction, which is, to the best of our knowledge, a new construction. Using this invariant, we are able to eliminate the bookkeeping steps used by Dodds et al., resulting in cleaner proofs. Finally, we design a construction to keep track of an ordering using ghost state, called *authoritative ordering*.

It has to be noted that there are some limitations to our work. First, we implement the barriers in HeapLang. This is a ML-like language part of Iris, mainly used for program verification. It has some low-level constructs allowing for fine-grained concurrency, but it is quite different from C, which would be a more common language to implement low-level constructs like barriers. Second, we assume a sequentially consistent memory model, as opposed to a relaxed memory model which is used by multiprocessors nowadays. However, these limitations are regular in this field of study.

We start this thesis by introducing HeapLang, the programming language we use in this thesis, and give an introduction to separation logic in Chapter 2. In Chapter 3, we verify the simple barrier implementation given in (Dodds et al., 2016, Section 4). This chapter mainly serves as an introduction to Iris. Chapter 4 discusses the barrier implementation that supports chains (Dodds et al., 2016, Section 6), and an intuitive verification is presented. The formalization of this verification is discussed in Chapter 5. Finally, we discuss related work in Chapter 6, and we conclude in Chapter 7 with future work.

The Coq formalization, accompanied with installation instructions and references to the relevant section, can be found here (van Collem, 2023).

Chapter 2

Preliminaries

In this chapter we give a brief introduction to separation logic and HeapLang. We start by presenting Iris's HeapLang (§2.1), a programming language used in the rest of this thesis. We then proceed by giving a short introduction to separation logic (§2.2), and conclude by showing how we can prove program specifications (§2.3). We do not discuss Iris specific features in this chapter, so readers familiar with separation logic may safely skim this chapter.

2.1 HeapLang

We start by introducing HeapLang. HeapLang is an untyped ML style language which supports higher-order heap (e.g. storing functions on the heap), concurrency and common atomic operations. It is a language which is primarily used to provide examples and show what Iris is capable of. There do, until now, not exist ways to run HeapLang outside of Coq.

We partially present the syntax of HeapLang. In some places we cut some corners to simplify the story. One can refer to (The Iris Team, 2022, §12) for a more thorough formalization of HeapLang.

$$\begin{aligned}
v ::=\ & () \mid z \mid \ell \mid \texttt{true} \mid \texttt{false} \mid \texttt{rec}\, f(x) := e \mid \ldots \qquad (z \in \mathbb{Z}, \ell \in \mathit{Loc}) \\
& \texttt{inl}(v) \mid \texttt{inr}(v) \mid \ldots \\
e ::=\ & v \mid x \mid e_1 e_2 \mid \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mid \\
& -e \mid e_1 + e_2 \mid \ldots \\
& \texttt{inl}(e) \mid \texttt{inr}(e) \mid (\texttt{match}\, e\, \texttt{with}\, \texttt{inl}(x) \Rightarrow e_1 \mid \texttt{inr}(y) \Rightarrow e_2\, \texttt{end}) \mid \\
& \texttt{ref}(e) \mid \texttt{Free}(e) \mid !\, e \mid e_1 \leftarrow e_2 \mid \texttt{CAS}(e_1, e_2, e_3) \mid \texttt{FAA}(e_1, e_2) \mid \\
& e_1 \mid\mid e_2 \mid \ldots
\end{aligned}$$

HeapLang is an untyped lambda calculus. It furthermore has recursive functions which we use to define non-recursive functions, let expressions and the sequence of two expressions:

$$\begin{aligned} \lambda x.\, e &\triangleq \texttt{rec}\ \_(x) := e \\ \texttt{let}\, x = e_1\, \texttt{in}\, e_2 &\triangleq (\lambda x.\, e_2) e_1 \\ e_1; e_2 &\triangleq \texttt{let}\ \_ = e_1\, \texttt{in}\, e_2 \end{aligned}$$

We also frequently use optionals, which we can encode in the following way:[1]

$$\begin{aligned} \texttt{none} &\triangleq \texttt{inl}(()) \\ \texttt{some}(e) &\triangleq \texttt{inr}(e) \end{aligned}$$

We furthermore use indentation for if-statements to indicate their scopes. For example, the code on the left should be parsed as the code on the right.

```
if $e_1$
   then $e_2$;
         $e_3$                  (if $e_1$ then ($e_2$; $e_3$) else ($e_4$; $e_5$)); $e_6$
   else $e_4$;
         $e_5$
$e_6$
```

HeapLang is also able to interact with the heap. We can use $\texttt{ref}(e)$ to store a value on the heap. It returns a location ℓ which can be thought of as a pointer. Note that there is no "null pointer", but we can represent it as $\texttt{none}$, and all other pointers as $\texttt{some}(\ell)$. We use $\texttt{Free}(e)$ to free a location, $!\, e$ to read a value stored at a location, and $e_1 \leftarrow e_2$ to update the value stored at e_1 to e_2. These four operations are atomic, meaning that they do this in a single program step.[2] Apart from these heap operations, HeapLang also has two other atomic operations. $\texttt{CAS}(e_1, e_2, e_3)$ (compare and swap) compares the value stored at e_1 with e_2. When they are equal, e_1 gets updated to e_3 and $\texttt{true}$ is returned, otherwise it returns $\texttt{false}$. $\texttt{FAA}(e_1, e_2)$ (fetch and add) loads the value stored in e_1, adds e_2 to it and returns the original value. Finally, we can use $e_1 \mid\mid e_2$ to execute e_1 and e_2 in parallel and join them together when they are both finished.[3] It is called the *par* operation.

As HeapLang has support for multithreading, we need to talk about the semantics of how the threads interleave. Thread scheduling is completely

[1] The definition of $\texttt{none}$ looks a bit weird but we just pass $()$, the unit value, to the $\texttt{inl}$ constructor.

[2] Do note that $\ell \leftarrow f(n)$ first reduces $f(n)$ to a value, before storing it in ℓ. That means that $\ell \leftarrow f(n)$ is not atomic, while $\ell \leftarrow 1$ is.

[3] The par operation actually is not a primitive of HeapLang. Rather, it has $\texttt{fork}\ \{e\}$, which forks a new thread to execute e. We can implement par using $\texttt{fork}$. However, for our examples, it is easier to think of par as a primitive of the language.

nondeterministic meaning a programmer can make no assumptions about the order threads execute in.

We now present a few example programs, which we later give and prove specifications for.

Example 1 (Parallel execution)**.** We define a function `par_exec`, which takes a location ℓ. It then allocates a new location ℓ' where we store 37. It doubles the value stored in ℓ and adds 3 to the value stored in ℓ'. These operations are done in parallel. Finally it returns the newly allocated location ℓ'.

$$\begin{array}{l} \texttt{par\_exec} \triangleq \lambda \ell.\, \texttt{let}\, \ell' = \texttt{ref}(37)\, \texttt{in} \\ \qquad\qquad\qquad \left(\begin{array}{l} \texttt{let}\, n = !\,\ell\, \texttt{in} \\ \ell \leftarrow 2 \cdot n \end{array} \middle\| \begin{array}{l} \texttt{let}\, m = !\,\ell'\, \texttt{in} \\ \ell' \leftarrow m + 3 \end{array} \right) \\ \qquad\qquad\qquad \ell' \end{array}$$

Example 2 (List append)**.** Before we can write a function to append two lists, we first need a way to encode lists in HeapLang. There are many possible ways to do this, but we use a representation which is sort of analogous to linked lists in C-style languages. An empty list is represented by `none`, and a list of values $a :: \vec{x}$ is represented by $\texttt{some}(\ell)$, such that the value stored at ℓ is a pair containing x and the representation of $\vec{x}$. Using this representation, we represent the list $[1,2]$ by $\texttt{some}(\ell_1)$, where the value stored at ℓ_1 is $(1, \texttt{some}(\ell_2))$ and the value stored at ℓ_2 is $(2, \texttt{none})$. Now that we have a way to represent lists in HeapLang, we can define `append` as follows:

$$\begin{array}{l} \texttt{append} \triangleq \texttt{rec}\; \mathit{append}(v_1, v_2) := \texttt{match}\, v_1\, \texttt{with} \\ \qquad\qquad\qquad\qquad\qquad\qquad \begin{array}{ll} \;\;\texttt{none} & \Rightarrow v_2 \\ \mid \texttt{some}(\mathit{hd}) & \Rightarrow \texttt{let}\, x = \texttt{fst}(!\,\mathit{hd})\, \texttt{in} \\ & \quad\; \texttt{let}\, \mathit{xs} = \texttt{snd}(!\,\mathit{hd})\, \texttt{in} \\ & \quad\; \texttt{let}\, v_1' = \mathit{append}(\mathit{xs}, v_2)\, \texttt{in} \\ & \quad\; \mathit{hd} \leftarrow (x, v_1'); \\ & \quad\; \texttt{some}(\mathit{hd}) \end{array} \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{end} \end{array}$$

If v_1 represents the empty list, then we can just return v_2. In the other case, we first read the values stored in the hd location. We then do a recursive call to append v_2 to the tail of our current list. Finally, we store the original value x together with the new tail, v_1', in hd, and return $\texttt{some}(\mathit{hd})$, the representation of our appended list.

Finally, it is good to stress that this language is not safe at all. We can write all kind of programs like $\texttt{true} \leftarrow \texttt{false}$, $()()$, which have undefined behavior. It is furthermore possible to write programs with use-after-free bugs:

$$\begin{array}{l} \texttt{let}\,\ell = \texttt{ref}(\texttt{true})\,\texttt{in} \\ \texttt{Free}(\ell); \\ \ell \leftarrow \texttt{false} \end{array}$$

This is of course undesirable. We thus want a version of Hoare logic which not only allows us to prove specifications for programs, but furthermore also allows us to argue that a program is safe.

2.2 Hoare and separation logic

We can use separation logic to reason about HeapLang programs. Separation logic is an extension of Hoare logic. We write Hoare triples $\{P\}\, e\, \{\Phi\}$ as specifications for programs. Intuitively, they represent the fact that (i) e is a safe program, and (ii) if precondition P holds and program e terminates with value v, then postcondition $\Phi(v)$ holds. We often write $\{P\}\, e\, \{v.\, Q\}$ as a shorthand for $\{P\}\, e\, \{\lambda v.\, Q\}$, and $\{P\}\, e\, \{Q\}$ as a shorthand for $\{P\}\, e\, \{\lambda\_.\, Q\}$ whenever we are not interested in the return value. Furthermore, it is good to stress that these Hoare triples only specify partial correctness of a program. Take for example the following recursive function which loops forever:

$$\texttt{diverge} \triangleq \texttt{rec}\; \mathit{diverge}(\_) := \mathit{diverge}\,()$$

We are able to prove the following specification for it:[4]

$$\{\mathsf{True}\}\, \texttt{diverge}\,()\, \{\mathsf{False}\}$$

This makes sense as $\texttt{diverge}\,()$ does not terminate, allowing us to state any postcondition we would like.

We now present the Iris propositions, which can be used for pre and postconditions.[5]

$$\begin{array}{rll} P ::= & \varphi \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid & \text{Higher order logic} \\ & \ell \mapsto v \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \{P\}\, e\, \{\Phi\} \mid & \text{Separation logic} \\ & \triangleright P \mid \Box P \mid \mathrel{|\!\!\!\Rrightarrow} P \mid \boxed{P}^{\mathcal{N}} \mid \boxed{a}^{\gamma} \mid \ldots & \text{Iris} \end{array}$$

The first line corresponds with standard higher order logic. We let φ range over all propositions of the meta logic, Coq in this case. This allows us to, for example, reason about lists in our logic. The second line has the separation

[4] The proof uses Löb induction which we do not present until §3.3.

[5] We again cut a corner here by taking Hoare triples as a primitive of Iris, while they are actually defined inside the logic itself (Jung et al., 2018, §6).

logic primitives. The intuitive idea behind separation logic propositions is that they describe some resource. Proposition P holding corresponds to owning said resource. Because of this correspondence, we often refer to propositions as resources. Programs can only use and update resources they own. We shortly see an example of this when we discuss $\ell \mapsto v$. In the original separation logic papers (O'Hearn et al., 2001; Reynolds, 2002), resources were heap fragments, describing a finite part of the heap. In Iris it is also possible to own other kinds of resources,[6] some of which are shown on the third line. For simplicity sake, we only consider heap fragments as our type of resources for the rest of this chapter.

The *separating conjunction* $*$ has the following meaning: $P * Q$ holds whenever P and Q both hold and describe disjoint resources. In our case this corresponds to P and Q describing disjoint parts of the heap. We typically say that a program owns P and Q when it owns $P * Q$.

The proposition $\ell \mapsto v$ should be read as ℓ *points to* v and is often referred to as *the points-to predicate of* ℓ. It states that value v is stored at location ℓ in the heap. Only when a program owns $\ell \mapsto v$, it is allowed to read or update the value stored at ℓ. This is what makes sure that we cannot have use-after-free-bugs, as a program loses the points-to predicate of ℓ after freeing ℓ. In a similar way, this also ensures that a program cannot write to unallocated memory. Also note that if a program owns $\ell \mapsto v * \ell' \mapsto w$, then we know that $\ell \neq \ell'$ as both resources cannot describe the same part of the heap.

The *magic wand* $-\!\!*$ can be thought of as an implication for separation logic: whenever a program owns $P -\!\!* Q$ and combines it, using $*$, with a disjoint resource P, the program owns Q.

Hoare triples $\{P\}\, e\, \{\Phi\}$ are, as already discussed, used to state specifications for programs. Furthermore, P also describes a *footprint* of e. As e can only use and update resources it owns itself, we can be sure that it does not use any resources disjoint with P.

Finally the last line consists of propositions which are not present in the original separation logics (O'Hearn et al., 2001; Reynolds, 2002). We discuss them on demand whenever we come across them. The $\triangleright$ modality and invariants $\boxed{P}^{\mathcal{N}}$ are discussed in §3.3, and the $\square$ and $\mid\!\Rrightarrow$ modalities are discussed in §4.3. Ghost state ownership $\boxed{a}^{\gamma}$ is used in Chapter 5. One can refer to Jung et al., 2018, §3 for a discussion.

Now that we can describe resources, we can use them to reason about concurrent programs. Instead of saying that a program owns a resource, we assign ownership of resources to individual threads. We furthermore make sure that the resources owned by threads are all disjoint with each other. This allows us to reason locally about each thread. When a thread owns

[6] One can refer to (Jung et al., 2018, §3) for a more thorough discussion on resource ownership in Iris.

$\ell \mapsto v$, it can be sure that no other thread also owns a points-to predicate for the same location. Otherwise the two threads own resources describing intersecting heap fragments which cannot happen. Since a thread knows that it exclusively owns $\ell \mapsto v$, it can be sure that no other thread changes the value stored at ℓ, thus never invalidating $\ell \mapsto v$.

We conclude this section by discussing plausible specifications for the example programs from the previous section. We prove these specifications in the next section.

Example 3 (Parallel execution specification)**.** The `par_exec` function, which we defined in Example 1, has the following specification:

$$\textsc{ParExec-Spec} \\ \{\ell \mapsto k\}\ \texttt{par\_exec}(\ell)\ \{\ell'.\, \ell \mapsto (2 \cdot k) * \ell' \mapsto 40\}$$

In this case ℓ is a location and k an integer. We often leave these variables unbound, especially when it is clear from the context what their type should be.

This specification says that, if we own $\ell \mapsto k$ and execute $\texttt{par\_exec}(\ell)$, the function returns a location ℓ' and ownership of the resources $\ell \mapsto (2 \cdot k)$, $\ell' \mapsto 40$. As we own both resources, they should be disjunct, meaning that $\ell \neq \ell'$. This essentially tells us that the location ℓ' is fresh, i.e. a new location.

Example 4 (List append specification)**.** When we make a call $\texttt{append}(v_1, v_2)$, we should make sure that both v_1, v_2 are actual representations of lists of values. Otherwise the execution gets stuck at some point. We do this using an abstract predicate isList: $\mathit{Val} \to \mathit{List}(\mathit{Val}) \to \mathit{iProp}$. This predicate takes a value v and a list of values $\vec{x}$, and asserts whether the value v represents the list $\vec{x}$, according to the representation we discussed in Example 2. Before we discuss the definition of isList, let us look at how we can use it in the specification for `append`:

$$\textsc{Append-Spec} \\ \{\mathrm{isList}(v_1, \vec{x_1}) * \mathrm{isList}(v_2, \vec{x_2})\}\ \texttt{append}(v_1, v_2)\ \{v.\ \mathrm{isList}(v, \vec{x_1} \mathbin{++} \vec{x_2})\}$$

This specification states that, if we have two lists of values $\vec{x_1}, \vec{x_2}$, and two values v_1, v_2 representing these lists respectively, and we execute $\texttt{append}(v_1, v_2)$, the function returns a value v, which represents the list $\vec{x_1} \mathbin{++} \vec{x_2}$.

We define isList by recursion on the list, as follows:

$$\begin{aligned} \mathrm{isList}(v, [\,]) &\triangleq v = \texttt{none} \\ \mathrm{isList}(v, a :: \vec{x}) &\triangleq \exists \ell, v'.\, v = \texttt{some}(\ell) * \ell \mapsto (a, v') * \mathrm{isList}(v', \vec{x}) \end{aligned}$$

This corresponds with the intuitive representation we discussed in Example 2. The empty list is represented by `none` and a non empty list is represented by $\texttt{some}(\ell)$, where the value stored at ℓ is a pair containing the head of the list and the representation of the tail of the list.

2.3 Proving specifications

Until now we have seen how we can create programs and what kind of specifications we can write down. The final piece of the puzzle is to be able to prove specifications. We start by giving some general proof rules, after which we prove the specifications we discussed in Examples 3, 4.

Before we can give proof rules we need to present the *entailment* relation $\vdash$. Whenever we write $P \vdash Q$ we mean that if P holds, then Q also holds. Furthermore, we write $\vdash P$ when P holds without any assumptions, and $P \dashv\vdash Q$ when both $P \vdash Q$ and $Q \vdash P$. Although we present Hoare triples are first-class citizens of Iris, we write them without the entailment in from of them, for simplicity sake.

We start by introducing the rules for separating conjunction and the magic wand.

$$\textsc{Sep-Unit}\quad \mathsf{True} * P \dashv\vdash P \qquad \textsc{Sep-Weaken}\quad P * Q \vdash P \qquad \textsc{Sep-Comm}\quad P * Q \dashv\vdash Q * P$$

$$\textsc{Sep-Assoc}\quad P * (Q * R) \dashv\vdash (P * Q) * R \qquad \textsc{Sep-Mono}\quad \frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \qquad \textsc{Wand-Intro}\quad \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

$$\textsc{Wand-Elim}\quad \frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}$$

The unit resource of $*$ is True. We can use Sep-Weaken to throw away resources.[7] We furthermore have that the separating conjunction is both commutative and associative. If we want to prove a separating conjunction, we can use Sep-Mono to prove both parts separately by also splitting our assumptions. Finally, Wand-Intro and Wand-Elim describe that the magic wand can be seen as an implication for separation logic. These rules are regularly used when proving specifications. We typically use them implicitly, just like we normally do when using proof rules of the logical connectives in higher order logic. Iris of course also has the regular proof rules for higher order logic which we do not state here.

[7] One could notice that by having this rule, it is impossible to reason about memory leaks. Some separation logics therefore exclude this rule. A discussion on why this rule is present in Iris can be found in (Jung et al., 2018, §9). We discuss how this rule restricts a possible extension of the specification in Chapter 7.

We now discuss the following proof rules for Hoare triples:

$$\textsc{Hoare-Frame}\quad \frac{\{P\}\, e\, \{v.\, Q\}}{\{P * R\}\, e\, \{v.\, Q * R\}}$$

$$\textsc{Hoare-Conseq}\quad \frac{P \vdash P' \qquad \{P'\}\, e\, \{v.\, Q'\} \qquad \forall u.\, (Q'[u/v] \vdash Q[u/v])}{\{P\}\, e\, \{v.\, Q\}}$$

$$\textsc{Hoare-Pure}\quad \frac{\{P\}\, e_2\, \{v.\, Q\} \qquad e_1 \rightarrow_{\text{pure}} e_2}{\{P\}\, e_1\, \{v.\, Q\}} \qquad\qquad \textsc{Hoare-Val}\quad \{\mathsf{True}\}\, v\, \{w.\, w = v\}$$

The most important rule of separation logic is Hoare-Frame. Like we discussed earlier, P can be seen as a footprint of e. As e only uses those resources in P, we can be sure that a disjoint *frame* R, still holds after e has finished executing. This rule is what allows us to write specifications which only describe a small part of the heap, but instantiate them for any heap satisfying that precondition. We could for example use it to state another specification for par_exec:

$$\{\ell \mapsto k * \ell'' \mapsto v\}\, \texttt{par\_exec}(\ell)\, \{\ell'.\, (\ell \mapsto (2 \cdot k) * \ell' \mapsto 40) * \ell'' \mapsto v\}$$

As location ℓ'' is not used in par_exec, we can be sure that $\ell'' \mapsto v$ still holds in the postcondition. We often use this rule implicitly in proofs.

Hoare-Conseq is similar to the consequence rule from Hoare logic. It allows us to weaken the precondition and strengthen the postcondition. This rules is also used implicitly in proofs most of the times.

If we can do a pure program step from e_1 to e_2, i.e. one which does not involve using the heap, we can use Hoare-Pure to prove a Hoare triple for e_2. These pure program steps are defined like one would expect for a lambda calculus. The complete definition of $\rightarrow_{\text{pure}}$ is given in (The Iris Team, 2022, §12).[8]

Finally, Hoare-Val states that a program consisting of a single value returns that value.

We now give proofs of the specifications for par_exec, append we gave in Examples 3, 4. We give our proofs as decorated programs, accompanied with extra explanation where needed. In these decorated programs we write resources in green, code in blue and comments explaining intermediate steps in ML comment style: (* Comment *).

Example 5 (Proof of specification of par_exec)**.** Before we discuss the proof

[8] Note that they use a different notation for pure reductions, namely $\xrightarrow{\epsilon}_{\mathsf{h}}$.

$\{\ell \mapsto k\}$
$\mathtt{let}\ \ell' = \mathtt{ref}(37)\ \mathtt{in}$
(* Use Hoare-Alloc *)
$\{\ell \mapsto k * \ell' \mapsto 37\}$
(* Use Hoare-Par and give $\ell \mapsto k$ to the left thread,
 and $\ell' \mapsto 37$ to the right thread *)

$$\left(\begin{array}{l||l}
\{\ell \mapsto k\} & \{\ell' \mapsto 37\} \\
\mathtt{let}\ n = !\,\ell\ \mathtt{in} & \mathtt{let}\ m = !\,\ell'\ \mathtt{in} \\
\text{(* Use Hoare-Load *)} & \text{(* Use Hoare-Load *)} \\
\{n = k * \ell \mapsto n\} & \{m = 37 * \ell' \mapsto 37\} \\
\ell \leftarrow 2 \cdot n & \ell' \leftarrow m + 3 \\
\text{(* Use Hoare-Store *)} & \text{(* Use Hoare-Store *)} \\
\{n = k * \ell \mapsto 2 \cdot k\} & \{m = 37 * \ell' \mapsto 40\} \\
\text{(* Use Sep-Weaken *)} & \text{(* Use Sep-Weaken *)} \\
\{\ell \mapsto 2 \cdot k\} & \{\ell' \mapsto 40\}
\end{array}\right)$$

$\{\ell \mapsto (2 \cdot k) * \ell' \mapsto 40\}$
ℓ'
(* Use Hoare-Val *)
$\{\ell'.\, \ell \mapsto (2 \cdot k) * \ell' \mapsto 40\}$

Figure 2.1: Proof sketch of `par_exec`.

sketch given in Figure 2.1, we need the following proof rules:

Hoare-Par
$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 \,||\, e_2\, \{Q_1 * Q_2\}}$$

Hoare-Alloc
$$\{\mathsf{True}\}\ \mathtt{ref}(v)\ \{\ell.\, \ell \mapsto v\}$$

Hoare-Load
$$\{\ell \mapsto v\}\ !\,\ell\ \{w.\, w = v * \ell \mapsto v\}$$

Hoare-Store
$$\{\ell \mapsto v\}\ \ell \leftarrow w\ \{u.\, u = () * \ell \mapsto w\}$$

We discuss them on demand, whenever we come across a use of them.

We start with the precondition $\ell \mapsto k$. We then use Hoare-Alloc which gives us a fresh location together with its points-to predicate $\ell' \mapsto 37$. By framing $\ell \mapsto k$ implicitly, we end up with $\ell \mapsto k * \ell' \mapsto 37$. We now need to use Hoare-Par. It allows us to split our precondition and prove Hoare triples for both threads separately. We then get both postconditions of the individual threads, as a postcondition of the par operation. The reason that this rule is sound, is that if e_1, e_2 use disjoint resources, their execution order does not matter. We can thus focus on the individual threads. Their postconditions still describe disjunct resources, so we can safely take the separating conjunction of them. In our proof sketch, we only discuss the left thread, as the right thread has a completely analogous proof. We first use Hoare-Load to load the current value from ℓ. It says that the return value, is the value asserted by the points-to predicate. We furthermore get the points-to predicate back via the postcondition. We then use Hoare-Store to update the value stored at ℓ. It takes a points-to predicate as a precondition and gives it back with the value stored updated to w. We now know that $n = k * \ell \mapsto 2 \cdot k$ holds. As we do not need the fact that $n = k$ for the rest of the proof, we can use Hoare-Conseq and Sep-Weaken to forget this. Now that we have proven Hoare triples for both threads, we know that both their postconditions hold. Finally, the program returns ℓ' so we conclude by using Hoare-Val.

Example 6 (Proof of specification of `append`)**.** We now prove the following specification for `append`:

$$\{\text{isList}(v_1, \vec{x_1}) * \text{isList}(v_2, \vec{x_2})\}\ \texttt{append}(v_1, v_2)\ \{v.\ \text{isList}(v, \vec{x_1} \mathbin{++} \vec{x_2})\}$$

We prove this by induction on $\vec{x_1}$. Its proof sketch is given in Figure 2.2. We prove this specification by showing that it holds for both cases of the `match` statement. We first discuss the first case, where we know that $v_1 = \texttt{none}$. If we look at the definition of isList, which we gave in Example 4, we see that it has to be the case that $\vec{x_1} = \texttt{none}$. We can thus conclude that v_2 represents $\vec{x_1} \mathbin{++} \vec{x_2}$. We then use Hoare-Val, to get the desired postcondition. In the `some` case, we can reason in a similar way to the `none` case to conclude that $\vec{x_1} = a :: \vec{x_1'}$, for some value a and list of values $\vec{x_1'}$. If we then unfold isList, we get the value w_1, which represents $\vec{x_1'}$. We thus have $hd \mapsto (a, w_1) * \text{isList}(w_1, \vec{x_1'})$. We then use Hoare-Load twice to read the values of the pair stored in hd. We can use our induction hypothesis for the inductive call to *append*. We know $\text{isList}(w_1, \vec{x_1'}) * \text{isList}(v_2, \vec{x_2})$ and $\vec{x_1'}$ is the parameter we can use our induction hypothesis for. By our induction hypothesis, we know that $\text{isList}(v_1', \vec{x_1'} \mathbin{++} \vec{x_2})$ holds. We now use Hoare-Store to update the tail of our list. Finally, we fold the definition of isList and use Hoare-Val to conclude our postcondition.

```
$\{\mathrm{isList}(v_1, \vec{x_1}) * \mathrm{isList}(v_2, \vec{x_2})\}$
match $v_1$ with
  none      $\Rightarrow$ $\{v_1 = \texttt{none} * \mathrm{isList}(v_1, \vec{x_1}) * \mathrm{isList}(v_2, \vec{x_2})\}$
              (* $\vec{x_1} = []$ as $v_1 = \texttt{none}$ *)
              $\{\vec{x_1} = [] * \mathrm{isList}(v_2, \vec{x_2})\}$
              (* $\vec{x_2} = \vec{x_1} \mathbin{++} \vec{x_2}$ as $\vec{x_1} = []$ *)
              $\{\mathrm{isList}(v_2, \vec{x_1} \mathbin{++} \vec{x_2})\}$
              $v_2$
              (* Use Hoare-Val *)
              $\{v.\ \mathrm{isList}(v, \vec{x_1} \mathbin{++} \vec{x_2})\}$
| some($hd$) $\Rightarrow$ $\{v_1 = \texttt{some}(hd) * \mathrm{isList}(v_1, \vec{x_1}) * \mathrm{isList}(v_2, \vec{x_2})\}$
              (* $\vec{x_1} = a :: \vec{x'_1}$ as $v_1 = \texttt{some}(hd)$, unfold $\mathrm{isList}(v_1, \vec{x_1})$ *)
              $\left\{\vec{x_1} = a :: \vec{x'_1} * hd \mapsto (a, w_1) * \mathrm{isList}(w_1, \vec{x'_1}) * \mathrm{isList}(v_2, \vec{x_2})\right\}$
              let $x$ = fst(! $hd$) in
              (* Use Hoare-Load *)
              $\left\{x = a * hd \mapsto (a, w_1) * \mathrm{isList}(w_1, \vec{x'_1}) * \mathrm{isList}(v_2, \vec{x_2})\right\}$
              let $xs$ = snd(! $hd$) in
              (* Use Hoare-Load *)
              $\left\{xs = w_1 * hd \mapsto (a, w_1) * \mathrm{isList}(w_1, \vec{x'_1}) * \mathrm{isList}(v_2, \vec{x_2})\right\}$
              let $v'_1 = append(xs, v_2)$ in
              (* Induction hypothesis *)
              $\left\{hd \mapsto (a, w_1) * \mathrm{isList}(v'_1, \vec{x'_1} \mathbin{++} \vec{x_2})\right\}$
              $hd \leftarrow (x, v'_1);$
              (* Use Hoare-Store and fact that $x = a$ *)
              $\left\{hd \mapsto (a, v'_1) * \mathrm{isList}(v'_1, \vec{x'_1} \mathbin{++} \vec{x_2})\right\}$
              (* Fold isList *)
              $\left\{\mathrm{isList}(\texttt{some}(hd), a :: \vec{x'_1} \mathbin{++} \vec{x_2})\right\}$
              some($hd$)
              (* Use Hoare-Val *)
              $\left\{v.\ \mathrm{isList}(v, a :: \vec{x'_1} \mathbin{++} \vec{x_2})\right\}$
              (* Rewrite $\vec{x_1} = a :: \vec{x'_1}$ *)
              $\{v.\ \mathrm{isList}(v, \vec{x_1} \mathbin{++} \vec{x_2})\}$
end
$\{v.\ \mathrm{isList}(v, \vec{x_1} \mathbin{++} \vec{x_2})\}$
```

Figure 2.2: Proof sketch of append.

Chapter 3

Simple Barrier

In this chapter we give an implementation of a simple barrier (§3.1). This implementation is a direct port of the one given in (Dodds et al., 2016, Figure 8) and was also used in (Jung et al., 2016, Figure 9). We proceed by giving the specification (§3.2) presented in (Dodds et al., 2016, Section 4). Finally, we discuss how we could verify this specification (§3.3) and conclude with a proof sketch of the specification. Our verification and proof go in a different fashion than those in (Dodds et al., 2016; Jung et al., 2016), as we do not use *state transition systems*.[1] This chapter mainly serves as an Iris tutorial. Readers familiar with both Iris and barriers may want to skip this chapter.

3.1 Implementation

We start by giving an implementation for a simple barrier in HeapLang. We define 3 methods: `new_barrier` creates a new barrier, `signal` gets a barrier and signals it, and `wait` also gets a barrier and waits until the barrier is signalled. The code of this barrier is straightforward. We only need to store whether our barrier has been signalled yet. That means that `new_barrier` can return a reference to `false`, `signal` updates this reference to `true` and `wait` recursively calls itself until the reference is set to `true`.

$$\begin{aligned} \texttt{new\_barrier} &\triangleq \lambda\_.\ \texttt{ref}(\texttt{false}) \\ \texttt{signal} &\triangleq \lambda b.\, b \leftarrow \texttt{true} \\ \texttt{wait} &\triangleq \texttt{rec}\ \mathit{wait}(b) := \texttt{if}\ !\, b\ \texttt{then}\ ()\ \texttt{else}\ \mathit{wait}\ b \end{aligned}$$

Before we give the specification, let us look at a motivating example. Suppose we have two computationally expensive functions `expensive_bool` and `expensive_int`, which return a boolean and an integer respectively.

[1] We discuss this difference in Chapter 6

Now consider the following code:

$$\begin{array}{l}
\texttt{example} \triangleq \\
\quad \lambda r.\, \texttt{let}\ x = \texttt{ref}(\texttt{false})\ \texttt{in} \\
\quad\quad \texttt{let}\ b = \texttt{new\_barrier}\ ()\ \texttt{in} \\
\quad\quad \left(\begin{array}{l||l}
 & \texttt{let}\ n = \texttt{expensive\_int}\ ()\ \texttt{in} \\
 & \texttt{wait}\ b \\
x \leftarrow \texttt{expensive\_bool}\ (); & \texttt{let}\ b' = !\,x\ \texttt{in} \\
\texttt{signal}\ b & \texttt{if}\ b' \\
 & \quad \texttt{then}\ r \leftarrow 2 \cdot n \\
 & \quad \texttt{else}\ r \leftarrow n
\end{array} \right)
\end{array}$$

Recall that the double line in the middle is notation for the `par` operation, which executes both expressions in parallel. The left thread first computes `expensive_bool` and then signals the barrier to indicate that that computation has finished. The right threads starts by computing `expensive_int`. It then needs to wait for the left thread to finish, as it needs to be sure that the `expensive_bool` computation has finished. This is achieved by waiting on the barrier. Finally, when it is done waiting, it checks the value of x, and based on that it makes a decision on what to store in r.

When we want to prove a specification for this example we can see that both threads need to have access to x. We are thus unable to split our resources which means we cannot use the Hoare-Par rule directly. However, we as the programmer, know that this code is thread safe: We do not use x in the left thread after the barrier got signalled, and we only access it in the right thread after we are done waiting on the barrier. We would thus desire a specification for our barrier where we can transfer the ownership of x from the left to the right thread.

3.2 Specification

To give a specification for our barrier, we make use of two abstract predicates: $\text{send}(b, P)$ and $\text{recv}(b, P)$. For now, we do not look at their definitions yet, but rather at their use in the specification. The specifications of our methods are as follows:

New Barrier-Spec

$$\{\mathsf{True}\}\ \texttt{new\_barrier}\ ()\ \{b.\ \text{recv}(b, P) * \text{send}(b, P)\}$$

Signal-Spec

$$\{\text{send}(b, P) * P\}\ \texttt{signal}\ b\ \{\mathsf{True}\}$$

Wait-Spec

$$\{\text{recv}(b, P)\}\ \texttt{wait}\ b\ \{P\}$$

Intuitively, ownership of $\text{send}(b, P)$ represents the fact that when someone owns resource P, they should be able to signal barrier b and give up ownership of P. Ownership of $\text{recv}(b, P)$ represents the fact that when someone waits on b, they should get ownership of P after b got signalled. The specification of `new_barrier` just states that it returns a barrier b together with the predicates $\text{recv}(b, P)$ and $\text{send}(b, P)$ for a resource P specified by the user. This makes it possible to verify `signal` and `wait` calls on the returned barrier.

To verify that this specification is indeed strong enough for our example, we prove the following specification for `example`:

$$\{r \mapsto v\}\, \texttt{example}\, r\, \{r \mapsto 74\}$$

To prove this we assume that `expensive_bool`, `expensive_int` return `true`, 37 respectively:

$$\{\mathsf{True}\}\, \texttt{expensive\_bool}\, ()\, \{b.\, b = \texttt{true}\}$$
$$\{\mathsf{True}\}\, \texttt{expensive\_int}\, ()\, \{z.\, z = 37\}$$

The decorated program can be found in Figure 3.1. We start by using Hoare-Alloc to allocate a reference x and store `false` in it. Before we can use New Barrier-Spec, we should decide which resource we want to send and receive. We can see that the right thread needs access to the points-to predicate of x. As `expensive_bool` always returns `true`, we know that `true` is stored at x when we call `signal` b. We thus use

$$P \triangleq x \mapsto \texttt{true}$$

as the resource to send and receive. We can now create a new barrier using New Barrier-Spec. We give $x \mapsto \texttt{false} * \text{send}(b, P)$ to the left thread and the $r \mapsto v * \text{recv}(b, P)$ to the right. In the left thread we first compute `expensive_bool` and use Hoare-Store to store the result in x. As we now know that $x \mapsto \texttt{true}$, we can signal the barrier using Signal-Spec. In the right thread we start by computing `expensive_int`. After that we can wait on the barrier using Wait-Spec. After we are done waiting, the right threads owns $x \mapsto \texttt{true}$. This allows us to use Hoare-Load and conclude that $b' = \texttt{true}$. In the then-case, we use Hoare-Store to store 74 in r, after which we use Sep-Weaken. In the else-case, we have a contradiction $\texttt{false} = \texttt{true}$. As we can conclude anything from a contradiction, we can also conclude that $r \mapsto 74$ holds. As we showed that $r \mapsto 74$ holds after both cases, we can conclude that $r \mapsto 74$ holds after the if statement. Finally, we conclude by using Hoare-Par to combine the postconditions of both threads, and conclude with $r \mapsto 74$ as the final postcondition.

3.3 Verification

We can now define the send and recv predicates. These predicates should relate the physical state, i.e. the values stored on the heap, to a suitable

```
$\{r \mapsto v\}$
let $x$ = ref(false) in
$\{x \mapsto \texttt{false} * r \mapsto v\}$
(* Use Hoare-Alloc *)
let $b$ = new_barrier () in
(* Use New Barrier-Spec with $P \triangleq x \mapsto \texttt{true}$ *)
$\{x \mapsto \texttt{false} * r \mapsto v * \text{send}(b, P) * \text{recv}(b, P)\}$
(* Give $x \mapsto \texttt{false} * \text{send}(b, P)$ to the left thread,
     and $r \mapsto v * \text{recv}(b, P)$ to the right thread *)
(
    $\{x \mapsto \texttt{false} * \text{send}(b, P)\}$
    $x$ ← expensive_bool ();
    (* Spec of expensive_bool *)
    $\{x \mapsto \texttt{true} * \text{send}(b, P)\}$
    signal $b$
    (* Use Signal-Spec *)
    $\{\mathsf{True}\}$
||
    $\{r \mapsto v * \text{recv}(b, P)\}$
    let $n$ = expensive_int () in
    (* Spec of expensive_int *)
    $\{n = 37 * r \mapsto v * \text{recv}(b, P)\}$
    wait $b$
    (* Use Wait-Spec *)
    $\{r \mapsto v * x \mapsto \texttt{true}\}$
    let $b'$ = !$x$ in
    (* Use Hoare-Load *)
    $\{b' = \texttt{true} * r \mapsto v * x \mapsto \texttt{true}\}$
    if $b'$
      then $\{r \mapsto v * x \mapsto \texttt{true}\}$
           (* Use Hoare-Store *)
           $r \leftarrow 2 \cdot n$
           $\{r \mapsto 74 * x \mapsto \texttt{true}\}$
           (* Use Sep-Weaken *)
           $\{r \mapsto 74\}$
      else $\{\texttt{false} = \texttt{true} * r \mapsto v * x \mapsto \texttt{true}\}$
           (* Contradiction *)
           $r \leftarrow n$
           $\{r \mapsto 74\}$
    $\{r \mapsto 74\}$
)
(* Use Hoare-Par to combine the postconditions of both threads *)
$\{r \mapsto 74\}$
```

Figure 3.1: Proof sketch of example.

notion of *logical* or *ghost state*. This is a type of state which, unlike physical state, is not present during the execution of a program. Rather, it is only available during the verification of a program. We can read and write information to this logical state by indexing it with a *ghost name* γ. Making use of this logical state, allows us to prove specifications of more complex programs, where multiple threads may read or write to the same location on the heap. In this section, we design a logical state for this barrier. This allows us to define the send and recv predicates. Afterwards, we show that these definitions are indeed correct, by giving decorated programs to prove the New Barrier-Spec, Signal-Spec, Wait-Spec rules from §3.2.

Both `signal` and `wait` need to read or write to the stored boolean. We can thus see that both the send and recv predicates need knowledge about the points-to predicate of the stored boolean. As we cannot split this points-to predicate,[2] we cannot give it to both $\mathrm{send}(b, P)$ and $\mathrm{recv}(b, P)$.

In situations like this, we can use an invariant. The proposition $\boxed{I}^{\mathcal{N}}$ states that proposition I holds at any moment, hence being an invariant. Each invariant has a namespace $\mathcal{N}$, which use we discuss later. To allocate an invariant we use the following rule:

$$\textsc{Hoare-Inv-Alloc}\quad \frac{\left\{\boxed{I}^{\mathcal{N}} * P\right\} e \left\{v.\, Q\right\}}{\{I * P\}\, e\, \{v.\, Q\}}$$

At first it does not seem like this rule is useful. However, invariants are duplicable: $\boxed{I}^{\mathcal{N}}$ states that I always holds, so duplicating this fact is sound. We thus get the following rule:

$$\textsc{Inv-Dup}\quad \boxed{I}^{\mathcal{N}} \vdash \boxed{I}^{\mathcal{N}} * \boxed{I}^{\mathcal{N}}$$

This means we can give the invariant to multiple threads. This obviously comes at a cost on how we can access I, the resource inside the invariant. Any thread may want to temporarily break the invariant, do some computations and then satisfy it again. The problem is that this could mean that the invariant is also broken for all other threads, who may do some computations while the invariant is broken. This would break the promise that the invariant holds at any time. To prevent this, we require that $\boxed{I}^{\mathcal{N}}$ can only be broken, or rather opened, around atomic expressions. Atomic expressions reduce to a value in one step. That means that no other threads execute during an atomic expression so we can safely open the invariant. This gives rise to the

[2] Technically we can, but we would get two read only points-to predicates, i.e. $\ell \overset{1/2}{\longmapsto} b * \ell \overset{1/2}{\longmapsto} b$, which means we can no longer store new values (Bornat et al., 2005; Boyland, 2003).

following rule:

Hoare-Inv-Open

$$\frac{\mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E} \qquad \{\triangleright I * P\}\, e\, \{v.\, \triangleright I * Q\}_{\mathcal{E}\setminus\mathcal{N}}}{\left\{\boxed{I}^{\mathcal{N}} * P\right\} e \left\{v.\, \boxed{I}^{\mathcal{N}} * Q\right\}_{\mathcal{E}}}$$

The essential part of this rule is that we can temporarily open invariants around atomic expressions. It furthermore has some technicalities, which we discuss now. Opening the same invariant twice could lead to inconsistencies. To prevent this we use a mask $\mathcal{E}$. Hoare triples are written with a mask subscript, indicating which invariants may still be opened in its proof.[3] When opening an invariant we do not directly get access to I, rather to $\triangleright I$. The $\triangleright$ is a *later* modality. Intuitively, $\triangleright P$ means that P holds after one program step. Without the $\triangleright$ guard in Hoare-Inv-Open, Iris would become inconsistent (Jung et al., 2018, §8.2). Therefore, Iris internally uses *step indexing* (Appel & McAllester, 2001; Birkedal et al., 2012), which results in these $\triangleright$ modalities appearing in the rule. However, in many cases we can get rid of these $\triangleright$ modalities as we discuss in §3.3.2. To introduce a later modality, one can use the following rule:

Later-Intro

$$P \vdash \triangleright P$$

The idea behind this rule is that if P already holds, then it will also hold after a program step.

Going back to the specification of our barrier, we need to define a suitable invariant. We can see that for each barrier there can be three possible states:

1. The barrier is not signalled yet.
2. The barrier is signalled but the resource is not received yet.
3. The barrier is signalled and the resource is received.

In the first state the `false` is stored in the reference, while in the other two states, `true` is stored. To distinguish the last two states, we notice that when the barrier gets signalled, P is given up by the sender. We can thus store this resource in the invariant. When transitioning to the last state, we want to transfer this ownership from the invariant to the receiver. However, the receiver should only be able to receive P once, otherwise we could for example duplicate points-to predicate by receiving them multiple times. We therefore make use of an exclusive token, denoted as $\mathsf{token}(\gamma)$.[4] If a receiver

[3] Note that we typically do not write this subscript in proof rules when the mask is $\top$, indicating that no invariants are currently open.

[4] In this chapter we consider $\mathsf{token}(\gamma)$ as a primitive. In Iris, it is defined using *ghost state*. We could formally define it, using the *exclusive* RA, as $\mathsf{token}(\gamma) \triangleq \boxed{\mathsf{ex}()}^{\gamma}$ (Jung et al., 2018, §3).

owns this token, they can exchange it with the invariant for the resource P. This makes sure they can only receive P once. The exclusivity of token(γ) refers to the following rule:

$$\textsc{Token-Exclusive}\\ \text{token}(\gamma) * \text{token}(\gamma) \vdash \mathsf{False}$$

This is similar to the points-to predicate where we have

$$\ell \mapsto v * \ell \mapsto v' \vdash \mathsf{False}$$

We can allocate an exclusive token as follows:

$$\textsc{Token-Alloc}\\ \frac{\{P * \exists \gamma.\ \text{token}(\gamma)\}\ e\ \{v.\, Q\}}{\{P\}\ e\ \{v.\, Q\}}$$

The γ one gets by this rule is a ghost name, which, as we discussed earlier, serves as a key into the logical state.

We can now define the invariant:

$$I(\ell, \gamma, P) \triangleq \underbrace{\ell \mapsto \texttt{false}}_{\substack{\text{Barrier is not}\\ \text{signalled yet}}} \vee \underbrace{(\ell \mapsto \texttt{true} * P)}_{\substack{\text{Barrier is signalled,}\\ \text{P is not received yet}}} \vee \underbrace{(\ell \mapsto \texttt{true} * \text{token}(\gamma))}_{\substack{\text{Barrier is signalled,}\\ \text{P is received}}}$$

The three disjuncts corresponds with the three states we described above. To transition from the first to the second state, the reference should be updated to `true` and P should be given to the invariant. This transition happens when `signal` gets called, as the reference will be updated and P is provided via the precondition of Signal-Spec. To transition from the second to the last state, one should provide token(γ) and gets P in return. This transition will be done by the receiver, so we initially give the ownership of token(γ) to the receiver. Although this invariant exactly describes the barrier's state system, it can be a bit cumbersome to work with because each time we open the invariant, we have to do a case distinction just to get the points-to predicate. We thus use the following invariant, which is logically equivalent, but a bit nicer to work with:

$$I(\ell, \gamma, P) \triangleq \exists b.\, \ell \mapsto b * (\texttt{if}\, b\, \texttt{then}\, (\text{token}(\gamma) \vee P)\ \texttt{else}\ \mathsf{True})$$

Note that the we use a meta level (Coq) if statement here, not the one from the HeapLang program syntax.

We can now use this invariant to define our send and recv predicates. Both of them should have type $\mathit{Val} \to \mathit{iProp} \to \mathit{iProp}$, as they take the value returned by `new_barrier` and the resource being received or sent. We define them as follows:

$$\text{send}(\ell, P) \triangleq \exists \gamma.\, \ell \in \mathit{Loc} * \boxed{I(\ell, \gamma, P)}^{\mathcal{N}}$$
$$\text{recv}(\ell, P) \triangleq \exists \gamma.\, \ell \in \mathit{Loc} * \text{token}(\gamma) * \boxed{I(\ell, \gamma, P)}^{\mathcal{N}}$$

$\{\mathsf{True}\}$
$\texttt{ref}(\texttt{false})$
(* Use Hoare-Alloc *)
$\{\ell.\, \ell \mapsto \texttt{false}\}$
(* Use Token-Alloc *)
$\{\ell.\, \ell \mapsto \texttt{false} * \mathrm{token}(\gamma)\}$
(* Fold definition of I *)
$\{\ell.\, I(\ell, \gamma, P) * \mathrm{token}(\gamma)\}$
(* Use Hoare-Inv-Alloc *)
$\left\{\ell.\, \boxed{I(\ell, \gamma, P)}^{\mathcal{N}} * \mathrm{token}(\gamma)\right\}$
(* Use Inv-Dup *)
$\left\{\ell.\, \boxed{I(\ell, \gamma, P)}^{\mathcal{N}} * \boxed{I(\ell, \gamma, P)}^{\mathcal{N}} * \mathrm{token}(\gamma)\right\}$
(* Fold definitions of send, recv *)
$\{\ell.\ \mathrm{send}(\ell, P) * \mathrm{recv}(\ell, P)\}$

Figure 3.2: `new_barrier` proof sketch.

Both predicates assert that the value should be a location and that the invariant should hold. Furthermore, $\mathrm{recv}(\ell, P)$ has ownership of $\mathrm{token}(\gamma)$. This can be used to transition from the second to the last state and receive P. We can also be sure that the invariant is never in state 3 when we have a recv predicate: otherwise both the invariant and $\mathrm{recv}(\ell, P)$ own $\mathrm{token}(\gamma)$, allowing us to prove False using Token-Exclusive.

With these definitions, we can proceed to verify the specification by giving a decorated program for each of `new_barrier`, `signal`, `wait`.

3.3.1 Verification of New Barrier-Spec

The decorated program is given in Figure 3.2. We start by allocating a location where `false` is stored. We then allocate the exclusive token using Token-Alloc. We are now ready to allocate the invariant. To do so, we first need to prove $I(\ell, \gamma, P)$. This holds because we have $\ell \mapsto \texttt{false}$. Now that we know $\boxed{I(\ell, \gamma, P)}^{\mathcal{N}}$, we can duplicate it using Inv-Dup. Finally we can split our resources and prove

$$\boxed{I(\ell, \gamma, P)}^{\mathcal{N}} \vdash \mathrm{send}(\ell, P)$$
$$\boxed{I(\ell, \gamma, P)}^{\mathcal{N}} * \mathrm{token}(\gamma) \vdash \mathrm{recv}(\ell, P)$$

by instantiating the existential quantifiers and framing.

3.3.2 Verification of Signal-Spec

The decorated program is given in Figure 3.3. We start by destructing the send predicate and beta reducing the lambda. We now want to prove

$\{\text{send}(\ell, P) * P\}$

(* Unfold send *)

$\left\{\boxed{I(\ell, \gamma, P)}^{\mathcal{N}} * P\right\}$

(* Open invariant *)

$\{\triangleright I(\ell, \gamma, P) * P\}$

(* Unfold I *)

$\{\triangleright(\exists b.\, \ell \mapsto b * (\texttt{if}\; b\; \texttt{then}\; (\text{token}(\gamma) \vee P)\; \texttt{else}\; \mathsf{True})) * P\}$

(* Distribute $\triangleright$ *) **(1)**

$\{\triangleright(\ell \mapsto b) * \triangleright(\texttt{if}\; b\; \texttt{then}\; (\text{token}(\gamma) \vee P)\; \texttt{else}\; \mathsf{True}) * P\}$

(* Timelessness of $\ell \mapsto b$ *) **(2)**

$\{\ell \mapsto b * \triangleright(\texttt{if}\; b\; \texttt{then}\; (\text{token}(\gamma) \vee P)\; \texttt{else}\; \mathsf{True}) * P\}$

$\ell \leftarrow \texttt{true}$

(* Use Hoare-Store and Hoare-Later-Eliminate *) **(3)**

$\{\ell \mapsto \texttt{true} * (\texttt{if}\; b\; \texttt{then}\; (\text{token}(\gamma) \vee P)\; \texttt{else}\; \mathsf{True}) * P\}$

(* Weaken *)

$\{\ell \mapsto \texttt{true} * P\}$

(* Introduce $\vee$ *)

$\{\ell \mapsto \texttt{true} * (\text{token}(\gamma) \vee P)\}$

(* Fold I *)

$\{I(\ell, \gamma, P)\}$

(* Use Later-Intro *)

$\{\triangleright I(\ell, \gamma, P)\}$

(* Close invariant *)

$\left\{\boxed{I(\ell, \gamma, P)}^{\mathcal{N}}\right\}$

(* Weaken *)

$\{\mathsf{True}\}$

Figure 3.3: `signal` proof sketch.

$\left\{\boxed{I(\ell, \gamma, P)}^{\mathcal{N}} * P\right\}\; \ell \leftarrow \texttt{true}\; \{\mathsf{True}\}$ As storing a value is an atomic expression, we can open up the invariant. We now want to store `true` in ℓ but this is not possible yet. If we unfold I, we can see that we have the following:

$$\triangleright(\exists b.\, \ell \mapsto b * (\texttt{if}\; b\; \texttt{then}\; (\text{token}(\gamma) \vee P)\; \texttt{else}\; \mathsf{True}))$$

but we need $\ell \mapsto b$ before we can store a value in ℓ. This $\triangleright$ modality is introduced by the Hoare-Inv-Open rule. We earlier stated that we can get rid of this later modality if most cases, which we now explain. There are basically three ways to eliminate later modalities:[5]

1. Commuting rules for $\triangleright$

2. Timelessness

3. Program steps

[5] Since Iris 4.0, there is also a fourth way, later credits (Spies et al., 2022), but we do not discuss that method in this thesis.

These three ways correspond to the numbers is Figure 3.3. We first distribute the $\triangleright$ using the following commuting rules:[6]

$$\textsc{Later-Exists}\quad \frac{\tau \text{ is inhabited} \qquad \triangleright \exists x : \tau.\, P}{\exists x : \tau.\, \triangleright P} \qquad\qquad \textsc{Later-Sep}\quad \frac{\triangleright(P * Q)}{\triangleright P * \triangleright Q}$$

We now have $\triangleright(\ell \mapsto b)$. To strip the later here we can make use of the fact that points-to predicates are $\mathsf{timeless}$. Intuitively, a proposition P is $\mathsf{timeless}$, written as $\mathsf{timeless}(P)$, if it is first-order, does not refer to other invariants and does not contain nested Hoare triples (Jung et al., 2018, §5.7). Because of this property we can strip the $\triangleright$ from a $\mathsf{timeless}$ proposition in the precondition using the following rule:

$$\textsc{Hoare-Timeless-Pre}\quad \frac{\mathsf{timeless}(P) \qquad \{P * Q\}\, e\, \{v.\, R\}}{\{\triangleright P * Q\}\, e\, \{v.\, R\}}$$

We now own $\ell \mapsto b$ which means ℓ can be updated. We are also allowed to remove the only remaining $\triangleright$ as we did a program step. As explained earlier, $\triangleright P$ intuitively mean that P holds after one program step. This intuition is captured by the following rule:

$$\textsc{Hoare-Later-Eliminate}\quad \frac{e \text{ is not a value} \qquad \{P\}\, e\, \{v.\, \Phi(v)\}}{\{\triangleright R * P\}\, e\, \{v.\, R * \Phi(v)\}}$$

We can now prove $I(\ell, \gamma, P)$ and use Later-Intro to close the invariant. The postcondition we want to prove is True, so we can conclude by weakening.

3.3.3 Verification of Wait-Spec

The decorated program is given in Figure 3.4. As `wait` is a recursive function, we need to use induction. However, unlike the verification of `append` in Example 6, we do not have a inductive argument to perform induction on. In such a case, we can use Löb induction:

$$\textsc{Löb Induction}\quad \frac{\triangleright P \vdash P}{\vdash P}$$

If we want to prove P, it is enough to assume the induction hypothesis $\triangleright P$, and then prove P. At first this rule may seem weird, but it makes sense if instantiate it with a Hoare triple:

$$\frac{\triangleright\, (\{\mathrm{recv}(b, P)\}\, \texttt{wait}\, b\, \{P\}) \vdash \{\mathrm{recv}(b, P)\}\, \texttt{wait}\, b\, \{P\}}{\{\mathrm{recv}(b, P)\}\, \texttt{wait}\, b\, \{P\}}$$

[6] We normally do not write the domain of an existential variable, but for the Later-Exists rule it is needed as that type should have an inhabitant.

Our induction hypothesis now states that the Hoare triple holds, but under a $\triangleright$ modality. This means that we can use it, after we did at least one program step. We can thus use it when we do a recursive call. This is sound because Hoare triples only assert partial correctness.

In this proof we did a let expansion on $!\,\ell$. This is because we want to open the invariant which we can only do around an atomic expression. Using the following rule we can bind to the first argument of a let expression:

Hoare-Let

$$\frac{\{P\}\, e_1\, \{x.\, Q\} \qquad \forall v.\, \{Q[v/x]\}\, e_2[v/x]\, \{w.\, R\}}{\{P\}\, \mathtt{let}\, x := e_1\, \mathtt{in}\, e_2\, \{w.\, R\}}$$

In our proof sketch, this allows us to open up the invariant around $!\,\ell$.

In actual proofs, this method of doing a let expansion is unpractical. We would have to show that the semantics of our program does not change after this expansion. One possible method is to restrict our language to only allow programs which are in A-normal form (Flanagan et al., 1993), which means the programmer has to let expand all their code by hand. Iris handles this problem is a different way, by introducing evaluation contexts and the following bind rule:

Hoare-Bind

$$\frac{K \text{ is an evaluation context} \qquad \{P\}\, e\, \{v.\, Q\} \qquad \forall v.\, \{Q\}\, K[v]\, \{w.\, R\}}{\{P\}\, K[e]\, \{w.\, R\}}$$

In our proof sketch we could take K as $\mathtt{if}\ \bullet\ \mathtt{then}\ ()\ \mathtt{else}\ \mathtt{wait}\ \ell$ and $K[e]$ would fill e into the hole in our context, which is denoted by $\bullet$. However, this approach does not have a nice representation in our decorated program, hence why we do the let expansion and use Hoare-Let.[7]

Now that we can open the invariant around $!\,\ell$, we can again distribute the later and use the timelessness of $\ell \mapsto b$. We dereference ℓ and remove a $\triangleright$ because we did a program step. For (1) we argue that the following holds:

$$(\mathtt{if}\ b\ \mathtt{then}\ (\mathsf{token}(\gamma) \vee P)\ \mathtt{else}\ \mathsf{True}) * \mathsf{token}(\gamma) \vdash$$
$$(\mathtt{if}\ b\ \mathtt{then}\ (\mathsf{token}(\gamma) \vee P)\ \mathtt{else}\ \mathsf{True}) * (\mathtt{if}\ b\ \mathtt{then}\ P\ \mathtt{else}\ \mathsf{token}(\gamma))$$

We do so by case distinction on b. Suppose b is $\mathtt{true}$. This means that we have $(\mathsf{token}(\gamma) \vee P) * \mathsf{token}(\gamma)$. As we know by Token-Exclusive, $\mathsf{token}(\gamma)$ is exclusive which means that $P * \mathsf{token}(\gamma)$ holds. We can then prove our goal by giving $\mathsf{token}(\gamma)$ to the left and P to the right hand side of the $*$. When b is $\mathtt{false}$ we need to prove $\mathsf{True} * \mathsf{token}(\gamma) \vdash \mathsf{True} * \mathsf{token}(\gamma)$ which holds trivially. We can now close the invariant again and continue to the if statement. If b is $\mathtt{true}$, then we own P, so after evaluating $()$ we are done. If b is $\mathtt{false}$, then we still own $\mathsf{token}(\gamma)$. If we combine this with $\boxed{I(\ell, \gamma, P)}^{\mathcal{N}}$

[7] One may notice that Hoare-Let is an instantiation of Hoare-Bind.

we get $\text{recv}(\ell, P)$ again. We now need to execute `wait` ℓ. We can use our induction hypothesis and get P via the postcondition of the recursive call. As we receive P in both cases of the if statement, we can conclude that afterwards we always own P, which concludes the proof.

```
$\{\mathrm{recv}(\ell, P)\}$
(* Unfold recv *)
$\left\{\boxed{I(\ell,\gamma,P)}^{\mathcal{N}} * \mathrm{token}(\gamma)\right\}$
let $f$ = $\left\{\boxed{I(\ell,\gamma,P)}^{\mathcal{N}} * \mathrm{token}(\gamma)\right\}$
        (* Open invariant *)
        $\{\triangleright I(\ell,\gamma,P) * \mathrm{token}(\gamma)\}$
        (* Unfold $I$ *)
        $\{\triangleright(\exists b.\, \ell \mapsto b * (\texttt{if}\ b\ \texttt{then}\ (\mathrm{token}(\gamma) \lor P)\ \texttt{else}\ \mathsf{True})) * \mathrm{token}(\gamma)\}$
        (* Distribute $\triangleright$ and timelessness of $\ell \mapsto b$ *)
        $\{\ell \mapsto b * \triangleright(\texttt{if}\ b\ \texttt{then}\ (\mathrm{token}(\gamma) \lor P)\ \texttt{else}\ \mathsf{True}) * \mathrm{token}(\gamma)\}$
        $!\,\ell$
        (* Dereference $\ell$, remove $\triangleright$ by program step *)
        $\{b.\, \ell \mapsto b * (\texttt{if}\ b\ \texttt{then}\ (\mathrm{token}(\gamma) \lor P)\ \texttt{else}\ \mathsf{True}) * \mathrm{token}(\gamma)\}$
        (* See note (1) *)
        $\{b.\, \ell \mapsto b * (\texttt{if}\ b\ \texttt{then}\ (\mathrm{token}(\gamma) \lor P)\ \texttt{else}\ \mathsf{True}) * (\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ \mathrm{token}(\gamma))\}$
        (* Fold $I$ *)
        $\{b.\, I(\ell,\gamma,P) * (\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ \mathrm{token}(\gamma))\}$
        (* Introduce $\triangleright$ *)
        $\{b.\, \triangleright I(\ell,\gamma,P) * (\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ \mathrm{token}(\gamma))\}$
        (* Close invariant *)
        $\left\{b.\, \boxed{I(\ell,\gamma,P)}^{\mathcal{N}} * (\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ \mathrm{token}(\gamma))\right\}$
$\left\{f = b * \boxed{I(\ell,\gamma,P)}^{\mathcal{N}} * (\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ \mathrm{token}(\gamma))\right\}$
if $f$ then
    $\left\{b = \texttt{true} * \boxed{I(\ell,\gamma,P)}^{\mathcal{N}} * (\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ \mathrm{token}(\gamma))\right\}$
    (* Reduce if *)
    $\left\{\boxed{I(\ell,\gamma,P)}^{\mathcal{N}} * P\right\}$
    ()
    (* Weaken *)
    $\{P\}$
else
    $\left\{b = \texttt{false} * \boxed{I(\ell,\gamma,P)}^{\mathcal{N}} * (\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ \mathrm{token}(\gamma))\right\}$
    (* Reduce if *)
    $\left\{\boxed{I(\ell,\gamma,P)}^{\mathcal{N}} * \mathrm{token}(\gamma)\right\}$
    (* Fold recv *)
    $\{\mathrm{recv}(\ell, P)\}$
    wait $\ell$
    (* Induction hypothesis *)
    $\{P\}$
$\{P\}$
```

Figure 3.4: `wait` proof sketch.

Chapter 4

Chainable Barrier

In this chapter, we extend the barrier we have seen in the previous chapter. We start by describing some key features of this barrier (§4.1). We then proceed by giving the implementation of this barrier (§4.2). Our implementation is a slight adaptation of the on given in (Dodds et al., 2016, Section 6). This allows us to present a richer specification (§4.3) than the one given in (Dodds et al., 2016, Figure 2). We extend the implementation by adding a `clone` operation, and the specification by adding rules for *send splitting*, *send strengthening* and *receive weakening*.[1] Finally, we conclude by giving an intuitive idea of the verification of the specification (§4.4). Our verification takes a different approach compared with (Dodds et al., 2016, Section 6). By taking a different invariant we are able to prove the specification in a more straightforward way. The verification presented in §4.4, is meant to explain the intuitive idea. The formal definitions are presented in Chapter 5.

4.1 Key features

In this section we discuss key features of the chainable barrier. We do this via example programs and discuss how the code needs to change to support these individual features. We start of by discussing *receive splitting* (§4.1.1). Next, we introduce *send splitting* (§4.1.2), and finally, we discuss how to create a chain of barriers and how to use *renunciation* (§4.1.3). Both *receive splitting* and *renunciation* were already present in Dodds et al., 2016, while *send splitting* is a new feature introduced in this thesis.

[1] Note that *receive weakening* was already present the Coq formalization of Jung et al., 2016. However, they used a slightly weaker version in the paper, namely Recv-Mono.

4.1.1 Receive splitting

It is common that we want to wait on the same barrier in different threads. Let us consider the following example program:

$$\texttt{let}\ b = \texttt{new\_barrier}\,()\ \texttt{in}$$

$$\left(\begin{array}{l} \texttt{wait}\ b; \\ \ell_1 \leftarrow (!\,\ell_1) + 5 \end{array} \;\middle\|\; \begin{array}{l} \ell_1 \leftarrow 37; \\ \ell_2 \leftarrow 42; \\ \texttt{signal}\ b \end{array} \;\middle\|\; \begin{array}{l} \texttt{wait}\ b; \\ \ell_2 \leftarrow (!\,\ell_2) - 5 \end{array}\right)$$

The middle thread stores values in ℓ_1, ℓ_2 and then signals the barrier. The left and right thread wait on the barrier before updating one of the locations. This code is safe since the left and right thread only use ℓ_1, ℓ_2 after the barrier is signalled. We furthermore have that after the signal, ℓ_1 is only used in the left thread, and ℓ_2 is only used in the right thread. However, the specification of Chapter 3 is not strong enough to verify this example. Using New Barrier-Spec we only get a single $\mathrm{recv}(b, P)$ resource. As we always need $\mathrm{recv}(b, P)$ for the precondition of a `wait` call, we can currently only have one `wait` call in verified programs. We therefore introduce the following rule to the specification:

Recv-Split

$$\mathrm{recv}(b, P * Q) \vdash {\mid\!\!\Rrightarrow}\, \mathrm{recv}(b, P) * \mathrm{recv}(b, Q)$$

If we have a $\mathrm{recv}(b, P * Q)$ resource, we can split it and get two receive predicates back. We can use $\mathrm{recv}(b, P)$ for one `wait` call to receive P, and $\mathrm{recv}(b, Q)$ for another `wait` call to receive Q. The ${\mid\!\!\Rrightarrow}$ symbol, a ghost state update modality, is a technicality which is discussed in §4.3. For now, it is only important to note that we can eliminate these ${\mid\!\!\Rrightarrow}$ modalities at any point during verification of a program, which means we can more or less ignore them.

With this rule we could verify our example program as shown in Figure 4.1. We first obtain $\mathrm{recv}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42)$ and $\mathrm{send}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42)$ by using New Barrier-Spec. We can use Recv-Split to split the single receive predicate into two to get $\mathrm{recv}(b, \ell_1 \mapsto 37)$ and $\mathrm{recv}(b, \ell_2 \mapsto 42)$. We then give $\mathrm{recv}(b, \ell_1 \mapsto 37)$ to the left thread, $\mathrm{send}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42)$ and the points-to predicates of ℓ_1, ℓ_2 to the middle thread, and $\mathrm{recv}(b, \ell_2 \mapsto 42)$ to the right thread. We now have a receive predicate for b in both the left and right thread, and a send predicate in the middle thread. The rest of the verification follows by using Hoare-Load, Hoare-Store, Wait-Spec and Signal-Spec.

Note that we do not have to change the code of the barrier to introduce this rule, as there is no problem with multiple threads executing a `wait` call. We only need to change the invariant we use for the verification to add support for this rule, which we discuss in §4.4.

$\{\ell_1 \mapsto v * \ell_2 \mapsto w\}$

$\mathtt{let}\, b = \mathtt{new\_barrier}\,()\ \mathtt{in}$

(* Use $\textsc{New Barrier-Spec}$ with $P \triangleq \ell_1 \mapsto 37 * \ell_2 \mapsto 42$ *)

$$\left\{\begin{array}{l} \ell_1 \mapsto v * \ell_2 \mapsto w \\ * \operatorname{send}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42) \\ * \operatorname{recv}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42) \end{array}\right\}$$

(* Use $\textsc{Recv-Split}$ *)

$$\left\{\begin{array}{l} \ell_1 \mapsto v * \ell_2 \mapsto w \\ * \operatorname{send}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42) \\ * \operatorname{recv}(b, \ell_1 \mapsto 37) * \operatorname{recv}(b, \ell_2 \mapsto 42) \end{array}\right\}$$

(* Split resources over the three threads *)

$$\left(\begin{array}{l||l||l}
\begin{array}{l}
\{\operatorname{recv}(b, \ell_1 \mapsto 37)\} \\
\mathtt{wait}\, b; \\
\text{(* Use }\textsc{Wait-Spec}\text{ *)} \\
\{\ell_1 \mapsto 37\} \\
\ell_1 \leftarrow (!\,\ell_1) + 5 \\
\text{(* Use }\textsc{Hoare-Load}, \\
\quad \textsc{Hoare-Store}\text{ *)} \\
\{\ell_1 \mapsto 42\}
\end{array}
&
\begin{array}{l}
\left\{\begin{array}{l} \operatorname{send}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42) \\ * \ell_1 \mapsto v * \ell_2 \mapsto w \end{array}\right\} \\
\ell_1 \leftarrow 37; \\
\text{(* Use }\textsc{Hoare-Store}\text{ *)} \\
\left\{\begin{array}{l} \operatorname{send}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42) \\ * \ell_1 \mapsto 37 * \ell_2 \mapsto w \end{array}\right\} \\
\ell_2 \leftarrow 42; \\
\text{(* Use }\textsc{Hoare-Store}\text{ *)} \\
\left\{\begin{array}{l} \operatorname{send}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42) \\ * \ell_1 \mapsto 37 * \ell_2 \mapsto 42 \end{array}\right\} \\
\mathtt{signal}\, b \\
\text{(* Use }\textsc{Signal-Spec}\text{ *)} \\
\{\mathsf{True}\}
\end{array}
&
\begin{array}{l}
\{\operatorname{recv}(b, \ell_2 \mapsto 42)\} \\
\mathtt{wait}\, b; \\
\text{(* Use }\textsc{Wait-Spec}\text{ *)} \\
\{\ell_2 \mapsto 42\} \\
\ell_2 \leftarrow (!\,\ell_2) - 5 \\
\text{(* Use }\textsc{Hoare-Load}, \\
\quad \textsc{Hoare-Store}\text{ *)} \\
\{\ell_2 \mapsto 37\}
\end{array}
\end{array}\right)$$

(* Combine the postconditions of all threads *)

$\{\ell_1 \mapsto 42 * \ell_2 \mapsto 37\}$

Figure 4.1: Verification of receive splitting example.

4.1.2 Send splitting

It is also possible that we have multiple worker threads signalling a single thread when they are all finished. Let us consider the following program:

$$\begin{array}{l}\texttt{let}\ b = \texttt{new\_barrier}\ ()\ \texttt{in}\\ \texttt{clone}\ b;\end{array}$$

$$\left(\begin{array}{l}\ell_1 \leftarrow 37;\\ \texttt{signal}\ b\end{array} \middle\| \begin{array}{l}\texttt{wait}\ b;\\ \ell_1 \leftarrow (!\,\ell_1) + 5;\\ \ell_2 \leftarrow (!\,\ell_2) - 5\end{array} \middle\| \begin{array}{l}\ell_2 \leftarrow 42;\\ \texttt{signal}\ b\end{array}\right)$$

We start by creating a barrier, after which we clone it. This basically tells the barrier that it needs to wait on an extra signal before it is done waiting. This means that after this call, the barrier waits for two separate signals. The left and right thread both signal the barrier after updating ℓ_1, ℓ_2 respectively, after which the middle thread is done waiting and can continue executing. The specification for `clone` is the following:

Clone-Spec
$$\{\mathrm{send}(b, P * Q)\}\ \texttt{clone}\ b\ \{\mathrm{send}(b, P) * \mathrm{send}(b, Q)\}$$

If we need to send $P * Q$, we can use `clone` to send this via two separate `signal` calls. The first call sends P and the second one sends Q. We therefore get two send predicates via the postcondition.

By extending our specification with this rule, we can verify the example, as shown in Figure 4.2. We start by obtaining $\mathrm{recv}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42)$ and $\mathrm{send}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42)$ by using New Barrier-Spec. We then use Clone-Spec to split the send predicate and get $\mathrm{send}(b, \ell_1 \mapsto 37)$ and $\mathrm{send}(b, \ell_2 \mapsto 42)$. We then give $\mathrm{send}(b, \ell_1 \mapsto 37)$ and the points-to predicate of ℓ_1 to the left thread, $\mathrm{recv}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42)$ to the middle thread, and $\mathrm{recv}(b, \ell_2 \mapsto 42)$ and the points-to predicate of ℓ_2 to the right thread. We now have a send predicate for b in both the left and right thread, and a receive predicate in the middle thread. The rest of the verification proceeds by using Hoare-Load, Hoare-Store, Wait-Spec and Signal-Spec.

To add this functionality, we need to change the implementation of the barrier. Previously there was a boolean flag, which indicated whether the barrier was signalled. This is now replaced with an integer counter, which states how many times the barrier needs to be signalled before it is done waiting. The counter starts at 1 and `wait` waits until it is equal to 0. The counter gets decremented by calls to `signal` and incremented by calls to `clone`. The changes in the code are discussed in §4.2.

Note that, although they look similar, Clone-Spec and Recv-Split have an important difference. The former is a Hoare triple, which means one can use it to split a send predicate after a `clone` call, while the latter can be used to split a receive predicate at any moment during the verification of a program.

$\{\ell_1 \mapsto v * \ell_2 \mapsto w\}$

$\texttt{let}\ b = \texttt{new\_barrier}\ ()\ \texttt{in}$

(* Use New Barrier-Spec with $P \triangleq \ell_1 \mapsto 37 * \ell_2 \mapsto 42$ *)

$$\left\{\begin{array}{l} \ell_1 \mapsto v * \ell_2 \mapsto w \\ * \operatorname{recv}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42) \\ * \operatorname{send}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42) \end{array}\right\}$$

$\texttt{clone}\ b;$

(* Use Clone-Spec *)

$$\left\{\begin{array}{l} \ell_1 \mapsto v * \ell_2 \mapsto w \\ * \operatorname{recv}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42) \\ * \operatorname{send}(b, \ell_1 \mapsto 37) * \operatorname{send}(b, \ell_2 \mapsto 42) \end{array}\right\}$$

(* Split resources over the three threads *)

$$\left(\begin{array}{l||l||l}
\begin{array}{l}
\left\{\begin{array}{l} \ell_1 \mapsto v \\ * \operatorname{send}(b, \ell_1 \mapsto 37) \end{array}\right\} \\
\ell_1 \leftarrow 37; \\
\text{(* Use \textsc{Hoare-Store} *)} \\
\left\{\begin{array}{l} \ell_1 \mapsto 37 \\ * \operatorname{send}(b, \ell_1 \mapsto 37) \end{array}\right\} \\
\texttt{signal}\ b \\
\text{(* Use \textsc{Signal-Spec} *)} \\
\{\mathsf{True}\}
\end{array}
&
\begin{array}{l}
\{\operatorname{recv}(b, \ell_1 \mapsto 37 * \ell_2 \mapsto 42)\} \\
\texttt{wait}\ b; \\
\text{(* Use \textsc{Wait-Spec} *)} \\
\{\ell_1 \mapsto 37 * \ell_2 \mapsto 42\} \\
\ell_1 \leftarrow (!\,\ell_1) + 5; \\
\text{(* Use \textsc{Hoare-Load},} \\
\quad \text{\textsc{Hoare-Store} *)} \\
\{\ell_1 \mapsto 42 * \ell_2 \mapsto 42\} \\
\ell_2 \leftarrow (!\,\ell_2) - 5 \\
\text{(* Use \textsc{Hoare-Load},} \\
\quad \text{\textsc{Hoare-Store} *)} \\
\{\ell_1 \mapsto 42 * \ell_2 \mapsto 37\}
\end{array}
&
\begin{array}{l}
\left\{\begin{array}{l} \ell_2 \mapsto w \\ * \operatorname{send}(b, \ell_2 \mapsto 42) \end{array}\right\} \\
\ell_2 \leftarrow 42; \\
\text{(* Use \textsc{Hoare-Store} *)} \\
\left\{\begin{array}{l} \ell_2 \mapsto 42 \\ * \operatorname{send}(b, \ell_2 \mapsto 42) \end{array}\right\} \\
\texttt{signal}\ b \\
\text{(* Use \textsc{Signal-Spec} *)} \\
\{\mathsf{True}\}
\end{array}
\end{array}\right)$$

(* Combine the postconditions of all threads *)

$\{\ell_1 \mapsto 42 * \ell_2 \mapsto 37\}$

Figure 4.2: Verification of send splitting example.

4.1.3 Chains and renunciation

Up til now we have only considered individual barriers. However, there are cases where there is a order in which threads should execute. Let us consider the following program:

$$\begin{array}{c}
\texttt{let}\, b_1 = \texttt{new\_barrier}\,()\,\texttt{in} \\
\texttt{let}\, b_2 = \texttt{new\_barrier}\,()\,\texttt{in} \\
\left(\begin{array}{l||l||l}
\begin{array}{l} \ell \leftarrow (!\,\ell) + x \\ \texttt{signal}\, b_2 \end{array}
&
\begin{array}{l} \texttt{if}\, y = 0 \\ \quad \texttt{then}\, \texttt{signal}\, b_1 \\ \quad \texttt{else}\, \texttt{wait}\, b_2; \\ \qquad\quad \ell \leftarrow (!\,\ell) + y; \\ \qquad\quad \texttt{signal}\, b_1 \end{array}
&
\begin{array}{l} \texttt{wait}\, b_2; \\ \texttt{wait}\, b_1; \\ \ell \leftarrow (!\,\ell) + z \end{array}
\end{array}\right)
\end{array}$$

The left thread first adds x to ℓ, after which it signals b_2. The execution of the middle thread depends on the value of y. If $y = 0$ it instantly signals b_1. Otherwise, it first waits on b_2, then adds y to ℓ, and finally signals b_1. The right thread has to wait on both barriers before it can add z to ℓ. One can thus observe that, first x is added to ℓ, then, depending on the value of y, y may be added, and finally, z is added. We essentially created a chain of barriers. One may observe that extending this pattern to more threads results in quite ugly code. Furthermore, we cannot dynamically add more barriers to this chain. For example, the middle thread could create another barrier, but the right thread does not have access to it, so it cannot wait on it.

We therefore put this chain construction in the barrier itself. From now on, we consider a barrier as a linear chain of nodes. A programmer can use `signal` and `wait` to signal and wait on individual nodes. A `wait` call not only has to wait until the node itself got signalled, but also until all nodes which occur earlier in the chain have been signalled. To add a new node in the chain, one can use `extend`. It takes a node b as an argument, and returns a new node b' which is inserted in the chain, such that it is the first node which occurs earlier than b. This means that a $\texttt{wait}\, b$ call also has to wait until b' got signalled. We can thus use `extend` to dynamically add new nodes to the chain.

With this new construction, we can simplify the example program, like follows:

$$
\begin{array}{l}
\texttt{let}\, b_1 = \texttt{new\_barrier}\,()\,\texttt{in}\\
\texttt{let}\, b_2 = \texttt{extend}\, b_1\,\texttt{in}
\end{array}
$$

$$
\left(\begin{array}{l||l||l}
\begin{array}{l} \ell \leftarrow (!\,\ell) + x \\ \texttt{signal}\, b_2 \end{array}
&
\begin{array}{l} \texttt{if}\; y = 0 \\ \quad \texttt{then}\,\texttt{signal}\, b_1 \\ \quad \texttt{else}\,\texttt{wait}\, b_2; \\ \qquad\quad \ell \leftarrow (!\,\ell) + y; \\ \qquad\quad \texttt{signal}\, b_1 \end{array}
&
\begin{array}{l} \texttt{wait}\, b_1; \\ \ell \leftarrow (!\,\ell) + z \end{array}
\end{array}\right)
$$

Instead of creating a new barrier, we extend b_1 to get b_2. The $\texttt{wait}\, b_1$ call in the right thread terminates once both b_1 and b_2 have been signalled, so we now only need a single wait call.

Let us now look at a the specification of `extend`:[2]

Extend-Spec
$$\{\mathrm{send}(b, P)\}\ \texttt{extend}\, b\ \{b'.\, b' \prec b * \mathrm{send}(b', Q) * \mathrm{recv}(b', Q) * \mathrm{send}(b, P)\}$$

If a client owns $\mathrm{send}(b, P)$, they can extend b. The function returns a new node b' for which $b' \prec b$ holds. This $\prec$ (earlier) predicate represents the fact that b' occurs earlier in the chain than b. Furthermore, the client gets ownership of $\mathrm{send}(b', Q)$ and $\mathrm{recv}(b', Q)$ for arbitrary Q, and also gets back ownership of $\mathrm{send}(b, P)$.

In programs, it can be the case that a thread may or may not do a `wait` call. This happens, for example, in the program above, where the middle thread only calls $\texttt{wait}\, b_2$ if y is unequal to 0. This means that while verifying the middle thread, we do not receive a resource from b_1 if $y = 0$. In situations like this, we can use the following rule:

Renunciation
$$\mathrm{recv}(b, P) * \mathrm{send}(b', Q) * b \prec b' \vdash \mathrel{|\!\!\Rrightarrow} \mathrm{send}(b', P \mathrel{-\!\!*} Q)$$

Again, we do not worry about the $\mathrel{|\!\!\Rrightarrow}$ modality for now. The Renunciation rule intuitively says that if we can receive P from node b, signal node b' with resource Q and $b \prec b'$ holds, then it is enough to just signal node b' with resource $P \mathrel{-\!\!*} Q$. Recall that $P \mathrel{-\!\!*} Q$ describes a resource, which when combined with P, describes Q. The reason why this rule is sound, is because we give up ownership of $\mathrm{recv}(b, P)$. We can therefore no longer receive P via a $\texttt{wait}\, b$ call. Instead, when P is sent to b, b can forward P to b'. We thus only need to send $P \mathrel{-\!\!*} Q$ to b'.

With these rules, we can verify that the example program sums x, y and z, as shown in in Figure 4.3. As a precondition we assume that $\ell \mapsto 0$. We start by using New Barrier-Spec. If we inspect the code, we can notice

[2] This is a slightly weaker version. The stronger specification is presented in §4.3.

```
{$\ell \mapsto 0$}
let $b_1$ = new_barrier () in
(* Use NEW BARRIER-SPEC with $P \triangleq \ell \mapsto x + y$ *)
{$\ell \mapsto 0 * \text{recv}(b_1, \ell \mapsto x + y) * \text{send}(b_1, \ell \mapsto x + y)$}
let $b_2$ = extend $b_1$ in
(* Use EXTEND-SPEC with $Q \triangleq \ell \mapsto x$ *)
{$\ell \mapsto 0 * \text{recv}(b_1, \ell \mapsto x + y) * \text{send}(b_1, \ell \mapsto x + y)$
  $* b_2 \prec b_1 * \text{recv}(b_2, \ell \mapsto x) * \text{send}(b_2, \ell \mapsto x)$}
(
  {$\text{send}(b_2, \ell \mapsto x)$
    $* \ell \mapsto 0$}
  $\ell \leftarrow (!\,\ell) + x$;
  (* Use HOARE-LOAD,
      HOARE-STORE *)
  {$\text{send}(b_2, \ell \mapsto x)$
    $* \ell \mapsto x$}
  (* Use SIGNAL-SPEC *)
  signal $b_2$
  {True}
||
  {$\text{send}(b_1, \ell \mapsto x + y)$
    $* b_2 \prec b_1 * \text{recv}(b_2, \ell \mapsto x)$}
  if $y = 0$
    then (* Substitute $y = 0$ *)
         {$\text{send}(b_1, \ell \mapsto x + 0)$
           $* b_2 \prec b_1 * \text{recv}(b_2, \ell \mapsto x)$}
         (* Use RENUNCIATION *)
         {$\text{send}(b_1, (\ell \mapsto x) \mathrel{-\!\!*} (\ell \mapsto x + 0))$}
         (* Use SIGNAL-SPEC, as
              $(\ell \mapsto x) \mathrel{-\!\!*} (\ell \mapsto x + 0)$
              is a tautology *)
         signal $b_1$
         {True}
    else {$\text{send}(b_1, \ell \mapsto x + y)$
           $* b_2 \prec b_1 * \text{recv}(b_2, \ell \mapsto x)$}
         wait $b_2$;
         (* Use WAIT-SPEC *)
         {$\text{send}(b_1, \ell \mapsto x + y)$
           $* \ell \mapsto x$}
         $\ell \leftarrow (!\,\ell) + y$
         (* Use HOARE-LOAD,
             HOARE-STORE *)
         {$\text{send}(b_1, \ell \mapsto x + y)$
           $* \ell \mapsto x + y$}
         signal $b_1$
         (* Use SIGNAL-SPEC *)
         {True}
  {True}
||
  {$\text{recv}(b_1, \ell \mapsto x + y)$}
  wait $b_1$;
  (* Use WAIT-SPEC *)
  {$\ell \mapsto x + y$}
  $\ell \leftarrow (!\,\ell) + z$
  (* Use HOARE-LOAD,
      HOARE-STORE *)
  {$\ell \mapsto x + y + z$}
)
(* Combine postconditions of all threads *)
{$\ell \mapsto x + y + z$}
```

Figure 4.3: Verification of chain example.

that ℓ always contains $x + y$ after the `wait` b_1 call in the right thread. Even if $y = 0$, we still know that ℓ contains x, which is equal to $x + y$ in this case, as the `wait` b_1 call also waits for b_2 to get signalled. We can therefore receive $\ell \mapsto x + y$, so we take $P \triangleq \ell \mapsto x + y$ in our application of New Barrier-Spec. We then use Extend-Spec to create an extra node. We take $Q \triangleq \ell \mapsto x$, as this is the value stored at ℓ when we signal b_2. We then give $\text{send}(b_2, \ell \mapsto x) * \ell \mapsto 0$ to the left thread. It can use this to update ℓ and signal b_2. To the right thread we give $\text{recv}(b_1, \ell \mapsto x + y)$. After the `wait` b_1 call, it receives $\ell \mapsto x + y$, allowing it to add z to the sum. Verifying the middle thread is the most interesting, as we need to use Renunciation there. In the else-case of the if statement, we can first receive $\ell \mapsto x$, then update ℓ, and finally, send $\ell \mapsto x + y$ to b_1. In the then-case, we first use Renunciation, after which we only have to send $(\ell \mapsto x) \mathrel{-\!\!*} (\ell \mapsto x + 0)$. We can prove this proposition by using Wand-Intro and the fact that 0 is a right identity for $+$. We can thus use Signal-Spec, after which the verification of the middle thread is done. Finally, we can combine the postconditions of all threads, to see that ℓ indeed contains the sum of x, y and z.

To implement this feature, each node needs to keep track of an optional previous node. The `wait` function can then recursively call on previous nodes, until all of them are signalled. The `extend` function updates the previous node to insert a new one. In the next section, we discuss the complete implementation.

4.2 Implementation

We now discuss the implementation of this barrier. The complete implementation is given in Figure 4.4. We first need a way to represent the barriers. Every node in a barrier consists of a counter and an optional previous node. We can therefore use a representation similar to the one we used in Example 2 for lists. The representation we use here is slightly different for reasons that become apparent when we discuss the implementation of `signal`. A node is represented by a location ℓ, similar to how a barrier was represented by a location in Chapter 3. We store the counter of the node in ℓ and store the optional previous node at $\ell +_{\mathrm{L}} 1$. This is the location directly after ℓ and can be thought of as pointer addition.

With this new representation in mind, `new_barrier` needs to allocate two adjacent locations. We can do this using $\texttt{AllocN}(2, ())$, which returns a location ℓ such that $\ell \mapsto ()$ and $(\ell +_{\mathrm{L}} 1) \mapsto ()$. As the counter should be initialized to 1, we set ℓ to 1, and since there is no previous node, we store `none` in $\ell +_{\mathrm{L}} 1$. Finally we return the representation of this new node, namely ℓ.

The implementations of `signal` and `clone` are quite similar to each other.

```
new_barrier ≜ $\lambda\_.$ let $\ell$ = AllocN$(2, ())$ in
                  $\ell \leftarrow 1$;
                  $(\ell +_{\mathtt{L}} 1) \leftarrow$ none;
                  $\ell$
     signal ≜ $\lambda b.$ FAA$(b, -1)$;
                  $()$
      clone ≜ $\lambda b.$ FAA$(b, 1)$;
                  $()$
       wait ≜ rec $wait(b)$ := if $(!\,b) = 0$
                                 then match $!(b +_{\mathtt{L}} 1)$ with
                                        none     $\Rightarrow ()$
                                      | some$(b')$ $\Rightarrow$ $wait(b')$
                                      end
                                 else $wait(b)$
     extend ≜ rec $extend(b)$ := let $b' = !(b +_{\mathtt{L}} 1)$ in
                                 let $\ell$ = AllocN$(2, ())$ in
                                 $\ell \leftarrow 1$;
                                 $(\ell +_{\mathtt{L}} 1) \leftarrow b'$;
                                 if CAS$(b +_{\mathtt{L}} 1, b', $some$(\ell))$
                                   then $\ell$
                                   else Free$(\ell)$;
                                        Free$(\ell +_{\mathtt{L}} 1)$;
                                        $extend(b)$
```

Figure 4.4: Implementation of the barrier

The argument b is exactly where the counter is stored. We thus use `FAA` (fetch and add) to atomically increment or decrement the counter, and afterwards return the unit value. We need an atomic operation to change the counter, as there can be multiple threads signalling or cloning at the same time. This is also the reason why we cannot use a pair to store both the counter and the previous node, as we cannot do a `FAA` operation on the first element of a pair.

The implementation of `wait` changes slightly, as it also needs to wait on previous nodes. We first check if the counter (i.e. the value stored in b) is equal to 0. If the counter is not equal to 0 yet, we do a recursive call on the same barrier. If the counter is set to 0, we check whether there is a previous node. We are done waiting whenever there is no previous node, in which case we return. If there is a previous node b' we recursively wait on b'.

The implementation of `extend` is bit more complex. We first load the previous node of b and call it b'. We then allocate a location ℓ for the new node, and initialize its counter to 1. We want to insert this new node immediately after b, which means that the previous node of the new node should be set to b'. This creates a data race, as another thread may have already updated the previous node of b after we read it into b'. We therefore need a `CAS` (compare and swap) check.[3] If b' is still stored at $b +_{\mathtt{L}} 1$, we update the previous node of b to $\mathtt{some}(\ell)$, i.e. the newly created node. Afterwards, we return this new node. If the previous node of b has changed, we can conclude that we lost the race. We free the allocated locations and try to extend the node once again with a recursive call.

4.3 Specification

We now discuss the full specification of this barrier, which is given in Figure 4.5. We start by looking at the rules of the $\prec$-predicate. Firstly, Earlier-Trans says that the $\prec$-predicate is transitive. If we know that b_1 occurs earlier in the chain than b_2, and b_2 occurs earlier than b_3, then b_1 also occurs earlier than b_3. Furthermore, the $\prec$-predicate is timeless, which means we can eliminate $\triangleright$ guards around them.[4] Finally, Earlier-Persistent states that the $\prec$-predicate is persistent. We have not seen persistency up til now, so we discuss it shortly. Propositions are persistent when they can describe a resource without needing exclusive ownership over that resource. For example, $\ell \mapsto 1$ is not persistent as it describes exclusive ownership of ℓ. On the other hand, $b_1 \prec b_2$ is persistent, because multiple threads can know it at the same time. Once $b_1 \prec b_2$ holds, it holds forever. This is because

[3] Note that Dodds et al. did not need a `CAS` check in their implementation of `extend`. This is because in the absence of send splitting there is at most one sender, so the extend race is always won. Their implementation thus had a data race, but this was not a problem as the specification forbids it.

[4] Linguistically, it is nice that we can eliminate *later* guards around *earlier* predicates.

Earlier-Trans

$$b_1 \prec b_2 * b_2 \prec b_3 \vdash b_1 \prec b_3$$

Earlier-Timeless

$$\mathsf{timeless}(b_1 \prec b_2)$$

Earlier-Persistent

$$b_1 \prec b_2 \vdash \Box(b_1 \prec b_2)$$

New Barrier-Spec

$$\{\mathsf{True}\}\ \texttt{new\_barrier}\ ()\ \{b.\ \mathrm{recv}(b, P) * \mathrm{send}(b, P)\}$$

Signal-Spec

$$\{\mathrm{send}(b, P) * P\}\ \texttt{signal}\ b\ \{\mathsf{True}\}$$

Wait-Spec

$$\{\mathrm{recv}(b, P)\}\ \texttt{wait}\ b\ \{P\}$$

Clone-Spec

$$\{\mathrm{send}(b, P * Q)\}\ \texttt{clone}\ b\ \{\mathrm{send}(b, P) * \mathrm{send}(b, Q)\}$$

Extend-Spec

$$\left\{\mathrm{send}(b, P) * \circledast_{e \in E}\, e \prec b\right\}$$

$$\texttt{extend}\ b$$

$$\left\{b'.\ \mathrm{send}(b, P) * b' \prec b * \mathrm{send}(b', Q) * \mathrm{recv}(b', Q) * \circledast_{e \in E}\, e \prec b'\right\}$$

Recv-Weaken

$$\mathrm{recv}(b, P) * (P \mathrel{-\!\!*} Q) \vdash \mathrm{recv}(b, Q)$$

Send-Strengthen

$$\mathrm{send}(b, P) * (Q \mathrel{-\!\!*} P) \vdash \mathrm{send}(b, Q)$$

Recv-Split

$$\mathrm{recv}(b, P * Q) \vdash \Rrightarrow \mathrm{recv}(b, P) * \mathrm{recv}(b, Q)$$

Renunciation

$$\mathrm{recv}(b, P) * \mathrm{send}(b', Q) * b \prec b' \vdash \Rrightarrow \mathrm{send}(b', P \mathrel{-\!\!*} Q)$$

Figure 4.5: Specification of the barrier

there is no function to delete or shuffle nodes. We thus do not need exclusive ownership of this fact. Persistent propositions have the property that they are duplicable. This means that once we know that $b_1 \prec b_2$ holds, we can duplicate it such that we can distribute the information to multiple threads.

The specifications of `new_barrier`, `signal` and `wait` are the same as in §3.2, and the specification of `clone` is the same as discussed in §4.1.2. The specification of `extend`, given by Extend-Spec, is slightly stronger than discussed in §4.1.3. The precondition now takes a set E of nodes which occurs earlier than b, as $e \prec b$ for all $e \in E$. It then states in the postcondition that $e \prec b'$ for all $e \in E$ holds. This represents the fact that if a node was earlier than b, then it is now also earlier than b', as b' was inserted as the previous node of b. We cannot assume that b' is still the previous node of b, as other threads may already have extended b with other new nodes, but we do know that all nodes which were earlier than b before the `extend` b call, are earlier than b' after the call.

Finally, we have rules for the receive and send predicates. Let us first look at Recv-Weaken and Send-Strengthen as we have not seen those before. We can use Recv-Weaken to weaken the resource we can receive. For example, suppose that we have a function which takes as precondition $\mathrm{recv}(b, \exists n.\ \mathrm{even}(n) * \ell \mapsto n)$, while we own $\mathrm{recv}(b, \ell \mapsto 2)$. We can then use Recv-Weaken and the fact that

$$\ell \mapsto 2 \vdash \exists n.\ \mathrm{even}(n) * \ell \mapsto n$$

to prove the precondition. It thus allows us to receive a weaker resource. Similarly, we can use Send-Strengthen to send a stronger resource. Let us now look at Recv-Split and Renunciation, and especially the update modality $\mid\!\Rrightarrow$, which we did not discuss before. Intuitively, one can read Recv-Split in the following way: if one owns $\mathrm{recv}(b, P * Q)$, they can own $\mathrm{recv}(b, P) * \mathrm{recv}(b, Q)$ after a ghost state update. This ghost state update is a technicality. Similarly to how we defined send and recv in §3.3, their definitions again assert some invariant. During the verification of this rule, we need to temporarily open this invariant, change some ghost state, and close the invariant again. The $\mid\!\Rrightarrow$ modality expresses this. However, this is not a problem for a client using this specification. That is because the following rule allows us to do ghost state updates at any point during a program:

$$\textsc{Hoare-Conseq-Upd}\quad \frac{P \vdash \mid\!\Rrightarrow P' \qquad \{P'\}\, e\, \{v.\, Q'\} \qquad \forall u.\, (Q'[u/v] \vdash \mid\!\Rrightarrow Q[u/v])}{\{P\}\, e\, \{v.\, Q\}}$$

Thus, if we want to prove $\{\mathrm{recv}(b, P * Q)\}\, e\, \{v.\, Q\}$, we can use Hoare-Conseq-Upd combined with Recv-Split after which we need to prove $\{\mathrm{recv}(b, P) * \mathrm{recv}(b, Q)\}\, e\, \{v.\, Q\}$.

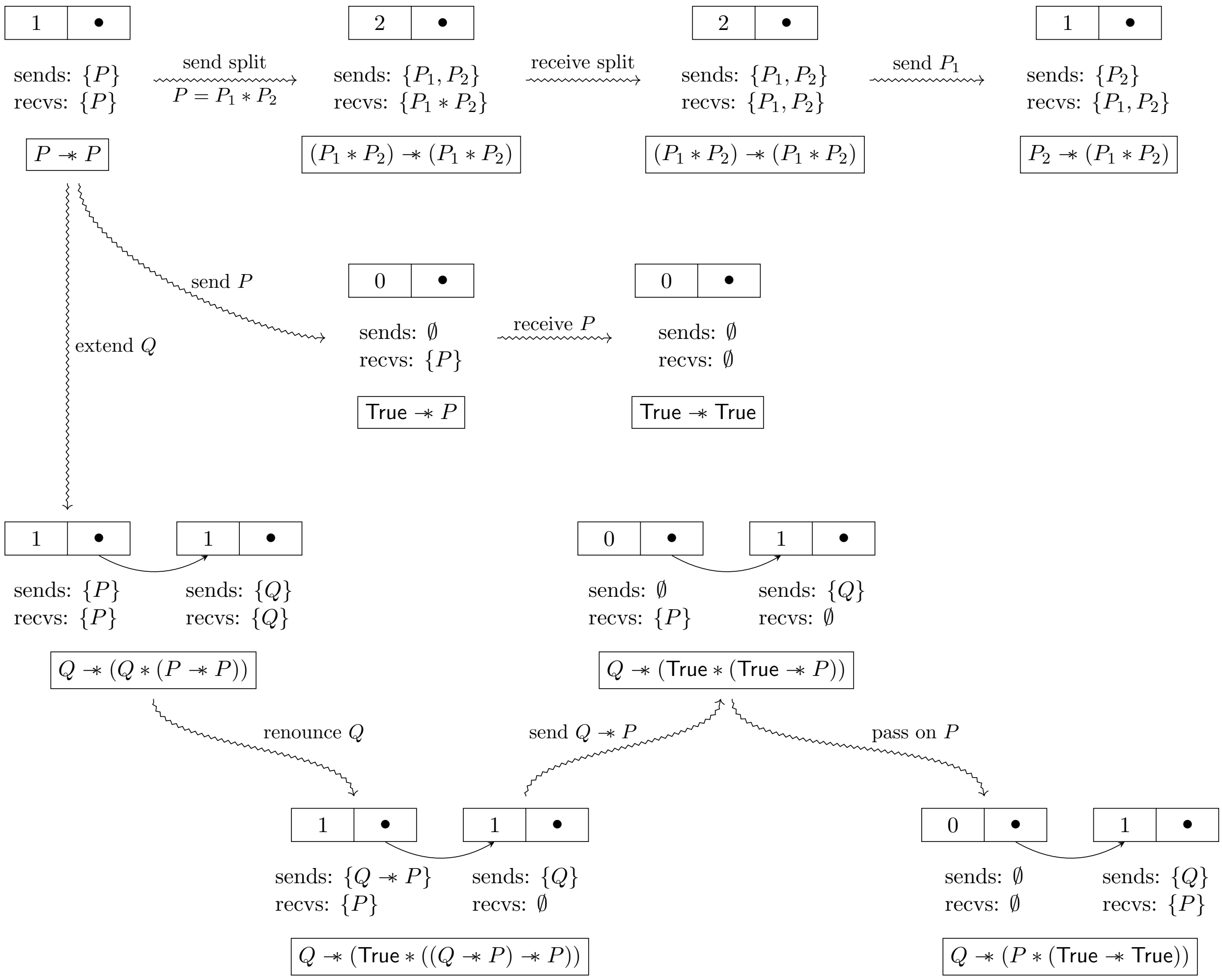


Figure 4.6: Barrier operations

4.4 Intuitive verification

We now present an intuitive verification of the specification. We do this by examining the possible operations a client can perform on a barrier. Figure 4.6 gives an overview of how the physical state relates to the logical state, before and after each operation. We start this section by discussing the invariant which relates the physical state to the logical state (§4.4.1). We also discuss how we represented both the physical and logical state in Figure 4.6. Afterwards, we look at every operation from Figure 4.6 individually, and discuss how we can alter the physical and logical state in such a way that the invariant still holds after the operation.

4.4.1 Invariant

To define a invariant, we first need a way to represent the nodes of a barrier. As discussed in §4.2, a node is represented in the code, or physically, as an optional location ℓ, such that its counter is stored at ℓ and it previous node is stored at $\ell +_{\mathtt{L}} 1$. Logically we represent them using the following inductive record:

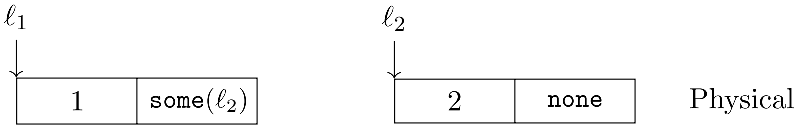
$$
\begin{aligned}
&\mathit{Node} \triangleq \{\\
&\quad \begin{array}{lcl}
\ell & : & \mathit{Loc},\\
\mathit{sends} & : & \mathit{FinMSet}(\mathit{iProp}),\\
\mathit{recvs} & : & \mathit{FinMSet}(\mathit{iProp}),\\
\mathit{prev} & : & \mathit{Option}(\mathit{Node})
\end{array}\\
&\}
\end{aligned}
$$

A node contains a location where we physically store the counter and previous node. It furthermore has two finite multisets of propositions, or resources. The resources that a client still has to send via `signal` are stored in *sends*, and the resources a client can still receive via `wait` are stored in *recvs*. They are both multisets as it should be possible to send the same resource multiple times. Finally, we store an optional previous *Node* in *prev*. This makes *Node* inductive, meaning we can represented a chain of nodes using a single head node n_{hd}.

Note that we do not need to add a field for the counter. That is because the counter stores the number of times a node still needs to be signalled. This is equal to $|\mathit{sends}|$, the size of *sends*, as *sends* contains all the resources that still needs to be sent via `signal`. That means that the value of the counter is already implicitly stored in this record.

To see how this logical *Node* record relates to a physical node, let us consider Figure 4.7. In the top half we represented the physical state of a barrier. There are two nodes. The first one is stored ad ℓ_1 and has its counter set to 1 and its previous node stored in ℓ_2. The second node has its counter set to 2 and no previous node. Logically we could represent this using the *Node* given in the bottom half of the diagram. Note that the sizes of the *sends* fields indeed match up with the counters which are physically stored.

Intuitively, one can interpret the definitions of the send and recv predicates in the following way: for a physical node b represented by a logical node n, there are $\mathrm{send}(b, P)$ predicates for all $P \in n.\mathit{sends}$, and $\mathrm{recv}(b, P)$ predicates for all $P \in n.\mathit{recvs}$. For the rest of this chapter this intuitive definition is enough. On the other hand, for physical nodes b, b' represented by logical nodes n, n', $b \prec b'$ holds whenever either n is the previous node of n', or n is earlier than the previous node of n'. Formally we would define these predicates using ghost state, but we delay this discussion until Chapter 5.



Logical

$$\begin{array}{l}
\{\ell = \ell_1, \\
\quad \mathit{sends} = \{R\}, \\
\quad \mathit{recvs} = \{Q, R\}, \\
\quad \mathit{prev} = \mathsf{Some}(\{ \\
\qquad \ell = \ell_2, \\
\qquad \mathit{sends} = \{P, Q\}, \\
\qquad \mathit{recvs} = \{P\}, \\
\qquad \mathit{prev} = \mathsf{None} \\
\quad \})\} \\
\}
\end{array}$$

Figure 4.7: Physical and logical state

With this logical representation at hand, we can define an invariant. The invariant asserts information about n_{hd}, the head of the chain, and consists of two main parts. The first part describes the physical state, so we refer to it as *the physical invariant*. For each node n in the chain, the physical invariant asserts that the size of $n.\mathit{sends}$ is stored at $n.\ell$, and the value stored at $n.\ell +_{\mathtt{L}} 1$ is `none` when n has no previous node, and $\mathtt{some}(n'.\ell)$ whenever n' is the previous node of n.

The second part of the invariant describes how the resources stored in *sends* and *recvs* relate to each other. From now on, we refer to it as *the resource invariant*. Before we define it, let us look at the barrier in Figure 4.8. This representation mixes the physical and logical state. The values b_1, b_2

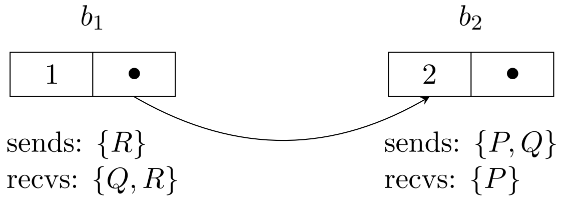


Figure 4.8: Mixed representation

denote the physical representations of the two nodes. The previous node of b_1 is b_2, indicated by the arrow, and b_2 has no previous node. The counters of the nodes are displayed in the their left boxes, and the *sends* and *recvs* multisets are displayed beneath each node. Nodes b_1, b_2 have their counter set to $1, 2$ respectively, as that is the number of resources in their *sends* field. A sender still needs to send R to b_1, and a receiver will be able to receive Q and R from b_1 once the counters of both b_1, b_2 are set to 0. Node b_2 still needs to be send P and Q by a sender, and a receiver will be able to receive R from b_2 once the counter of b_2 is set to 0. One may observe that this is the same barrier as pictured in Figure 4.7. It is however represented in a more visual way, so we use this representation for the rest of the chapter.

Given the barrier from Figure 4.8, we can write its resource invariant in the following way:

$$\underbrace{(P * Q)}_{b_2.sends} \mathrel{-\!\!*} (\underbrace{P}_{b_2.recvs} * (\underbrace{R}_{b_1.sends} \mathrel{-\!\!*} \underbrace{(Q * R)}_{b_1.recvs}))$$

Once both P, Q have been sent to b_2, a receiver can receive P from b_2, as at that point the counter of b_2 will be 0 and it has no previous node. If not only P, Q are sent to b_2, but R is also sent to b_1, then a receiver can also receive Q, R from b_1, as the counters of both itself and its previous node are set to 0.

In general, we can write such a resource invariant for every chain of nodes. For simplicity's sake, we only consider chains with a maximum of two nodes in this chapter. The general case is discussed in Chapter 5. For a chain consisting of a single node b, we can write the resource invariant in the following way:

$$(\circledast_{P \in b.sends} \,.\, P) \mathrel{-\!\!*} (\circledast_{R \in b.recvs} \,.\, R)$$

Once all the resources have been sent to b, the big separating conjunction $\circledast$ ranges over an empty domain, resulting in the following resource invariant:

$$\mathsf{True} \mathrel{-\!\!*} (\circledast_{R \in b.recvs} \,.\, R)$$

This represents the fact that the resources in $b.recvs$ are owned by the resource invariant and are ready to be received. This is similar to how the invariant owned P is §3.3.

For a two node chain with head b_1 and tail b_2, i.e. b_2 is the previous node of b_1, the resource invariant becomes the following:

$$\left(\underset{P \in b_2.sends}{\circledast} P\right) \mathrel{-\!\!*} \left(\left(\underset{R \in b_2.recvs}{\circledast} R\right) * \left(\left(\underset{P \in b_1.sends}{\circledast} P\right) \mathrel{-\!\!*} \left(\underset{R \in b_1.recvs}{\circledast} R\right)\right)\right)$$

Once all resources in $b_2.sends$ have been sent to b_2, a receiver can start receiving resources from $b_2.recvs$. As b_2 is the previous node of b_1, a receiver has to wait until all resources in both $b_2.sends$ and $b_1.sends$ have been sent, before it can receive resources from $b_1.recvs$.

With this invariant at hand, we can verify the operations from Figure 4.6. To verify these operations, we make use of diagrams similar to Figure 4.8.

Additionally, for each barrier, we write its resource invariant beneath it. We now verify that under the assumption that the invariant holds before an operation, we can modify both the physical and logical state in such a way that the invariant also holds after the operation. This is similar to how we verified `signal` and `wait` is Section 3.3. Let us first discuss what happens when we create a barrier in the next section.

4.4.2 Barrier creation

At the end of a `new_barrier` () call, we have physically created a barrier, i.e. we have a value b such that $b = \texttt{some}(\ell)$ where $\ell \mapsto 1$ and $(\ell +_{\mathtt{L}} 1) \mapsto \texttt{none}$. This physical representation can also be seen in Figure 4.9. We now need to create a logical Node n_{hd} such that we can satisfy the invariant. We take

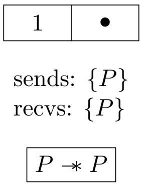


Figure 4.9: Barrier creation

$$n_{hd} \triangleq \{\ell = \ell, \mathit{sends} = \{P\}, \mathit{recvs} = \{P\}, \mathit{prev} = \mathsf{None}\}$$

The $n_{hd}.\ell$ field should obviously be set to ℓ, and $n_{hd}.\mathit{prev}$ should be set to None, as there is no previous node yet. As can be seen in New Barrier-Spec, a client has to pick a resource P they want to send and receive. We thus set both $n.\mathit{sends}$ and $n.\mathit{recvs}$ to $\{P\}$. The physical and logical state now line up with Figure 4.9. To satisfy the invariant, we first notice that the physical part is satisfied if we give up ownership of the points-to predicates for ℓ and $\ell +_{\mathtt{L}} 1$, which we received after the `AllocN` call. Now all that remains is to prove that the resource invariant holds, but this is trivial as $P \mathrel{-\!\!*} P$ is a tautology. We can also prove the postcondition of New Barrier-Spec: we set $n.\mathit{sends} = n.\mathit{recvs} = \{P\}$, so we get ownership of the $\mathrm{send}(b, P)$ and $\mathrm{recv}(b, P)$, which we can give to the postcondition.

4.4.3 Send splitting

As stated by Clone-Spec, if a client owns $\mathrm{send}(b, P_1 * P_2)$, they can give up ownership of $\mathrm{send}(b, P_1 * P_2)$, call `clone` b, and get ownership of $\mathrm{send}(b, P_1)$ and $\mathrm{send}(b, P_2)$. Now suppose that we own $\mathrm{send}(b, P_1 * P_2)$, where b is the barrier represented in the left hand side of Figure 4.10. At the end of the `clone` b call, the counter is incremented by 1, resulting in the physical state represented in the right hand side of Figure 4.10. We now need to update the

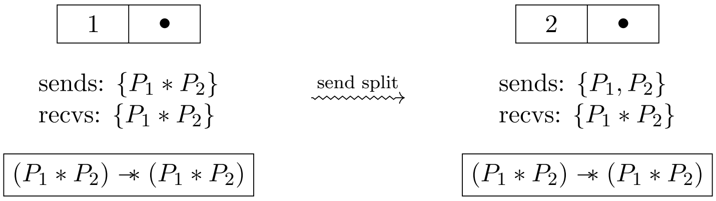


Figure 4.10: Send splitting

logical state in such a way that we can satisfy the invariant again. We do this by removing $P_1 * P_2$ from *sends* and adding P_1, P_2 separately to *sends*. The size of *sends* is now again equal to the counter, so the physical invariant is satisfied. Although we changed the contents of *sends*, the resource invariant stays the same. As it held before the `clone` b operation, it still has to hold after it. All that remains is to prove the postcondition $\text{send}(b, P_1) * \text{send}(b, P_2)$, which holds because both P_1, P_2 are in the *sends* field.

4.4.4 Receive splitting

The verification of Recv-Split is similar. The difference is that there is no physical change, only a logical one. As can be seen in Figure 4.11, we now replace the $P_1 * P_2$ in the *recvs* field with P_1 and P_2 separately. As there is no physical change, the physical invariant still holds. For the same reason as in the verification of send splitting, the resource invariant does not change, so it also still holds. The postcondition $\text{recv}(b, P_1) * \text{recv}(b, P_2)$ is satisfied because both P_1, P_2 are part of the *recvs* field.

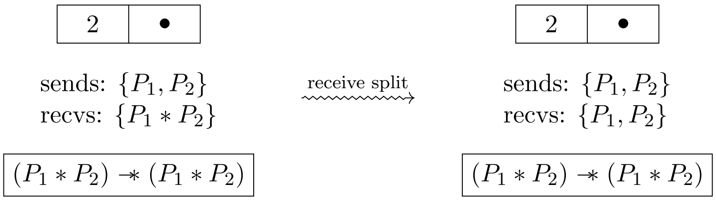


Figure 4.11: Receive splitting

4.4.5 Signalling

The verification of Signal-Spec is interesting, as the resource invariant will no longer be a tautology like it was before. Suppose we own $\text{send}(b, P_1) * P_1$ for the barrier represented in the left hand side of Figure 4.12. At the end of a `signal` b call, the counter has decremented by 1, so the barrier transitions to the physical state represented in the right hand side of Figure 4.12. To satisfy the physical invariant, we remove P_1 from *sends*, as this makes the size of *sends* equal to the counter. The resource invariant however, now

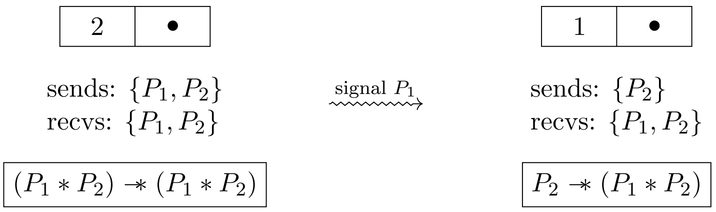


Figure 4.12: Signalling

becomes $P_2 \twoheadrightarrow (P_1 * P_2)$. We can however still prove this, as the precondition of Signal-Spec states we have ownership of P_1. We can thus give up ownership of P_1, and give it to the resource invariant. This is similar to the verification of Signal-Spec in §3.3.2, where we also gave ownership of the resource to the invariant. This makes sure we satisfy the invariant again. The verification is now complete, as there is no postcondition.

4.4.6 Extending

As `extend` is a recursive function, the verification of Extend-Spec makes use of Löb induction, just like we did in §3.3.3. If the extend race is lost, i.e. some other thread has already extended the node we tried to extend, we can use the induction hypothesis. This is possible because the state did not change which means that the invariant still holds. For the rest of the proof we can thus assume that we won the extend race. That means that the physical state updates like in Figure 4.13. We update the logical state by adding a new node b' to the chain. Like in the verification of New Barrier-Spec (§4.4.2), the client picks a resource Q that they want to send to and receive from the new node. We thus set the *sends* and *recvs* field of the new node to $\{Q\}$. The physical invariant is now satisfied as the size of *sends* if set to 1. The resource invariant becomes $Q \twoheadrightarrow (Q * (P \twoheadrightarrow P))$. As the previous resource invariant $P \twoheadrightarrow P$ already holds, we can easily prove $Q \twoheadrightarrow (Q * (P \twoheadrightarrow P))$. Finally, we need to prove the postcondition. As we did not change $b.sends$, we still own $\text{send}(b, P)$, and because we set $b'sends = b'.recvs = \{Q\}$, we also own $\text{recv}(b', Q) * \text{send}(b', Q)$. We know that $b' \prec b$ holds because we set b' as

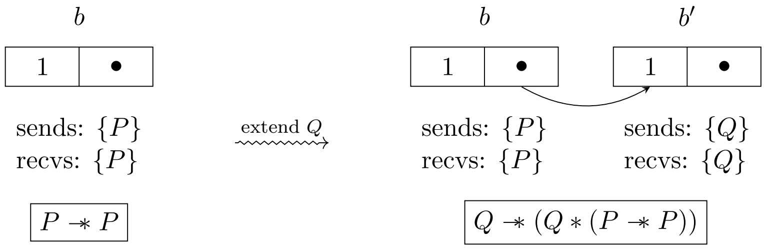


Figure 4.13: Extending

the previous node of b. Finally, to prove $\circledast_{e \in E}\, e \prec b'$, we assume that $e \in E$. By the precondition we can thus assume that $e \prec b$. As this was already true before we updated the previous node of b, and we inserted b' just after b, it has to be the case that $e \prec b'$ also holds, which concludes this verification.

4.4.7 Renunciation

Just like in the verification of Recv-Split (§4.4.4), there is no physical state change, only a logical one. As can be seen in Figure 4.14, we own send(b, P) and recv(b', Q), and we also know that $b' \prec b$. We now want to update the logical state such that we get ownership of send$(b, Q \mathrel{-\!\!*} P)$. We do this by replacing P with $Q \mathrel{-\!\!*} P$ in $b.sends$ and removing Q from $b'.recvs$. We did not change the sizes of the *sends* fields, so the physical invariant still holds. The resource invariant changes from $Q \mathrel{-\!\!*} (Q * (P \mathrel{-\!\!*} P))$ to $Q \mathrel{-\!\!*} (\mathsf{True} * ((Q \mathrel{-\!\!*} P) \mathrel{-\!\!*} P))$, which still holds. As $b.sends$ contains $Q \mathrel{-\!\!*} P$, we now own send$(b, Q \mathrel{-\!\!*} P)$, which concludes the verification.

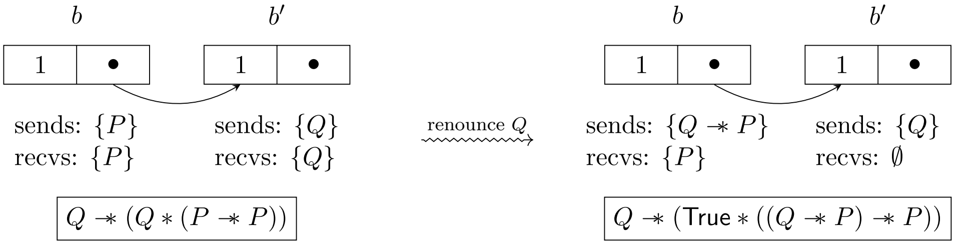


Figure 4.14: Renunciation

4.4.8 Receiving

The verification of Wait-Spec is interesting, as it involves two different operations. The operation which is performed, depends on whether a node is the last in the chain. As `wait` is recursive, we again use Löb induction, so we only consider the interesting case where the counter is set to 0.

Let us first consider the case where we try to receive a resource P from the last node in a chain, like in Figure 4.15. In this case the resource invariant

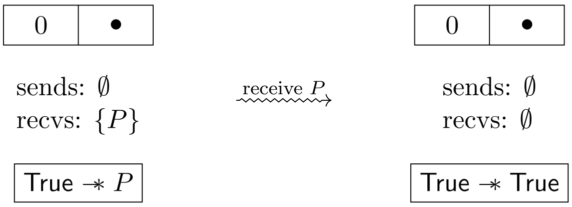


Figure 4.15: Receiving

is just simply $\mathsf{True} \mathrel{-\!\!*} P$. We can thus update the logical state by removing P from *recvs*, after which the new resource invariant becomes $\mathsf{True} \mathrel{-\!\!*} \mathsf{True}$. This means we can receive P from the invariant, as it no longer needs it to satisfy the resource invariant. This is similar to the proof of Wait-Spec in §3.3.3, where we also received P from the invariant. The postcondition of Wait-Spec is P, which we now own, so that concludes this case.

We now consider the case where we try to receive a resource P from a node b which has a previous node b', like in Figure 4.16. Before we can receive P, we still need to wait for the second node to be signalled, as `wait` waits on all previous nodes. We thus *pass on* P to its previous node. This means we remove P from $b.\mathit{recvs}$, and add it to $b'.\mathit{recvs}$. After this update to the logical state, the resource invariant changes from $Q \mathrel{-\!\!*} (\mathsf{True} * (\mathsf{True} \mathrel{-\!\!*} P))$ to $Q \mathrel{-\!\!*} (P * (\mathsf{True} \mathrel{-\!\!*} \mathsf{True}))$. Under the assumption that the first resource invariant holds, we can easily prove that the new resource invariant also holds. Furthermore, we now have ownership of recv(b', P) as we passed on P to b'. We can thus use our induction hypothesis to receive P after the recursive `wait` b' call.

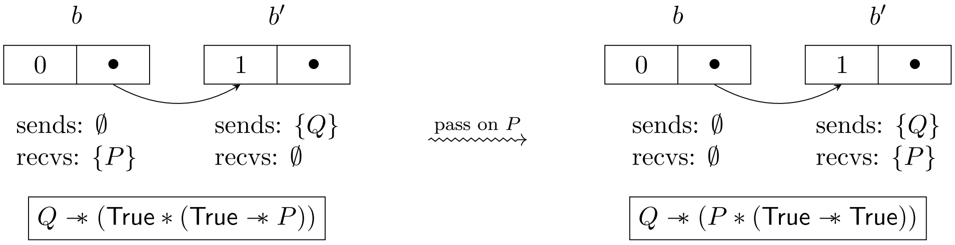


Figure 4.16: Passing on

We can now see how in general a resource P is received. The resource P gets passed on through the chain every time a counter is set to 0. At some point, P will be passed to the last node in the chain. Once the counter of the last node is also set to 0, we can receive P.

Chapter 5

Formalization

In this chapter we present our formalization to verify the specification of the chainable barrier (Figure 4.5). Like we did in the verification of the simple barrier (§3.3), we have to find the right definitions for the abstract send, recv and $\prec$-predicates. Given the right definitions, we verify the rules from Figure 4.5 using the proof rules of Iris. With the verification of the barrier at hand, we can use it to verify clients, like we did in Figures 4.1, 4.2, 4.3. To obtain confidence that our barrier specification is indeed the right one, we can use Iris' adequacy theorem (Jung et al., 2018, §7.4) to obtain a closed proof stating that the client is safe.

In order to find the right definitions, we use existing Iris techniques, like we did in the verification of the simple barrier (§3.3), and combine these with two new constructions. The first new construction is called *authoritative ordering*, which we present in §5.1. It can be used to keep track of the order of elements in a list via ghost state. The second new construction is *recursively nested wands*. This construction is discussed together with the definitions of the abstract send, recv and $\prec$-predicates in §5.2. This chapter is quite technical, so readers are assumed to have sufficient knowledge from "Iris from the group up" (Jung et al., 2018).

5.1 Authoritative ordering

To model the $\prec$-relation we need a resource which keeps track of the order of the nodes. In this section we describe a construction which does exactly that. This construction does no depend on the use of locations. In fact, we can use this construction to assert an ordering on lists of any type. The orderings we can describe using this construction are persistent, and we can always extend an ordering by adding an element somewhere in the ordering. However, we cannot remove elements from the ordering. The design of *authoritative ordering* is inspired by a similar construction in Iris, which models partial bijections (The Iris Team, 2023, `iris/base_logic/lib/gset_bij.v`). We discuss its differences and similarities in Chapter 6. We now look at the specification of *authoritative ordering*, and then discuss its implementation.

Specification. We write $\mathsf{auth}_\gamma(\vec{x})$ for *authoritative* ownership of the ordering on list $\vec{x}$. The ordering of the elements is according to their order in $\vec{x}$. The owner of this resource allowed to update the ordering by adding new elements to $\vec{x}$. No one else can own this resource as it is exclusive. We therefore call this ownership authoritative. We write $a \prec_\gamma b$ for *fragmental* ownership of the same ordering. It asserts that a has a lower index in $\vec{x}$ than b. This ownership is fragmental, as it does not say anything about the other elements in $\vec{x}$. The ghost variable γ relates the fragments to the authoritative ordering. For example, given $\gamma \neq \gamma'$, if we own $\mathsf{auth}_\gamma(\vec{x}) * a \prec_\gamma b$ and $\mathsf{auth}_{\gamma'}(\vec{x'})$, we know that a occurs earlier in $\vec{x}$ than b, but we do not know anything about the order of a and b in $\vec{x'}$, or if they are even elements of $\vec{x'}$.

Let us now look at the specification of these resources.

auth_γ-Timeless
$$\mathsf{timeless}(\mathsf{auth}_\gamma(\vec{x}))$$

$\prec_\gamma$-Timeless
$$\mathsf{timeless}(a \prec_\gamma b)$$

auth_γ-Exclusive
$$\mathsf{auth}_\gamma(\vec{x}) * \mathsf{auth}_\gamma(\vec{x'}) \vdash \mathsf{False}$$

$\prec_\gamma$-Persistent
$$a \prec_\gamma b \vdash \Box(a \prec_\gamma b)$$

$\prec_\gamma$-Trans
$$a \prec_\gamma b * b \prec_\gamma c \vdash a \prec_\gamma c$$

$\prec_\gamma$-Get
$$\frac{\exists i < j.\ \wedge \vec{x}(i) = a \wedge \vec{x}(j) = b}{\mathsf{auth}_\gamma(\vec{x}) \vdash a \prec_\gamma b}$$

$\prec_\gamma$-Indices
$$\mathsf{auth}_\gamma(\vec{x}) * a \prec_\gamma b \vdash \exists i < j.\ \wedge \vec{x}(i) = a \wedge \vec{x}(j) = b$$

auth_γ-NoDup
$$\mathsf{auth}_\gamma(\vec{x}) \vdash \mathsf{NoDup}(\vec{x})$$

auth_γ-Alloc
$$\mathsf{NoDup}(\vec{x}) \vdash |\!\dot{\Rrightarrow} \exists \gamma.\ \mathsf{auth}_\gamma(\vec{x})$$

auth_γ-Update
$$\frac{a \notin \vec{x} +\!\!+ \vec{x'}}{\mathsf{auth}_\gamma(\vec{x} +\!\!+ \vec{x'}) \vdash |\!\dot{\Rrightarrow}\ \mathsf{auth}_\gamma(\vec{x} +\!\!+ a :: \vec{x'})}$$

Both auth_γ and $\prec_\gamma$ are timeless, which allows us to remove $\triangleright$ modalities around them when we are proving Hoare triples. The exclusivity of auth_γ is captured by auth_γ-Exclusive. As we use the authoritative ordering to model the $\prec$-relation on barrier nodes, it makes sense that we have $\prec_\gamma$-Persistent and $\prec_\gamma$-Trans. It does however mean that one cannot remove elements from the ordering, as $\prec_\gamma$ is persistent.

The $\prec_\gamma$-Get rule states that the owner of the authoritative element can prove that $a \prec_\gamma b$ by showing that the index of a in $\vec{x}$ is smaller than the index of b. On the other hand, if the owner of the authoritative element knows that $a \prec_\gamma b$, they can get the indices of a and b using $\prec_\gamma$-Indices.

The rule auth_γ-NoDup states that an ordering cannot contain duplicate elements. Without this requirement, one could use $\prec_\gamma$-Trans and $\prec_\gamma$-Indices to prove that c has a lower index than b in the list $[a, b, c, a]$.

To allocate a new ordering, one can use auth_γ-Alloc. The premise requires that the list $\vec{x}$ does not contain duplicate elements, for the same reason as explained above. The conclusion states that one receives a ghost variable γ and ownership of $\mathsf{auth}_\gamma(\vec{x})$, after executing a ghost state update. Recall that this can be done during program execution using Hoare-Conseq-Upd.

Finally, one can use auth_γ-Update to add a new element to the ordering, given that it was not in the ordering before. This element is added to the ordering after a ghost state update has happened.

Implementation. To implement this, we use the following definitions:

$$\begin{aligned}\mathsf{auth}_\gamma(\vec{x}) \triangleq \exists X.\, \mathsf{NoDup}(\vec{x}) * \boxed{\bullet X}^{\gamma} * \boxed{\circ X}^{\gamma} \\ * \forall a, b.\ ((a, b) \in X \leftrightarrow \exists i < j.\ \wedge \vec{x}(i) = x \wedge \vec{x}(j) = y)\end{aligned}$$

$$a \prec_\gamma^{\mathrm{pre}} b \triangleq \boxed{\circ\{(a, b)\}}^{\gamma} \qquad\qquad a \prec_\gamma b \triangleq a\ (\prec_\gamma^{\mathrm{pre}})^{+}\ b$$

The resource algebra being used is $\mathrm{Auth}((\mathit{FinSet}(A \times A), \cup))$, where A is any type. The authoritative element quantifies over a set X. This set contains pairs (a, b), whenever a has a lower index in $\vec{x}$ than b. Furthermore, we assert that $\vec{x}$ does not contain duplicates, and we assert ownership of both $\boxed{\bullet X}^{\gamma}$ and $\boxed{\circ X}^{\gamma}$.[1]

To define $\prec_\gamma$, we first define $\prec_\gamma^{\mathrm{pre}}$, which asserts ownership of the fragment $\boxed{\circ\{(a, b)\}}^{\gamma}$. Due to the definition of auth_γ, this makes sure that a has a lower index in the list than b. Finally, to allow us to prove $\prec_\gamma$-Trans, we define $\prec_\gamma$ as the transitive closure.[2] of $\prec_\gamma^{\mathrm{pre}}$

The proof that this implementation satisfies the implementation can be found in the Coq formalization (van Collem, 2023). Let us next look at how we can use this construction to define the send, recv and $\prec$-predicates.

5.2 Definitions

The complete definitions can be found in Figure 5.1. Let us start of by discussing the logical state. The *Node* record is similar to the one from §4.4. The only difference is that the *sends* and *recvs* fields no longer have type

[1] The ownership of the fragment $\boxed{\circ X}^{\gamma}$ may seem a bit weird, but it is actually needed to prove $\prec_\gamma$-Get. This trick was also used in the partial bijection construction we mentioned earlier.

[2] This is the transitive closure for Iris propositions, which we added to Iris as part of this thesis: https://gitlab.mpi-sws.org/iris/iris/-/merge_requests/862

$$
\begin{array}{lll}
\mathit{Node} \triangleq \{ & & \\
\quad \ell & : & \mathit{Loc}, \\
\quad \mathit{sends} & : & \mathit{FinSet}(\mathit{GName}), \\
\quad \mathit{recvs} & : & \mathit{FinSet}(\mathit{GName}), \\
\quad \mathit{prev} & : & \mathit{Option}(\mathit{Node}) \\
\} & &
\end{array}
\qquad\qquad
\begin{array}{lll}
\mathit{BName} \triangleq \{ & & \\
\quad s & : & \mathit{GName}, \\
\quad o & : & \mathit{GName}, \\
\quad r & : & \mathit{GName} \\
\} & &
\end{array}
$$

$$
\begin{aligned}
&\mathrm{chain}\colon \mathit{Node} \to \mathit{List}(\mathit{Node}) \\
&\mathrm{chain}(n) \triangleq \\
&\quad \begin{cases} [n] & \text{if } n.\mathit{prev} = \mathsf{None} \\ n :: \mathrm{chain}(n') & \text{if } n.\mathit{prev} = \mathsf{Some}(n') \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{resources}\colon \mathit{Node} \to \mathit{iProp} \to \mathit{iProp} \\
&\mathrm{resources}(n, P_{acc}) \triangleq \\
&\quad \mathtt{let}\ P'_{acc} = \left(\mathop{\circledast}_{\gamma \in n.\mathit{sends}} \exists P.\ \triangleright P * \gamma \Mapsto P \right) \mathrel{-\!\!*} \left(\left(\mathop{\circledast}_{\gamma \in n.\mathit{recvs}} \exists P.\ \triangleright P * \gamma \Mapsto P \right) * P_{acc} \right)\ \mathtt{in} \\
&\quad \begin{cases} P'_{acc} & \text{if } n.\mathit{prev} = \mathsf{None} \\ \mathrm{resources}(n', P'_{acc}) & \text{if } n.\mathit{prev} = \mathsf{Some}(n') \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&I\colon \mathit{BName} \to \mathit{iProp} \\
&I(\gamma) \triangleq \exists n_{hd}.\ \boxed{\bullet \left(\bigcup \{ n.\ell \mapsto n.\mathit{sends} \mid n \in \mathrm{chain}(n_{hd}) \}\right)}^{\gamma.s} * && (1) \\
&\qquad\quad \boxed{\bullet \left(\biguplus \{ n.\ell \mapsto n.\mathit{recvs} \mid n \in \mathrm{chain}(n_{hd}) \}\right)}^{\gamma.r} * && (2) \\
&\qquad\quad \mathsf{auth}_{\gamma.o}([n.\ell \mid n \in \mathrm{chain}(n_{hd})]) * && (3) \\
&\qquad\quad \mathrm{resources}(n_{hd}, \mathsf{True}) * && (4) \\
&\qquad\quad \left(\mathop{\circledast}_{n \in \mathrm{chain}(n_{hd})} \begin{cases} n.\ell \mapsto_{\square} 0 * & (n.\ell +_{\mathtt{L}} 1) \mapsto \lceil n.\mathit{prev}.\ell \rceil \quad \text{if } |n.\mathit{sends}| = 0 \\ n.\ell \mapsto |n.\mathit{sends}| * & (n.\ell +_{\mathtt{L}} 1) \mapsto \lceil n.\mathit{prev}.\ell \rceil \quad \text{otherwise} \end{cases} \right) && (5)
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{send}\colon \mathit{Val} \to \mathit{iProp} \to \mathit{iProp} \\
&\mathrm{send}(\ell, P) \triangleq \exists \gamma, \gamma_P, P'.\ \ell \in \mathit{Loc} * (P \mathrel{-\!\!*} P') * \boxed{\circ\{\ell \mapsto \{\gamma_P\}\}}^{\gamma.s} * \\
&\qquad\qquad\qquad\qquad \gamma_P \Mapsto P' * \mathsf{meta}(\ell, \mathcal{N}, \gamma) * \boxed{I(\gamma)}^{\mathcal{N}}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{recv}\colon \mathit{Val} \to \mathit{iProp} \to \mathit{iProp} \\
&\mathrm{recv}(\ell, P) \triangleq \exists \gamma, \gamma_P, P'.\ \ell \in \mathit{Loc} * (P' \mathrel{-\!\!*} P) * \boxed{\circ\{\ell \mapsto \{\gamma_P\}\}}^{\gamma.r} * \\
&\qquad\qquad\qquad\qquad \gamma_P \Mapsto P' * \mathsf{meta}(\ell, \mathcal{N}, \gamma) * \boxed{I(\gamma)}^{\mathcal{N}}
\end{aligned}
$$

$$
\begin{aligned}
&\prec\colon \mathit{Val} \to \mathit{Val} \to \mathit{iProp} \\
&\ell \prec \ell' \triangleq \exists \gamma.\ \ell \in \mathit{Loc} * \ell' \in \mathit{Loc} * \mathsf{meta}(\ell, \mathcal{N}, \gamma) * \mathsf{meta}(\ell', \mathcal{N}, \gamma) * \ell' \prec_{\gamma.o} \ell
\end{aligned}
$$

Figure 5.1: Definitions

$\mathit{FinMSet}(\mathit{iProp})$, but rather $\mathit{FinSet}(\mathit{GName})$. This is because we use *saved propositions* (Dodds et al., 2016, Section 5.1) to keep track of the resources that still needs to be sent or received. That means that *sends* is a finite set of ghost names, where each ghost name corresponds with a resource that still needs to be sent. There can be multiple ghost names with the same associated resource, so we can still send the same resource multiple times. The *recvs* field stores the same information, just for resources that still need to be received. Again, the counter of a node is stored implicitly in this record, as it corresponds to the size of *sends*.

Now that we have a notion of a logical state, let us look at the barrier invariant I. It takes a structure γ which contains three ghost names. We use $\gamma.s$ to store the resources which still have to be sent, $\gamma.r$ to store the resources which still have to be received, and $\gamma.o$ to store the order of elements in the chain. The invariant starts by extensionally quantifying over a node n_{hd}, which represents the head of the chain. The rest of the invariant can be split into five separate parts. Parts (1), (2) keep track of all the resources that still have to be sent or received. We use $\mathrm{chain}(n_{hd})$ to get an ordered list of all the nodes in the chain, and map all their locations to their corresponding *sends* and *recvs* sets. We use the $\mathrm{Auth}(\mathit{FinMap}(\mathit{Loc}, (\mathit{FinSet}(\mathit{GName}), \uplus)))$ resource algebra to store these. This allows us to add new resources with fresh identifiers, and remove resources if we have ownership of some fragment $\circ\{\ell \mapsto \gamma_P\}$. It is important to note that we use a disjoint union $\uplus$ in (2). This asserts that the values of the map, i.e. $n.\mathit{recvs}$ for $n \in \mathrm{chain}(n_{hd})$, are distinct. This is essential for the *passing on* of resources we discussed in §4.4.8. In this case we need to pass on an identifier γ_P, which is only possible whenever γ_P is not yet part of the *recvs* field of the previous node. We can keep all $n.\mathit{recvs}$ sets distinct, as we can always allocate saved propositions with fresh identifiers.

In (3), the invariant asserts the authoritative ordering of the locations in the chain. It is important to note that $\mathrm{chain}(n)$ returns a list where the order of the nodes is reversed, i.e. the last node in the list corresponds to the *earliest* node. We take this into account in the definition of the $\prec$-predicate.

In (4), the resource invariant is asserted using $\mathrm{resources}(n, \mathsf{True})$. It is similar to how we defined it for chains with a maximal length of 2 in §4.4.1, but it has a few technical differences. First, the big separating conjunctions ranges over ghost names γ due to the use of saved propositions. We must therefore extensionally quantify over a resource P which is saved at γ, as asserted by $\gamma \Mapsto P$. Second, all resources are guarded by a $\triangleright$ modality. This is because saved propositions do not agree directly, as can be seen in the following rule:

Saved Prop-Agree

$$\gamma \Mapsto P * \gamma \Mapsto Q \vdash \triangleright(P \leftrightarrow Q)$$

By guarding all resources with a $\triangleright$ modality, we can use their agreement at a later step index. This has the consequence that the resource received in `wait` is received under a $\triangleright$. However, the resource is received after the previous node is read. Afterwards the program still does some steps, which allows us to eliminate the $\triangleright$ modality. Finally, to allow for chains of arbitrary size, the resources function is recursive. It uses a resource accumulator P_{acc}, which is nested under a wand each recursive call. This is what we call a *recursively nested wands* construction. It has the consequence that the $n_{hd}.recvs$ resources are nested under a wand for each earlier node. This is exactly the behavior we want, as this means all the *sends* resources of the earlier nodes have to be sent, before we can receive any resource of $n_{hd}.recvs$.

Finally, the invariant relates the logical state to the physical state in (5). It holds the points-to predicates of all the nodes in the chain, asserting their counter and previous values. We write $\lceil n.prev.\ell \rceil$ to lift the location of the optional previous node from the meta logic to a HeapLang optional location value. Once a counter has become 0, it can never increase again. We therefore make its points-to predicate persistent in that case. The reason why we do this, has to do with the implementation of `wait`. In Figure 4.4, we can see that we read both *counter* and *prev*, with two separate reads. We thus need to open the invariant twice, and we want to be sure that the counter is still 0 when we do the second read. This then allows us to either pass on the resource to the previous node, or extract it from the resource invariant and give it to the postcondition, like we discussed in §4.4.8.

The send and recv predicates are similar to each other. We first discuss the send predicate, and then discuss the differences with recv. The $\text{send}(\ell, P)$ predicate states that the value ℓ is actually a location. It furthermore extensionally quantifies over γ, γ_P and P'. The γ is used to assert the barrier invariant $\boxed{I(\gamma)}^{\mathcal{N}}$. The resource P' is saved at γ_P, as asserted by $\gamma_P \Mapsto P'$. It relates to P by $P \mathrel{-\!\!*} P'$. This allows to verify Send-Strengthen, as P, the resource we send, may be stronger than the stored resource P'. Ownership of $\boxed{\circ\{\ell \mapsto \{\gamma_P\}\}}^{\gamma.s}$ indicates that γ_P is indeed one of the identifiers which has not been sent yet. Due to the saved proposition we thus know that P' is part of the resource invariant. Finally, we need to make sure that when two nodes are in the same chain, they agree on the same invariant, and thus, agree on the same value for γ. We do this using `meta` predicates, which allows us to associate logical data with locations (The Iris Team, 2023, `iris/base_logic/lib/gen_heap.v`). We explain how this asserts the agreement on γ record, when we discuss the $\prec$-predicate.

The differences between the send and recv predicates are subtle. To verify Recv-Weaken, we assert $P' \mathrel{-\!\!*} P$, as P, the resource a client receives, may be weaker than the stored resource P'. Furthermore, the ownership of $\boxed{\circ\{\ell \mapsto \{\gamma_P\}\}}^{\gamma.s}$ is changed to ownership of $\boxed{\circ\{\ell \mapsto \{\gamma_P\}\}}^{\gamma.r}$ (the ghost name is changed from $\gamma.s$ to $\gamma.r$), as $\text{recv}(b, P)$ should assert that P is yet to be received.

The $\ell \prec \ell'$ predicate again states that the values ℓ, ℓ' are actual locations. It also asserts ownership of $\ell' \prec_{\gamma.o} \ell$. Note that we swapped the order here: $\ell \prec \ell'$ asserts that $\ell' \prec_{\gamma.o} \ell$. This is because, as explained earlier, the authoritative ordering in the invariant (3) is reversed. Finally, we use meta predicates to associate γ with both locations. Two meta predicates always agree on the associated data with a location:

$$\mathsf{meta}(\ell, \mathcal{N}, \gamma) * \mathsf{meta}(\ell, \mathcal{N}, \gamma') \vdash \gamma = \gamma'$$

Thus, if we have ownership of $\mathrm{recv}(\ell, P)$, $\mathrm{send}(\ell', Q)$ and $\ell \prec \ell'$, like in the verification of Renunciation, we can conclude that both nodes agree on the same invariant, as $\ell \prec \ell'$ states that they are both associated with the same γ.

Chapter 6

Related Work

In this chapter we discuss related work. We start of by discussing the differences between our work and the verification by Dodds et al. We then compare our two new constructions, *authoritative ordering* and *recursively nested wands*, to earlier work, and conclude by discussing a separation logic with Pthreads-style barriers.

Specification and implementation. Our specification from Figure 4.5 is slightly different from the one presented by Dodds et al., 2016. First of all, we added a `clone` operation which allows for send splitting. We furthermore added Recv-Weaken and Send-Strengthen. Jung et al., 2016 already added Recv-Weaken to the specification in their Coq formalization, and we added Send-Strengthen in a similar way.

Our specification of `extend` is also slightly different from Dodds et al., 2016. This is because their `extend` function returned a tuple of nodes, where the second node was the same as the argument. However, this was not clear from the specification. Our `extend` only returns the new node, which allows for a shorter Extend-Spec rule.

Dodds et al., 2016 had to reason about *stability* of resources (Wickerson et al., 2010) in their specification. They had to do this because some iCAP propositions are not *stable*, which means that a proposition may be invalidated after another thread has executed. We, however, did no have to reason about stability, as all Iris propositions are stable (Jung, 2020, §7.1).

Finally, we had to add a `CAS` loop to the implementation of `extend`. By introducing send splitting, it is possible that multiple threads try to extend a node at the same time. To deal with this data race, we use a `CAS` loop. As Dodds et al., 2016 did not have send splitting, they were sure that their was only one thread allowed to extend a node, hence why they did not have to take care of the data race in their implementation.

Logical state. The main difference between our verification in Chapter 5 and Dodds et al., 2016, Section 6, is how we defined our logical state. Their CNode structure, which corresponds to our *Node* structure, has an additional $\mathcal{W}$ field, which holds a finite set of ghost names.[1] With this field, each node keeps track of the resources that are renounced on earlier nodes, and to which they will later get access. In our verification, when we renounce a resource we directly move it from the *earlier* to the *later* node. They instead keep the resource in the *recvs* set of the earlier node, but also add it to the $\mathcal{W}$ set of the later node. Their resource invariant then roughly becomes the following:[2]

$$\mathop{\circledast}_{n \in \mathsf{chain}(n_{hd})} \left(\mathop{\circledast}_{\gamma \in n.sends \cup n.\mathcal{W}} \exists P.\, \triangleright P * \gamma \Mapsto P \right) \mathrel{-\!\!*} \left(\mathop{\circledast}_{\gamma \in n.recvs} \exists P.\, \triangleright P * \gamma \Mapsto P \right)$$

A resource $P \in n.recvs$ can thus only be received once all resources in $n.sends$ and $n.\mathcal{W}$ have been supplied. This makes their resource invariant simpler, as their is no need for recursively nested wands. However, receiving a renounced resource becomes more involved. They show that, for a node n, once all previous and its own counter are set to 0, one can remove all resources from $n.\mathcal{W}$, by cancelling them with earlier renounced resources (Lemma 6.3). To prove this they need an additional property in their invariant, namely well-formedness, which states that for each node n, the resources stored in $n.\mathcal{W}$ are actually available from nodes earlier in the chain. Proving preservation of well-formedness is conceptually not too hard, but it does require a lot of bookkeeping, which can become quite involved in a proof assistant. We do not have to do this bookkeeping, as we move resources immediately when they get renounced, as opposed to doing it when all resources are sent.

State transition systems. Our verification made use of *resource algebras*, while both Dodds et al., 2016 and Jung et al., 2016 used *state transitions systems* (STS) for the verification of the barriers.[3] State transition systems, whose use in separation logic was pioneered by CaReSL (Turon, Dreyer, & Birkedal, 2013; Turon, Thamsborg, et al., 2013), are an intuitive way to write down how different threads interact with a program. It consists of a set of states, transitions between them and a state interpretation function. Some transitions are only allowed by certain threads, which is modelled by the transition requiring a certain token from a thread. The state interpretation

[1] The terminology and field names they use is a bit different, e.g. they have *region identifiers* instead of ghost names, and their $\mathcal{I}$ field corresponds to our *recvs* field. For simplicity, we stick to the terminology used in this thesis.

[2] One can clearly see that our resource invariant is inspired by theirs.

[3] The verification of the simple barrier by Jung et al., 2016 was later redone using resource algebras, see https://gitlab.mpi-sws.org/iris/examples/-/merge_requests/16.

function relates the state of the STS with both the physical and the logical state.

While a STS is an intuitive representation of a protocol on paper, reasoning with it in Coq is rather tedious. In Iris, it is more common to reason with resources algebras (Jung et al., 2018, §2.1). As Jung, 2020 stated in his PhD thesis: "This is probably the biggest hurdle someone has to overcome to become proficient in Iris". However, once proficient with them, proofs typically become smaller.[4]

Authoritative ordering. Our authoritative ordering construction (§5.1) is inspired by a construction which models partial bijections (The Iris Team, 2023, `iris/base_logic/lib/gset_bij.v`). In a similar way to authoritative ordering, there is an authoritative element which keeps track of the complete partial bijection, and fragments which keep track of individual pairs which are related by the bijection. They make use of the *view camera* (The Iris Team, 2023, `iris/algebra/view.v`), which is a generalization of the *authoritative camera* (Jung et al., 2015, §3.6) we use. If we want to generalize the authoritative ordering construction, on which we shortly comment in Chapter 7, it would be good to use *view camera* as well, although that probably means we would need another abstraction layer to close the fragmental elements under transitivity.

Recursively nested wands. The recursively nested wands construction is somewhat similar to the *accessor* pattern in Iris (Jung, 2020, §5.6), which is related to a *ramification* (Hobor & Villard, 2013). A ramification is of the following form:

$$R \vdash P * (Q \mathrel{-\!\!*} R')$$

If one has ownership of R, they can exchange it for ownership of P, some smaller part of the resource. They can then update that resource to Q and update the whole resource to R' using $Q \mathrel{-\!\!*} R'$. This allows one to temporarily focus on a specific part of R and then update it to R'.

If we compare this to the resource invariant, we can see a similar pattern. As an illustration, let us take the following resource invariant, which we used as an example in §4.4.1:

$$\underbrace{(P * Q)}_{b_2.\mathit{sends}} \mathrel{-\!\!*} (\underbrace{P}_{b_2.\mathit{recvs}} * (\underbrace{R}_{b_1.\mathit{sends}} \mathrel{-\!\!*} \underbrace{(Q * R)}_{b_1.\mathit{recvs}}))$$

Before one can receive, or access, R, all of P, Q, R have to be sent. The main difference is that these resources are gradually being sent. One typically

[4] For example, when the simple barrier verification in Iris was rewritten to use resources algebras instead of a STS, there was a `+87,-224` line difference https://gitlab.mpi-sws.org/iris/examples/-/merge_requests/16/diffs.

uses the accessor pattern to temporarily access a small part of the resource, where as the premises of the resource invariant are gradually satisfied until one can receive a resource.

Pthreads-style barrier. Hobor and Gherghina, 2011 designed a separation logic with pthreads-style barriers as a primitive. This is different from our work, as we implemented a barrier with low-level operations, while they took barriers as a primitive and assumed operational semantics for them. Using this operational semantics they proved a higher order specification for the barriers. They verified the soundness of their separation logic using Coq.

The semantics of the pthreads barriers studied by Hobor and Gherghina, differs slightly from the barriers studied by us: a **barrier** bn call, signals the barrier, but also waits until a number of other threads made a **barrier** bn call as well. They can thus be used to synchronize multiple threads. After the threads are synchronized, the barrier is reset and can be used again for another synchronization. This is different from our implementation, where the `signal` and `wait` functions are split up, and a barrier cannot be reset after its counter is set to 0.

Their key insight was that these barrier are used to redistribute ownership of resources between threads. They furthermore noticed that this can be modelled using a finite automata, where each barrier synchronization corresponds to a transition in the automata. To model this in their separation logic, they added a $\mathsf{barrier}(bn, \pi, cs)$ proposition, which states that barrier bn, owned with fractional permission π (Bornat et al., 2005; Boyland, 2003), is in state cs. To make sure that a state transition from cs to ns only redistributes resources, they added the following conditions:

$$\left.\begin{aligned} \circledast_i \, Pre_i &= F * \mathsf{barrier}(bn, \blacksquare, cs) \\ \circledast_i \, Post_i &= F * \mathsf{barrier}(bn, \blacksquare, ns) \end{aligned}\right\} \tag{6.1}$$

$$\exists \pi.\, Pre_i \Rightarrow \top * \mathsf{barrier}(bn, \pi, cs) \tag{6.2}$$

For a thread i, its pre and postconditions of the **barrier** bn are Pre_i, $Post_i$ respectively. Equation 6.1 requires that the combination of all preconditions should include full ownership of the barrier, denoted by the $\blacksquare$. Furthermore, the combination of all postconditions should be the almost the same: only the barrier resource is allowed to transition from state cs to ns. Equation 6.2 requires that each thread has ownership of some part of the barrier resource. Using these requirements,[5] a user can define a automata and add it to the context Γ. This gives rise to their Hoare rule for **barrier** bn:

$$\frac{\Gamma[bn] = bd \qquad \mathsf{lookup\_move}(bd, cs, dir, mv) = (P, Q)}{\Gamma \vdash \{P\}\ \textbf{barrier}\ bn\ \{Q\}}$$

[5] and a few more technical requirements

If there is a transition in the automata bd with precondition P and postcondition Q, then one can use these as pre and postconditions of the **barrier** bn call. Note by construction of the automata, condition 6.2 has to hold for P.

In follow-up work, Hobor and Gherghina, 2012 integrated this logic into the HIP/SLEEK program verification toolset (Gherghina et al., 2011; Nguyen & Chin, 2008). In their earlier work, they showed how one could define a barrier automata in their Coq development. Even for a quite simple automata, with only 4 states and 4 transitions, they had a Coq file of 2700 lines, to verify that this automata was well defined. Verification of this script took approximately 48 seconds. The definition of the same automata in SLEEK consisted of only 20 lines, and was verified in less than 2.3 seconds.

Chapter 7

Conclusions and Future Work

In this thesis we formally verified two of the three barrier implementations discussed in (Dodds et al., 2016). To do so, we designed two new constructions: *recursively nested wands* and *authoritative ordering*. These constructions allowed us to define a different invariant, which required less bookkeeping, ultimately leading to more straightforward proofs.

One possible venture for future work is a formal verification of the third barrier implementation discussed in (Dodds et al., 2016). This implementation uses a tree structure, instead of a linear chain. The tree has the invariant that if the signal flag of a node is set, all the flags of its children are also set. This means that `wait` has to check less flags, resulting in a more efficient implementation. We reckon that this implementation can be verified in a similar way to our verification presented in Chapter 5. To do so, one would need to generalize *authoritative ordering*. Currently, it models the ordering of a list, which can be seen as a path graph or linear graph. This could however be extended to arbitrary finite graphs, allowing one to model the $\prec$-predicate for the tree based implementation.

Another possibility for future work, is to add deallocation to the implementation and specification. Currently, it is not possible to deacllocate a barrier. To implement deallocation, one could use reference counting. However, we cannot describe the absence of memory leaks in the specification, as Iris is an *affine* separation logic (Jung et al., 2018, §9.5): using SEP-WEAKEN, one is able to throw away a $\text{send}(b, P)$ predicate, which means the barrier will never be deallocated. To completely eliminate the possibility of memory leaks, one could use Iron (Bizjak et al., 2019). This is a separation logic modelled on top of Iris, which allows one to reason about the absence of leaked resources.

Finally, we only verified partial correctness for the barriers, i.e. if the `wait` calls terminate, the appropriate resource is given to the client. This does mean that a client cannot use these specifications to get a total correctness proof of their program. However, there are many possible programs, where a client would like to use a barrier, but still have the guarantee that their program eventually terminates. It would thus be interesting to see if it is possible to write a specification which guarantees termination of `wait` under fair use, i.e. if someone owns $\text{send}(b, P)$, they eventually call $\texttt{signal}\, b$. However, Iris cannot be used to prove termination under fair scheduling. TaDA Live (D'Osualdo et al., 2021) is a separation logic which allows to reason about termination of blocking fine-grained concurrent programs. Our barrier is such a program, as `wait` is essentially a busy wait loop. However, TaDA Live is not higher order, which means we cannot write the specification using higher order predicates like $\text{send}(b, P)$ and $\text{recv}(b, P)$, with P being an arbitrary separation logic proposition.

Bibliography

Appel, A. W., & McAllester, D. A. (2001). An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, *23*(5), 657–683. https://doi.org/10.1145/504709.504712

Birkedal, L., Møgelberg, R. E., Schwinghammer, J., & Støvring, K. (2012). First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. *Log. Methods Comput. Sci.*, *8*(4). https://doi.org/10.2168/LMCS-8(4:1)2012

Bizjak, A., Gratzer, D., Krebbers, R., & Birkedal, L. (2019). Iron: Managing obligations in higher-order concurrent separation logic. *Proc. ACM Program. Lang.*, *3*(POPL), 65:1–65:30. https://doi.org/10.1145/3290378

Bornat, R., Calcagno, C., O'Hearn, P., & Parkinson, M. (2005). Permission accounting in separation logic. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 259–270. https://doi.org/10.1145/1040305.1040327

Boyland, J. (2003). Checking interference with fractional permissions. In R. Cousot (Ed.), *Static analysis* (pp. 55–72). Springer Berlin Heidelberg.

Brookes, S. (2007). A semantics for concurrent separation logic. *Theor. Comput. Sci.*, *375*(1-3), 227–270. https://doi.org/10.1016/j.tcs.2006.12.034

Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. J., & Vafeiadis, V. (2010). Concurrent abstract predicates. In T. D'Hondt (Ed.), *ECOOP 2010 - object-oriented programming, 24th european conference, maribor, slovenia, june 21-25, 2010. proceedings* (pp. 504–528, Vol. 6183). Springer. https://doi.org/10.1007/978-3-642-14107-2_24

Dodds, M., Jagannathan, S., & Parkinson, M. J. (2011). Modular reasoning for deterministic parallelism. In T. Ball & M. Sagiv (Eds.), *Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2011, austin, tx, usa, january 26-28, 2011* (pp. 259–270). ACM. https://doi.org/10.1145/1926385.1926416

Dodds, M., Jagannathan, S., Parkinson, M. J., Svendsen, K., & Birkedal, L. (2016). Verifying custom synchronization constructs using higher-order separation logic. *ACM Trans. Program. Lang. Syst.*, *38*(2). https://doi.org/10.1145/2818638

D'Osualdo, E., Sutherland, J., Farzan, A., & Gardner, P. (2021). Tada live: Compositional reasoning for termination of fine-grained concurrent programs. *ACM Trans. Program. Lang. Syst.*, *43*(4), 16:1–16:134. https://doi.org/10.1145/3477082

Flanagan, C., Sabry, A., Duba, B. F., & Felleisen, M. (1993). The essence of compiling with continuations. *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 237–247. https://doi.org/10.1145/155090.155113

Gherghina, C., David, C., Qin, S., & Chin, W. (2011). Structured specifications for better verification of heap-manipulating programs. In M. J. Butler & W. Schulte (Eds.), *FM 2011: Formal methods - 17th international symposium on formal methods, limerick, ireland, june 20-24, 2011. proceedings* (pp. 386–401, Vol. 6664). Springer. https://doi.org/10.1007/978-3-642-21437-0_29

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, *12*(10), 576–580. https://doi.org/10.1145/363235.363259

Hobor, A., & Gherghina, C. (2011). Barriers in concurrent separation logic. In G. Barthe (Ed.), *Programming languages and systems - 20th european symposium on programming, ESOP 2011, held as part of the joint european conferences on theory and practice of software, ETAPS 2011, saarbrücken, germany, march 26-april 3, 2011. proceedings* (pp. 276–296, Vol. 6602). Springer. https://doi.org/10.1007/978-3-642-19718-5_15

Hobor, A., & Gherghina, C. (2012). Barriers in concurrent separation logic: Now with tool support! *Log. Methods Comput. Sci.*, *8*(2). https://doi.org/10.2168/LMCS-8(2:2)2012

Hobor, A., & Villard, J. (2013). The ramifications of sharing in data structures. In R. Giacobazzi & R. Cousot (Eds.), *The 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '13, rome, italy - january 23 - 25, 2013* (pp. 523–536). ACM. https://doi.org/10.1145/2429069.2429131

Jung, R. (2020). *Understanding and evolving the rust programming language* [Doctoral dissertation, Saarland University, Saarbrücken, Germany]. https://doi.org/10.22028/D291-31946

Jung, R., Krebbers, R., Birkedal, L., & Dreyer, D. (2016). Higher-order ghost state. *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 256–269. https://doi.org/10.1145/2951913.2951943

Jung, R., Krebbers, R., Jourdan, J.-H., Bizjak, A., Birkedal, L., & Dreyer, D. (2018). Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, *28*, e20. https://doi.org/10.1017/S0956796818000151

Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., & Dreyer, D. (2015). Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In S. K. Rajamani & D. Walker (Eds.), *Proceedings of the 42nd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2015, mumbai, india, january 15-17, 2015* (pp. 637–650). ACM. https://doi.org/10.1145/2676726.2676980

Krebbers, R., Timany, A., & Birkedal, L. (2017). Interactive proofs in higher-order concurrent separation logic. In G. Castagna & A. D. Gordon (Eds.), *Proceedings of the 44th ACM SIGPLAN symposium on principles of programming languages, POPL 2017, paris, france, january 18-20, 2017* (pp. 205–217). ACM. https://doi.org/10.1145/3009837.3009855

Nguyen, H. H., & Chin, W. (2008). Enhancing program verification with lemmas. In A. Gupta & S. Malik (Eds.), *Computer aided verification, 20th international conference, CAV 2008, princeton, nj, usa, july 7-14, 2008, proceedings* (pp. 355–369, Vol. 5123). Springer. https://doi.org/10.1007/978-3-540-70545-1_34

O'Hearn, P. W. (2007). Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, *375*(1-3), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

O'Hearn, P. W., Reynolds, J. C., & Yang, H. (2001). Local reasoning about programs that alter data structures. *Proceedings of the 15th International Workshop on Computer Science Logic*, 1–19.

Reynolds, J. (2002). Separation logic: A logic for shared mutable data structures. *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Spies, S., Gäher, L., Tassarotti, J., Jung, R., Krebbers, R., Birkedal, L., & Dreyer, D. (2022). Later credits: Resourceful reasoning for the later modality. *Proc. ACM Program. Lang.*, *6*(ICFP). https://doi.org/10.1145/3547631

Svendsen, K., & Birkedal, L. (2014). Impredicative concurrent abstract predicates. In Z. Shao (Ed.), *Programming languages and systems - 23rd european symposium on programming, ESOP 2014, held as part of the european joint conferences on theory and practice of software, ETAPS 2014, grenoble, france, april 5-13, 2014, proceedings* (pp. 149–168, Vol. 8410). Springer. https://doi.org/10.1007/978-3-642-54833-8_9

The Iris Team. (2022). *The Iris 4.0 reference* [https://plv.mpi-sws.org/iris/appendix-4.0.pdf].

The Iris Team. (2023, February 16). *The iris coq development* (Version dev.2023-02-16.2.bcedf1c7). https://gitlab.mpi-sws.org/iris/iris/

Turon, A., Dreyer, D., & Birkedal, L. (2013). Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In G. Morrisett & T. Uustalu (Eds.), *ACM SIGPLAN international conference on functional programming, icfp'13, boston, ma, USA - september 25 - 27, 2013* (pp. 377–390). ACM. https://doi.org/10.1145/2500365.2500600

Turon, A., Thamsborg, J., Ahmed, A., Birkedal, L., & Dreyer, D. (2013). Logical relations for fine-grained concurrency. In R. Giacobazzi & R. Cousot (Eds.), *The 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '13, rome, italy - january 23 - 25, 2013* (pp. 343–356). ACM. https://doi.org/10.1145/2429069.2429111

van Collem, S. (2023). *Verifying a barrier using Iris*. Zenodo. https://doi.org/10.5281/zenodo.7749189

Wickerson, J., Dodds, M., & Parkinson, M. J. (2010). Explicit stabilisation for modular rely-guarantee reasoning. In A. D. Gordon (Ed.), *Programming languages and systems, 19th european symposium on programming, ESOP 2010, held as part of the joint european conferences on theory and practice of software, ETAPS 2010, paphos, cyprus, march 20-28, 2010. proceedings* (pp. 610–629, Vol. 6012). Springer. https://doi.org/10.1007/978-3-642-11957-6_32