Verified Lock-Free Session Channels with Linking

THOMAS SOMERS, Radboud University Nijmegen, The Netherlands
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands

Type systems and program logics based on session types provide powerful high-level reasoning principles for message-passing concurrency. Modern versions employ bidirectional session channels that (1) are asynchronous so that **send** operations do not block, (2) have buffers in both directions so that both parties can send messages in parallel, and (3) feature a **link** operation (also called **forward**) to concisely write programs in *process style*. These features complicate a low-level lock-free implementation of channels and therefore increase the gap between the meta theory of prior work—which is verified w.r.t. a high-level semantics of channels (*e.g.,* π-calculus)—and the code that runs on an actual computer.

We address this problem by verifying a low-level lock-free implementation of session channels w.r.t. a high-level specification based on session types. We carry out our verification in a layered manner by employing the Iris framework for concurrent separation logic. We start with an abstract specification of (unidirectional) queues—of which we provide a linked-list and array-segment based implementation—and gradually build up to session channels with all of the aforementioned features. To make a layered verification possible we develop two logical abstractions—*queues with ghost linking* and *pairing invariants*—to reason about the atomicity and changing endpoints due to linking, respectively. All our results are mechanized in the Coq proof assistant.

CCS Concepts: • **Theory of computation → Logic and verification**; **Separation logic**.

Additional Key Words and Phrases: Message passing, session types, separation logic, Iris, Coq

ACM Reference Format:
Thomas Somers and Robbert Krebbers. 2024. Verified Lock-Free Session Channels with Linking. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 292 (October 2024), 30 pages. https://doi.org/10.1145/3689732

1 Introduction

Message passing through channels is a pervasive method of communication in concurrent programming. Languages such as Go and Erlang have native support for message-passing, while many other languages provide libraries for it. To ensure the end-to-end correctness of message-passing programs, we desire powerful high-level reasoning principles for message-passing programs, and prove soundness of these reasoning principles w.r.t. low-level lock-free implementations of channels. There have been two important lines of work to obtain this goal:

- Type systems and program logics based on session types [18, 25, 26] provide high-level reasoning to establish safety and functional correctness of message-passing programs.
- Modern separation logics such as Iris [31–33, 35, 36] enable the verification of strong correctness conditions (*e.g.,* linearizability [20]) of implementations of concurrent data structures in a proof assistant, including queues which are essential to channel implementations.

Unfortunately, these two lines of work cannot readily be combined to obtain end-to-end correctness. Soundness of high-level reasoning principles is typically proved w.r.t. an abstract semantics of

Authors' Contact Information: Thomas Somers, Radboud University Nijmegen, The Netherlands, thomas.somers@ru.nl; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl.

ACM 2475-1421/2024/10-ART292
https://doi.org/10.1145/3689732

Implementation Layers		**Specification Layers**	
Bidirectional channels	(§ 3.4)	Actris's dependent separation protocols [21, 22]	(§ 2.2)
		Logically Atomic Triples (to hide data representation)	(§ 5.2)
Unidirectional queues	(§ 3.2)	Ghost linking (to linearize **link_queue** calls)	(§ 5.1)
		Logically Atomic Triples (to hide data representation)	(§ 4.4)
Atomic heap operations	(§ 3.1)	Hoare triples (from Iris)	

Fig. 1. Overview of layers used in the verification.

channels—often some form of process calculus such as the π-calculus [43]—which is far removed from the low-level lock-free implementation of channels that has been verified. To illustrate the differences, we consider the **link** operation (also called **forward**). This operation is present in many theoretical papers on session types [17, 39, 57, 60], and is also used to enable an elegant way of writing programs in *process style* as demonstrated by the session-typed languages Concurrent C0 [61], Rast [15] and CLASS [51]. This operator forwards messages between two channels, combining them into a single channel. Let us consider a simple use:

$$\begin{aligned} prog_1 \triangleq\ & \mathbf{let}\ (c : !\mathbb{Z}.?\mathbb{Z}.\mathbf{end}) = \mathbf{start}\ (\lambda(\overline{c} : ?\mathbb{Z}.!\mathbb{Z}.\mathbf{end}).\ \mathbf{let}\ n = \overline{c}.\mathbf{recv}\ ()\ \mathbf{in}\ \overline{c}.\mathbf{send}\ (n+1))\ \mathbf{in} \\ & \mathbf{let}\ (d : ?\mathbb{Z}.\mathbf{end}) = \mathbf{start}\ (\lambda(\overline{d} : !\mathbb{Z}..\mathbf{end}).\ c.\mathbf{send}\ 2; \overline{d}.\mathbf{link}\ c)\ \mathbf{in} \\ & d.\mathbf{recv}\ () \end{aligned}$$

This program creates two bidirectional channels using **start** f, which returns one channel endpoint and forks off a new thread that runs the function f on the other endpoint. Thread 1 (for $\overline{c}$) receives an integer and sends back its successor. Thread 2 (for $\overline{d}$) sends 2 over c, hence thread 1 receives 2 over $\overline{c}$. Thread 2 then links $\overline{d}$ to c, and hence the incremented value (3) is sent from the thread 1 (over $\overline{c}$) to the main thread (over d). The red annotations are session types, which are protocols consisting of sequences of send ($!T$) and receive ($?T$) actions. When creating a channel using **start** : $(T \multimap ()) \multimap \overline{T}$, both endpoints have *dual* protocols T and $\overline{T}$, in which sending **!** and receiving **?** actions are swapped. The **link** : $T \times \overline{T} \multimap ()$ operation is opposite and takes two channels with dual protocols. (By the Curry-Howard correspondence for session types [10, 60] **link** corresponds to the identity rule in linear logic.)

In π-calculus, where endpoints are represented by variables (bound by the name restriction operator ν), one can give a concise operational semantics of **link** as substitution (see *e.g.,* [60]): $\nu c.\ (c.\mathbf{link}\ d \mid\mid P) \rightarrow P[d/c]$. In a single step in the semantics, all uses of c in other threads (the process P can contain nested $\mid\mid$ operators) are replaced by d.

A low-level implementation of channels with a **link** operator is much more challenging. Channel endpoints are not represented by variables, but by pointers into a buffer on the heap. Consequently a lock-free linking operator cannot be implemented by global substitution, but needs to be implemented using the atomic heap operations that the implementation language provides. Modern versions of asynchronous session types furthermore use *bidirectional* channels with buffers in both directions, so that parties can send messages simultaneously provided this is done in the order the other party expects to receive them [44, 45]. As we explain momentarily, bidirectional channels further complicate a low-level implementation and the verification thereof.

Key Idea 1: Suitable abstraction layers. Channels can be implemented in many different ways, using linked lists, arrays, or combinations of those. We aim to make our work as independent from these implementation choices as possible. This is important for two reasons:

- To ensure feasibility, we want to obtain a clear separation of concerns. We want to separate low-level implementation aspects from other key aspects of the proof.

- To ensure reusability, we want to be able to support different channel implementations while reusing as much of our proofs as possible.

We achieve these goals by building up our development using multiple abstraction layers, as shown in Figure 1. Rather than implementing bidirectional channels with **link** directly, we implement them on top of two unidirectional channels with a **link_queue** operator. We provide two implementations of queues: a simple linked-list based version where each node contains a single value, and an array-segment based version that avoids allocations and improves cache locality.

Each implementation layer has two specifications, a basic specification that hides the internal data representation by linking it to a mathematical representation, and a flexible specification that provides further abstraction. Crucially, the verification of the each higher specification layer can be reused for any implementation matching the lower specification layer.

The flexible specification layer for unidirectional queues is used to prove that the **link** operator on bidirectional channels is linearizable (*i.e.,* it behaves as if it were an atomic operation). This is non-trivial, as **link** performs two calls to **link_queue** for the unidirectional queues in both directions. Our flexible specification layer employs an idea we call 'ghost linking' (detailed in 'Key Idea 2' below), allowing us to treat multiple **link_queue** calls as a single atomic operation.

The flexible specification layer for bidirectional channels abstracts over the exact state of the buffers and the other endpoint, hiding that another thread may link the other endpoint. This layer is based on *dependent separation protocols* from the Actris framework [21, 22] in Iris, which extend session types with separation logic propositions to prove functional correctness. A key ingredient to verify this layer is our notion of 'pairing invariants', which we detail in 'Key Idea 3' below.

Our Coq development shows that the use of abstraction layers pays off. Particularly, we can indeed swap one queue implementation for another and reuse the abstraction layers above. The implementation and proofs for linked-list based queues involves 350 LOC, the array-based version involves 680 LOC, while the rest of the verification effort involves 2,330 LOC.

Specifications in separation logic. To describe our key ideas, we first provide some background on (concurrent) separation logic [9, 47, 48, 50]. In separation logic one typically specifies programs using Hoare triples $\{P\}\ p\ \{Q\}$, which say that if the precondition P holds, then (1) the expression p executes safely (without memory errors), and (2) if p terminates, then the postcondition Q holds. In a concurrent setting Hoare triples are rather weak—they say nothing about atomicity of p, *i.e.,* whether p is linearizable [20]. For example, one could prove $\{\ell \mapsto n\}\ p\ \{\ell \mapsto (n+1)\}$ for $p \triangleq \ell \leftarrow (!\,\ell + 1)$, where $\leftarrow$ is the assignment and ! the dereference. Clearly, running p two times in parallel is not guaranteed to increment ℓ by two. Hence a Hoare triple $\{P\}\ p\ \{Q\}$ can only be used if a thread has unique ownership of P, not when P is shared between threads.

One can give stronger specifications in separation logic using *logically-atomic Hoare triples (LATs)* [13, 33]. A LAT $\langle P\rangle\ p\ \langle Q\rangle$ implies the Hoare triple $\{P\}\ p\ \{Q\}$, but additionally makes sure that p is atomic. Therefore, a LAT can be used even if P is shared between threads. The LAT for **link_queue** on unidirectional queues is (simplified):

$$\langle \mathbf{Q}\, d_1\, e_1\, \vec{v}_1 * \mathbf{Q}\, d_2\, e_2\, \vec{v}_2\rangle\ e_1.\mathtt{link\_queue}\ d_2\ \langle \mathbf{Q}\, d_1\, e_2\, (\vec{v}_1 \mathbin{++} \vec{v}_2)\rangle \qquad \text{(LINK-QUEUE-LAT)}$$

Here, $\mathbf{Q}\, d\, e\, \vec{v}$ is the queue representation predicate, which provides unique ownership of the internal data structures representing a queue with head pointer d (*i.e.,* dequeue handle), tail pointer e (*i.e.,* enqueue handle), and messages $\vec{v}$.

Key Idea 2: Ghost Linking. The LAT for **link_queue** is a-priori unusable for the verification of **link** on bidirectional asynchronous channels. It only says that linking of one pair of queues takes place atomically, while we need to link two pairs of queues (both channel directions) in a single atomic step. We provide a more flexible specification with a notion of *ghost linking* (simplified):

$$\mathbf{Q}_{👻}\, d_1\, e_1\, \vec{v}_1 * \mathbf{Q}_{👻}\, d_2\, e_2\, \vec{v}_2 \Rrightarrow e_1 \leftsquigarrow d_2 * \mathbf{Q}_{👻}\, d_1\, e_2\, (\vec{v}_1 \mathbin{++} \vec{v}_2) \qquad \text{(MAKE-GHOST-LINK)}$$

$$\{e_1 \leftsquigarrow d_2\}\; e_1.\mathtt{link\_queue}\; d_2\; \{\mathsf{True}\} \qquad \text{(LINK-QUEUE-GHOST)}$$

Rule MAKE-GHOST-LINK allows us to logically link two queues, giving the permission $e_1 \leftsquigarrow d_2$ to call **link_queue** later through the ordinary Hoare triple LINK-QUEUE-GHOST. Crucially, MAKE-GHOST-LINK is not tied to a physical program step, so we can perform it at any point during the verification. (For now, one can think of Iris's *update* $\Rrightarrow$ as being implication.) In particular, this allows us to logically link multiple queues at the same time, and postpone the actual calls to **link_queue**.

We show that given any queue implementation that satisfies LINK-QUEUE-LAT we can derive a specification with ghost linking. This is a non-trivial construction because the queue representation predicate $\mathbf{Q}_{👻}\, d\, e\, \vec{v}$ no longer provides ownership of a single queue. In between the time that queues have been logically linked, and the actual call to **link_queue** takes place, elements can be dequeued or enqueued via the handles not involved in the linking, and these handles themselves could be linked as well. Hence, $\mathbf{Q}_{👻}\, d\, e\, \vec{v}$ provides ownership of a chain of queues whose linking has been postponed, and $e_1 \leftsquigarrow d_2$ provides the permission to perform the call to **link_queue**.

Using our notion of ghost linking, we are able to prove the following LAT for **link** on bidirectional asynchronous channels (simplified):

$$\langle \mathbf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2 * \mathbf{C}\, c_3\, c_4\, \vec{v}_3\, \vec{v}_4 \rangle\; c_2.\mathtt{link}\; c_3\; \langle \mathbf{C}\, c_1\, c_4\, (\vec{v}_1 \mathbin{++} \vec{v}_3)\, (\vec{v}_4 \mathbin{++} \vec{v}_2) \rangle \qquad \text{(LINK-SPEC-LAT)}$$

Here, $\mathbf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2$ is the channel representation predicate, which provides unique ownership of a channel with endpoints c_1 and c_2 that has messages $\vec{v}_1$ and $\vec{v}_2$ in its buffers.

Dependent separation protocols. The specification LINK-SPEC-LAT says that linking is atomic—without exposing that internally two linking operations on queues are performed—but it is still low-level. The representation predicate $\mathbf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2$ involves *both* channel endpoints c_1 and c_2 and the messages $\vec{v}_1$ and $\vec{v}_2$ in both buffers. We want to verify a thread w.r.t. its own endpoint and without explicit reasoning about the buffers. It is particularly important that the verification does not concern the other endpoint, as another thread may link and thus change the other endpoint.

To enable thread-local verification akin to session types, we verify a high-level (and our final) specification for channels using *dependent separation protocols* from the Actris framework [21, 22] in Iris. For example, the dependent separation protocol for c in $prog_1$ on Page 2 is:

$$c \rightarrowtail \mathbf{!}\,(n : \mathbb{Z})\,\langle n \rangle.\, \mathbf{?}\,\langle n+1 \rangle.\, \mathbf{end}$$

Here, $c \rightarrowtail prot$ provides ownership of endpoint c and says that it behaves according to protocol *prot*. The protocol describes receiving any integer n, followed by sending the successor $n+1$. Similar to session types, after sending 2 in the second thread, we obtain $c \rightarrowtail \mathbf{?}\,\langle 3 \rangle.\,\mathbf{end}$. Since we also have $d \rightarrowtail \mathbf{!}\,\langle 3 \rangle.\,\mathbf{end}$, which is dual, we can use our novel high-level specification for **link**:

$$\{c \rightarrowtail prot * d \rightarrowtail \overline{prot}\}\; c.\mathtt{link}\; d\; \{\mathsf{True}\} \qquad \text{(LINK-SPEC)}$$

Note that the Hoare triple for **link** is similar to the typing rule in session types (and the identity rule in linear logic), so we have obtained our goal of obtaining high-level reasoning principles that are verified all the way to a low-level lock-free implementation of asynchronous channels.

Key Idea 3: Pairing Invariants. The final challenge is to define $c \rightarrowtail prot$ in terms of $\mathbf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2$, and to prove LINK-SPEC in terms of LINK-SPEC-LAT. Naively, we would use an Iris invariant per channel to connect $\mathbf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2$ to the protocols of c_1 and c_2. However, these endpoints c_1, c_2 can change over time due to calls to **link**, so a naive invariant per channel would not hold invariantly.

We address this challenge through our second key idea: *pairing invariants.* A usual invariant in separation logic (*e.g.,* Iris) says that a proposition I can be shared between threads provided all threads ensure I hold invariantly (*i.e.,* at all time). A pairing invariant takes a binary Iris relation $R : A \to A \to \mathrm{iProp}$. To allocate a pairing invariant, one trades ownership of $R\,a\,b$ for tokens $\mathsf{Tok}\;a$ and $\mathsf{Tok}\;b$ that provide shared access to $R\,a\,b$. To verify session channels we use (simplified):

$$R\,(c_1, \mathit{prot}_1)\;(c_2, \mathit{prot}_2) \triangleq \exists \vec{v}_1, \vec{v}_2.\, \mathsf{C}\;c_1\;c_2\;\vec{v}_1\;\vec{v}_2 * P\;\mathit{prot}_1\;\mathit{prot}_2\;\vec{v}_1\;\vec{v}_2$$
$$c \rightarrowtail \mathit{prot} \triangleq \mathsf{Tok}\;(c, \mathit{prot})$$

Here, P is a predicate (provided by Actris) that formalizes that the messages $\vec{v}_1$ and $\vec{v}_2$ in the channel buffers comply with the dependent separation protocols prot_1 and prot_2 of the endpoints. Pairing invariants provide a concise and abstract set of rules (which are independent of channels) that allow one to access $R\,a\,b$ through $\mathsf{Tok}\;a$ without knowledge of b. Particularly, when one owns $c_1 \rightarrowtail \mathit{prot}_1$, these rules provide access to $\mathsf{C}\;c_1\;c_2\;\vec{v}_1\;\vec{v}_2$ without knowledge of c_2.

Contributions and Outline. We provide the first low-level implementation of session channels that is lock-free, asynchronous with buffers in both directions, and supports programming in process style using `link`. We verify this implementation in layers (see Figure 1), where the top layer enables functional verification using a high-level session-type-based specification in the style of Actris [21, 22] (§ 2). This encompasses the following contributions:

- We parametrize our development by the implementation of a unidirectional queue, for which we provide a linked-list and array-segment instance (§ 3).
- On top of any unidirectional queue that enjoys a logically-atomic specification (§ 4), we provide a specification with *ghost linking* (§ 5), which makes it possible to linearize multiple `link` operations in one step. This allows us to verify a logically-atomic channel specification.
- We develop *pairing invariants* (§ 6), which provide a generic logical abstraction to reason locally without knowledge of the other component.
- We show that our results scale to the *higher-order* setting where channel endpoints and functions are used as messages (§ 7).
- We mechanize all our results in the Coq proof assistant (§ 8).

We conclude the paper with a discussion of related and future work (§ 9).

Limitations/non-goals. Similar to most papers on concurrent separation logic and Iris, we consider a sequentially consistent rather than relaxed memory model, and consider safety and functional correctness instead of deadlock-freedom or liveness. Similar to most papers on session types, we prohibit channel endpoints to be shared between threads, allowing us to use single-reader/single-writer queues. We restrict to binary instead of multiparty session types.

2 High-level Verification of Message-Passing Programs with Link

In this section we show the API of session channels and their high-level specification using Actris's dependent separation protocols. We introduce these concepts using simple examples (§ 2.1) before showing the proof rules (§ 2.2). Using an insertion sort example inspired by Concurrent C0 [61], we show that our proof rules can be used to not only prove safety (as in Concurrent C0), but also functional correctness of programs in *process style* that use `link` (§ 2.3). We conclude with a variant of insertion sort that uses the channel buffers in both directions (§ 2.4). Our high-level specification has all the existing proof rules of Actris—including a novel rule for `link`—and has been verified on top a low-level implementation of channels that we will define in § 3 (Actris uses a lock-based implementation and does not support `link`).

2.1 Basic Examples

Bidirectional asynchronous session channels provide the following operations:

$\texttt{new\_chan}\ ()$	Create a new channel returning the two endpoints.
$c.\texttt{send}\ v$	Send the message v over channel endpoint c.
$c.\texttt{recv}\ ()$	Wait for and receive the next message from channel endpoint c.
$c.\texttt{link}\ c'$	Link the endpoints c and c', forwarding all messages between them.

Using **new_chan** and $\texttt{fork}\ \{p\}$, which runs p in a new thread, we define the **start** operation, which runs a function on one endpoint of a channel in a new thread, and returns the other endpoint:

$$\texttt{start}\ f \triangleq \texttt{let}\ (c, \overline{c}) = \texttt{new\_chan}\ ()\ \texttt{in}\ \texttt{fork}\ \{f\ \overline{c}\}\ ;\ c$$

The example program $prog_2$ below uses **start** to spawn a process *service* that receives positive integers and sends back the sum once a non-positive value has been received:

$$\begin{aligned} \mathit{service}\ d\ \mathit{sum} \triangleq\ & \texttt{let}\ n = d.\texttt{recv}\ ()\ \texttt{in} \\ & \texttt{if}\ n > 0\ \texttt{then}\ \mathit{service}\ d\ (\mathit{sum} + n) \\ & \texttt{else}\ d.\texttt{send}\ \mathit{sum} \end{aligned} \qquad \begin{aligned} \mathit{prog}_2 \triangleq\ & \texttt{let}\ c = \texttt{start}\ (\lambda \overline{c}.\ \mathit{service}\ \overline{c}\ 0)\ \texttt{in} \\ & c.\texttt{send}\ 20;\ c.\texttt{send}\ 22;\ c.\texttt{send}\ 0; \\ & \texttt{assert}\ (c.\texttt{recv}\ () == 42) \end{aligned}$$

The $\texttt{assert}\ p$ operation succeeds when p evaluates to **true** and gets stuck otherwise. Verification in Iris ensures that no **assert** operation gets stuck.

Dependent Separation Protocols. To prove functional correctness of these programs—in this case that $prog_2$ does not get stuck—we need to reason about the messages sent over the channel. For this we use the *channel ownership assertion* $c \rightarrowtail \mathit{prot}$, which describes that a channel endpoint c sends and receives messages according to Actris's dependent separation protocol *prot*. For $prog_2$ the channel endpoints c and $\overline{c}$ can be described as channel ownership assertions with dual protocols:

$$c \rightarrowtail\ !\,\langle 20\rangle.\,!\,\langle 22\rangle.\,!\,\langle 0\rangle.\,?\,\langle 42\rangle.\,\mathbf{end} \qquad \overline{c} \rightarrowtail\ ?\,\langle 20\rangle.\,?\,\langle 22\rangle.\,?\,\langle 0\rangle.\,!\,\langle 42\rangle.\,\mathbf{end}$$

Here, $!\langle n\rangle$ and $?\langle n\rangle$ correspond to sending and receiving the value n, respectively. Protocols are *dual* iff all occurrences of **!** and **?** are swapped. The dual of a protocol *prot* is denoted $\overline{\mathit{prot}}$.

Channel Linking. The **link** operation combines two channels with dual protocols such that all messages between them are forwarded. It can be used to verify programs such as the following:

$$\begin{aligned} \mathit{prog}_3 \triangleq\ & \texttt{let}\ c = \texttt{start}\ (\lambda \overline{c}.\ \mathit{service}\ \overline{c}\ 0)\ \texttt{in} \\ & \texttt{let}\ d = \texttt{start}\ \big(\lambda \overline{d}.\ c.\texttt{send}\ 20;\ \overline{d}.\texttt{link}\ c\big)\ \texttt{in} \\ & d.\texttt{send}\ 22;\ d.\texttt{send}\ 0;\ \texttt{assert}\ (d.\texttt{recv}\ () == 42) \end{aligned}$$

In this program, a new process $\overline{d}$ first sends 20 over c, followed by linking with c. At the time of linking we have to verify that the protocols for $\overline{d}$ and c are duals:

$$\overline{d} \rightarrowtail\ ?\,\langle 22\rangle.\,?\,\langle 0\rangle.\,!\,\langle 42\rangle.\,\mathbf{end} \qquad c \rightarrowtail\ !\,\langle 22\rangle.\,!\,\langle 0\rangle.\,?\,\langle 42\rangle.\,\mathbf{end}$$

Quantified and Recursive Protocols. More general protocols for c and $\overline{c}$ abstract over the exact integers that are transferred (and thus the number of recursive calls):

$$\begin{aligned} c &\rightarrowtail \mathit{prot}\ 0 \quad \text{where} \quad \mathit{prot}\ s \triangleq\ !\,(n:\mathbb{Z})\,\langle n\rangle.\ \texttt{if}\ n > 0\ \texttt{then}\ \mathit{prot}\ (s+n)\ \texttt{else}\ ?\,\langle s\rangle.\,\mathbf{end} \\ \overline{c} &\rightarrowtail \overline{\mathit{prot}}\ 0 \quad \text{where} \quad \overline{\mathit{prot}}\ s \triangleq\ ?\,(n:\mathbb{Z})\,\langle n\rangle.\ \texttt{if}\ n > 0\ \texttt{then}\ \overline{\mathit{prot}}\ (s+n)\ \texttt{else}\ !\,\langle s\rangle.\,\mathbf{end} \end{aligned}$$

In quantified dependent separation protocols, both sent values and the remainder of the protocol can be arbitrary mathematical/Coq functions of the quantified variables. In this case, the protocol branches depending on the sent value by using an **if** expression in the remainder of the protocol. Note that the **if** in protocols is fundamentally different from the **if** in programs: the first lives at the level of higher-order logic/Coq, whereas the latter is part of the programming language.

Separation Logic Resources. In addition to sent values, protocols can also describe resources sent over the channel such as location ownership $l \mapsto v$ or channel ownership assertions $c \rightarrowtail prot$ (generalizing *delegation* in session types). This is used to verify programs such as the following, in which a location is received, the value at that location is incremented, and **true** is sent back:

$$prog_4\, c \triangleq \mathbf{let}\, l = c.\mathbf{recv}\ ()\ \mathbf{in}\, l \leftarrow (!\, l + 1);\ c.\mathbf{send}\,\mathbf{true}$$

In the protocol for c we receive not only the location ℓ, but also the location ownership $\ell \mapsto n$:

$$c \rightarrowtail \mathbf{?}\,(\ell : \mathsf{Loc})(n : \mathbb{Z})\ \langle \ell \rangle \{\ell \mapsto n\}.\, \mathbf{!}\,(b : \mathbb{B})\ \langle b \rangle \{\ell \mapsto (n+1)\}.\, \mathbf{end}$$

The $\{P\}$ notation states that resources matching the separation logic proposition $(P : \mathsf{iProp})$ are sent along with the message. When $\{P\}$ is omitted, it means $\{\mathsf{True}\}$, *i.e.,* no resources are sent along. In the example, the location ownership $\ell \mapsto n$ is sent along with the location ℓ. The location ownership $\ell \mapsto (n+1)$ is sent back along with the Boolean value b, meaning that after receiving, the other side can read the incremented value from the same location it sent earlier.

Neither the quantified variable n, nor the resources are physically sent over the channel, as they only exist during verification. As the resources are described by a separation logic proposition, they can also contain pure propositions such as $(\exists m : \mathbb{Z}.\, n = 2m)$, or combination of pure propositions and resources $(b \Leftrightarrow \exists m : \mathbb{Z}.\, n = 2m) * \ell \mapsto (n+1)$. The resources can also contain Hoare triples, which is useful to reason about programs in which functions/closures are sent over channels.

2.2 High-Level Specifications

In Iris, specifications are written as Hoare triples $\{P\}\, p\, \{\Phi\}$, where $(P : \mathsf{iProp})$ is a precondition, $(p : \mathsf{Expr})$ an expression, and $(\Phi : \mathsf{Val} \to \mathsf{iProp})$ the postcondition. The triple ensures that for all heaps in which P holds, executing p is safe (*i.e.,* does not get stuck), and if p evaluates to a value v, then $\Phi\ v$ holds for the heap after executing p. The Hoare triples of the channel operations are:

$$\{\mathsf{True}\}\ \mathbf{new\_chan}\ ()\ \left\{(c, \overline{c}).\, c \rightarrowtail prot * \overline{c} \rightarrowtail \overline{prot}\right\}$$
$$\{c \rightarrowtail \mathbf{!}\,\vec{x}\ \langle v \rangle \{P\}.\, prot * P[\vec{y}/\vec{x}]\}\ c.\mathbf{send}\ (v[\vec{y}/\vec{x}])\ \{c \rightarrowtail prot[\vec{y}/\vec{x}]\}$$
$$\{c \rightarrowtail \mathbf{?}\,\vec{x}\ \langle v \rangle \{P\}.\, prot\}\ c.\mathbf{recv}\ ()\ \{w.\, \exists \vec{y}.\, w = v[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}] * c \rightarrowtail prot[\vec{y}/\vec{x}]\}$$
$$\left\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\right\}\ c.\mathtt{link}\, c'\ \{\mathsf{True}\}$$

The **new_chan** specification says we can always create a channel with two endpoints that adhere to a protocol *prot* and its dual $\overline{prot}$. This protocol is chosen during the verification, and not part of the program, meaning that the implementation of channels is not dependent on the protocol.

The **send** specification states that to send a value w over c, we need to give up a sending channel ownership assertion $c \rightarrowtail \mathbf{!}\,\vec{x}\ \langle v \rangle \{P\}.\, prot$ and pick a suitable instantiation $\vec{y}$ of variables $\vec{x}$ such that $w = v[\vec{y}/\vec{x}]$ and we can give up resources $P[\vec{y}/\vec{x}]$. After sending we continue with the remaining protocol $c \rightarrowtail prot[\vec{y}/\vec{x}]$, which can also depend on the chosen instantiation $\vec{y}$.

The **recv** specification is similar to **send**. To receive from c we need a receiving channel ownership assertion $c \rightarrowtail \mathbf{?}\,\vec{x}\ \langle v \rangle \{P\}.\, prot$. After receiving there is an instantiation $\vec{y}$ such that the received value is $v[\vec{y}/\vec{x}]$ and we gain the resources $P[\vec{y}/\vec{x}]$ and the continuation of the protocol $c \rightarrowtail prot[\vec{y}/\vec{x}]$.

Finally, the **link** specification states linking is safe if we give up two channel ownership assertions $c \rightarrowtail prot$ and $c' \rightarrowtail \overline{prot}$ with dual protocols.

The specifications of **new_chan**, **send** and **recv** are the same as those for lock-based channels in Actris and can be used to verify the same programs, but against a lock-free implementation. The **link** rule is fully novel. The rule for **start** can be derived from those of **new_chan** and Iris's **fork**:

$$\frac{\forall \overline{c}.\ \left\{P_1 * \overline{c} \rightarrowtail \overline{prot}\right\}\ f\ \overline{c}\ \{\mathsf{True}\}}{\{P_1 * P_2\}\ \mathbf{start}\ \ f\ \{c.\, P_2 * c \rightarrowtail prot\}} \qquad \frac{\{P_1\}\ p\ \{\mathsf{True}\}}{\{P_1 * P_2\}\ \mathbf{fork}\ \{p\}\ \{P_2\}}$$

2.3 Insertion Sort

As a more challenging example that uses **link**, we consider insertion sort written in *process style*. In process languages such as Concurrent C0 [61], Rast [15] and CLASS [51], algorithms are modeled by concurrent processes, which provide a channel to the caller for communication. Such algorithms are written in our language as functions that use **start** to spawn a new thread and return the other endpoint to the client.

The insertion sort client can insert elements by sending a **some** value, and receives the elements of the list in sorted order after sending a **none** value. This is described by the following protocol:

$$\begin{aligned}&\mathit{prot}_{\mathit{recv}}, \mathit{prot}_{\mathit{sort}} : \mathsf{List}\ \mathbb{Z} \to \mathsf{iProto}\\ &\quad \mathit{prot}_{\mathit{recv}}\ \vec{x} \triangleq \mathbf{match}\ \vec{x}\ \mathbf{with}\ [\,] \Rightarrow ?\,\langle \mathbf{none}\rangle.\,\mathbf{end} \mid x :: \vec{x} \Rightarrow ?\,\langle \mathbf{some}\ x\rangle.\,\mathit{prot}_{\mathit{recv}}\ \vec{x}\ \mathbf{end}\\ &\quad \mathit{prot}_{\mathit{sort}}\ \vec{x} \triangleq !\,v\,\langle v\rangle.\,\mathbf{match}\ v\ \mathbf{with}\ \mathbf{none} \Rightarrow \mathit{prot}_{\mathit{recv}}\ \vec{x} \mid \mathbf{some}\ y \Rightarrow \mathit{prot}_{\mathit{sort}}\ (\mathit{insert}\ y\ \vec{x})\ \mathbf{end}\end{aligned}$$

The protocol $\mathit{prot}_{\mathit{sort}}\ \vec{x}$ is parameterized by a sorted list of integers $\vec{x}$ that are sent. If a **some** y value is sent, we use the mathematical function $\mathit{insert}\ y\ \vec{x}$ to insert y into a sorted list $\vec{x}$, and proceed recursively. If a **none** value is sent, we continue with the protocol $\mathit{prot}_{\mathit{recv}}\ \vec{x}$, which says that the list $\vec{x}$ is received in order, terminated by **none**. We implement insertion sort in process style, using a process for each element (the **send** operation in red will be removed in § 2.4):

```
elem x t ≜ start (λc. match c.recv () with
                     | none   ⇒ t.send none; c.send (some x); c.link t
                     | some y ⇒ if y ≤ x then let t′ = elem x t in c.link (elem y t′)
                                 else t.send (some y); c.link (elem x t)
                     end                                                          )
empty () ≜ start (λc. match c.recv () with
                     | none   ⇒ c.send none
                     | some x ⇒ let t = empty () in c.link (elem x t)
                     end                                            )
```

The *empty* process does not contain any values, and the *elem* x t process consists of a single value x and a tail process t. Both processes allow for inserting a new value by sending a **some** value, and receiving all sorted values in order by sending a **none** value. Recursion is done using the **link** operation. The **link** operation in the **none** case of *elem* (blue) allows the client to communicate directly with the tail process, rather than having to manually forward all messages.

A client uses this version of insertion sort by creating and communicating with a new *empty* process. It sends a list of **some** values terminated by **none**, and then receives a sorted list of **some** values, terminated by **none**. This mode of use is formalized by the specification of *empty*:

$$\{\mathsf{True}\}\ \mathit{empty}\ ()\ \{c.\ c \rightarrowtail \mathit{prot}_{\mathit{sort}}\ [\,]\}$$

Using this specification, we can verify that clients are safe and functionally correct, for example (each line is annotated with the current protocol):

```
client () ≜ let c = empty () in                                   {c ↣ prot_sort []}
            c.send (some 5); c.send (some 2); c.send (some 3);    {c ↣ prot_sort [2, 3, 5]}
            c.send none                                            {c ↣ prot_recv [2, 3, 5]}
            assert(c.recv () == some 2);                           {c ↣ prot_recv [3, 5]}
            assert(c.recv () == some 3);                           {c ↣ prot_recv [5]}
            assert(c.recv () == some 5);                           {c ↣ prot_recv []}
            assert(c.recv () == none)                              {c ↣ end}
```

To verify *empty* we need a suitable protocol for *elem*. Rather than giving a brand new protocol, we notice that $\mathit{prot}_{\mathit{sort}}\ (y :: \vec{x})$ is a protocol for $\mathit{elem}\ y\ t$ when $\mathit{prot}_{\mathit{sort}}\ \vec{x}$ is a protocol for t:

$$\{t \rightarrowtail \mathit{prot}_{\mathit{sort}}\ \vec{x}\}\ \mathit{elem}\ y\ t\ \{c.\ c \rightarrowtail \mathit{prot}_{\mathit{sort}}\ (y :: \vec{x})\}$$

2.4 Modified Insertion Sort

Channels with buffers in both directions allow clients to send messages earlier than required by the protocol. Inspired by *asynchronous subtyping* in session types [44, 45], dependent separation protocols in Actris provide *asynchronous subprotocols* [22, §6] to move sends ahead of receives. Formally, $c \rightarrowtail {?}\,\vec{x}\,\langle v\rangle\{P\}.\,{!}\,\vec{y}\,\langle w\rangle\{Q\}.\,\mathit{prot}$ can be converted into $c \rightarrowtail {!}\,\vec{y}\,\langle w\rangle\{Q\}.\,{?}\,\vec{x}\,\langle v\rangle\{P\}.\,\mathit{prot}$ as long as w and Q do not depend on $\vec{x}$.

To demonstrate asynchronous subprotocols, we modify the insertion sort algorithm to only retrieve and remove the least element when **none** is sent. This is similar to the original version in Concurrent C0, but our client goes beyond Concurrent C0, which does not support asynchronous subtyping. This change consists of removing the t.**send none** operation from the *elem* process in § 2.3. We update the protocols accordingly:

$$\mathit{prot}'_{\mathit{recv}}\ \vec{x} \triangleq \mathbf{match}\ \vec{x}\ \mathbf{with}\ [\,] \Rightarrow {?}\,\langle\mathbf{none}\rangle.\,\mathbf{end} \mid x :: \vec{x} \Rightarrow {?}\,\langle\mathbf{some}\ x\rangle.\,\mathit{prot}'_{\mathit{sort}}\ \vec{x}\ \mathbf{end}$$

$$\mathit{prot}'_{\mathit{sort}}\ \vec{x} \triangleq {!}\,v\,\langle v\rangle.\,\mathbf{match}\ v\ \mathbf{with}\ \mathbf{none} \Rightarrow \mathit{prot}'_{\mathit{recv}}\ \vec{x} \mid \mathbf{some}\ y \Rightarrow \mathit{prot}'_{\mathit{sort}}\ (\mathit{insert}\ y\ \vec{x})\ \mathbf{end}$$

The difference compared to $\mathit{prot}_{\mathit{sort}}$ from § 2.3 is that whereas $\mathit{prot}_{\mathit{recv}}$ retrieves and removes all elements by recursively using $\mathit{prot}_{\mathit{recv}}$, the protocol $\mathit{prot}'_{\mathit{recv}}$ continues with $\mathit{prot}'_{\mathit{sort}}$ after retrieving and removing the first element (marked in blue).

Unlike Concurrent C0—which does not use buffers in both directions, and channels must follow their protocol exactly—we can verify programs that send messages earlier than required. The following annotated client sends multiple **none** values before receiving the results:

$$\begin{array}{lr}
\mathit{client}'\ ()\triangleq \mathbf{let}\ c = \mathit{empty}\ ()\ \mathbf{in} & \{c \rightarrowtail \mathit{prot}'_{\mathit{sort}}\ [\,]\} \\
\quad c.\mathbf{send}\ (\mathbf{some}\ 5);\ c.\mathbf{send}\ (\mathbf{some}\ 2);\ c.\mathbf{send}\ (\mathbf{some}\ 3); & \{c \rightarrowtail \mathit{prot}'_{\mathit{sort}}\ [2,3,5]\} \\
\quad c.\mathbf{send}\ \mathbf{none}; & \{c \rightarrowtail \mathit{prot}'_{\mathit{recv}}\ [2,3,5]\}\quad (*) \\
\quad c.\mathbf{send}\ \mathbf{none}; & \{c \rightarrowtail {?}\,\langle\mathbf{some}\ 2\rangle.\,\mathit{prot}'_{\mathit{recv}}\ [3,5]\} \\
\quad \mathbf{assert}(c.\mathbf{recv}\ () == \mathbf{some}\ 2); & \{c \rightarrowtail \mathit{prot}'_{\mathit{recv}}\ [3,5]\} \\
\quad \mathbf{assert}(c.\mathbf{recv}\ () == \mathbf{some}\ 3); & \{c \rightarrowtail \mathit{prot}'_{\mathit{sort}}\ [5]\} \\
\quad c.\mathbf{send}\ \mathbf{none}; \mathbf{assert}(c.\mathbf{recv}\ () == \mathbf{some}\ 5); & \{c \rightarrowtail \mathit{prot}'_{\mathit{sort}}\ [\,]\} \\
\quad c.\mathbf{send}\ \mathbf{none}; \mathbf{assert}(c.\mathbf{recv}\ () == \mathbf{none}) & \{c \rightarrowtail \mathbf{end}\}
\end{array}$$

At $(*)$, we unfold the definition of $\mathit{prot}'_{\mathit{recv}}\ [2,3,5]$ as follows:

$${?}\,\langle\mathbf{some}\ 2\rangle.\,{!}\,v\,\langle v\rangle.\,\mathbf{match}\ v\ \mathbf{with}\ \mathbf{none} \Rightarrow \mathit{prot}'_{\mathit{recv}}\ \overset{\rightarrow}{[}3,5] \mid \mathbf{some}\ y \Rightarrow \mathit{prot}'_{\mathit{sort}}\ (\mathit{insert}\ y\ [3,5])\ \mathbf{end}$$

At which point we can use asynchronous subprotocols to get a channel with the protocol:

$${!}\,v\,\langle v\rangle.\,{?}\,\langle\mathbf{some}\ 2\rangle.\,\mathbf{match}\ v\ \mathbf{with}\ \mathbf{none} \Rightarrow \mathit{prot}'_{\mathit{recv}}\ [3,5] \mid \mathbf{some}\ y \Rightarrow \mathit{prot}'_{\mathit{sort}}\ (\mathit{insert}\ y\ [3,5])\ \mathbf{end}$$

We can therefore send another **none** value before retrieving the value 2.

3 Implementation

In this section we provide a low-level lock-free implementation of asynchronous session channels. We start by describing the implementation language (Iris's HeapLang) and its low-level operations (§ 3.1). We then implement lock-free queues with non-blocking **enqueue** and **link_queue** operations using linked lists (§ 3.2) and array segments (§ 3.3), which we pair to implement session channels (§ 3.4). Since Actris's session-type discipline ensures that endpoints are owned by a unique party, we consider single-reader/single-writer queues.

3.1 Description of HeapLang

We implement our channels in HeapLang—a low-level untyped functional language with concurrency, mutable state and garbage collection. HeapLang is the main low-level language included in the Coq version of Iris [28]. We use the following HeapLang operations:

alloc $n\,v$	Allocate n adjacent locations ℓ to $(\ell + n - 1)$ each containing value v and return ℓ.
ref v	Allocate and return a single location containing v. Short for **alloc** $1\,v$.
$\ell \leftarrow v$	Write the value v to location ℓ.
$!\,\ell$	Read and return the value at location ℓ.

3.2 Linked-List Implementation of Unidirectional Queues

We start by building an implementation of unidirectional queues with linking, whose API is:

new_queue $()$	Create a new queue and return a pair (d, e) of dequeue and enqueue handles.
d.**dequeue**$()$	Dequeue the next value from the queue with dequeue handle d.
e.**enqueue** v	Enqueue v to the queue with enqueue handle e.
e.**link_queue** d'	Combine the queues with enqueue handle e and dequeue handle d'.

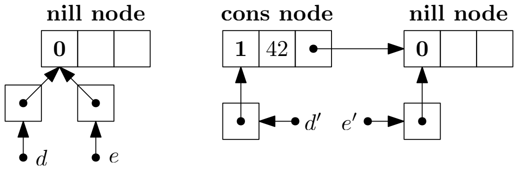
The **dequeue** operation is blocking, as it waits for the next value to dequeue when none are available. The **link_queue** operation is non-blocking and combines the queues (d, e) and (d', e') into a single queue (d, e'). Intuitively, the result of **link_queue** is that all messages from d' are forwarded to e without explicitly forwarding each message. By splitting up the enqueue and dequeue handles for queues, **link_queue** does not have to update the other ends of the queues d and e'.

To see our queues in action, consider the following example:

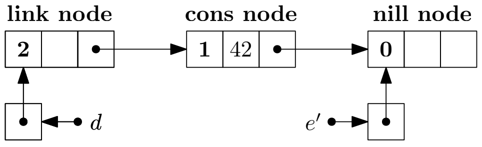
$$\begin{aligned} prog_5 \triangleq\ & \textbf{let}\ (d, e) = \textbf{new\_queue}\ ()\ \textbf{in} \\ & \textbf{fork}\ \{\textbf{let}\ (d', e') = \textbf{new\_queue}\ ()\ \textbf{in}\ e'.\textbf{enqueue}\ 42; e.\textbf{link\_queue}\ d'\}\,; \\ & d.\textbf{dequeue}() \end{aligned}$$

A new queue with dequeue handle d and enqueue handle e is created. The new thread creates a new queue (d', e'), enqueues 42 to the new queue and then links e to d'. The main thread waits for and dequeues the forwarded value 42 from d. This dequeued value 42 is returned from the program.

The queue is implemented using a singly-linked list with 3 types of nodes identified by an integer tag: `NIL` (0), `CONS` (1) and `LINK` (2). Each node consists of 3 adjacent locations containing the integer tag, contained value and next node location, respectively. Though both `CONS` and `LINK` nodes have a next node location, only the `CONS` nodes contain values. The enqueue and dequeue handles are locations containing pointers to the head and tail nodes respectively. The following diagram depicts the queues (d, e) and (d', e') in $prog_5$ before the **link_queue** operation:



The boxes represent locations on the heap, whereas arrows represent location values, pointing to a location on the heap. When linking e and d', the `NIL` node at e is replaced by a `LINK` node pointing to the head of the other queue at d', but containing no value. This results in the combined queue:



As the LINK node does not contain a value, the dequeue operation skips the LINK node and continues dequeueing from the next node. Hence, the next value dequeued from d in $prog_5$ would be 42.

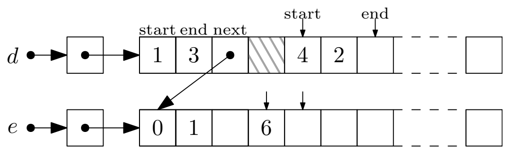
The queue operations are implemented as follows

$$\begin{aligned}
\textbf{new\_queue}\,() &\triangleq \textbf{let}\ nill = \textbf{alloc}\ 3\,\texttt{NIL}\ \textbf{in}\ (\textbf{ref}\ nill,\ \textbf{ref}\ nill)\\
d.\textbf{dequeue}() &\triangleq \textbf{let}\ node = !\,d\ \textbf{in}\\
&\quad\ \textbf{if}\ !node == \texttt{NIL}\ \textbf{then}\ d.\textbf{dequeue}()\\
&\quad\ \textbf{else if}\ !node == \texttt{CONS}\ \textbf{then}\ d \leftarrow !\,(node+2);\ !\,(node+1)\\
&\quad\ \textbf{else}\ d \leftarrow !\,(node+2);\ d.\textbf{dequeue}()\\
e.\textbf{enqueue}\ v &\triangleq \textbf{let}\ node = !\,e\ \textbf{in let}\ nill = \textbf{alloc}\ 3\,\texttt{NIL}\ \textbf{in}\\
&\quad\ (node+2) \leftarrow nill;\ (node+1) \leftarrow v;\ node \leftarrow \texttt{CONS};\ e \leftarrow nill\\
e.\textbf{link\_queue}\ d' &\triangleq \textbf{let}\ node = !\,e\ \textbf{in}\ (node+2) \leftarrow !\,d';\ node \leftarrow \texttt{LINK}
\end{aligned}$$

Thread safety depends on there being a single reader (that uses the dequeue handle) and single writer (that uses the enqueue handle), which will be enforced by the session-type discipline of Actris. For thread safety it is crucial that **enqueue** and **link_queue** update the tag last.

3.3 Array-Segment Implementation of Unidirectional Queues

In addition to the linked-list based queue, we also implement an array-segment based version with the same API. The queue consists of multiple array segments with a fixed length (*e.g.,* 1024). The first 3 fields are occupied by the start and end indices, and a pointer to the next segment. Dequeueing is performed on the first segment, whereas enqueueing is performed on the last segment. When enqueueing to a full array, a new array segment is allocated and the next field is set to the location of the new array. Similarly when linking, the next field is set to the location of the first array segment of the linked queue. As an example, the following diagram depicts a queue containing 4, 2 and 6 consisting of a segment containing 4 and 2, and a second segment containing 6:



The implementation and verification details of this queue can be found in our Coq development [3].

3.4 Implementation of Session Channels

We implement lock-free bidirectional channels as a pair of two single-reader/single-writer queues:

$$\begin{aligned}
\textbf{new\_chan}\,() &\triangleq \textbf{let}\ (d,e) = \textbf{new\_queue}\,()\ \textbf{in}\\
&\quad\ \textbf{let}\ (d',e') = \textbf{new\_queue}\,()\ \textbf{in}\\
&\quad\ ((d,e'),(d',e))\\
(d,e).\textbf{recv}\,() &\triangleq d.\textbf{dequeue}()\\
(d,e).\textbf{send}\ v &\triangleq e.\textbf{enqueue}\ v\\
(d,e).\textbf{link}\ (d',e') &\triangleq e.\textbf{link\_queue}\ d';\ e'.\textbf{link\_queue}\ d
\end{aligned}$$

Creating a channel consists of creating two queues, and creating endpoints containing one dequeue handle for receiving, and the other enqueue handle for sending. The **recv** operation corresponds to dequeueing, **send** corresponds to enqueueing, and **link** consists of linking both pairs of queues.

3.5 End-to-End Correctness

End-to-end correctness is captured by Iris's adequacy theorem [32, §6.4]: If $\{\mathsf{True}\}\ p\ \{\mathsf{True}\}$, then p does not get stuck w.r.t. the operational semantics of HeapLang. Programs get stuck if operators are applied to wrong arguments, there are memory errors, or **assert**s fail. Hence adequacy ensures safety and functional correctness of both the message-passing program subject to verification and the low-level implementation of channels.

The Coq mechanization (§ 8) contains closed proofs (*i.e.,* proofs without any axioms) of the adequacy theorem and the Hoare rules for the operations on session channel. The adequacy theorem is provided by Iris, and the Hoare rules for channels are a contribution of this paper. Using these ingredients we can derive closed proofs of safety for concrete programs with channels. Hence the trusted computing base consists of the statement of safety and the operational semantics of the HeapLang language, which are both standard as they are provided by Iris.

4 Specification and Verification of Queues

In this section we use logical atomic triples (LATs) in Iris to verify that the queues with linking from § 3 are linearizable—*i.e.,* all operations behave as if they were atomic. We first give a sequential specification (§ 4.1), and explain invariants and logical atomicity in Iris (§ 4.2 and § 4.3). We then present the concurrent specification (§ 4.4), and show how it is verified (§ 4.5). In the next section (§ 5), we extend the concurrent specification with our novel concept of *ghost linking*, allowing for the logical linking of multiple queues at a single linearization point.

4.1 Sequential Specification of Unidirectional Queues

A queue is commonly specified in separation logic as follows:

$$\{\mathsf{True}\}\ \mathbf{new\_queue}\ ()\ \{(d,e).\ \mathbf{Q}\,d\,e\,[\,]\}$$
$$\{\mathbf{Q}\,d\,e\,\vec{v}\}\ d.\mathbf{dequeue}()\ \{w.\ \exists \vec{w}.\ \vec{v} = w :: \vec{w} * \mathbf{Q}\,d\,e\,\vec{w}\}$$
$$\{\mathbf{Q}\,d\,e\,\vec{v}\}\ e.\mathbf{enqueue}\,v\ \{\mathbf{Q}\,d\,e\,(\vec{v} \mathbin{++} [v])\}$$
$$\{\mathbf{Q}\,d\,e\,\vec{v} * \mathbf{Q}\,d'\,e'\,\vec{w}\}\ e.\mathtt{link\_queue}\,d'\ \{\mathbf{Q}\,d\,e'\,(\vec{v} \mathbin{++} \vec{w})\}$$
$$\{\mathbf{Q}\,d\,e\,\vec{v}\}\ e.\mathtt{link\_queue}\,d\ \{\mathsf{True}\}$$

The queue representation predicate $\mathbf{Q}\,d\,e\,\vec{v}$ provides unique ownership of a queue with head pointer d (*i.e.,* dequeue handle), tail pointer e (*i.e.,* enqueue handle), and values $\vec{v}$ represented by a mathematical list. The postcondition $\mathbf{Q}\,d\,e\,[\,]$ of **new_queue** says that an empty queue is created for the returned dequeue and enqueue handles d and e. The specifications for **dequeue** and **enqueue** update the mathematical specification of the queue **Q** by dequeueing the head of the list and appending to the tail of the list respectively. The **dequeue** specification only requires that the queue state $\vec{v}$ is non-empty in the postcondition. This relies on partial correctness, as in the empty case the dequeue operation waits for the next value and hence does not terminate. The **link_queue** operation has two distinct cases. In the first case, the operation takes two distinct queues and links these queues into a single queue with the appended values from both queues. In the second case, the operation is provided both handles to the same queue. As no handles to the queue remain after linking, the postcondition is simply True. A definition of the **Q** representation predicate is given in § 4.5.

4.2 Invariants in Iris

Let us consider a small example to see why sequential specifications, such as those from § 4.1, are insufficient to verify concurrent programs:

$$prog_6 \triangleq \mathtt{let}\ (d,e) = \mathbf{new\_queue}\ ()\ \mathtt{in}\ \mathtt{fork}\ \{e.\mathbf{enqueue}\,42\}\ ;\ d.\mathbf{dequeue}()$$

The desired specification is $\{\mathsf{True}\}\ prog_6\ \{n.\, n = 42\}$. Verifying the **fork** requires splitting the predicate $\mathbf{Q}\, d\, e\, []$ into $P_1 * P_2$, where P_1 is used to prove $\{P_1\}\ e.\mathbf{enqueue}\ 42\ \{\mathsf{True}\}$ and P_2 to prove $\{P_2\}\ d.\mathbf{dequeue}()\ \{42\}$. But we only have one $\mathbf{Q}$, which may be used for P_1 or P_2, but not both.

To share resources such as the queue representation predicate $\mathbf{Q}$ between threads, Iris provides *invariants* $\boxed{P}$ corresponding to a proposition P that holds at all times (*i.e.,* invariantly). The invariant assertion $\boxed{P}$ is duplicable and can thus be shared between threads. Since an invariant needs to hold at all times during execution, it can only be opened and used for *physically-atomic operations*. Physically-atomic operations are primitive operations in the language, such as loading or storing to a location, and therefore take effect at a single instant in time. For the duration of a physically-atomic operation p, the contents P of an invariant $\boxed{P}$ can be used, as long as the invariant is closed after executing p by giving up P. These intuitive ideas are captured by the following rules:

$$\textsc{inv-alloc}\quad \frac{\{\boxed{P}^{\mathcal{N}} * S\}\ p\ \{\Phi\}_{\mathcal{E}}}{\{\triangleright P * S\}\ p\ \{\Phi\}_{\mathcal{E}}} \qquad \textsc{inv-dup}\quad \boxed{P}^{\mathcal{N}} \dashv\vdash \boxed{P}^{\mathcal{N}} * \boxed{P}^{\mathcal{N}} \qquad \textsc{inv-open}\quad \frac{\mathcal{N} \subseteq \mathcal{E} \qquad \mathsf{atomic}\, p \qquad \{\triangleright P * S\}\ p\ \{v.\ \triangleright P * \Phi\, v\}_{\mathcal{E}\setminus\mathcal{N}}}{\{\boxed{P}^{\mathcal{N}} * S\}\ p\ \{\Phi\}_{\mathcal{E}}}$$

Rule inv-alloc allows for proving a Hoare triple by making a part of the precondition (P) invariant, meaning that P holds invariantly during the execution of p. Rule inv-dup allows an invariant to be duplicated and thus to be shared among threads. Finally, inv-open allows unopened invariants to be opened for the duration of a physically atomic operation p.

For logical consistency, invariants in Iris $\boxed{P}^{\mathcal{N}}$ have a *namespace* $\mathcal{N}$, and Hoare triples $\{P\}\ p\ \{\Phi\}_{\mathcal{E}}$ have a *mask* $\mathcal{E}$. When an invariant is opened, its namespace is removed from the mask, ensuring that any invariant can only be opened once at any given time. Throughout this paper we generally omit namespaces and masks considering they are mostly an administrative detail.

The *later modality* ($\triangleright R$) corresponds to R holding after the next program step, and is required for logical consistency in the presence of higher-order reasoning [32, §5.5]. Until we consider higher-order concepts in § 7, the reader may ignore the later modality and read $\triangleright R$ as R.

Before describing how to reason about linearizability of functions (such as those for queues) in § 4.3, let us consider a simpler example to see invariants in action:

```
incr l ≜ let n = ! l in                    prog_7 ≜ let l = ref 40 in
         if CAS l n (n + 1) then ()                 fork {incr l} ; fork {incr l} ;
         else incr l                                ! l
```

The function **incr** l increments the value at location l. It uses **CAS** $l\ n\ m$, which is a *physically atomic* operation that compares the value at l with n, replacing it with m and returning **true** on success, and returning **false** on failure. The program $prog_7$ calls **incr** l twice in parallel and reads the value at location l. We prove that a value bigger than 40 is returned, *i.e.,* $\{\mathsf{True}\}\ prog_7\ \{n.\, n \geq 40\}$.

A sequential specification for **incr** would be $\{l \mapsto n\}\ \mathbf{incr}\ l\ \{l \mapsto (n+1)\}$. Similar to the sequential queue specifications, this specification is insufficient to verify $prog_7$—the heap ownership $l \mapsto 40$ can only be used in one of the threads. We instead use inv-alloc to turn $l \mapsto 40$ into the invariant $I \triangleq \boxed{\exists n \geq 40.\ l \mapsto n}$. Using inv-dup we share the invariant I between the threads. We use inv-open to prove $\{I\}\ \mathbf{incr}\ l\ \{\mathsf{True}\}$ and $\{I\}\ !\, l\ \{n.\, n \geq 40\}$, concluding the proof of $prog_7$.

4.3 Logically-Atomic Triples in Iris

We explain how *logically-atomic triples* (LATs) [13, 33] in Iris are used to give concise and abstract specifications of *linearizable* functions [20], *i.e.,* functions that appear to behave atomically. To motivate the need for LATs, let us consider the **incr** function, for which we proved $\{I\}\ !\, l\ \{n.\, n \geq 40\}$ in § 4.2. A similar specification holds for other invariants, *e.g.,* $I \triangleq \boxed{\exists n \geq 40, m \leq n.\ l \mapsto n * k \mapsto m}$, which also contains information about a location k. Note that the proof of **incr** for any invariant has the same structure. There is a single *physically atomic* step, namely the succesful **CAS**, called *the linearization point*, at which the current value $l \mapsto n$ is retrieved from the invariant (using INV-OPEN), updated to $l \mapsto (n+1)$ and then stored back into the invariant.

It would be undesirable to verify a function again and again for any invariant that shows up. Instead, we want a generic specification from which we can derive a specific specification for any invariant—this can be done using a LAT. In their simplest form, a LAT is of the following form:

$$\langle \vec{x}.\, P\rangle\; p\; \langle \vec{y}.\, Q\rangle$$

The *atomic precondition* P can refer to the variables $\vec{x}$, and the *atomic postcondition* Q can refer to $\vec{x}$ and $\vec{y}$. The LAT expresses that P holds for some (possibly changing) $\vec{x}$ at all points until the *linearization point*, at which the current P is transformed into Q for the same $\vec{x}$ and some $\vec{y}$ in a single atomic step. A LAT for **incr** is $\langle n.\, l \mapsto n\rangle\; \mathbf{incr}\, l\; \langle l \mapsto (n+1)\rangle$, as l contains some value n until the successful **CAS** operation, at which point l is updated to $n+1$. To understand how to both verify and use LATs, we unfold the definition (we omit the binder for the return value for now):

$$\langle \vec{x}.\, P\rangle\; p\; \langle \vec{y}.\, Q\rangle_{\mathcal{E}} \triangleq \forall R.\, \{\langle \vec{x}.\, P \mid \vec{y}.\, Q \Rrightarrow R\rangle_{\top\setminus\mathcal{E}}\}\; p\; \{R\}$$

A LAT is encoded as a Hoare triple, but with an *atomic update* $\langle x.\, P \mid y.\, Q \Rrightarrow R\rangle_{\mathcal{E}}$ in the precondition. The atomic update describes the update at the linearization point. When verifying a LAT, an atomic update can either (1) be used as the precondition of another LAT, or (2) be opened around a *physically atomic* step. The rule for the second mode of use is:

AU-OPEN

$$\frac{\mathcal{E}' \subseteq \mathcal{E} \qquad \mathsf{atomic}\, p \qquad \forall \vec{x}.\, \{P * S\}\; p\; \left\{v.\begin{array}{l}(\exists \vec{y}.\, Q * (R \mathrel{-\!\!*} \Phi\, v)) \\ \vee (P * (\langle \vec{x}.\, P \mid \vec{y}.\, Q \Rrightarrow R\rangle_{\mathcal{E}'} \mathrel{-\!\!*} \Phi\, v))\end{array}\right\}_{\mathcal{E}\setminus\mathcal{E}'}}{\{\langle \vec{x}.\, P \mid \vec{y}.\, Q \Rrightarrow R\rangle_{\mathcal{E}'} * S\}\; p\; \{\Phi\}_{\mathcal{E}}}$$

This rule is similar to INV-OPEN. It 'opens' the atomic update by adding the atomic precondition P to the precondition. The 'closing' is more complicated. One can either *commit* by establishing the atomic postcondition Q in the postcondition, or *defer* by reestablishing the atomic precondition P. In the first case, one gets R in return, while in the second case, one gets the atomic update back.

For **incr** we need to prove $\{\mathsf{AU}_{incr}\}\; \mathbf{incr}\, l\; \{R\}$ with $\mathsf{AU}_{incr} \triangleq \langle n.\, l \mapsto n \mid l \mapsto (n+1) \Rrightarrow R\rangle$ for any R. To access $l \mapsto n$ for the $!\, l$ operation, we apply AU-OPEN and *defer* by picking the second disjunct in the postcondition. This means we keep ownership of AU_{incr} to verify the remainder of the function. For **CAS**, we again use AU-OPEN and depending on its success *commit* or *defer*. Specifically, we use $\Phi\, v \triangleq \mathbf{if}\; v = \mathbf{true}\; \mathbf{then}\; R\; \mathbf{else}\; \mathsf{AU}_{incr}$. If the **CAS** succeeds (returns **true**), we *commit* by using the first disjunct. This means we give up the updated location $l \mapsto (n+1)$ and prove $\Phi\, \mathbf{true} = R$. If the **CAS** fails (returns **false**), we *defer* by using the second disjunct. This means we give up the unchanged $l \mapsto n$ and get back the atomic update AU_{incr}, allowing us to prove $\Phi\, \mathbf{false} = \mathsf{AU}_{incr}$. We then recursively apply the **incr** specification.

The simple version of LATs we have shown does not specify the return value, nor does it allow to specify resources that are not shared and changed before the linearization point. To support these, LATs can be extended with private pre- and postconditions:

$$\langle P_p \mid \vec{x}.\, P_a\rangle\; p\; \langle \vec{y}.\, Q_a \mid v.\, Q_p\rangle_{\mathcal{E}} \triangleq \forall \Phi.\, \{P_p * \langle \vec{x}.\, P_a \mid \vec{y}.\, Q_a \Rrightarrow \forall v.\, Q_p \mathrel{-\!\!*} \Phi\, v\rangle_{\top\setminus\mathcal{E}}\}\; p\; \{\Phi\}$$

The *private precondition* P_p cannot refer to any of the variables $\vec{x}$ and $\vec{y}$. The *private postcondition* Q_p can refer to $\vec{x}$ and $\vec{y}$ and the return value v. Proofs of these extended LATs proceed the same as their simpler form, but P_p can be used freely (subject to the same conditions as ordinary Hoare triples) throughout the verification of the whole expression P. Dually, the private postcondition Q_p only needs to be established at the very end of P.

The key feature of LATs—compared to ordinary Hoare triples—is that they allow one to open invariants without the physical-atomicity side-condition (atomic p):

LA-INV-OPEN
$$\frac{\mathcal{N} \subseteq \mathcal{E} \qquad \langle P_p \mid \vec{x}.\ \triangleright P * P_a \rangle\ p\ \langle \vec{y}.\ \triangleright P * Q_a \mid v.\ Q_p \rangle_{\mathcal{E}\setminus\mathcal{N}}}{\langle \vec{x}.\ \boxed{P}^{\mathcal{N}} * P_p \mid P_a \rangle\ p\ \langle \vec{y}.\ Q_a \mid v.\ Q_p \rangle_{\mathcal{E}}}$$

LA-ELIM
$$\frac{\langle P_p \mid \vec{x}.\ P_a \rangle\ p\ \langle \vec{y}.\ Q_a \mid v.\ Q_p \rangle_{\mathcal{E}}}{\left\{P_p * \exists \vec{x}.\ P_a\right\}\ p\ \left\{v.\ \exists \vec{x}, \vec{y}.\ Q_a * Q_p\right\}_{\mathcal{E}}}$$

The content P of the invariant appears only in the atomic precondition, not the private one—meaning that one gets access to P only at the linearization point, not throughout the whole expression p. The LA-ELIM rule allows ordinary Hoare triples to be proven using LATs by providing both the private and atomic preconditions. The witness of $\vec{x}$ in the postcondition may differ from the precondition, as it does not have to stay constant during the execution of p. These rules can be derived from the rules for atomic updates. In fact, in Coq one uses the rules for atomic updates directly rather than the derived LAT rules. However, Iris's rules for atomic updates are beyond the scope of this paper.

4.4 Logically-Atomic Specification of Unidirectional Queues

Using LATs, we verify the following specification for queues with linking:

$$\{\mathsf{True}\}\ \mathbf{new\_queue}\ ()\ \{(d, e).\ \mathbf{Q}\, d\, e\, [\,] * \mathbf{Deq}\, d * \mathbf{Enq}\, e\}$$
$$\langle \mathbf{Deq}\, d \mid e, \vec{v}.\ \mathbf{Q}\, d\, e\, \vec{v} \rangle\ d.\mathbf{dequeue}()\ \langle v, \vec{w}.\ \vec{v} = v :: \vec{w} * \mathbf{Q}\, d\, e\, \vec{w} \mid w.\ w = v * \mathbf{Deq}\, d \rangle$$
$$\langle \mathbf{Enq}\, e \mid d, \vec{v}.\ \mathbf{Q}\, d\, e\, \vec{v} \rangle\ e.\mathbf{enqueue}\, v\ \langle \mathbf{Q}\, d\, e\, (\vec{v} \mathbin{++} [v]) \mid \mathbf{Enq}\, e \rangle$$
$$\langle \mathbf{Enq}\, e * \mathbf{Deq}\, d' \mid b, d, \vec{v}, e', \vec{w}.\ \mathbf{Q}\, d\, e\, \vec{v} * (\mathbf{if}\ b\ \mathbf{then}\ \mathbf{Q}\, d'\, e'\, \vec{w}\ \mathbf{else}\ d = d' * e = e') \rangle$$
$$e.\mathbf{link\_queue}\, d'$$
$$\langle (\mathbf{if}\ b\ \mathbf{then}\ \mathbf{Q}\, d\, e'\, (\vec{v} \mathbin{++} \vec{w})\ \mathbf{else}\ \mathsf{True}) \mid \mathsf{True} \rangle$$

This specification encapsulates the data representation of the queue, and thus holds for both the linked-list based queue from § 3.2 and the array-segment based queue from § 3.3.

Creating a new queue results not only in the queue representation predicate (**Q**), but also an enqueue and a dequeue handle (**Enq**, **Deq**) representing the single writer and single reader, respectively. As the queue **Q** occurs only in the atomic pre- and postconditions, it can be placed in an invariant and shared between threads, whereas the handles **Enq** and **Deq** cannot. The mathematical state of the queue ($\vec{v}$) is quantified for the atomic preconditions, as it may change during the execution of the operation. For instance, during the execution of a **dequeue** operation, the state of the queue may be changed by an **enqueue** operation in a different thread.

To verify $prog_6$ from the start of § 4.2 we use the invariant $I \triangleq \boxed{\exists n \geq 0.\ \mathbf{Q}\, d\, e\, (\mathbf{replicate}\ n\ 42)}$. The queue starts empty ($\mathbf{replicate}\ 0\ 42 = [\,]$), the value 42 is enqueued ($\mathbf{replicate}\ 1\ 42 = [42]$) and then the value 42 is dequeued ($\mathbf{replicate}\ 0\ 42 = [\,]$).

Unlike the sequential specification, the cases for linking two separate queues or both ends of the same queue are combined in a single LAT and distinguished by the Boolean parameter b. This is necessary as the applicable case can change depending on other threads, such as in the following:

$$\begin{aligned} prog_8 \triangleq\ & \mathbf{let}\ (d, e) = \mathbf{new\_queue}\ ()\ \mathbf{in}\ \mathbf{let}\ (d', e') = \mathbf{new\_queue}\ ()\ \mathbf{in} \\ & \mathbf{fork}\ \{e.\mathbf{link\_queue}\, d'\}\ ;\ e'.\mathbf{link\_queue}\, d \end{aligned}$$

Here, the first executed **link_queue** operates on two distinct queues, whereas the second executed **link_queue** operates on both ends of the same queue. As the execution order of the two operations is non-deterministic, the LAT of **link_queue** has to support both possibilities.

4.5 Verification of the Logically-Atomic Specification of Unidirectional Queues

So far we left the representation predicate $\mathbf{Q}\, d\, e\, \vec{v}$ and handles $\mathbf{Deq}\, d$ and $\mathbf{Enq}\, e$ abstract. This is intentional—these predicates allow one to reason abstractly about the queue without knowledge of its implementation. Indeed, the layers in § 5 and § 6 rely neither on the queue implementation nor the definition of these predicates. We now give concrete definitions to verify the linked-list based version from § 3.2. The definitions for the array-segment based version from § 3.3 follow a similar but more complicated structure, which can be found in our Coq development [3].

The following auxiliary predicate describes that the queue is backed by a singly-linked list:

```
is_queue_list $\ell_h\ \ell_t\ \vec{v}_?$ ≜
    match $\vec{v}_?$ with
    | []            ⇒ $\ell_h = \ell_t$
    | some $v :: \vec{v}_?$ ⇒ $\exists \ell.\ \ell_h \mapsto_\square \texttt{CONS} * (\ell_h + 1) \mapsto_\square v * (\ell_h + 2) \mapsto_\square \ell * \texttt{is\_queue\_list}\ \ell\ \ell_t\ \vec{v}_?$
    | none $:: \vec{v}_?$   ⇒ $\exists \ell.\ \ell_h \mapsto_\square \texttt{LINK} * (\ell_h + 2) \mapsto_\square l * \texttt{is\_queue\_list}\ \ell\ \ell_t\ \vec{v}_?$
    end
```

The parameter $\vec{v}_?$ is a mathematical list of options where **none** accounts for LINK nodes. The parameters ℓ_h and ℓ_t correspond to the locations of the head and tail nodes and are equal for the empty queue. For CONS or LINK nodes, ℓ_h contains the tag, the message v (if applicable), and the location ℓ of the next node. Permission of the NIL node is in the **Q** predicate below.

The points-to connective $\ell \mapsto_\pi v$ is equipped with a permission $\pi \in \{\square\} \cup (0,1]_{\mathbb{Q}}$, where $\pi = \square$ is the *discarded permission* [58] and $\pi \in (0,1]_{\mathbb{Q}}$ a *fractional permission* [8]. Any permission π may be used to read from a location ℓ, but only the entire permission (fraction 1) can be used to write to ℓ. A discarded permission conceptually corresponds to an unknown fractional permission, *i.e.,* think of $\ell \mapsto_\square v$ as $\exists q \in (0,1]_{\mathbb{Q}}.\ \ell \mapsto_q v$, meaning ℓ is immutable and its value cannot change. The discarded permission is important in the verification of **dequeue**—where the tag, message, and location of the next node are read during multiple heap accesses—to ensure that their values remain unchanged throughout the **dequeue** operation.

The predicates **Q**, **Deq** and **Enq** are defined as:

$$\begin{aligned}\mathbf{Q}\, d\, e\, \vec{v} &\triangleq \exists \ell_h, \ell_t, \vec{v}_?, \gamma.\ \vec{v} = \texttt{filtero}\ \vec{v}_? * d \mapsto_{3/4} \ell_h * \mathbf{meta}\ e\ \gamma * \gamma \hookrightarrow_{1/2} \ell_t\ * \\ &\qquad \texttt{is\_queue\_list}\ \ell_h\ \ell_t\ \vec{v}_? * \ell_t \mapsto \texttt{NIL} \\ \mathbf{Deq}\, d &\triangleq \exists \ell_h.\ d \mapsto_{1/4} \ell_h \\ \mathbf{Enq}\, e &\triangleq \exists \gamma, \ell_t.\ \mathbf{meta}\ e\ \gamma * \gamma \hookrightarrow_{1/2} \ell_t * e \mapsto \ell_t * (\ell_t + 1) \mapsto - * (\ell_t + 2) \mapsto -\end{aligned}$$

The `is_queue_list` predicate uses a list of options whereas the queue **Q** describes a list of values. Hence, the queue contents $\vec{v}$ correspond to `filtero` $\vec{v}_?$, which filters out only the **some** values.

The **Q** predicate contains the permissions for each non-NIL node using `is_queue_list`, as well as the full permission $\ell_t \mapsto \texttt{NIL}$ of the tag of the tail node. The permissions of the other locations are distributed between **Q**, **Deq** and **Enq** so that one can dequeue and enqueue exactly when one has private ownership of **Deq** and **Enq**, respectively.

Ownership of the head pointer d is distributed between **Q** and **Deq**. Only if both **Q** and **Deq** are owned do the permissions $3/4$ and $1/4$ sum up to the entire permission 1 allowing the value of d to be updated. (We use different fractions $3/4$ and $1/4$ instead of equal halves $1/2$ to obtain the uniqueness property $\mathbf{Q}\, d\, e_1\, \vec{v}_1 * \mathbf{Q}\, d\, e_2\, \vec{v}_2 \vdash \text{False}$, which is relevant for the pairing invariant in § 6.2).

The distribution of permissions between **Q** and **Enq** is more complicated. The **enqueue** function performs a number of heap operations, but $\mathbf{Q}\, d\, e\, \vec{v}$ must be updated to $\mathbf{Q}\, d\, e\, (\vec{v} \mathbin{++} [v])$ at exacty the linearization point (*i.e.,* when applying the atomic update). This occurs at the store operation (*node* ← CONS) where the tag is set to CONS. There are also stores before the linearization point (to update the next pointer and value) and after (to update the tail pointer). To verify these stores, we put the entire permissions of $(\ell_t + 1)$ and $(\ell_t + 2)$ and e in the enqueue handle $\mathbf{Enq}$.

To ensure that the tail node locations l_t in $\mathbf{Q}$ and $\mathbf{Enq}$ are the same, we use two halves of a *ghost variable* $\gamma \hookrightarrow \ell_t$. A ghost variable is similar to the points-to connective, but is not associated with a physical heap locations, and as such can be updated at any point. We update the ghost variable γ at the linearization point, but only perform the physical update to e in the next step, allowing us to restore $\mathbf{Enq}\, e$ at the end of **enqueue**. To ensure that the existentially quantified ghost variables γ in $\mathbf{Q}$ and $\mathbf{Enq}$ are the same, we use Iris's *meta-tokens* [28, `iris/base_logic/lib/gen_heap.v`]. When allocating a new location ℓ, Iris allows one to allocate a (duplicable) meta-token $\mathbf{meta}\ \ell\ x$, which associates additional information x to the location ℓ. The rule $\mathbf{meta}\ \ell\ x_1 * \mathbf{meta}\ \ell\ x_2 \vdash x_1 = x_2$ allows us to conclude that the γs are the same.

5 Ghost Linking of Queues

The logically-atomic specification of queues in § 4.4 is a-priori unusable to prove the logically-atomic specification of channels implemented as pairs of queues as done in § 3.4. The **link** operation consists of two sequential **link_queue** operations and therefore does not have a single linearization point. To solve this, we introduce a ghost linking mechanism, which separates the logical linking of queues from the physical linking of queues. This approach allows multiple queues to be linked logically in a single atomic step even though the physical linking of queues is done in multiple steps. We describe the altered specification of queues using ghost linking (§ 5.1), show how this updated specification is used to derive a logically-atomic specification for channels (§ 5.2), and finally verify the specification of queues with ghost linking itself (§ 5.3). Ghost linking and higher layers depend only on the logically atomic specification, rather than implementation of queues. Hence the queue implementation can be swapped, for instance from the linked-list version (§ 3.2) to array-segment version (§ 3.3), whilst reusing ghost linking and the higher layers.

5.1 Specification of Queues with Ghost Linking

Our specification with ghost linking uses two new predicates: the *ghost queue representation predicate* $\mathbf{Q}_{👻}\, d\, e\, \vec{v}$ and the *link permission* $e \leftsquigarrow d$. The ghost queue representation predicate $\mathbf{Q}_{👻}\, d\, e\, \vec{v}$ is similar to $\mathbf{Q}\, d\, e\, \vec{v}$, but represents a chain of queues that remain to be linked, where d is the dequeue handle of the first queue, e is the enqueue handle of the last queue, and $\vec{v}$ is the concatenation of the messages in the queues. The link permission $e \leftsquigarrow d$ allows one to physically link a queue with enqueue handle e to a queue with dequeue handle d. The rules are as follows (some details are elided and will be discussed in § 5.3):

$$\{\mathsf{True}\}\ \mathbf{new\_queue}\ ()\ \big\{(d, e).\ \mathbf{Q}_{👻}\, d\, e\, []\ *\ \mathbf{Deq}\, d\ *\ \mathbf{Enq}\, e\big\}$$

$$\big\langle \mathbf{Deq}\, d \mid e, \vec{v}.\ \mathbf{Q}_{👻}\, d\, e\, \vec{v}\big\rangle\ d.\mathbf{dequeue}()\ \big\langle v, \vec{w}.\ \vec{v} = v :: \vec{w} * \mathbf{Q}_{👻}\, d\, e\, \vec{w} \mid w.\ w = v * \mathbf{Deq}\, d\big\rangle$$

$$\big\langle \mathbf{Enq}\, e \mid d, \vec{v}.\ \mathbf{Q}_{👻}\, d\, e\, \vec{v}\big\rangle\ e.\mathbf{enqueue}\, v\ \big\langle \mathbf{Q}_{👻}\, d\, e\, (\vec{v} +\!\!+ [v]) \mid \mathbf{Enq}\, e\big\rangle$$

$$\mathbf{Q}_{👻}\, d_1\, e_1\, \vec{v}_1 * \mathbf{Q}_{👻}\, d_2\, e_2\, \vec{v}_2 \Rrightarrow\!\!\ast\ e_1 \leftsquigarrow d_2 * \mathbf{Q}_{👻}\, d_1\, e_2\, (\vec{v}_1 +\!\!+ \vec{v}_2) \qquad \text{(MAKE-GHOST-LINK)}$$

$$\mathbf{Q}_{👻}\, d\, e\, \vec{v} \Rrightarrow\!\!\ast\ e \leftsquigarrow d \qquad \text{(MAKE-GHOST-LINK-SELF)}$$

$$\{\mathbf{Enq}\, e * \mathbf{Deq}\, d * e \leftsquigarrow d\}\ e.\mathbf{link\_queue}\, d\ \{\mathsf{True}\}$$

The specifications of **new_queue**, **dequeue** and **enqueue** are the same as those in § 4.4 but use the ghost queue representation predicate $\mathbf{Q}_{👻}\, d\, e\, \vec{v}$ instead of $\mathbf{Q}\, d\, e\, \vec{v}$. Unlike § 4.4, linking is performed in two steps. First, two ghost queues are combined into a single ghost queue, providing the permission $e \leftsquigarrow d$. Second, $e \leftsquigarrow d$ is used to verify the physical **link_queue** operation.

The rule MAKE-GHOST-LINK allows two different ghost queues to be combined into a single ghost queue, producing a link permission $e_1 \leftrightsquigarrow d_2$. Similarly, MAKE-GHOST-LINK-SELF allows both ends of the same ghost queue to be linked, also producing a link permission $e \leftrightsquigarrow d$, but no new ghost queue. These rules are similar to the $b = \texttt{true}$ and $b = \texttt{false}$ cases in the specification of **`link_queue`** without ghost linking in § 4.4. Unlike LATs, the rules MAKE-GHOST-LINK and MAKE-GHOST-LINK-SELF are not tied to program steps—they use Iris's *update* $\Rrightarrow\!\!*$. During the same program step (*e.g.*, a linearization point) multiple such updates can be performed, and hence multiple ghost queues can be logically linked. The obtained link permissions can then be used in program steps after the linearization point to physically link the queues using the Hoare triple for **`link_queue`**.

In the verification of queues (§ 4.5) we already used ghost state (specifically, ghost variables). Unlike location ownership, which represents values in the heap, ghost state is only used during verification and not updated by physical operations. Ghost state is updated using Iris's update $P \Rrightarrow\!\!* Q$, defined as $P \mathrel{-\!\!*} {\Rrightarrow} Q$, where the *update modality* ${\Rrightarrow} Q$ means that after (possibly) some ghost state updates we have the resources Q. Relevant rules are:

UPD-INTRO
$$P \vdash {\Rrightarrow} P$$

UPD-IDEMP
$$ {\Rrightarrow}{\Rrightarrow} P \vdash {\Rrightarrow} P$$

UPD-SEP
$$({\Rrightarrow} P) * ({\Rrightarrow} Q) \vdash {\Rrightarrow}(P * Q)$$

UPD-EXEC
$$\frac{\{P * S\}\ p\ \{\Phi\}}{\{({\Rrightarrow} P) * S\}\ p\ \{\Phi\}}$$

The first 3 rules state that performing no updates is a valid update, and that updates can be combined. By applying UPD-EXEC multiple times, multiple ghost updates such as MAKE-GHOST-LINK or MAKE-GHOST-LINK-SELF can be performed during a single program step, such as a linearization point. Such updates can similarly be eliminated in atomic updates and hence LATs using the following rule (omitting binders), which derives one atomic update from another:

AUPD-AUPD
$$P_1 \Rrightarrow\!\!* P_2 * ((P_2 \Rrightarrow\!\!* P_1) \land (Q_2 \Rrightarrow\!\!* Q_1 * (R_1 \Rrightarrow\!\!* R_2))) \vdash \langle P_1 \mid Q_1 \Rrightarrow R_1 \rangle \mathrel{-\!\!*} \langle P_2 \mid Q_2 \Rrightarrow R_2 \rangle$$

This rule states that an atomic update can be converted into another atomic update if the precondition, postconditions for both defer and commit, and the result can be updated. As LATs are defined using atomic updates, the AUPD-AUPD rule can also be applied to prove a LAT using another LAT.

5.2 Logically-Atomic Specification of Channels

We utilize the ghost queue specification to prove a logically-atomic specification for the implementation of channels as pairs of queues from § 3.4. We first define the representation predicates:

$$\mathbf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2 \triangleq \exists d_1, e_1, d_2, e_2.\, c_1 = (d_1, e_1) * c_2 = (d_2, e_1) * \mathbf{Q}_{\text{👻}}\, d_1\, e_1\, \vec{v}_1 * \mathbf{Q}_{\text{👻}}\, d_2\, e_2\, \vec{v}_2$$
$$\mathbf{Ch}\, c \triangleq \exists d, e.\, c = (d, e) * \mathbf{Deq}\, d * \mathbf{Enq}\, e$$

We derive the logically-atomic specifications for **`new_chan`**, **`send`** and **`recv`** directly from the ghost queue specifications:

$$\{\mathsf{True}\}\ \texttt{new\_chan}\ ()\ \{(c_2, c_2).\, \mathbf{C}\, c_1\, c_2\, []\, []\, * \mathbf{Ch}\, c_1 * \mathbf{Ch}\, c_2\}$$
$$\langle \mathbf{Ch}\, c_1 \mid c_2, \vec{v}_1, \vec{v}_2.\, \mathbf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2 \rangle\ c_1.\texttt{send}\ v\ \langle \mathbf{C}\, c_1\, c_2\, \vec{v}_1\, (\vec{v}_2 \mathbin{++} [v]) \mid \mathbf{Ch}\, c_1 \rangle$$
$$\langle \mathbf{Ch}\, c_1 \mid c_2, \vec{v}_1, \vec{v}_2.\, \mathbf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2 \rangle\ c_1.\texttt{recv}\ ()\ \langle v, \vec{v}_1'.\, \vec{v}_1 = v :: \vec{v}_1' * \mathbf{C}\, c_1\, c_2\, \vec{v}_1'\, \vec{v}_2 \mid w.\, w = v * \mathbf{Ch}\, c_1 \rangle$$
$$\mathbf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2 \dashv\vdash \mathbf{C}\, c_2\, c_1\, \vec{v}_2\, \vec{v}_1 \qquad \text{(CHAN-SYMMETRIC)}$$

Most rules are written from the point of view of the left endpoint, CHAN-SYMMETRIC makes it possible to send/receive using the right endpoint. To verify **`new_chan`** we use the specification of **`new_queue`** twice, resulting in two ghost queue representation predicates, and two dequeue and two enqueue handles. These are converted into the channel representation predicate **C** and a pair of channel handles **Ch**. The specifications of **`send`** and **`recv`** are derived from the specifications with ghost linking for **enqueue** and **dequeue** by retrieving the relevant queue handle from the channel handle from **Ch**, and the relevant ghost queue from the channel representation predicate **C**.

Finally, we derive a logically-atomic specification for **link**, which links two channels in a single logically-atomic step:

$$\begin{array}{l}\langle \mathbf{Ch}\, c_2 * \mathbf{Ch}\, c_3 \mid b, c_1, c_4, \vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{v}_4.\ \mathbf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2 * (\mathbf{if}\ b\ \mathbf{then}\ \mathbf{C}\, c_3\, c_4\, \vec{v}_3\, \vec{v}_4\ \mathbf{else}\ c_1 = c_3)\rangle \\ \quad c_2.\mathtt{link}\, c_3 \\ \langle \mathbf{if}\ b\ \mathbf{then}\ \mathbf{C}\, c_1\, c_4\, (\vec{v}_1 \mathbin{++} \vec{v}_3)\, (\vec{v}_4 \mathbin{++} \vec{v}_2)\ \mathbf{else}\ \mathsf{True} \mid \mathsf{True}\rangle\end{array}$$

The variable b in this specification is similar to that for **link_queue**, as the possibility of linking two channels or a single channel can change depending on other threads.

The proof of the LAT for **link** is more complex than the other LATs, as the ghost and physical linking are performed in different steps. The ghost linking of channels (using either MAKE-GHOST-LINK if $b = \mathbf{true}$ or MAKE-GHOST-LINK-SELF if $b = \mathbf{false}$) can be done during any program step before or including the first use of **link_queue**. This results in link permissions that are then used to verify the physical calls to the **link_queue** operations.

5.3 Verification of Queues with Ghost Linking

To verify the specification of queues with ghost linking we define the ghost queue representation predicate $\mathbf{Q}_{👻}\, d\, e\, \vec{v}$ and the link permission $e \leftrightsquigarrow d$. We then derive the rules presented in § 5.1 from those in § 4.4. The definitions are carried out in two layers. The first layer is the representation of queue resources. As these resources are shared between ghost queues and link permissions, they are placed in an invariant. To support linking of different ghost queues, we use a single invariant for all ghost queues, rather than a separate invariant for each ghost queue. The second layer defines the representation of ghost queues and link permissions to access and update specific queues in the invariant. These permissions are modeled by Iris ghost state.

Layer #1: Representation of Queue Resources. Fundamentally, ghost linking represents a ghost queue not as single queue, but as a chain of queues, where each adjacent pair of queues is to be linked. This allows different ghost queues to be linked logically by merging two chains (MAKE-GHOST-LINK). When linking a ghost queue to itself (MAKE-GHOST-LINK-SELF), the first and last queue are logically linked, resulting in a cycle of queues. As the actual queues are physically linked in later steps, the cycle cannot be discarded at the point of logical linking. These chains and cycles are represented by the following predicates:

$$\begin{array}{rl}\mathtt{lchain}\, \vec{L}\, d\, e\, \vec{v} \triangleq & \mathbf{match}\ \vec{L}\ \mathbf{with} \\ & \mid [\,] \qquad\quad\ \Rightarrow \mathbf{Q}\, d\, e\, \vec{v} \\ & \mid (e', d') :: \vec{L}' \Rightarrow \exists\, \vec{v}_1, \vec{v}_2.\ \vec{v} = \vec{v}_1 \mathbin{++} \vec{v}_2 * \mathbf{Q}\, d\, e'\, \vec{v}_1 * \mathtt{lchain}\, \vec{L}'\, d'\, e\, \vec{v}_2 \\ & \mathbf{end} \\ \mathtt{lcycle}\, \vec{L} \triangleq & \mathbf{match}\ \vec{L}\ \mathbf{with} \\ & \mid [\,] \qquad\quad \Rightarrow \mathsf{True} \\ & \mid (e, d) :: \vec{L} \Rightarrow \exists\, \vec{v}.\ \mathtt{lchain}\, \vec{L}\, d\, e\, \vec{v} \\ & \mathbf{end}\end{array}$$

In $\mathtt{lchain}\, \vec{L}\, d\, e\, \vec{v}$, the parameters d and e are the dequeue and enqueue handles of the first and last queue, and $\vec{v}$ the concatenation of messages in queues. In both $\mathtt{lchain}\, \vec{L}\, d\, e\, \vec{v}$ and $\mathtt{lcycle}\, \vec{L}$, the list $\vec{L}$ contains all links between the queues. Note that two separate resources $\mathtt{lchain}\, \vec{L}_1\, d_1\, e_1\, \vec{v}_1$ and $\mathtt{lchain}\, \vec{L}_2\, d_2\, e_2\, \vec{v}_2$ can be combined into $\mathtt{lchain}\, (\vec{L}_1 \mathbin{++} (e_1, d_2) :: \vec{L}_2)\, d_1\, e_2\, (\vec{v}_1 \mathbin{++} \vec{v}_2)$, meaning that two different chains can be linked logically without a program step. Similarly, when linking a ghost queue to itself (MAKE-GHOST-LINK-SELF), the resource $\mathtt{lchain}\,\vec{L}\,d\,e\,\vec{v}$ can be turned into a cycle resource $\mathtt{lcycle}\,((e,d) :: \vec{L})$ without a program step.

Layer #2: Ghost Queues and Link Permissions. A ghost queue $\mathbf{Q}_{\text{👻}}\,d\,e\,\vec{v}$ corresponds to a chain $\mathtt{lchain}\,\vec{L}\,d\,e\,\vec{v}$ in the invariant, and each link permission $e' \leftrightsquigarrow d'$ corresponds to a link $(e',d') \in \vec{L}$ in either a chain $\mathtt{lchain}\,\vec{L}\,d\,e\,\vec{v}$ or cycle $\mathtt{lcycle}\,\vec{L}$ in the invariant. This is formalized as:

$$\begin{aligned}
\mathbf{Qctx}_{\text{👻}\,\gamma} &\triangleq \boxed{\begin{array}{l}\exists \mathcal{L}, m.\ \mathbf{map\_auth}_{\gamma_{chains}}\ \{d \mapsto (e,\vec{v}) \mid d \mapsto (e,\vec{v},\vec{L}) \in m\} \ast \\ \qquad\quad \mathbf{set\_auth}_{\gamma_{links}}\left(\biguplus \{\vec{L} \mid \vec{L} \in \mathcal{L} \vee d \mapsto (e,\vec{v},\vec{L}) \in m\}\right) \ast \\ \qquad\quad \left(\mathop{\ast}_{d\mapsto(e,\vec{v},\vec{L})\in m}.\ \mathtt{lchain}\,d\,e\,\vec{v}\,\vec{L}\right) \ast \left(\mathop{\ast}_{\vec{L}\in\mathcal{L}}.\ \mathtt{lcycle}\,\vec{L}\right)\end{array}} \\
\mathbf{Q}_{\text{👻}\,\gamma}\,d\,e\,\vec{v} &\triangleq \mathbf{map\_own}_{\gamma_{chains}}(d \mapsto (e,\vec{v})) \\
e \leftrightsquigarrow_{\gamma} d &\triangleq \mathbf{set\_own}_{\gamma_{links}}\,(e,d)
\end{aligned}$$

Let us describe all components of these definitions. The quantified variable $\mathcal{L}$ describes the list of links for each cycle, and m maps each dequeue handle d to a tuple $(e,\vec{v},\vec{L})$ of enqueue handle e, messages $\vec{v}$ and list of links $\vec{L}$. The parameter γ is a pair of ghost names γ_{chains} and γ_{links}, which are used to link the ghost state in the invariant and other predicates.

The connection between each ghost queue $\mathbf{Q}_{\text{👻}}\,d\,e\,\vec{v}$ and the chain $\mathtt{lchain}\,\vec{L}\,d\,e\,\vec{v}$ in the invariant is represented by a ghost map, consisting of a single authoritative tokens $\mathbf{map\_auth}_{\gamma}\,m$ and multiple fragment tokens $\mathbf{map\_own}_{\gamma}(k \mapsto x)$. The fragment token $\mathbf{map\_own}_{\gamma}(k \mapsto x)$ states that the key k is mapped to x in the map m governed by the authoritative token $\mathbf{map\_auth}_{\gamma}\,m$. Only a single fragment token exists for each key k in the map m, so $\mathbf{map\_own}_{\gamma}(k \mapsto x)$ gives the unique permission to update and delete k in m. Each ghost queue $\mathbf{Q}_{\text{👻}\,\gamma}\,d\,e\,\vec{v}$ thus corresponds to a unique entry in m.

The link permission $e \leftrightsquigarrow d$ corresponds to a link in either a chain or a cycle. Links are represented by a ghost set, consisting of a single authoritative token $\mathbf{set\_auth}_{\gamma}\,X$ and multiple fragment tokens $\mathbf{set\_own}_{\gamma}\,x$. This ghost state indicates that X is the disjoint union of the sets $\{x\}$ for all fragment tokens $\mathbf{set\_own}_{\gamma}\,x$. Each link permission $e \leftrightsquigarrow_{\gamma} d$ thus corresponds to a unique link in $\mathcal{L}$ or m.

Verification of Ghost Linking Rules. With these definitions at hand, the exact queue specification from § 5.1 needs to be adjusted slightly. First of all, we need to include the ghost name γ that connects the invariant and the predicates. This also means that the channel representation predicate $\mathbf{C}$ from § 5.2 actually has a ghost name. Second, not only the ghost queues and link permissions are needed in the precondition, but also the invariant $\mathbf{Qctx}_{\text{👻}\,\gamma}$ is necessary to retrieve and use the relevant queue predicates. For instance, the **enqueue** specification is adjusted as follows:

$$\left\langle \mathbf{Qctx}_{\text{👻}\,\gamma} \ast \mathbf{Enq}\,e \mid d,\vec{v}.\ \mathbf{Q}_{\text{👻}\,\gamma}\,d\,e\,\vec{v}\right\rangle\ e.\mathbf{enqueue}\ v\ \left\langle \mathbf{Q}_{\text{👻}\,\gamma}\,d\,e\,(\vec{v} \mathbin{++} [v]) \mid \mathbf{Enq}\,e\right\rangle$$

The proof of this specification uses LA-INV-OPEN to add the ghost linking invariant to the atomic pre- and postcondition, retrieves the relevant queue and then applies the non-ghost linking **enqueue** specification from § 4.4.

The MAKE-GHOST-LINK and MAKE-GHOST-LINK-SELF rules are adjusted in a similar way, by adding $\mathbf{Qctx}_{\text{👻}\,\gamma}$ as an additional assumption:

$$\mathbf{Qctx}_{\text{👻}\,\gamma} \ast \mathbf{Q}_{\text{👻}\,\gamma}\,d_1\,e_1\,\vec{v}_1 \ast \mathbf{Q}_{\text{👻}\,\gamma}\,d_2\,e_2\,\vec{v}_2 \Rrightarrow\!\!\ast\ e_1 \leftrightsquigarrow_{\gamma} d_2 \ast \mathbf{Q}_{\text{👻}\,\gamma}\,d_1\,e_2\,(\vec{v}_1 \mathbin{++} \vec{v}_2)$$
$$\mathbf{Qctx}_{\text{👻}\,\gamma} \ast \mathbf{Q}_{\text{👻}\,\gamma}\,d\,e\,\vec{v} \Rrightarrow\!\!\ast\ e \leftrightsquigarrow_{\gamma} d$$

The proofs of MAKE-GHOST-LINK and MAKE-GHOST-LINK-SELF rely on the fact that Iris allows invariants to be opened around updates $\Rrightarrow\!\!\ast$.

A new ghost invariant $\mathbf{Qctx}_{👻\gamma}$ can be allocated with no channels and no cycles by allocating an empty set and empty map ghost state, using the following rule:

$$\mathsf{True} \Rrightarrow \exists \gamma.\, \mathbf{Qctx}_{👻\gamma} \qquad \text{(GHOST-CTX-ALLOC)}$$

6 Dependent Separation Protocol Channels

In § 5.2 we defined a representation predicate $\mathbf{C}\; c_1\; c_2\; \vec{v}_1\; \vec{v}_2$ for channels and corresponding logically-atomic specifications for the channel operations. This channel predicate has two drawbacks: (1) it describes the exact buffered values $\vec{v}_1$ and $\vec{v}_2$, rather than a high-level protocol, and (2) it involves the other endpoint c_2, which might change over time due to linking. We address these two drawbacks by deriving the high-level specification based on Actris's dependent separation protocols that we described in § 2.2. We address the first drawback by extending Actris's ghost theory to support protocol linking (§ 6.1). The key to addressing the second drawback is our notion of *pairing invariant* to reason abstractly about shared channels with changing endpoints (§ 6.2). Using these, we then derive the desired high-level specifications (§ 6.3).

6.1 Protocol Linking

We use the Actris ghost theory [22, §9.4] to relate dependent separation protocols to the current values in the channel buffers. To support linking of channels, we extend the Actris ghost theory with a rule to combine two protocol contexts into a single context.

The Actris ghost theory provides an authoritative token $\mathbf{prot\_auth}_{\gamma_1,\gamma_2}\; \vec{v}_1\; \vec{v}_2$ governing the state of the channel buffers, and two tokens $\mathbf{prot\_own}_{\gamma_1}\; \mathit{prot}_1$ and $\mathbf{prot\_own}_{\gamma_2}\; \mathit{prot}_2$ describing the local state of the protocol at each endpoint. It has the following rules:

$$\mathsf{True} \Rrightarrow \exists \gamma_1, \gamma_2.\, \mathbf{prot\_auth}_{\gamma_1,\gamma_2}\; []\; [] * \mathbf{prot\_own}_{\gamma_1}\; \mathit{prot} * \mathbf{prot\_own}_{\gamma_2}\; \overline{\mathit{prot}} \qquad \text{(PROTO-ALLOC)}$$

$$\begin{aligned}&\mathbf{prot\_own}_{\gamma_1}\; (!\,\vec{x}\,\langle v\rangle\{P\}.\,\mathit{prot}) * \mathbf{prot\_auth}_{\gamma_1,\gamma_2}\; \vec{v}_2\; \vec{v}_1 * P[\vec{y}/\vec{x}] \Rrightarrow \\ &\qquad \mathbf{prot\_own}_{\gamma_1}\; (\mathit{prot}[\vec{y}/\vec{x}]) * \mathbf{prot\_auth}_{\gamma_1,\gamma_2}\; (\vec{v}_2 \mathrel{++} [v[\vec{y}/\vec{x}]])\; \vec{v}_1 \qquad \text{(PROTO-SEND)}\end{aligned}$$

$$\begin{aligned}&\mathbf{prot\_own}_{\gamma_1}\; (?\,\vec{x}\,\langle v\rangle\{P\}.\,\mathit{prot}) * \mathbf{prot\_auth}_{\gamma_1,\gamma_2}\; \vec{v}_2\; (v :: \vec{v}_1) \Rrightarrow \\ &\qquad \exists \vec{y}.\, v' = v[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}] * \mathbf{prot\_own}_{\gamma_1}\; (\mathit{prot}[\vec{y}/\vec{x}]) * \mathbf{prot\_auth}_{\gamma_1,\gamma_2}\; \vec{v}_2\; \vec{v}_1 \qquad \text{(PROTO-RECV)}\end{aligned}$$

$$\mathbf{prot\_auth}_{\gamma_1,\gamma_2}\; \vec{v}_1\; \vec{v}_2 \dashv\vdash \mathbf{prot\_auth}_{\gamma_2,\gamma_1}\; \vec{v}_2\; \vec{v}_1 \qquad \text{(PROTO-SYMMETRIC)}$$

The rule PROTO-ALLOC allows for creating new authoritative and protocol tokens corresponding to a chosen protocol *prot*. The rules PROTO-SEND and PROTO-RECV ensure for the left endpoint that when sending and receiving values, those values adhere to the protocol, and that resources P defined in the protocol are transmitted. Finally, PROTO-SYMMETRIC allows the endpoint arguments in the authoritative token to be swapped to apply the PROTO-SEND and PROTO-RECV rules to the right endpoint. The actual rules of the Actris ghost theory contain later modalities ($\triangleright$) to support higher-order protocols, which are omitted here for brevity and will be discussed in § 7.

To support linking of endpoints, we extend these rules with a new rule to combine two authoritative tokens into a single one. This rule is key to verify the high-level specification of **`link`**:

$$\begin{aligned}&\mathbf{prot\_own}_{\gamma_2}\; \mathit{prot} * \mathbf{prot\_auth}_{\gamma_1,\gamma_2}\; \vec{v}_2\; \vec{v}_1 * \mathbf{prot\_own}_{\gamma_3}\; \overline{\mathit{prot}} * \mathbf{prot\_auth}_{\gamma_3,\gamma_4}\; \vec{v}_4\; \vec{v}_3 \Rrightarrow \\ &\qquad \mathbf{prot\_auth}_{\gamma_1,\gamma_4}\; (\vec{v}_4 \mathrel{++} \vec{v}_2)\; (\vec{v}_1 \mathrel{++} \vec{v}_3) \qquad \text{(PROTO-JOIN)}\end{aligned}$$

Let us first show how the Actris ghost theory is used to verify channels without linking and why this attempt does not scale to support linking. We would define the channel ownership assertion as:

$$c \rightarrowtail \mathit{prot} \triangleq \exists \gamma, c', \gamma'.\, \mathbf{Ch}\; c * \mathbf{prot\_own}_{\gamma}\; \mathit{prot} * \boxed{\exists \vec{v}_1, \vec{v}_2.\, \mathbf{C}\; c\; c'\; \vec{v}_1\; \vec{v}_2 * \mathbf{prot\_auth}_{\gamma,\gamma'}\; \vec{v}_2\; \vec{v}_1}$$

This definition relies on the fact that channel predicates $\mathbf{C}$ and protocol contexts **proto_ctx** are symmetric (CHAN-SYMMETRIC and PROTO-SYMMETRIC).

This definition of $c \rightarrowtail prot$ fixes the other endpoint c' in the invariant. It would therefore not extend to linking, in which the other endpoint of a channel ownership assertion can be changed without updating the given assertion. Our approach to this problem is to introduce a single pairing invariant for all channels, which maintains not only the shared resources for channels, but also a changeable—rather than fixed—pairing of endpoints.

6.2 Pairing Invariants

We introduce a new form of invariant—called a *pairing invariant*—which supports sharing resources between two paired parties, even when the pairing of the parties themselves can be changed. In the case of channels, the pairing invariant allows us to share resources between two endpoints of a channel, even when endpoints of different channels can be linked.

Pairing invariants are described by separation logic predicates $R : A \to A \to \mathsf{iProp}$, specifying the resources $R\,a\,b : \mathsf{iProp}$ for each pair $a, b \in A$. We require the following properties for R:

$$R\ a\ a' \vdash a \neq a' \qquad \text{(IRREFLEXIVE)}$$

$$R\ a\ b \dashv\vdash R\ b\ a \qquad \text{(SYMMETRIC)}$$

$$R\ a\ b * R\ a\ b' \vdash \mathsf{False} \qquad \text{(UNIQUE)}$$

The IRREFLEXIVE property states that we cannot have a paired resource between the same party. This ensures that when adding a resource $R\ a\ b$ to an invariant, we can always create two distinct parties a and b to access the resource. The SYMMETRIC property states that we can convert between $R\ a\ b$ and $R\ b\ a$, *i.e.,* the resources described by R are independent of the order of arguments. This allows us to always consider a party a to correspond to a resource with a as first argument. The UNIQUE property ensures that we cannot have multiple pairings with the same party. In combination with the first and second properties, this ensures that there is always a single unique resource for each party a, meaning that we can always add a new resource $R\ a\ b$ to the pairing invariant.

Placing $R\ a\ b$ directly in an Iris invariant of the form $\boxed{R\ a\ b}$ would fix the pairing between a and b. Instead, we add a form of indirection using Iris ghost state, using which we provide for each party a a witness that there is a b such that $R\ a\ b$, but not for which specific b this holds.

The ghost state consists of two parts: the first is a pairing invariant $\leftrightarrows_\gamma R$ which maintains the pairing of parties associated with the ghost name γ, and the second is the token $\mathbf{Tok}_\gamma\ a$, which states that a has an associated resource $R\ a\ b$ in the pairing invariant $\leftrightarrows_\gamma R$ for some b, but not which specific b. This indirection allows us to change the exact pairing of parties in $\leftrightarrows_\gamma R$, as long as we maintain a unique pairing for each token $\mathbf{Tok}_\gamma\ a$. The ghost state has the following rules:

$$\mathsf{True} \Rrightarrow \exists \gamma.\ \leftrightarrows_\gamma R \qquad \text{(PAIR-ALLOC)}$$

$$\leftrightarrows_\gamma R * R\ a\ b \Rrightarrow \mathbf{Tok}_\gamma\ a * \mathbf{Tok}_\gamma\ b * \leftrightarrows_\gamma R \qquad \text{(PAIR-ADD)}$$

$$\leftrightarrows_\gamma R * \mathbf{Tok}_\gamma\ a \Rrightarrow \exists b.\ R\ a\ b * (R\ a\ b \Rrightarrow \mathbf{Tok}_\gamma\ a * \leftrightarrows_\gamma R) \qquad \text{(PAIR-UPDATE)}$$

$$\begin{aligned} &\leftrightarrows_\gamma R * \mathbf{Tok}_\gamma\ b * \mathbf{Tok}_\gamma\ c \Rrightarrow \qquad \text{(PAIR-UPDATE-2)}\\ &\quad (R\ b\ c * (R\ b\ c \Rrightarrow \mathbf{Tok}_\gamma\ b * \mathbf{Tok}_\gamma\ c * \leftrightarrows_\gamma R) \wedge (\mathsf{True} \Rrightarrow \leftrightarrows_\gamma R))\\ &\quad \vee (\exists a, d.\ R\ a\ b * R\ c\ d * (R\ a\ b * R\ c\ d \Rrightarrow \mathbf{Tok}_\gamma\ b * \mathbf{Tok}_\gamma\ c * \leftrightarrows_\gamma R) \wedge (R\ a\ d \Rrightarrow \leftrightarrows_\gamma R)) \end{aligned}$$

The rule PAIR-ALLOC can be used to create a new pairing invariant $\leftrightarrows_\gamma R$ with a new ghost name γ. Using PAIR-ADD, a new paired resource $R\ a\ b$ can be added to a pairing invariant $\leftrightarrows_\gamma R$, resulting in tokens $\mathbf{Tok}_\gamma\ a$ and $\mathbf{Tok}_\gamma\ b$ associated with the pairing invariant $\leftrightarrows_\gamma R$.

Using PAIR-UPDATE, the pairing invariant $\leftrightarrows_\gamma R$ and a token $\mathbf{Tok}_\gamma\ a$ can be given up to temporarily access the resource $R\ a\ b$, after which the resource $R\ a\ b$ can be given up to get back the token $\mathbf{Tok}_\gamma\ a$ and the pairing invariant $\leftrightarrows_\gamma R$.

Finally, PAIR-UPDATE-2 is an extension of PAIR-UPDATE to retrieve resources associated with two tokens $\mathsf{Tok}_\gamma\ b$ and $\mathsf{Tok}_\gamma\ c$ from a pairing invariant $\leftrightarrows_\gamma R$. This results in either a single resource $R\ b\ c$ or two different resources $R\ a\ b$ and $R\ c\ d$. Later, we can either give up the same resources to get the tokens and pairing invariant $\leftrightarrows_\gamma R$ back, or give up the joined resources True and $R\ a\ d$ respectively to get back the pairing invariant $\leftrightarrows_\gamma R$.

As we will see in § 6.3 these four rules are sufficient to reason about the shared channel state in the context of linkable channels. When $\leftrightarrows_\gamma R$ is placed in an invariant $\boxed{\leftrightarrows_\gamma R}$, PAIR-ADD, PAIR-UPDATE and PAIR-UPDATE-2 describe how to retrieve resources from the opened invariant and which resources to give up in order to close the invariant after updating the resources.

Definition of the Pairing Invariant. The pairing invariant $\leftrightarrows_\gamma R$ and tokens $\mathsf{Tok}_\gamma\ a$ are defined using Iris ghost state mechanisms. At the base of this definition lies a partial pairing map from each party to its paired party. However, as only a single resource is shared between each pair of resources, the pairing map also assigns one of the paired parties to be the *primary party*. Hence a pairing map is a partial map from parties $a \in A$ to both a party $b \in A$ and a Boolean $s \in \mathbb{B}$ indicating whether a is the *primary party*, satisfying the following pairing map condition:

$$\mathtt{is\_pairing}\ (m : A \xrightarrow{\text{fin}} (A \times \mathbb{B})) \triangleq \forall a, b, s.\ m\,a = (b, s) \rightarrow m\,b = (a, \neg s)$$

This condition ensures that each paired a is paired to a unique b, and that only one of a and b is the primary party. Note that we implicitly have $a \neq b$ as the same party a cannot be both the primary ($m\,a = (a, \mathtt{true})$) and the non-primary ($m\,a = (a, \mathtt{false})$) party. Using this, the pairing invariant consists of the resources for each pairing in the map, and each token $\mathsf{Tok}_\gamma\ a$ corresponds to a key in the pairing map. This is formalized as:

$$\begin{aligned}
\mathtt{paired\_res}\ R\ E &\triangleq \exists m.\ \mathtt{is\_pairing}\ m * E = \operatorname{dom} m\ * \\
&\qquad\qquad \mathop{\scalebox{1.5}{$*$}}_{a \mapsto (b,s) \in m}.\ \mathtt{if}\ s\ \mathtt{then}\ R\ a\ b\ \mathtt{else}\ \mathrm{True} \\
\leftrightarrows_\gamma R &\triangleq \exists E.\ \mathtt{set\_auth}_\gamma\ E * \mathtt{paired\_res}\ R\ E \\
\mathsf{Tok}_\gamma\ a &\triangleq \mathtt{set\_own}_\gamma\ a
\end{aligned}$$

These definitions use a ghost set (similar to § 5.3) consisting of $\mathtt{set\_auth}_\gamma\ E$ and $\mathtt{set\_own}_\gamma\ a$ to ensure that each party a for a token $\mathsf{Tok}_\gamma\ a$ is in the domain E of the pairing map m.

These definitions allow us to prove the four rules of pairing invariants. For PAIR-ADD, we extend the map m with a pairing a, b, extending the domain of m. When extending E accordingly, we create new ghost tokens $\mathtt{set\_own}_\gamma\ a$ and $\mathtt{set\_own}_\gamma\ b$ for the new parties. For PAIR-UPDATE, we use the ghost state to determine that $a \in E$, and hence retrieve the relevant resource $R\ a\ b$ from the pairing invariant. Finally, for PAIR-UPDATE-2, we use the ghost state twice to determine that $b, c \in E$, and hence retrieve either $R\ b\ c$ from the pairing invariant when b and c are paired to each other, or $R\ a\ b$ and $R\ c\ d$ when b and c are paired to different parties. When giving up the joined resources $R\,a\,d$, the pairings (a, b) and (c, d) in m are replaced by the pairing (a, d) and parties b and c are removed from E by deallocating the tokens for b and c.

6.3 Specification of Protocol-Based Channels

In § 6.1 we defined $c \rightarrowtail prot$ for channels without linking, using a single invariant per channel, and remarked that this approach would not extend to linking because it fixes both endpoints of the channel. In section § 6.2 we introduced pairing invariants to obtain a form of indirection to allow for changing endpoints with shared resources. Combining these, we define the assertion $c \rightarrowtail prot$ for channels with linking, using a single pairing invariant for all channels.

The key step in this approach is to select the correct endpoint values A and pairing predicate $R : A \rightarrow A \rightarrow \mathrm{iProp}$ to describe channels with protocols, without fixing the endpoints. Note that the invariant in § 6.1 fixes the channel value c and protocol ghost name γ for each endpoint of the channel. Hence, we choose $A \triangleq \mathsf{Val} \times \mathsf{GName}$ as the endpoint value.

Next we choose the instance $\mathbf{R}_{\gamma_{chan}} : A \to A \to \mathsf{iProp}$ of the pairing predicate to be:

$$\mathbf{R}_{\gamma_{chan}}\ (c_1, \gamma_1)\ (c_2, \gamma_2) \triangleq \exists \vec{v}_1, \vec{v}_2.\ \mathbf{C}_{\gamma_{chan}}\ c_1\ c_2\ \vec{v}_1\ \vec{v}_2 * \mathbf{prot\_auth}_{\gamma_1,\gamma_2}\ \vec{v}_2\ \vec{v}_1$$

This relation describes the same shared state as in § 6.1, but using parameters for each of the two channel endpoints. In order to use this $\mathbf{R}$ in a pairing invariant, we need to establish the IRREFLEXIVE, SYMMETRIC, and UNIQUE properties for $\mathbf{R}$. Both $\mathbf{C}$ and $\mathbf{prot\_ctx}$ are symmetric, meaning that SYMMETRIC holds for $\mathbf{R}$. Furthermore, the endpoints of a ghost queue $\mathbf{Q}_{\text{👻}}$ are unique, meaning that atomic channels and hence $\mathbf{R}$ are both irreflexive and unique, thereby satisfying all three requirements for using $\mathbf{R}$ in a pairing invariant.

The resources for channels and ghost queues are shared between the endpoints. This shared state is captured by the channel invariant:

$$\mathbf{chan\_ctx}_{\gamma} \triangleq \mathbf{Qctx}_{\text{👻}}\,\gamma_{chan} * \boxed{\leftrightarrows_{\gamma_{pair}}\ R_{\gamma_{chan}}}$$

As both parts of the channel invariant are Iris invariants, the channel invariant can be duplicated. A channel invariant can be created using GHOST-CTX-ALLOC and PAIR-ALLOC:

$$\mathsf{True} \Rrightarrow \exists \gamma.\ \mathbf{chan\_ctx}_{\gamma}$$

The shared channel resources are captured by the pairing invariant using an endpoint token, resulting in the following definition of the channel ownership assertion:

$$c \rightarrowtail_{\gamma} \mathit{prot} \triangleq \mathbf{chan\_ctx}_{\gamma} * \exists \gamma_c.\ \mathbf{Ch}\,c * \mathbf{Tok}_{\gamma_{pair}}\ (c, \gamma_c) * \mathbf{prot\_own}_{\gamma_c}\ \mathit{prot}$$

Each channel endpoint $c \rightarrowtail_{\gamma} \mathit{prot}$ contains the channel handle $\mathbf{Ch}\,c$ for that endpoint, which can be used to send and receive values. They also contain an endpoint token $\mathbf{Tok}_{\gamma_{pair}}\ (c, \gamma_c)$, which can be used to retrieve the shared channel $\mathbf{R}_{\gamma_{chan}}\ (c, \gamma_c)\ (c', \gamma_{c'})$ resources from the pairing invariant $\leftrightarrows_{\gamma_{pair}}\ \mathbf{R}_{\gamma}$. And finally, the protocol token $\mathbf{prot\_own}_{\gamma}\ \mathit{prot}$ fixes the protocol for that endpoint, and is used to update the shared protocol invariant when sending and receiving values. The ownership assertion also contains the channel invariant. Using these channel ownership assertions, we derive the high-level specifications of channels with protocols:

$$\{\mathbf{chan\_ctx}_{\gamma}\}\ \mathbf{new\_chan}\ ()\ \{(c, \overline{c}).\ c \rightarrowtail_{\gamma} \mathit{prot} * \overline{c} \rightarrowtail_{\gamma} \overline{\mathit{prot}}\}$$
$$\{c \rightarrowtail_{\gamma} \mathbf{!}\,\vec{x}\,\langle v \rangle \{P\}.\ \mathit{prot} * P[\vec{y}/\vec{x}]\}\ c.\mathbf{send}\ (v[\vec{y}/\vec{x}])\ \{c \rightarrowtail_{\gamma} \mathit{prot}[\vec{y}/\vec{x}]\}$$
$$\{c \rightarrowtail_{\gamma} \mathbf{?}\,\vec{x}\,\langle v \rangle \{P\}.\ \mathit{prot}\}\ c.\mathbf{recv}\ ()\ \{w.\ \exists \vec{y}.\ w = v[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}] * c \rightarrowtail_{\gamma} \mathit{prot}[\vec{y}/\vec{x}]\}$$
$$\{c \rightarrowtail_{\gamma} \mathit{prot} * c' \rightarrowtail_{\gamma} \overline{\mathit{prot}}\}\ c.\mathbf{link}\ c'\ \{\mathsf{True}\}$$

The **new_chan** specification is derived from the one for atomic channels. Using PROTO-ALLOC we additionally create a protocol invariant, before adding the shared channel state to the pairing invariant using PAIR-ADD and creating the channel ownership assertions. The **send** and **recv** specifications are derived from the corresponding rules for atomic channels, using PAIR-UPDATE to retrieve the shared channel state and update the protocol context. Finally, the **link** specification follows from the atomic channel specification using PAIR-UPDATE-2 to retrieve the relevant shared channel states and the new PROTO-JOIN rule to combine the protocol contexts for the linked channel state. As the channel ownership assertions contain the channel context, only the **new_chan** specification differs from the specification in § 2.2. Additionally, ownership assertions for channels have an associated name γ corresponding to the context, as only channels for the same channel context can be linked. We hide the tracking of ghost names using type classes in Coq [52].

7 Higher-Order Protocols

Iris and Actris allow for the verification of higher-order programs, *i.e.,* programs in which channel endpoints and functions are treated as first-class data, and can be put into data structures, stored in references and sent over channels. In this section we give an example of a higher-order program and its specification using a higher-order protocol. We then show the changes that are necessary to support higher-order protocols in our work.

Example. Channel endpoints are first class and can thus be sent over channels, for example:

$$prog_9\, c \triangleq \mathtt{let}\, x = c.\mathtt{recv}\,()\, \mathtt{in}\, \mathtt{let}\, d = c.\mathtt{recv}\,()\, \mathtt{in}\, d.\mathtt{send}\, x;\ c.\mathtt{link}\, d$$

This program receives an arbitrary value x followed by a channel d over c. It then sends the value over d and finally links the two channels. We can prove the Hoare triple $\{c \rightarrowtail prot\}\ prog_9\ c\ \{\mathrm{True}\}$ with the following general protocol:

$$c \rightarrowtail \mathbf{?}\,(v : \mathsf{Val})\,\langle v\rangle.\,\mathbf{?}\,(d : \mathsf{Val})(prot' : \mathsf{iProto})\,\langle d\rangle\{d \rightarrowtail \mathbf{!}\,\langle v\rangle.\,prot'\}.\,\overline{prot'}$$

This protocol uses higher-order impredicative quantification over the protocol $prot'$: iProto and transfers a channel ownership assertion $d \rightarrowtail \mathbf{!}\,\langle v\rangle.\,prot'$ over the channel.

Changes. The key change is the definition of dependent separation protocols. These can no longer be simple first-order structures, but need to be defined in such a way that impredicative quantifiers are allowed and arbitrary Iris propositions (including channel ownership and Hoare triples) can be put into the resource $\{P\}$ part of protocols.

Actris [22, §9.1] defines protocols as the solution of a recursive domain equation in the category of metric spaces [2, 6], which is solved using the technique of step-indexing [1, 4]. As a consequence of employing step-indexing, judicious later modalities ($\triangleright$) [5, 46] show up in our definitions and rules that need to eliminated accordingly. In particular, the rules of the Actris ghost theory (§ 6.1) contain later modalities. Most interesting is our new rule PROTO-JOIN:

$$\mathtt{prot\_own}_{\gamma_2}\, prot * \mathtt{prot\_auth}_{\gamma_1,\gamma_2}\, \vec{v}_2\, \vec{v}_1 * \mathtt{prot\_own}_{\gamma_3}\, \overline{prot} * \mathtt{prot\_auth}_{\gamma_3,\gamma_4}\, \vec{v}_4\, \vec{v}_3 \Rrightarrow\!\!* \\ \triangleright^{|\vec{v}_2|+|\vec{v}_3|}\, \mathtt{prot\_auth}_{\gamma_1,\gamma_4}\, (\vec{v}_4 +\!\!+ \vec{v}_2)\ (\vec{v}_1 +\!\!+ \vec{v}_3)$$

This rule says that to merge two protocols, we need to eliminate a number of later modalities proportional to the number of messages in the buffers. The later modalities arise because the proof needs to unfold the recursive definition of protocols $|\vec{v}_2| + |\vec{v}_3|$ times.

In vanilla Iris, a single later modality can be eliminated by taking a program step, so $|\vec{v}_2| + |\vec{v}_3|$ program steps are needed to eliminate the necessary laters. However, as **prot_ctx** is in an invariant, all laters have to be eliminated at a single step during the linearization point. To enable more flexible reasoning using laters, we employ *(non-persistent)*[1] *time receipts* [42] and *later credits* [53]. The time receipt $⧗ n$ keeps track of the number of steps the program has performed, and the later credit $£\, n$ gives permission to eliminate n laters. After each program step we obtain a new time receipt and $£\,(n+1)$ later credits, as for example shown in the strengthened LAT for **link**:

$$\langle \mathsf{Ch}\, c_2 * \mathsf{Ch}\, c_3 \mid n, b, c_1, c_4, \vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{v}_4.\, ⧗ n * \mathsf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2 * (\mathtt{if}\, b\, \mathtt{then}\, \mathsf{C}\, c_3\, c_4\, \vec{v}_3\, \vec{v}_4\, \mathtt{else}\, c_1 = c_3)\rangle \\ c_2.\mathtt{link}\, c_3 \\ \langle ⧗ (n+1) * £\,(n+1) * \mathtt{if}\, b\, \mathtt{then}\, \mathsf{C}\, c_1\, c_4\, (\vec{v}_1 +\!\!+ \vec{v}_3)\, (\vec{v}_4 +\!\!+ \vec{v}_2)\, \mathtt{else}\, \mathrm{True} \mid \mathrm{True}\rangle$$

We update the channel pairing invariant **R** from § 6.3 with time receipts as follows:

$$\mathsf{R}\ (c_1, \gamma_1)\ (c_2, \gamma_2) \triangleq \exists \vec{v}_1, \vec{v}_2.\, ⧗ |\vec{v}_1| * ⧗ |\vec{v}_2| * \mathsf{C}\, c_1\, c_2\, \vec{v}_1\, \vec{v}_2 * \mathtt{prot\_auth}_{\gamma_1,\gamma_2}\, \vec{v}_2\, \vec{v}_1$$

[1]We use non-persistent time receipts $⧗ n$, rather than persistent time receipts $⧖ n$ as they are additive and hence can be added when merging channel predicates during linking.

To verify the Actris-style specification for **link**, we choose $n = |\vec{v}_1| + |\vec{v}_3|$ in the LAT for **link**. This gives us $n + 1$ later credits to eliminate the laters resulting from the PROTO-JOIN. Note that as the time receipts are stored in the channel pairing invariant, the Actris-style specification for channels remain unchanged. They do not leak that we internally use laters, time receipts, and later credits.

8 Mechanization in Coq

All our results are mechanized using the Iris Proof Mode [34, 36] in the Coq proof assistant. Our Coq development consists of ca. 4,250 LOC, with ca. 3,360 LOC for the layered verification and ca. 890 LOC for all examples throughout the paper. The line counts are measured without comments and consecutive blank lines.

The Coq development follows the same layered structure as the paper: the LATs for unidirectional queues based on linked lists and array-segments (§ 4.5, 350 and 680 LOC, respectively), ghost linking (§ 5.3, 550 LOC), the LATs for bidirectional channels (§ 5.2, 210 LOC), pairing invariants (§ 6.2, 350 LOC), and finally the high-level specification using dependent separation protocols (§ 6.3, 410 LOC). Our Coq development imports the model of dependent separation protocols and the ghost theory from Actris. It includes the extension of the Actris ghost theory with linking (§ 6.1, 170 LOC) and extensions of the Actris tactics for symbolic execution (500 LOC).

Since our system supports all rules and tactics of the existing Actris logic, one could redo the proofs of the examples in the Actris papers such as parallel merge sort, a load-balancing mapper and map-reduce. One could also redo the verification of the session type system by Hinrichsen et al. [24], who used the logical approach to type soundness [56] to give a semantic interpretation of session types in terms of Actris's dependent separation protocols. The benefit is that we get end-to-end correctness guarantees against a low-level implementation of channels. We have not explicitly done this since it involves copy/pasting the existing proofs in Coq.

9 Related and Future Work

Session types. Various approaches to the semantics of **link** have been studied in research on session types. Caires and Pfenning [10]'s seminal paper on the Curry-Howard correspondence for session types defines a process $id_A(x, y)$ that corresponds to the identity rule of linear logic. The process acts as a mediator between channels x and y: forwarding the messages received on one channel to the other. Toninho et al. [57] add linking as a primitive and define its semantics through substitution. This approach is also used in subsequent theoretical work by Wadler [60], Lindley and Morris [39] and Fowler et al. [17], and in the session-typed languages Rast [15] and CLASS [51]. The session-typed language Concurrent C0 [61] provides an efficient implementation of **link** using a special kind of forwarding message.

The mediator and substitution approach are well-suited for the study of the meta theory of session types, but not for efficient implementation. The mediator requires a thread to be spawned for each use of **link**, and a direct implementation of the substitution approach would require a global lock. Moreover, the mediator $id_A(x, y)$ is defined on the structure of the session type A in order to decide from which channel to send and which to receive. As such, the mediator only works for well-typed programs. Similarly, the channel used to transfer the forwarding message in Concurrent C0 depends on the direction of the session type. Our implementation of **link** is independent of types, and thus works for programs that are beyond the reach of vanilla session types, *e.g.*, with data dependencies such as the recursive example in § 2.1. Concurrent C0 has an asynchronous semantics, but uses buffers in a single direction, and therefore does not support asynchronous subtyping [44, 45]. The other results use a synchronous semantics.

Network verification. Gondelman et al. [19] verify a distributed implementation of asynchronous channels over UDP using the Iris-based Aneris framework in Coq [37]. Similar to us, they use Actris's dependent separation protocols to obtain high-level specifications. They consider a distributed implementation instead of a shared-memory implementation, do not consider `link`, and do not layer their verification using LATs. Supporting `link` in a distributed setting would be fundamentally different than linking queues, and an interesting challenge for future work.

Verification of concurrent queues. Queues are an important object of study in the verification of linearizability of concurrent data structures—they were the main data structure in Herlihy and Wing [20]'s seminal paper on linearizability. Today, there are many ways to specify linearizability: the classical definition based on traces [20], refinement [16, 38], and logical atomicity [13, 29]. We use logical atomicity because it provides good support for layering specifications.

Logical atomicity has been verified for many queues, but up to our knowledge, none of these verifications include a `link` operation. Most closely related is the single-reader/writer queue verified by Bizjak et al. [7]. They use a HOCAP [54] style specification instead of LATs, but we borrowed the idea of using enqueue and dequeue handles. They only consider a linked-list based version, not a version with array segments. Many of the queues that have been verified using Iris are more sophisticated in the sense that they support multiple readers/writers, *e.g.*, [12, 58, 59]. It is not obvious how to implement a queue that supports both `link` and multiple readers/writers. However, if one were to have such a queue with corresponding LATs, we expect that our abstraction layers above it (ghost linking and onwards) can be ported. Such a queue would make it possible to obtain LATs for the top layer (dependent separation protocol specifications), and thereby make it possible to put $c \rightarrowtail \mathit{prot}$ connectives in Iris invariants and share channel endpoints between threads. Tassarotti et al. [55] verify a refinement of an implementation of session channel w.r.t. a high-level semantics using Iris. They consider a simply-typed session language without `link`, but go beyond our work by considering preservation of termination under fair scheduling.

Future work. Our implementation of channels is much closer to the code that runs on an actual computer than the state-of-the-art on session types, but further improvements could be made. One could consider a real programming language such as C or Rust (instead of HeapLang). The closest work is that of Mansky et al. [40], who verified a message system written in C using the VST separation logic in Coq [11]. Their system is tailored to the communication between sensors and control systems and is therefore very different from session channels (particularly, it does not have unbounded buffers nor `link`). Another improvement would be to verify an implementation that uses relaxed memory accesses, but logical atomicity for relaxed memory concurrency is currently an active study on its own [14, 41, 49]. Finally, unlike some work on session types, the majority of work on Iris/Actris does not consider deadlock freedom. To remedy this, Jacobs et al. [30] presented a linear version of Iris/Actris that guarantees deadlock freedom for channels without `link`. In the work of Jacobs et al. [30], channels are primitive operations in the operational semantics, and not a low-level implementation. It remains an open challenge to verify deadlock freedom of low-level implementations of channel operations. Finally, it would be interesting to explore the verification of a low-level implementation of multiparty session types [27]. We expect that ideas from our bottom abstraction layers (queues, queues with linking, bidirectional channels) could be generalized to the multiparty case. However, the highest level (protocols) remains unknown since Actris-style specifications have only been considered for the multiparty case in a synchronous setting [23].

Acknowledgments

We thank Freek Wiedijk, Jules Jacobs, Jonas Kastberg Hinrichsen, and the POPL 2024 Student Research Competition reviewers for their feedback on an early version of this work. We also thank the anonymous reviewers and our shepherd Matthew Parkinson for their suggestions.

Data-Availability Statement

The layered channel implementation and verification described in this paper have been mechanized in Coq using the Iris and Actris frameworks. Our Coq mechanization, which also includes the verification of all example programs in this paper, is available on Zenodo [3].

References

[1] Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

[2] Pierre America and Jan J. M. M. Rutten. 1989. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. *JCSS* 39, 3 (1989), 343–375. https://doi.org/10.1016/0022-0000(89)90027-5

[3] Anonymous Authors. 2024. Coq mechanization: Verified Lock-Free Session Channels with Linking. See anonymous supplementary material in HotCRP, available on Zenodo for the final version.

[4] Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

[5] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*, Martin Hofmann and Matthias Felleisen (Eds.). 109–122. https://doi.org/10.1145/1190216.1190235

[6] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. The category-theoretic solution of recursive metric-space equations. *TCS* 411, 47 (2010), 4102–4122. https://doi.org/10.1016/J.TCS.2010.07.010

[7] Ales Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: managing obligations in higher-order concurrent separation logic. *PACMPL* 3, POPL (2019), 65:1–65:30. https://doi.org/10.1145/3290378

[8] John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS (LNCS)*, Vol. 2694. 55–72. https://doi.org/10.1007/3-540-44898-5_4

[9] Stephen Brookes. 2007. A semantics for concurrent separation logic. *TCS* 375, 1-3 (2007), 227–270. https://doi.org/10.1016/J.TCS.2006.12.034

[10] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR (LNCS)*, Vol. 6269. 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

[11] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *JAR* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/S10817-018-9457-5

[12] Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. 2022. Applying formal verification to microkernel IPC at Meta. In *CPP*. 116–129. https://doi.org/10.1145/3497775.3503681

[13] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP (LNCS)*, Vol. 8586. 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

[14] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. In *PLDI*. 792–808. https://doi.org/10.1145/3519939.3523451

[15] Ankush Das and Frank Pfenning. 2022. Rast: A Language for Resource-Aware Session Types. *LMCS* 18, 1 (2022). https://doi.org/10.46298/LMCS-18(1:9)2022

[16] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2009. Abstraction for Concurrent Objects. In *ESOP (LNCS)*, Vol. 5502. 252–266. https://doi.org/10.1007/978-3-642-00590-9_19

[17] Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. 2023. Separating Sessions Smoothly. *LMCS* 19, 3 (2023). https://doi.org/10.46298/LMCS-19(3:3)2023

[18] Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *JFP* 20, 1 (2010), 19–50. https://doi.org/10.1017/S0956796809990268

[19] Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. *PACMPL* 7, ICFP (2023), 847–877. https://doi.org/10.1145/3607859

[20] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *TOPLAS* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[21] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: session-type based reasoning in separation logic. *PACMPL* 4, POPL (2020), 6:1–6:30. https://doi.org/10.1145/3371074

[22] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *LMCS* 18, 2 (2022). https://doi.org/10.46298/LMCS-18(2:16)2022

[23] Jonas Kastberg Hinrichsen, Jules Jacobs, and Robbert Krebbers. 2024. Multris: Functional Verification of Multiparty Message Passing in Separation Logic. *PACMPL* 8, OOPSLA (2024), 322:1–322:29. https://doi.org/10.1145/3689762

[24] Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021. Machine-checked semantic session typing. In *CPP*. 178–198. https://doi.org/10.1145/3437992.3439914

[25] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR (LNCS)*, Vol. 715. 509–523. https://doi.org/10.1007/3-540-57208-2_35

[26] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP (LNCS)*, Vol. 1381. 122–138. https://doi.org/10.1007/BFB0053567

[27] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. 273–284. https://doi.org/10.1145/1328438.1328472

[28] Iris Development Team. 2024. The Coq mechanization of Iris. https://gitlab.mpi-sws.org/iris/iris/

[29] Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *POPL*. 271–282. https://doi.org/10.1145/1926385.1926417

[30] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. *PACMPL* 8, POPL (2024), 1385–1417. https://doi.org/10.1145/3632889

[31] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269. https://doi.org/10.1145/2951913.2951943

[32] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[33] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650. https://doi.org/10.1145/2676726.2676980

[34] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

[35] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS)*, Vol. 10201. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[36] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. https://doi.org/10.1145/3009837.3009855

[37] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *ESOP (LNCS)*, Vol. 12075. 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

[38] Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *PLDI*. 459–470. https://doi.org/10.1145/2491956.2462189

[39] Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *ESOP (LNCS)*, Vol. 9032. 560–584. https://doi.org/10.1007/978-3-662-46669-8_23

[40] William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A verified messaging system. *PACMPL* 1, OOPSLA (2017), 87:1–87:28. https://doi.org/10.1145/3133911

[41] Glen Mével and Jacques-Henri Jourdan. 2021. Formal verification of a concurrent bounded queue in a weak memory model. *PACMPL* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473571

[42] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *ESOP (LNCS)*, Vol. 11423. 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

[43] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *I&C* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

[44] Dimitris Mostrous and Nobuko Yoshida. 2015. Session typing and asynchronous subtyping for the higher-order π-calculus. *I&C* 241 (2015), 227–263. https://doi.org/10.1016/J.IC.2015.02.002

[45] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP (LNCS)*, Vol. 5502. 316–332. https://doi.org/10.1007/978-3-642-00590-9_23

[46] Hiroshi Nakano. 2000. A Modality for Recursion. In *LICS*. 255–266. https://doi.org/10.1109/LICS.2000.855774

[47] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *TCS* 375, 1-3 (2007), 271–307. https://doi.org/10.1016/J.TCS.2006.12.035

[48] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS)*, Vol. 2142. 1–19. https://doi.org/10.1007/3-540-44802-0_1

[49] Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. 2024. A Proof Recipe for Linearizability in Relaxed Memory Separation Logic. *PACMPL* 8, PLDI (2024). https://doi.org/10.1145/3656384

[50] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[51] Pedro Rocha and Luís Caires. 2023. Safe Session-Based Concurrency with Shared Linear State. In *ESOP (LNCS)*, Vol. 13990. 421–450. https://doi.org/10.1007/978-3-031-30044-8_16

[52] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *TPHOLs (LNCS)*, Vol. 5170. 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

[53] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *PACMPL* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

[54] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *ESOP (LNCS)*, Vol. 7792. 169–188. https://doi.org/10.1007/978-3-642-37036-6_11

[55] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP (LNCS)*, Vol. 10201. 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

[56] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. To appear in Journal of the ACM (JACM).

[57] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent session types via intuitionistic linear type theory. In *PPDP*. 161–172. https://doi.org/10.1145/2003476.2003499

[58] Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael–Scott queue (proof pearl). In *CPP*. 76–90. https://doi.org/10.1145/3437992.3439930

[59] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized verification of a fine-grained concurrent queue from Meta's Folly library. In *CPP*. 100–115. https://doi.org/10.1145/3497775.3503689

[60] Philip Wadler. 2014. Propositions as sessions. *JFP* 24, 2-3 (2014), 384–418. https://doi.org/10.1017/S095679681400001X

[61] Max Willsey, Rokhini Prabhu, and Frank Pfenning. 2016. Design and Implementation of Concurrent C0. In *LINEARITY (EPTCS)*, Vol. 238. 73–82. https://doi.org/10.4204/EPTCS.238.8

Received 2024-04-06; accepted 2024-08-18