Dependent Session Protocols in Separation Logic from First Principles (Functional Pearl)

JULES JACOBS, Radboud University Nijmegen, The Netherlands
JONAS KASTBERG HINRICHSEN, Aarhus University, Denmark
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands

We develop an account of dependent session protocols for a functional language with message-passing in concurrent separation logic. Inspired by minimalistic session calculi, we present a layered design: starting from mutable references, we build single-shot channels, session channels, and imperative channels. Whereas previous work on dependent session protocols in concurrent separation logic required advanced mechanisms such as recursive domain equations and higher-order ghost state, we only require the most basic mechanisms to verify that the single-shot channels satisfy single-shot protocols, and subsequently treat these specifications as a black box on top of which we define dependent session protocols. This has a number of advantages in terms of simplicity, elegance, and flexibility: support for subprotocols and guarded recursion *automatically* transfers from the single-shot protocols to the dependent session protocols, and we easily obtain various forms of channel closing. Because the meta theory of our results is so simple, we are able to give all definitions as part of this paper, and mechanize all our results using the Iris framework in less than 1000 lines of Coq.

1 INTRODUCTION

Message passing is a commonly used abstraction for concurrent programming, with languages such as Erlang and Go having native support for it, and languages such as Java, Scala, Rust, and C# having library support for it. Session types offer powerful type systems for message passing concurrency. Originally invented by Honda [1993]; Honda et al. [1998], session types have been extended with a number of exciting features:

(1) **Dependent protocols:** The key ingredient of a session type system is the notion of a *session protocol*, which describes what data should be exchanged. For example, the session protocol **!Z.!Z.?B.end** expresses that two integers are sent, after which a Boolean is received, and the channel is closed. In vanilla session types, protocols were meant to specify the types of the exchanged data. They cannot be used to express that the 'right' values are exchanged (*i.e.,* functional correctness), nor to express data-dependent protocols where the remaining protocol can depend on prior values that have been exchanged.
There have been two lines of work to extend session protocols with logical conditions to remedy this shortcoming. Lozes and Villard [2012]; Craciun et al. [2015]; Hinrichsen et al. [2020] develop program logics that combine concurrent separation logic [O'Hearn 2004; Brookes 2004] with concepts from session types. Bocchi et al. [2010]; Toninho et al. [2011]; Zhou et al. [2020]; Thiemann and Vasconcelos [2020] develop type systems that combine concepts from the theory of dependent and refinement types with session types.

(2) **Integration in functional languages:** While session types were originally developed in the context of π-calculus, a tempting direction is to combine session types with functional programming. In such languages, session-typed channels are considered first-class data, and can be stored in data types and sent over channels (similar to first-class mutable references in ML). The Good Variation (GV) family by Gay and Vasconcelos [2010]; Wadler [2012] extends linear lambda-calculus with channels. The SILL family by Toninho et al. [2013];

Authors' addresses: Jules Jacobs, Radboud University Nijmegen, The Netherlands, julesjacobs@gmail.com; Jonas Kastberg Hinrichsen, Aarhus University, Denmark, hinrichsen@cs.au.dk; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl.

Pfenning and Griffith [2015]; Toninho [2015] uses a monadic embedding of session types into an unrestricted language.

(3) **Minimalistic calculi:** Session-typed languages add a large number of additional constructs to the types and expressions of their base languages. Already in the early days of session types, Kobayashi [2002] showed that session types can be encoded into π-types; an approach that has later been formalized by Dardha et al. [2012, 2017], and Jacobs [2022] applied these ideas to GV-style functional languages.

(4) **Mechanization:** The meta theory of session types is notorious for its complexity. There exist various published broken proofs—including the failure of subject reduction for several multiparty systems [Scalas and Yoshida 2019]. As a result, over the last 5 years there has been an extensive amount of work on the mechanization of session types by among others Thiemann [2019]; Rouvoet et al. [2020]; Hinrichsen et al. [2020, 2021]; Tassarotti et al. [2017]; Goto et al. [2016]; Ciccone and Padovani [2020]; Castro-Perez et al. [2020]; Gay et al. [2020]; Jacobs et al. [2022]; Castro-Perez et al. [2021].

Up to our knowledge, there is no prior work that combines all four features under a single roof. The goal of this functional pearl is thus to do exactly that. We will develop an account of dependent session protocols for a GV-style language in a concurrent separation logic. Our work is built from *first principles*, forcing us to take a minimalistic approach. Our results have been mechanized in the Coq proof assistant using the Iris framework for concurrent separation logic [Jung et al. 2015, 2016; Krebbers et al. 2017a; Jung et al. 2018; Krebbers et al. 2018, 2017b]. In the remainder of the introduction, we give a teaser of our approach and list some of our key insights.

Key idea #1: Implicit buffers through single-shot channels. The first step to formalizing a language with message-passing concurrency is to decide on the semantics of channels. A common approach is to use an asynchronous semantics where the sender enqueues the messages in a buffer, from which the receiver dequeues them. In such a semantics, the receive operation can block if no message is present, but the send operation will always succeed immediately. To model the notion of a buffer, one typically incorporates a linked list in the formal definition of the language, and extends the language with operations to send (enqueue) and receive (dequeue) messages.

To be minimalistic, we want to avoid having to explicitly model the notion of a linked list in our semantics. Inspired by Kobayashi [2002]; Dardha et al. [2017]; Jacobs [2022] we build on top of single-shot channels. These come with functions $\texttt{new1}\ ()$, which creates a new channel; $\texttt{send1}\ c\ v$, which send a message v on channel c (without blocking); and $\texttt{recv1}\ c$, which receives a message v from c (blocks until a message has been sent). On top of the single-shot channels, we define regular multi-shot session channels. For example, the send operation of session channels is defined as:

$$\texttt{send}\ c\ v := \texttt{let}\ c' = \texttt{new1}\ ()\ \texttt{in}\ \texttt{send1}\ c\ (v, c');\ c'$$

This operation not only sends the message v. It also creates a new channel c' for the remainder of the communication, and sends the new channel tupled with the message.

While there is no explicit notion of a buffer/linked-list in the semantics of one-shot channels, nor in the definition of session channels, we will show that the buffer arises implicitly.

Key idea #2: Dependent session protocols via single-shot protocols. Program logics for message-passing concurrency typically come with a *channel points-to* connective $c \rightarrowtail p$, which provides unique ownership of a channel endpoint c that has to obey to a protocol p. These protocols typically have a sequenced structure, describing a dependent session of multiple exchanges. For example, an example of a *dependent separation protocol* in the Actris logic by Hinrichsen et al. [2020, 2022] is $!\,(n : \mathbb{N})\ \langle n \rangle .\, !\,(m : \mathbb{N})\ \langle m \rangle \{n \leq m\} .\, ?\langle m - n \rangle .\, \textbf{end}$. This protocol expresses that two natural numbers $n \leq m$ are sent, and the difference $m - n$ is returned.

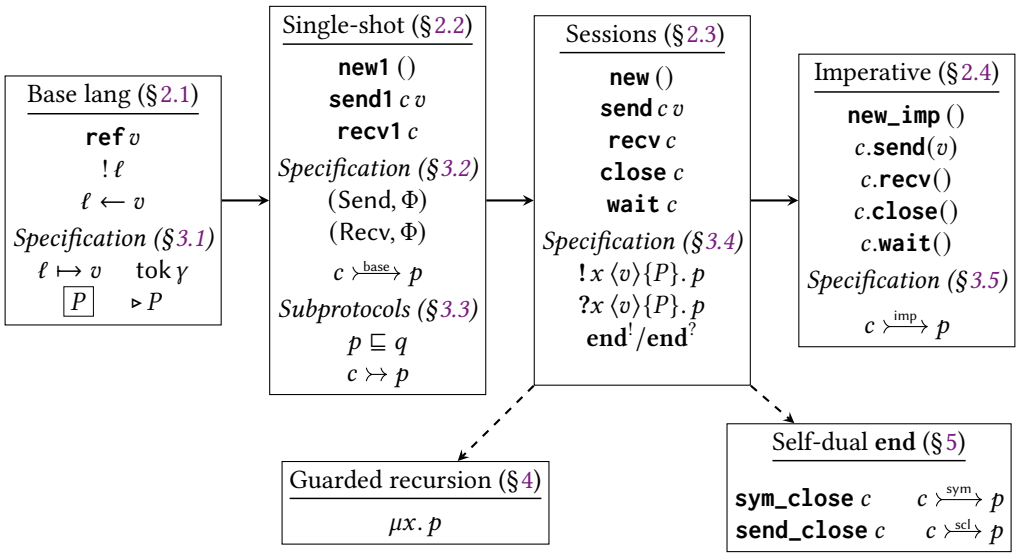

Fig. 1. Layered design of our development.

Similar to our desire for avoiding the need to explicitly model the buffers that underpin channels as linked lists, we would like to avoid having to inductively define such dependent session protocols. In our system, the channel points-to connective for the single-shot channels is simply $c \rightarrowtail (tag, \Phi)$, where $tag \in \{\text{Send}, \text{Recv}\}$ and Φ is a predicate over the exchanged value. While our protocols only describe a single message, dependent session protocols that can describe session channels are simply defined as combinators. This is achieved by recursively using the channel points-to connective for describing the channel continuation inside the base protocol Φ. Due to Iris's support for impredicativity [Svendsen and Birkedal 2014], we can use its fixpoint combinator to define recursive (and dependent) protocols by guarded recursion.

Key idea #3: Layered design. We define our session channels in terms of single-shot channels, and our dependent session protocols as combinators of single-shot protocols, but we wish to go further by layering our design—from below and above. The layered design is shown in Fig. 1.

From below, we do not start with a language that has single-shot channels as primitive. We build on top of a functional language with the usual operations for mutable references (allocation, deallocation, assignment and read) as found in languages of the ML family. One-shot channels are simply defined using ordinary references, and verified using (Iris's) separation logic rules for the verification of concurrent programs with references. Building on top of a language with mutable references has other tangible benefits. First, we can write and verify programs that transfer data by reference. Second, we can define both functional versions of session channels (that return a new endpoint) and imperative versions of channel endpoints (that mutate the channel).

From above, we demonstrate the freedom of our solution, by implementing multiple methods for closing a session. Session types and protocols are often terminated with an explicit tag (such as **end** in the aforementioned example), and it is non-trivial to extend the range of termination tags, in settings where the protocols are defined inductively. Since our session channels are defined as combinators on top of the single-shot channels—that inherently do not include a method for closing—we can freely choose how to close our channels, after the fact. Initially, we implement asymmetric closing, where one endpoint initiates the closing of a channel (protocol $\mathbf{end}^!$), while

the other waits and actually deallocates the memory backing the channel (protocol $\mathbf{end}^?$). We later provide two alternatives with a self-dual **end** protocol: symmetrically closing the channel, where there is a single closing operation, and a combined send-close operation, which sends a last message but does not create a continuation channel. The implementations make sure that the closing call by the last endpoint deallocates the memory.

Key idea #4: Mechanization using a subset of Iris. Our layered design proved beneficial for the meta theory and mechanization of our results. We only need the usual points-to connective $\ell \mapsto v$ for ownership of locations ℓ with value v in separation logic, a simple form of ghost state (unique tokens), and Iris's impredicative invariants. By comparison, the Actris logic by Hinrichsen et al. [2020, 2022] relies on a non-trivial model of recursive protocols using the technique from America and Rutten [1989] for solving recursive domain equations, and uses Iris's mechanism for higher-order ghost state [Jung et al. 2016] to define its channel points-to connective $c \rightarrowtail p$. Since the meta theory of our results is so simple, we are able to give all definitions as part of this paper (there is no appendix) and mechanize all our results in less than 1000 lines of Coq.

Contributions. This paper makes the following contributions:

- A layered implementation of higher-order shared-memory session channels, starting from mutable references, on which we build single-shot channels, session channels, and imperative channels (§2)
- A layered development of separation logic specifications for our channels. We start from a small subset of Iris, developing specifications for single-shot channels, which are then treated as a black box upon which we build high-level dependent separation protocols (§3)
- Support for subprotocols (§3.3) and guarded recursion (§4), which transfers *automatically* from single-shot protocols to dependent session protocols.
- A demonstration of the extensibility obtained by building on first principles, through various methods for closing session channels (§5)
- A small and intuitive mechanization in the Coq proof assistant, comprised of less than 1000 lines of Coq code (§6)

2 LAYERED IMPLEMENTATION OF CHANNELS

In this section we will implement message passing channels in terms of low-level operations. We build these channels in several layers:

- We start by describing the base language and its low-level operations (§2.1).
- We then build a library of single-shot channels (§2.2).
- On top of this we build functional multi-shot session channels (§2.3).
- As a final layer, we have imperative session channels (§2.4).
- Using an example we show that linked lists (buffers) implicitly emerge (§2.5).

In the subsequent §3, we develop specifications and proof for each of the layers, and demonstrate how to verify the correctness of the example.

2.1 Base Language

As a base language we use HeapLang, a low-level concurrent language that comes with the Iris separation logic framework. HeapLang has the purely functional operations that one would expect, such as arithmetic and conditionals, and also includes products, sums, and unit values. For the purpose of this paper, the following operations on mutable memory locations are the most relevant:

$\texttt{ref}\, v$ Allocate a new memory location that initially stores value v.
$!\, \ell$ Read the value from memory location ℓ.

$\ell \leftarrow v$ Write value v to location ℓ.
free ℓ Free the memory location ℓ

HeapLang additionally includes a primitive for spawning a new thread:

fork $\{e\}$ Run program e in a new thread.

The program e is allowed to refer to variables in the surrounding lexical context.

The following is a grammar of the most notable constructs that we will use:

$$e \in \text{Expr} ::= \textbf{ref}\, e \mid !\, e \mid e \leftarrow e \mid \textbf{fork}\ \{e\} \mid \textbf{free}\, e \mid \textbf{match}\, e\, \textbf{with Some}\, x \Rightarrow e;\, \textbf{None} \Rightarrow e\, \textbf{end} \mid x \mid e\ e \mid \lambda x.e \mid \textbf{assert}(e) \mid \textbf{for}(x = e..e)\ e \mid \ldots$$

2.2 Single-Shot Channels

At the base of our development lie single-shot channels, which communicate a single message from a sender to a receiver. The API consists of the following operations:

new1 () Create and return a new single-shot channel c.
send1 $c\, v$ Send message v on channel c (non-blocking).
recv1 c Receive message v from channel c (blocks until a message is sent).

The channels are single-shot; only one value is sent over the channel, after which point the channel is deallocated as a part of **recv1** c.

Example of using single-shot channels. These channels enable us to set up a communication between child and parent threads as in the following example:

```
prog_single :=
  let c = new1 () in
  fork {let l = ref 42 in send1 c l};
  assert(!(recv1 c) = 42)
```

The main thread creates a single-shot channel c, which is shared between the main thread and a forked-off thread. The forked-off thread then dynamically allocates a reference to 42, and sends the location over the channel. Finally, the main thread receives the reference, reads it, and asserts that the stored value is 42. To communicate several times, we could share several channels, but an interesting alternative style that allows unbounded communication is to send a new channel along with the message, as we shall see in §2.3.

In the HeapLang semantics, **assert** gets stuck if the condition is false. Safety (the fact that the **assert** does not fail) crucially depends on the forked-off thread not modifying the reference after it has sent it. We verify this safe transfer of ownership in §3.2. This example is safe as the exclusive permission to write and read the reference first belongs to the forked-off thread, after which it is transferred to the main thread. This goes beyond standard session types due to reference ownership and the verification of the **assert**.

Implementation of single-shot channels. In our development, channels are not primitive but implemented in terms of low-level mutable references. A channel is represented as a mutable reference that initially contains the value **None**. To send a value v to the channel, we set the mutable reference to **Some** v. To receive from the channel, we read the value of the mutable reference in a loop, until we see the **None** change to **Some** v. We then deallocate the mutable reference, and

return v. This gives us the following implementation:

$$\begin{aligned}
\texttt{new1}\ () &\triangleq \texttt{ref None}\\
\texttt{send1}\ c\ v &\triangleq c \leftarrow \texttt{Some}\ v\\
\texttt{recv1}\ c &\triangleq \texttt{match}\ !c\ \texttt{with}\\
&\quad \mid \texttt{Some}\ v \Rightarrow \texttt{free}\ c;\ v\\
&\quad \mid \texttt{None} \Rightarrow \texttt{recv1}\ c\\
&\quad \texttt{end}
\end{aligned}$$

This implementation shows that safety also depends on the fact that clients only call **recv** once, and does not call **send1** after a completed **recv1**. These would otherwise result in a double-free and use-after-free, which get stuck in the HeapLang semantics.

HeapLang has a sequentially consistent memory model. In a weaker memory model, the store/load instructions should use release/acquire memory order options (or stronger). However, in this paper, we use HeapLang's sequentially consistent memory model. This is a common assumption in the literature on Iris, with the exception of some papers specifically focused on weak memory [Mével and Jourdan 2021; Kaiser et al. 2017; Dang et al. 2020],

2.3 Session Channels

A session channel facilitates sequences of messages between two channel endpoints, which is useful for implementing client-server style concurrency. Session channels have the following API:

- **new** () Create a new session channel.
- **send** $c\ e$ Send message e on session channel c. Return a session channel on which to continue communication.
- **recv** c Receive from session channel c. Return a pair (v, c) of the received message v and the session channel on which to continue communication.
- **close** c Send termination message.
- **wait** c Wait for the termination message and deallocate the channel.

In this section we demonstrate how single-shot channels can be used to implement session channels. The session channels are obtained by allocating and exchanging a new single-shot channel whenever a value is sent. The new single-shot channel is then used as a continuation of the session. The session channels are implemented as follows:

$$\begin{aligned}
\texttt{new}\ () &:= \texttt{new1}\ ()\\
\texttt{send}\ c\ v &:= \texttt{let}\ c' = \texttt{new1}\ ()\ \texttt{in}\ \texttt{send1}\ c\ (v, c');\ c'\\
\texttt{recv}\ c &:= \texttt{recv1}\ c\\
\texttt{close}\ c &:= \texttt{send1}\ c\ ()\\
\texttt{wait}\ c &:= \texttt{recv1}\ c
\end{aligned}$$

The **new** function allocates an initial single-shot channel and returns it as the session channel. The **send** function allocates a new single-shot channel, and sends it along the original channel with the given message v, after which the new channel is returned. The **recv** function receives the value and continuation channel pair using the original single-shot channel receive function. The **close** function sends a final termination flag, without allocating a new single-shot channel, to terminate the session. The **wait** function receives the final termination flag, which deallocates the channel.

For the channel to be used safely—*i.e.,* to not cause memory errors such as use-after-free or double-free—it is crucial that the channel endpoints are used in a dual way. That is, if there is a

send on one endpoint, there should be a matching receive on the other endpoint, and *vice versa*. Similarly, a **close** should match up with a **wait**.

We discuss other options for closing the channel in §5.

Example of using session channels. An example of using the session channels is as follows:

```
prog_add :=
  let c = new () in
  fork {let (l, c) = recv c in l ← (! l + 2); let c = send c () in wait c} ;
  let l = ref 40 in let c = send c l in let (_, c) = recv c in close c; assert(! l = 42)
```

Here, the main thread initially creates a session channel c, which is shared between the main thread and forked-off thread. The main thread dynamically allocates a reference to 40, after which it sends the reference over the channel. The service thread receives the reference, adds 2 to it, and sends a flag back, to signal that the reference has been updated. The main thread receives the flag and then reads the updated value stored in the reference, and asserts that it is 42. Finally, the main thread sends the closing signal, which is received by the forked-off thread. Each operation on the channel binds the channel continuation to an overshadowing name c, to intuitively capture that they keep working on the same session.

Similar to the example presented in §2.1, this program is safe if the **assert** succeeds and there are no memory errors due to improper use of the channel API. Intuitively, this example achieves safe access to the reference l via ownership delegation over the channel. We verify this in §3.4.

2.4 Imperative Channels

Although session channels are more convenient to use than single-shot channels, they still require us to continuously pass around new channel references. On top of session channels we therefore define *imperative* channels, which have a traditional imperative channel API:

new_imp () Create a new imperative channel, and return a *pair* of two endpoints c_1 and c_2.
c.**send**(v) Send message v on channel c. Return nothing.
c.**recv**() Receive a message from channel c. Return only the message.
c.**close**() Send termination message and close the channel.
c.**wait**() Wait for termination message and close the channel.

We implement imperative channels in terms of session channels by storing a session channel in a mutable reference:

$$\begin{aligned}
\textbf{new_imp}\,() &:= \textbf{let}\; c = \textbf{new}\,()\; \textbf{in}\; (\textbf{ref}\, c, \textbf{ref}\, c)\\
c.\textbf{send}(v) &:= c \leftarrow \textbf{send}\,(!\, c)\, v\\
c.\textbf{recv}() &:= \textbf{let}\; (v, c') = \textbf{recv}\; !\, c\; \textbf{in}\; c \leftarrow c'; v\\
c.\textbf{close}() &:= \textbf{close}\; (!\, c); \textbf{free}\, c\\
c.\textbf{wait}() &:= \textbf{wait}\; (!\, c); \textbf{free}\, c
\end{aligned}$$

2.5 Emerging Linked List Buffers

We demonstrate the imperative API with the example from Fig. 2. The example creates a channel to communicate between the main thread and the forked-off thread. The main thread allocates a reference s and sends the message $(100, s)$ to the forked-off thread, which indicates that the main thread is going to send 100 further number messages to the forked-off thread. The forked-off thread receives each of these numbers, and mutates s to keep track of their sum. Finally, the forked-off

```
let (c1, c2) = new_imp () in          — create channel between main and worker
fork {                                — start the worker thread
  let (n, s) = c2.recv() in           — receive count n and answer reference s
  for(i = 1..n) s ← c2.recv() + !s    — sum n received numbers
  c2.send(())                         — signal that we are done
  c2.wait()                           — wait for closing signal
}
let s = ref(0) in                     — mutable reference to store the sum
c1.send((100, s))                     — we will send 100 numbers to be summed into s
for(i = 1..100) c1.send(i)            — send the numbers 1..100
c1.recv()                             — wait until the worker is done
c1.close()                            — send closing signal
assert(!s == 5050)                    — assert that the received answer is correct
```

Fig. 2. An example program using the imperative channels.

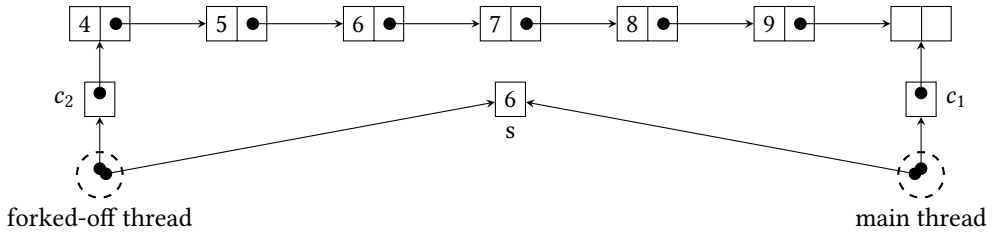

Fig. 3. The heap structure emerging from the example in Fig. 2, after the first 9 values $[1, \ldots, 9]$ have been sent, and the first 3 $[1, 2, 3]$ have been received and summed in the shared location s. The boxes with a number n and next pointer ℓ indicate that the memory location contains $\texttt{Some}(n, \ell)$, and the empty box on the right indicates that the memory location contains **None**.

thread sends an empty acknowledgment message () to the main thread, indicating that it is done with s and will not mutate s further. The main thread closes the session by sending the closing signal, which the forked-off thread waits for. The main thread then reads the value of the sum from s, and asserts that it is correctly computed.

The linked structures that emerge during execution are displayed in Fig. 3. In the picture, the main thread has sent the numbers $[1, \ldots, 9]$ in the forked-off thread has so far only received $[1, 2, 3]$. At run time, the forked-off thread will have a reference to c_2, which points to the head of a linked list structure. When the forked-off thread receives the next message (4), it updates c_2 to point to the next linked list element, and adds the value of the message to s. The main thread also has a reference to s, but it will not use it until the forked-off thread has sent the completion signal back, to avoid race conditions. Instead, the main thread is still busy working on the other end of the linked list. Each time the main thread sends a message, it allocates a new memory location, puts its message into the tail, and updates the tail of the existing linked list to point to the new location.

If the forked-off thread were to catch up with the main thread, it would wait until it sees a message. When the main thread is done, it tries to *receive* a message using the last linked list node it has created, which is initially still empty. When the client reaches that node, it puts the acknowledgment () into it, signaling that the main thread may now read s. More generally, the

threads switch roles when the polarity of the protocol changes: the thread that used to consume list cells now creates new list cells, and *vice versa*.

Note that the emergence of the buffer as a bi-directional linked list is somewhat implicit. We have built several layers of channels, but at no point did we have to think about the linked-list run-time structure as a whole. We will see a similar phenomenon when doing the proofs: we never need to think about the run-time structure as a whole. Instead, we will develop specifications in a layered way, following the layers of the implementation.

In the remainder of this paper, we will develop specifications for these different layers (corresponding to § 2.1 to 2.4), and prove the correctness of the channel implementations with respect to these specifications. We can then use the specifications to verify this example in §3.5.

3 LAYERED SPECIFICATIONS AND VERIFICATION

As the reader may have noticed, the implementations in the preceding section are *untyped*. Rather than assigning types to the channel APIs, we will provide *separation logic specifications*. These allow us to prove functional correctness of programs that make use of the channel API. We prove *partial correctness*, which guarantees that if a program satisfies a separation logic specification with trivial precondition, then the program is safe, *i.e.,* does not get stuck in the semantics due to run-time type errors, use-after-free or double-free bugs, or failing **`assert`** expressions. In terms of session types, our result should be compared with *type safety* and *session fidelity*. As is standard in papers that use Iris, we do not prove deadlock freedom or termination (which would only be true when assuming a fair scheduler as the spin-loop in **`recv1`** could otherwise trivially loop).

In this section we first present the Iris separation logic that we use to verify our implementation (§3.1). We then show how we verified the single-shot channel implementation using Iris primitives (§3.2), and layer subprotocols on top of it (§3.3). We verify dependent separation protocol [Hinrichsen et al. 2020, 2022] specifications of our session channel implementation directly on top of our single-shot specifications (§3.4). Finally, we verify our imperative channel implementation in terms of the session channel specifications (§3.5).

3.1 The Iris Separation Logic

To specify and verify the channel implementations and example clients, we use the Iris separation logic. Fig. 4 shows the grammar and a selection of rules of the subset of Iris that we use. Iris provides a program logic for HeapLang with Hoare-triples $\{P\}\ e\ \{\Phi\}$, which express that given the precondition ($P : \mathsf{iProp}$), the program ($e : \mathsf{Expr}$) is safe to execute, and yields the postcondition ($\Phi : \mathsf{Val} \to \mathsf{iProp}$). We often write $\{P\}\ e\ \{w.\ Q\} \triangleq \{P\}\ e\ \{\lambda w.\ Q\}$ and $\{P\}\ e\ \{Q\} \triangleq \{P\}\ e\ \{\lambda w.\ w = () * Q\}$.

Iris is a separation logic, meaning that propositions assert ownership over resources, such as references. This is made precise by the separation logic connectives, such as the separating conjunction $P * Q$, which describes that the propositions P and Q holds for *separate* parts of the heap. In particular, this lets us derive *exclusivity* of references; it is impossible to separately own the same reference: $\ell \mapsto v * \ell \mapsto w \mathrel{-\!\!*} \mathsf{False}$. Here $-\!\!*$ is the "separating implication" connective. It acts similarly to the regular implication, but for separation logic.

Separation logic facilitates *modular verification*, by virtue of the framing rule Ht-frame, which states that we can verify programs e in the presence of separate resources R. Non-structured concurrency is supported by the Ht-fork rule. Finally, Iris enjoys the conventional rules for mutable references Ht-alloc, Ht-load, Ht-store, and Ht-free, which respectively allow allocating, reading, updating, and freeing mutable references.

We use Iris's impredicative invariants $\boxed{P}$, ghost state tokens $\mathsf{tok}\,\gamma$, and later modality $\triangleright P$. We further discuss the meaning and importance of these throughout the section.

Iris propositions:

$$P, Q \in \mathsf{iProp} ::= \mathsf{True} \mid \mathsf{False} \mid P \land Q \mid P \lor Q \mid \quad \text{(Propositional logic)}$$
$$\forall x.\, P \mid \exists x.\, P \mid x = y \mid \quad \text{(Higher-order logic with equality)}$$
$$P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v \mid \{P\}\, e\, \{\Phi\} \mid \quad \text{(Separation logic)}$$
$$\boxed{P} \mid \mathsf{tok}\,\gamma \mid \triangleright P \mid \ldots \quad \text{(Invariants, ghost state, and step indexing)}$$

Separation logic:

$$\textsc{Ht-frame}\ \frac{\{P\}\, e\, \{w.\, Q\}}{\{P * R\}\, e\, \{w.\, Q * R\}} \qquad \textsc{Ht-val}\ \{\mathsf{True}\}\, v\, \{w.\, w = v\} \qquad \textsc{Ht-fork}\ \frac{\{P\}\, e\, \{\mathsf{True}\}}{\{P\}\ \mathbf{fork}\ \{e\}\ \{\mathsf{True}\}}$$

Heap manipulation:

$$\textsc{Ht-alloc}\ \{\mathsf{True}\}\ \mathbf{ref}\, v\, \{\ell.\, \ell \mapsto v\} \qquad \textsc{Ht-load}\ \{\ell \mapsto v\}\ !\ell\, \{w.\, (w = v) * \ell \mapsto v\}$$

$$\textsc{Ht-store}\ \{\ell \mapsto v\}\, \ell \leftarrow w\, \{\ell \mapsto w\} \qquad \textsc{Ht-free}\ \{\ell \mapsto v\}\ \mathbf{free}\, \ell\, \{\mathsf{True}\}$$

Invariants*, ghost state, and step indexing:

$$\textsc{Ht-inv-alloc}\ \frac{\{\boxed{P} * Q\}\, e\, \{\Phi\}}{\{\triangleright P * Q\}\, e\, \{\Phi\}} \qquad \textsc{Ht-inv-open}\ \frac{e \text{ is atomic} \qquad \{\triangleright P * Q\}\, e\, \{w.\, \triangleright P * R\}}{\{\boxed{P} * Q\}\, e\, \{w.\, R\}}$$

$$\textsc{Ht-later-frame}\ \frac{e \text{ is not a value} \qquad \{P\}\, e\, \{w.\, Q\}}{\{P * \triangleright R\}\, e\, \{w.\, Q * R\}} \qquad \textsc{Ht-ghost-alloc}\ \frac{\{P * \exists \gamma.\, \mathsf{tok}\,\gamma\}\, e\, \{\Phi\}}{\{P\}\, e\, \{\Phi\}} \qquad \textsc{Tok-excl}\ \frac{\mathsf{tok}\,\gamma * \mathsf{tok}\,\gamma}{\mathsf{False}}$$

Fig. 4. The grammar and a selection of rules of Iris.
*Iris uses *masks* to prevent opening the same invariant twice during a single step, as that is unsound [Jung et al. 2018]. We omit details about this mechanism because we only open at most one invariant at every step.

3.2 Single-Shot Channels

In Fig. 5 we show separation logic specifications for the single-shot channel implementation from §2.2. These specifications make use of *single-shot protocols* that describe the protocol for a single-shot channel. As a single-shot channel communicates a value, the protocol will carry a predicate describing which values are allowed to be communicated with that channel. Additionally, the protocol says whether we are allowed to send or receive. Therefore, we represent single-shot protocols as a pair (tag, Φ) where $tag \in \{\mathsf{Send}, \mathsf{Recv}\}$ and $\Phi \in \mathsf{Val} \to \mathsf{iProp}$. The predicate Φ is a separation logic predicate, so that protocols can express transfer of ownership.

To link protocols to actual channels, we have the *channel points-to* $c \overset{\text{base}}{\rightarrowtail} (tag, \Phi)$. The channel points-to provides unique ownership of one end of the channel and says that channel c satisfies protocol (tag, Φ). The channel points-to is analogous to the normal points-to $\ell \mapsto v$ of separation logic, in the sense that a points-to assertion is required to verify an invocation of a channel operation.

When we create a new channel using **new1** $()$, we may choose the protocol Φ, and we get *two* channel points-tos: $c \overset{\text{base}}{\rightarrowtail} (\mathsf{Send}, \Phi)$ and $c \overset{\text{base}}{\rightarrowtail} (\mathsf{Recv}, \Phi)$. As this will be useful later, we introduce

Protocols:	$p \in \mathsf{Prot} \triangleq \{\mathsf{Send}, \mathsf{Recv}\} \times (\mathsf{Val} \to \mathsf{iProp})$
Dual protocol:	$\overline{(\mathsf{Send}, \Phi)} \triangleq (\mathsf{Recv}, \Phi) \qquad \overline{(\mathsf{Recv}, \Phi)} \triangleq (\mathsf{Send}, \Phi)$
Channel points-to:	$c \overset{\text{base}}{\rightarrowtail} p \in \mathsf{iProp} \quad \text{where} \quad p \in \mathsf{Prot} \text{ and } c \in \mathsf{Val}$
Channel creation:	$\{\mathsf{True}\}\ \mathbf{new1}\ ()\ \{c.\ c \overset{\text{base}}{\rightarrowtail} p * c \overset{\text{base}}{\rightarrowtail} \overline{p}\}$
Send message:	$\{c \overset{\text{base}}{\rightarrowtail} (\mathsf{Send}, \Phi) * \Phi\ v\}\ \mathbf{send1}\ c\ v\ \{\mathsf{True}\}$
Receive message:	$\{c \overset{\text{base}}{\rightarrowtail} (\mathsf{Recv}, \Phi)\}\ \mathbf{recv1}\ c\ \{\Phi\}$

Fig. 5. Separation logic specifications for single-shot channels.

the *dual* function on protocols, given by $\overline{(\mathsf{Send}, \Phi)} \triangleq (\mathsf{Recv}, \Phi)$ and $\overline{(\mathsf{Recv}, \Phi)} \triangleq (\mathsf{Send}, \Phi)$. Using this definition, the specification for **new1** can be given as in Fig. 5.

Once we have the two channel-point-to predicates we may give one of them to another thread, and keep one of them in the current thread. This way we ensure that two threads use the protocol to agree on how the channel will be used.

We may then use the **send1** and **recv1** operations to perform the communication. The **send1** $c\ v$ operation requires ownership of $c \overset{\text{base}}{\rightarrowtail} (\mathsf{Send}, \Phi)$ as well as $\Phi\ v$ in its precondition. Dually, the **recv1** c operation requires ownership of $c \overset{\text{base}}{\rightarrowtail} (\mathsf{Recv}, \Phi)$ in its precondition. Its postcondition guarantees that **recv1** c returns a value v that satisfies $\Phi\ v$.

With these specifications we can verify the example from Fig. 2 presented in §2.2. We use the following protocol:

$$\Phi_{single} \triangleq (\mathsf{Send}, \lambda(v : \mathsf{Val}).\ \exists(\ell : \mathsf{Loc}).\ v = \ell * \ell \mapsto 42)$$

This protocol expresses that the exchanged value v is a location ℓ. We transfer the ownership of the exchanged reference ℓ along with the message. With this, we can symbolically apply the single-shot channel specifications, and finally assert that the value read from the received reference is 42.

Verifying the implementation with respect to the specification. We now prove that the single-shot channel implementation satisfies its specification. To do this, we *define* the channel points-to $c \overset{\text{base}}{\rightarrowtail} p$ in terms of Iris logic primitives (namely, ordinary points-to, ghost state and invariants). We then prove that the specifications for **new1**,**send1** and **recv1** follow from the rules of Iris. We first present the two key concepts from Iris needed for our proof: *ghost state* and *invariants*.

Ghost state. Ghost state is logical state that we can use to logically coordinate between parallel threads. Compared to the standard approach to ghost state in concurrency verification [Owicki and Gries 1976], ghost state in Iris is not part of the program text. It is introduced and manipulated solely in proofs. Just as the physical heap keeps track of the values of memory locations, Iris has a ghost heap that keeps track of the values of ghost locations. In our case we only need the very simplest form of ghost state: we need pure ownership over ghost heap locations; we do not need to store further information in the ghost locations. Given the ghost location γ, we have the ghost resource $\mathsf{tok}\ \gamma$, which is analogous to $\ell \mapsto ()$, *i.e.,* a location that points to a unit value. It may seem a bit puzzling that ghost locations that do not store any interesting contents can be helpful in a proof. The key is that ghost locations have the same *exclusivity* as memory locations. That is, we have the Tok-excl rule that says it is impossible to have ownership of two ghost locations with

the same name: $\mathsf{tok}\,\gamma * \mathsf{tok}\,\gamma \mathrel{-\!\!*} \mathsf{False}$. We shall see why this is useful in a moment. Finally, we can always allocate new pieces of ghost state, using the HT-GHOST-ALLOC rule.

Invariants. The points-to resource $\ell \mapsto v$ is an affine resource, and cannot be duplicated. This is a problem for verifying concurrent programs, where we would like to use the same memory location from multiple threads: when we fork off a child thread, we would like to keep ownership over the memory location in both the main thread and the child thread.

To solve this issue, concurrent separation logic has the notion of *invariants*. At any moment in the proof where we have ownership over $P \in \mathsf{iProp}$, we can choose to establish P as an invariant, denoted $\boxed{P} \in \mathsf{iProp}$. This is formally described by the HT-INV-ALLOC rule. The advantage of an invariant is that it can be freely duplicated, *i.e.*, $\boxed{P} \mathrel{-\!\!*} \boxed{P} * \boxed{P}$. In turn, we cannot directly access the P inside the invariant. Instead, we can only temporarily access it when the program takes an atomic step, such as a memory load $!\,\ell$ or store $\ell \leftarrow v$. After the atomic step has happened, we must immediately put P back into the invariant. This is formally described by the HT-INV-OPEN rule. The proposition P inside an invariant is typically a disjunction of several states, where the states may assert ownership over memory locations using $\ell \mapsto v$, and may assert that v has certain properties in that state. A state may also assert ownership over ghost resources.

Iris's invariants are *impredicative* [Svendsen and Birkedal 2014], which effectively lets us nest invariants inside of invariants, because $\boxed{P} \in \mathsf{iProp}$ for every $P \in \mathsf{iProp}$, including $P = \boxed{Q}$. Nesting of invariants is critical for the verification of our session channels, as will be covered in §3.4. To maintain soundness of the Iris logic, resources P extracted from an invariant $\boxed{P}$ are guarded by a *later modality* $\triangleright P$ [Nakano 2000; Appel et al. 2007]. This later can be seen in the HT-INV-OPEN rule. Resources guarded by a later modality can only be used after the program does the next step of execution. This is formally expressed by the HT-LATER-FRAME rule, which states that one can frame resources under a later, if the program has not terminated. Another means of stripping laters is if the guarded resources are *timeless*. The connectives for reference ownership $\ell \mapsto v$ and ghost ownership $\mathsf{tok}\,\gamma$ are timeless, which we leverage in the verification, as explained momentarily.

The single-shot channel invariant. To verify the single-shot channels, we need to define the connective $c \rightarrowtail^{\mathsf{base}} p$, whose key ingredient is an invariant. To explain the invariant, we start with a key observation. The single-shot channel can be in three different states: (1) no message has been sent ($\ell \mapsto \mathbf{None}$), (2) a message has been sent but not received ($\ell \mapsto \mathbf{Some}\,v$), and (3) the message has been both sent and received (ℓ has been deallocated). These states are reflected in the invariant $\mathsf{chan\_inv}\ \gamma_1\ \gamma_2\ \ell\ \Phi$ defined in Fig. 6. The arguments γ_1 and γ_2 are two ghost locations, whereas ℓ is the physical memory location where the channel is located, and Φ is the predicate associated with the protocol. The invariant captures each state with a separate disjunct. By virtue of the exclusion of the ghost resources, it is then possible to exclude possible states, based on local ghost ownership. In particular, if one owns $\mathsf{tok}\,\gamma_1$, the invariant must be in the first state (as the other states asserts ownership of the resource). Similarly, if one owns $\mathsf{tok}\,\gamma_2$, the invariant cannot be in the final state. The proof then follows by letting the sender own $\mathsf{tok}\,\gamma_1$ and the receiver own $\mathsf{tok}\,\gamma_2$, to let them locally determine which state the invariant is in, by the exclusivity rule of the ghost resources.

More formally, with the invariant in place, we can define the channel points-to $c \rightarrowtail^{\mathsf{base}} (tag, \Phi)$, as presented in Fig. 6. The definition captures (1) that c is a reference ($c = \ell$), (2) that the invariant is established ($\boxed{\mathsf{chan\_inv}\ \gamma_1\ \gamma_2\ \ell\ \Phi}$), (3) that the endpoint has ownership of either $\mathsf{tok}\,\gamma_1$ or $\mathsf{tok}\,\gamma_2$, if they are the sender or receiver, respectively. The later modalities $\triangleright$ are used to make $c \rightarrowtail^{\mathsf{base}} p$ *contractive* in p, which we will use in §4 to construct infinite protocols.

When the sender wants to send their message v, they temporarily open the invariant using the HT-INV-OPEN rule, and determine that they are in the first state, based on their $\mathsf{tok}\,\gamma_1$ token. They

$$\mathsf{chan\_inv}\ \gamma_1\ \gamma_2\ \ell\ \Phi \triangleq \underbrace{(\ell \mapsto \mathbf{None})}_{\text{(1) initial state}} \vee \underbrace{(\exists v.\ \ell \mapsto \mathbf{Some}\, v * \mathsf{tok}\,\gamma_1 * \Phi\, v)}_{\text{(2) message sent, but not yet received}} \vee \underbrace{(\mathsf{tok}\,\gamma_1 * \mathsf{tok}\,\gamma_2)}_{\text{(3) final state}}.$$

$$c \rightarrowtail^{\text{base}} (tag, \Phi) \triangleq \exists \gamma_1, \gamma_2, \ell.\ \triangleright(c = \ell) * \boxed{\mathsf{chan\_inv}\ \gamma_1\ \gamma_2\ \ell\ \Phi} * \triangleright \begin{cases} \mathsf{tok}\,\gamma_1 & \text{if } tag = \mathsf{Send} \\ \mathsf{tok}\,\gamma_2 & \text{if } tag = \mathsf{Recv} \end{cases}$$

Fig. 6. The channel invariant and channel points-to definition.

then get ownership over the reference $\ell \mapsto \mathbf{None}$. The sender then modifies the location to contain the sent value $\mathbf{Some}\, v$, and transfers the ownership back into the invariant. The sender also puts the token $\mathsf{tok}\,\gamma_1$ into the invariant, as well as the resources $\Phi\, v$ captured by the protocol. The invariant is restored in the second state.

When the receiver wants to receive, it temporarily opens up the invariant, using the Ht-inv-open rule, to get ownership over the reference. It reads the location, and if the value is **None**, it determines that it is in the first state, and so it loops. Once a value $\mathbf{Some}\, v$ is read, it is determined that we are in the second state, and so the receiver deallocates the reference. The receiver additionally takes the Φv resource out of the invariant, and re-establishes the invariant by putting its token $\mathsf{tok}\,\gamma_2$ into the invariant, which restores it in the third state.

The rule for **new1** is then proven as follows. We obtain ownership over the location $\ell \mapsto \mathbf{None}$ because **new1** allocates the reference. We also allocate two new ghost locations $\mathsf{tok}\,\gamma_1$ and $\mathsf{tok}\,\gamma_2$ obtaining the identifiers γ_1 and γ_2. We establish the invariant using the first disjunct, by putting $\ell \mapsto \mathbf{None}$ into the invariant, and allocate it with the Ht-inv-alloc rule. We then duplicate the invariant, and create $c \rightarrowtail^{\text{base}} (\mathsf{Send}, \Phi)$ and $c \rightarrowtail^{\text{base}} (\mathsf{Recv}, \Phi)$ using the two copies of the invariant, as well as $\mathsf{tok}\,\gamma_1$ and $\mathsf{tok}\,\gamma_2$, respectively.

Recall that the resources obtained from an invariant are guarded by a later modality $\triangleright$, which requires that we take a step to use them (Ht-later-frame). However, we can strip the later from the reference ownership ($\ell \mapsto -$) and the ghost locations ($\mathsf{tok}\,\gamma_1$ and $\mathsf{tok}\,\gamma_2$) as they are timeless. Even so, the protocol predicate Φ is arbitrary, so it may contain nested invariants (*i.e.,* as we crucially rely on in §3.4 to verify session channels), and so it will be guarded by a later when the receiver takes it out of the invariant. Luckily, we take a step when resolving the receive operation, which lets us strip the later, before returning it to the user.

3.3 Subprotocols

We define a subprotocol relation on dependent separation protocols as introduced by Actris [Hinrichsen et al. 2022], analogous to subtyping on session types [Gay and Hole 2005]. Whereas subtyping between session types is established by subtyping between the messages, the subprotocol relation between protocols is established by implications between the separation logic predicates.[1]

[1]Similarly to asynchronous subtyping [Mostrous et al. 2009; Mostrous and Yoshida 2015], the Actris subprotocols also enjoy *asynchronous subtyping*, which allow swapping sends in front of receives. Actris supports this due to its two-buffer semantics. As the semantics of our session channels, as built on single shot channels, corresponds to a single-buffer semantics, asynchronous subtyping is *unsound* in our setting. Our notion of subprotocols therefore focuses on the implication between separation logic predicates, and does not allow swapping sends in front of receives.

Protocols: $(!\, x\, \langle v \rangle \{P\}.\, p \mid ?x\, \langle v \rangle \{P\}.\, p \mid \mathbf{end}^{!} \mid \mathbf{end}^{?}) \in \mathsf{Prot}$

Dual protocol: $\overline{!\, x\, \langle v \rangle \{P\}.\, p} = ?x\, \langle v \rangle \{P\}.\, \overline{p} \qquad \overline{?x\, \langle v \rangle \{P\}.\, p} = !\, x\, \langle v \rangle \{P\}.\, \overline{p}$

$\overline{\mathbf{end}^{!}} = \mathbf{end}^{?} \qquad \overline{\mathbf{end}^{?}} = \mathbf{end}^{!} \qquad \overline{\overline{p}} = p$

Channel creation: $\{\mathsf{True}\}\ \mathtt{new}\ ()\ \{c.\, c * c \rightarrowtail p * c \rightarrowtail \overline{p}\}$

Send message: $\{c \rightarrowtail (!\, x\, \langle v \rangle \{P\}.\, p) * P\ t\}\ \mathtt{send}\ c\ (v\ t)\ \{c'.\, c' \rightarrowtail (p\ t)\}$

Receive message: $\{c \rightarrowtail (?x\, \langle v \rangle \{P\}.\, p)\}\ \mathtt{recv}\ c\ \{w.\, \exists y, c'.\, w = (v\ y, c') * c' \rightarrowtail (p\ y) * P\ y\}$

Close operation: $\{c \rightarrowtail \mathbf{end}^{!}\}\ \mathtt{close}\ c\ \{\mathsf{True}\}$

Wait operation: $\{c \rightarrowtail \mathbf{end}^{?}\}\ \mathtt{wait}\ c\ \{\mathsf{True}\}$

Fig. 7. Dependent Separation Protocols and session channel specifications

The subprotocol relation is denoted $p \sqsubseteq q$ where p, q are protocols, and is defined as follows:

$$(tag_1, \Phi_1) \sqsubseteq (tag_2, \Phi_2) \triangleq \begin{cases} \forall v.\, \Phi_2\ v \mathrel{-\!\!*} \Phi_1\ v & \text{if } tag_1 = tag_2 = \mathsf{Send} \\ \forall v.\, \Phi_1\ v \mathrel{-\!\!*} \Phi_2\ v & \text{if } tag_1 = tag_2 = \mathsf{Recv} \\ \mathsf{False} & \text{if } tag_1 \neq tag_2 \end{cases}$$

This relation is reflexive and transitive, and $p \sqsubseteq q$ iff $\overline{q} \sqsubseteq \overline{p}$. We layer subprotocols on top of our specification for single-shot channels by defining a new channel points-to $c \rightarrowtail p$ that is explicitly closed under subprotocols:

$$c \rightarrowtail p \triangleq \exists q.\, \triangleright(q \sqsubseteq p) * c \overset{\text{base}}{\rightarrowtail} q$$

We do not use a superscript on $c \rightarrowtail p$ because we consider it to be the main channel points-to, whereas we view $c \overset{\text{base}}{\rightarrowtail} q$ as an internal notion. The later modality $\triangleright$ is again used to make $c \rightarrowtail p$ *contractive* in p, which we will use in § 4 to construct infinite protocols. This channel points-to satisfies $(c \rightarrowtail p) * \triangleright(p \sqsubseteq q) \mathrel{-\!\!*} (c \rightarrowtail q)$, by transitivity of $\sqsubseteq$ and $P \mathrel{-\!\!*} \triangleright P$.

We can prove versions of the specifications for **new1**, **send1**, and **recv1** for $\rightarrowtail$. These proofs are straightforward, because we can prove these specifications *using* the existing specifications for $\overset{\text{base}}{\rightarrowtail}$ from Fig. 5, by using $p \sqsubseteq q$ at appropriate points to convert a $\Phi_1\ v$ into $\Phi_2\ v$ or *vice versa*. In particular, we apply this conversion in the send rule just before sending the message, and in the receive rule just after receiving the message. We also trivially have $(c \overset{\text{base}}{\rightarrowtail} p) \mathrel{-\!\!*} (c \rightarrowtail p)$, which is used to prove the **new1** rule for $\rightarrowtail$.

3.4 Session Channels

Now that we have established the specifications for the single-shot channels, we move on to the next layer: multi-shot session channels. A popular ever-expanding approach to specifying and verifying such multi-shot channels is the concept of *session types* [Honda et al. 2008], which lets a user ascribe session channel endpoints with a sequence of obligations to send or receive messages of certain types. More recently, the session type approach has been adopted in the separation logic setting [Craciun et al. 2015; Hinrichsen et al. 2022]. One such adaptation is *Dependent Separation Protocols* [Hinrichsen et al. 2022]. Rather than ascribing types to each exchange, dependent separation protocols ascribe logical variables, physical values, and propositions. The dependent separation protocols and the specifications for the session channels can be seen in Fig. 7.

The dependent separation protocols consists of four constructors: $!\, x\, \langle v \rangle \{P\}.\, p$, $?x\, \langle v \rangle \{P\}.\, p$, $\mathbf{end}^!$, and $\mathbf{end}^?$. The first two constructors describe the permission to send or receive the logical variable x, the value v, and the resources P, respectively, after which they follow the protocol tail p. Here, x binds into all of the remaining constituents. We often omit the binder when it is of the unit type: *e.g.,* $!\, \langle v \rangle \{P\}.\, p$. We similarly often omit the proposition if it is True: *e.g.,* $!\, x\, \langle v \rangle.\, p$. The last two constructors specify that the protocol has ended, meaning that no further operations can be made on the channel, and it the channel can be closed. We further detail alternative specifications for closing and deallocation in § 5.

The protocols are subject to the same notion of *duality*, as presented in § 3.2. The dual of a protocol is the same sequence of obligations, where the polarity has been flipped, *i.e.,* all sends (!) become receives (?), and *vice versa*, as made precise by the rules of the figure. Finally, we use the same channel endpoint ownership $c \rightarrowtail p$ as for the single-shot channels, as the dependent separation protocols share the same type as the single-shot protocols, as will be seen momentarily.

The dependent separation protocols can be used to specify and verify session channels. As an example, the following dependent separation protocol specifies the interactions of the `prog_add` example from § 2.3:

$$\texttt{prot\_add} \triangleq !\, ((\ell, x) : \mathsf{Loc} \times \mathbb{Z})\, \langle \ell \rangle \{\ell \mapsto x\}.\, ?\langle () \rangle \{\ell \mapsto x + 2\}.\, \mathbf{end}^!$$

The protocol captures that one must first send a reference to a number (captured by the logical variable $(\ell, x) : \mathsf{Loc} \times \mathbb{Z}$)), along with the ownership of the reference $\ell \mapsto x$. Afterwards, the updated reference can be reacquried, followed by the protocol termination. The dual of the protocol is $?((\ell, x) : \mathsf{Loc} \times \mathbb{Z})\, \langle \ell \rangle \{\ell \mapsto x\}.\, !\, \langle () \rangle \{\ell \mapsto x + 2\}.\, \mathbf{end}^?$.

The notion of duality is used in the specification for **`new`**. The specification states that we obtain separate exclusive ownership of the returned endpoint c, one with a freely picked protocol p and the other with its dual $\overline{p}$. This mimics the intuition from the single-shot channel, in which one endpoint had to release the specified resources, while the other could acquire them. The specification for **`send`** states that in order to send, the channel endpoint must have a sending protocol, and we must give up the specified resources $P\ t$, for a specific instantiation t of the variable x. Additionally, the sent value must correspond to the protocol, for the variable instantiation $v\ t$. As a result, the returned channel endpoint follows the protocol tail $c' \rightarrowtail p\ t$, for the same variable instantiation. Conversely, the specification for **`recv`** states that we can receive if the channel endpoint has a receiving protocol. As a result we obtain an instance of the logical variable y, and the resources specified by the protocol $Q\ y$. Additionally, the returned value is exactly the one specified by the protocol $v\ y$, and the new endpoint follows the protocol tail $c' \rightarrowtail p\ y$. Given these specifications the `prog_add` example can be verified by picking the `prot_add` protocol along with symbolic execution.

Verification of the session channel specifications. The definitions of the dependent separation protocols and the specification rules presented in Fig. 7 are derived directly on top of the single-shot channel definitions and specifications. In particular, the type of dependent separation protocols is the same as the one for the single-shot channel protocols, namely Prot. The definition of the sending protocol is as follows:

$$\begin{aligned}&\texttt{send\_prot}\ (\tau : \mathsf{Type})\ (v : \tau \to v)\ (P : \tau \to \mathsf{iProp})\ (p : \tau \to \mathsf{iProto}) : \mathsf{iProto} \triangleq \\ &\quad (\mathsf{Send}, \lambda(r : v).\, \exists (x : \tau), (c : v).\, r = (v\ x, c) * P\ x * c \rightarrowtail p\ x)\end{aligned}$$

$$!\, (x : \tau)\, \langle v \rangle \{P\}.\, p \triangleq \texttt{send\_prot}\ \tau\ (\lambda x.\, v)\ (\lambda x.\, P)\ (\lambda x.\, p)$$

The `send_prot` constructor takes four arguments, and constructs a sending single-shot channel protocol. In particular the constructor takes the type of its logical variable τ, the exchanged value v, the exchanged proposition P, and the protocol tail p. The latter three arguments all abstract over

the protocol variable, which is existentially quantified in the protocol body. The second projection captures that the actual exchanged value is a tuple of the value specified by the protocol ($v\ x$), and the continuation (c). It additionally includes ownership of the resources specified by the protocol ($P\ x$), and finally a single-shot channel ownership, of the continuation with the protocol tail ($c \rightarrowtail (p\ x)$). The notation $!\,(x : \tau)\ \langle v \rangle \{P\}.\ p$ then simply lets us instantiate the sending constructor, without explicitly repeating the variable abstraction for the three constituents.

The duality function of the session channels is the same as the one for the single-shot channel. We define the receive constructor in terms of the sending one, using the duality function as follows:

$$?x\ \langle v \rangle \{P\}.\ p \triangleq \overline{!\,x\ \langle v \rangle \{P\}.\ \overline{p}}$$

To specify the the **`close`** and **`wait`** operations we define two session protocols:

$$\mathbf{end}^{!} \triangleq (\mathsf{Send}, \lambda r.\ r = ())$$

$$\mathbf{end}^{?} \triangleq \overline{\mathbf{end}^{!}}$$

Finally, the channel endpoint ownership $c \mapsto p$ is identical to the one for the single-shot channels, as the type of the protocols are the same, they simply carry channel continuations now. This immediate reuse of the single-shot ownership is made possible by the higher-order nature of Iris. In particular, the internal invariant of the endpoint ownership refers to the session protocols, which internally includes a nested endpoint ownership, and so on. By virtue of the step-indexing of Iris, this is sound as we always take a step for each unfolding of the nested invariants.

With these definitions the soundness of the session channel specifications (Fig. 7) follow almost immediately from the sound specifications of the single-shot channel operations **`send1`** and **`recv1`**.

Subprotocols for session protocols. We have a notion of subprotocols for single-shot protocols (§3.3), but what about dependent session protocols? Because we have *defined* session protocols as particular forms of single-shot protocols, we get the appropriate notion of subprotocols for session protocols for free. The following lemmas for session subprotocols (and the imperative derivation on top of them) are *already true* and easily derived from the subprotocol rules in §3.3:

$$\frac{\forall x_1.\ \Phi_1\ x_1 \mathrel{-\!\!*} \exists x_2.\ (v_1\ x_1 = v_2\ x_2) * \Phi_2\ x_2 * \triangleright (p_1\ x_1 \sqsubseteq p_2\ x_2)}{?x_1\ \langle v_1 \rangle \{\Phi_1\}.\ p_1 \sqsubseteq ?x_2\ \langle v_2 \rangle \{\Phi_2\}.\ p_2}$$

$$\frac{\forall x_2.\ \Phi_2\ x_2 \mathrel{-\!\!*} \exists x_1.\ (v_2\ x_2 = v_1\ x_1) * \Phi_1\ x_1 * \triangleright (p_1\ x_1 \sqsubseteq p_2\ x_2)}{!\,x_1\ \langle v_1 \rangle \{\Phi_1\}.\ p_1 \sqsubseteq !\,x_2\ \langle v_2 \rangle \{\Phi_2\}.\ p_2}$$

At a high level, these lemmas state that a session protocol is a subprotocol of another, if for each logical message in the first protocol, there exists an appropriate logical message in the second protocol, such that we have a separating implication between separation logic assertions, and the tails of the protocols are in a subprotocol relationship. The stated lemmas are somewhat stronger than this high-level description; for instance, the user of the lemmas gets access to the assertion $\Phi_1\ x_1$ *before* having to provide the corresponding logical message x_2 for the other protocol. As an example, this strengthening allows one to perform a form of *framing* of resources within a protocol: if a resource is provided by an earlier send and needed by a later receive, we can frame these two resources (*i.e.,* remove both from the protocol by canceling them out). This property can be illustrated by the following rules:

$$!\,x\ \langle v \rangle \{P\}.\ ?x\ \langle w \rangle \{Q\}.\ p \sqsubseteq !\,x\ \langle v \rangle \{P * R\}.\ ?x\ \langle w \rangle \{Q * R\}.\ p$$

$$?x\ \langle v \rangle \{P * R\}.\ !\,x\ \langle w \rangle \{Q * R\}.\ p \sqsubseteq ?x\ \langle v \rangle \{P\}.\ !\,x\ \langle w \rangle \{Q\}.\ p$$

Channel points-to:	$c \rightarrowtail^{\text{imp}} p \in \text{iProp} \quad \text{where} \quad p \in \text{Prot and } c \in \text{Val}$
Channel creation:	$\{\text{True}\}\ \texttt{new\_imp}\ ()\ \{w.\ \exists c_1, c_2.\ w = (c_1, c_2) * c_1 \rightarrowtail^{\text{imp}} p * c_2 \rightarrowtail^{\text{imp}} \overline{p}\}$
Send message:	$\{c \rightarrowtail^{\text{imp}} (!\, x\ \langle v \rangle \{P\}.\ p) * P\ t\}\ c.\texttt{send}(v\ t)\ \{c \rightarrowtail^{\text{imp}} (p\ t)\}$
Receive message:	$\{c \rightarrowtail^{\text{imp}} (?x\ \langle v \rangle \{P\}.\ p)\}\ c.\texttt{recv}()\ \{w.\ \exists y.\ w = v\ y * c \rightarrowtail^{\text{imp}} (p\ y) * P\ y\}$
Close operation:	$\{c \rightarrowtail^{\text{imp}} \mathbf{end}^{!}\}\ c.\texttt{close}()\ \{\text{True}\}$
Wait operation:	$\{c \rightarrowtail^{\text{imp}} \mathbf{end}^{?}\}\ c.\texttt{wait}()\ \{\text{True}\}$

Fig. 8. Separation logic specifications for imperative channels.

3.5 Imperative Channels

Because our session channels create new pointers at each step, they return new channels, and are thus inconvenient to work with. For that reason, we have our final layer: the imperative channels from §2.4. These channels put a session channel in a mutable reference, so that we can use the same mutable reference throughout and use mutating operations to change the reference to a new session channel upon send and receive operations.

To handle these channels, we introduce a new channel points-to $c \rightarrowtail^{\text{imp}} p$. The specifications for the imperative channels can be found in Fig. 8. We note a couple of differences with respect to the session channels:

- The **new_imp** operation returns a pair of channels now, so the points-to connectives in the postcondition are for the two components of the pair.
- The **send** operation does not return a value. The new channel points-to in the postcondition refers to the original channel instead.
- The **recv** operation only returns one value—the message. The channel points-to in the postcondition once again refers to the original channel.

Verifying the imperative channel specifications. To verify the session channels we first define a new connective for channel endpoint ownership:

$$c \rightarrowtail^{\text{imp}} p \triangleq \exists(\ell : \text{Loc}), (c' : \text{Val}).\ c = \ell * \ell \mapsto c' * c' \rightarrowtail p$$

The new imperative channel ownership connective $c \rightarrowtail^{\text{imp}} p$ simply lifts the original connective $c' \rightarrowtail p$ to assert ownership of a mutable reference.

With this definition in hand, verifying the specification is trivial. We simply use the Iris rule for allocating, reading, and updating the reference, along with the specifications for the original channel endpoint ownership, to resolve the operations on the channel.

Verifying the example. We now explain how these specifications can be used to verify the example from §2.4. The example starts by allocating a new channel, so we use the specification for **new_imp**. In order to use this specification, we have to choose the session protocol p. We use the following protocol:

$$\texttt{prot\_sum}'\ x\ n \triangleq \textbf{if}\ n = 0\ \textbf{then}\ (?\langle () \rangle \{s \mapsto x\}.\ \mathbf{end}^{!})\ \textbf{else}\ (!\, (y : \mathbb{N})\ \langle y \rangle.\ \texttt{prot\_sum}'\ (x + y)\ n)$$
$$\texttt{prot\_sum} \triangleq !\, ((n, s) : \mathbb{N} \times \text{Loc})\ \langle (n, s) \rangle \{s \mapsto 0\}.\ \texttt{prot\_sum}'\ 0\ n$$

The protocol `prot_sum` says that we will first send the pair (n, s) of a number and a location, and the assertion that $s \mapsto 0$. We then continue with the protocol `prot_sum'` $0\ n$, which is recursively

defined. Its first argument keeps track of the sum of the messages sent so far, and the second argument keeps track of how many messages we still have to send. When the counter $n = 0$, we stop sending and instead receive a unit value, as well as the assertion that $s \mapsto x$, *i.e.,* the sum of the messages sent.

After the channel allocation, we have $c_1 \rightarrowtail^{\text{imp}} \texttt{prot\_sum}$ and $c_2 \rightarrowtail^{\text{imp}} \overline{\texttt{prot\_sum}}$. We verify the first interaction using the first step of `prot_sum`. We prove the loops correct using induction: the main thread does induction on 100, and the child thread induction on the received message n (which will be 100, but the child thread does not know this). After the final synchronization, the ownership over s has been transferred back to the main thread. According to the protocol, the location s points to the value $1 + 2 + \cdots + 100$, which we can verify to be equal to 5050 by mathematical reasoning.

As the reader can see, the reasoning about the pointer structure of the buffers is completely encapsulated in the higher-level session specifications. The nondeterminism present due to the asynchronous semantics of the send operation does not need to be reasoned about explicitly: although the depth of the linked list buffer changes non-deterministically according to the thread scheduling of the sends and receives, the proof does not explicitly reason about this at all.

4 GUARDED RECURSION

As we have seen in the example in §3.4, we can already create some recursive protocols by employing recursion over natural numbers (or other inductively defined data types in Coq). Recursion over natural numbers lets us verify the example from Fig. 2 where one side sends a number n, and then sends n further messages. Although recursion on inductive types is powerful, it does not allow us to create protocols for truly infinite interactions with services that run forever. We can create protocols that support truly infinite interactions with Iris's operator for *guarded recursion*.

Iris models guarded recursion via *step-indexing* [Appel and McAllester 2001; Ahmed 2004], meaning that separation logic propositions iProp are internally monotone predicates of a natural number i, the step index. Intuitively, the meaning of such a proposition is given by taking the limit to ever higher step indices. This allows us to model infinite protocols as a step-indexed protocol of unboundedly increasing depth. Iris does not expose the step index to the user of the logic, so we cannot define protocols by direct recursion over i. Instead, Iris provides a *logical* account of step-indexing [Appel et al. 2007; Dreyer et al. 2011] through the later modality $\triangleright P$ [Nakano 2000], and a guarded recursion operator $\mu x.F\ x$ for constructing recursive predicates . The $F\ x$ must be *contractive* in the sense that recursive usages of F must only occur under a later $\triangleright$. This ensures that creating such a recursive predicate does not result in any logical paradoxes. Our protocols $\mathsf{Prot} \triangleq (\mathsf{Send} \mid \mathsf{Recv}) \times (\mathsf{Val} \to \mathsf{iProp})$ contain separation logic predicates over values, so we can make direct use of Iris's guarded recursion mechanism to define recursive protocols.

The reader may have noticed that we have already inserted the later modality $\triangleright$ in certain places in our definitions, such as in the definition of $c \rightarrowtail^{\text{base}} p$ §3.2. This is to make sure that $c \rightarrowtail^{\text{base}} p$ is contractive in p, which in turn means that $!\, x\, \langle v \rangle \{P\}.\, p$ and $?x\, \langle v \rangle \{P\}.\, p$ are contractive in p. We are therefore able to take guarded fixed points of protocols, to create unbounded or infinite protocols, such as the following recursive variant of `prot_add`:

$$\texttt{prot\_add\_rec} \triangleq \mu p.\, !\, ((\ell, x) : \mathsf{Loc} \times \mathbb{Z})\, \langle \ell \rangle \{\ell \mapsto x\}.\, ?\langle () \rangle \{\ell \mapsto x + 2\}.\, p$$

A second component of guarded recursion is Iris's support for Löb induction. Löb induction allows us to verify unbounded or infinitely recursive programs that *use* recursive protocols. Ordinary induction only gives us an induction hypothesis for recursive calls where some measure is decreasing, and hence only works for terminating loops. Löb induction, on the other hand, gives us an induction hypothesis for *any* recursive call (not necessarily decreasing), but this induction hypothesis will be guarded under a later ($\triangleright$). These laters maintain logical consistency, but the resources guarded by

them may only be accessed after the next primitive program step. In this manner, Löb induction allows us to verify partial correctness of of a program that sends a stream of messages in an infinite tail-recursive loop, by instantiating the channel with the preceding recursive protocol.

The recursive protocols combined with Löb induction allow us to verify recursive programs such as the following recursive variant of the `prog_add` program from §2.3:

```
prog_add_rec :=
  let c = new () in
  fork {(rec f c = let (l, c) = recv c in l ← (! l + 2); let c = send c () in f c) c};
  let l = ref 38 in
  let c = send c l in let (_, c) = recv c in
  let c = send c l in let (_, c) = recv c in
  assert(! l = 42); c
```

Here **rec** f $x = e$ is a recursive function, where the recursive occurence is bound to f. Verifying the program is straightforward. Notably, the main thread unfolds the recursive protocol *prot_add_rec* twice, to verify its code. The forked-off thread is resolved using Löb induction. It unfolds the recursive protocol once, verifies one iteration, after which it uses the Löb induction hypothesis to verify the recursive call.

5 SELF-DUAL END

In the preceding sections, we had separate **close** and **wait** operations, with dual $\mathbf{end}^!$ and $\mathbf{end}^?$ protocols. In this section we investigate alternative operations to deallocate or close a channel, which result in a self-dual **end** protocol. We have two different options for achieving this:

- **Symmetric close.** Define one close operation, with protocol **end**, that both sides call, which dynamically determines who deallocates the channel (§5.1)
- **Send-close.** Define a combined send-close operation that sends the last message and closes the channel. The other side performs a **recv** that obtains no continuation channel (§5.2).

5.1 Symmetric Close

Suppose that we want only one **sym_close** operation, that both sides of the channel call. Because the channel consists of one memory location, we need to dynamically decide which caller gets to free the memory. We use compare-and-swap to achieve this effect:

$$\textbf{sym\_close}\ c \triangleq \textbf{if}\ \textbf{CAS}(c, \textbf{None}, \textbf{Some}())\ \textbf{then}\ ()\ \textbf{else}\ \textbf{free}\ c$$

To see how this works, consider two parallel close operations on the same channel: **sym_close** c ‖ **sym_close** c. The thread that does its **CAS** first will successfully set c from **None** to **Some**(), and return () from its **sym_close**. The second thread will then *fail* its **CAS**, since the value stored in c is no longer **None**. It will then go to the else branch and free c.

To verify this version of close, we need to make a change to our notion of protocols. So far, our protocols have all been single-shot protocols $p \in \mathsf{Prot} \triangleq (\mathsf{Send} \mid \mathsf{Recv}) \times (\mathsf{Val} \to \mathsf{iProp})$ under the hood; even the protocols $\mathbf{end}^!, \mathbf{end}^? \in \mathsf{Prot}$. For the symmetric **sym_close**, this does not work. We now have to explicitly distinguish **end** in the protocols:

$$q \in \mathsf{Prot}_{\mathsf{end}} ::= \mathbf{end} \mid p \qquad \text{where } p \in \mathsf{Prot}$$

We also need to extend duality with $\overline{\mathbf{end}} \triangleq \mathbf{end}$ and subprotocols with $\mathbf{end} \sqsubseteq \mathbf{end}$. With this additional protocol, we have the following specification for **sym_close**:

$$\textbf{Close operation:} \qquad \{c \overset{\text{sym}}{\rightarrowtail} \mathbf{end}\}\ \textbf{sym\_close}\ c\ \{\mathsf{True}\}$$

Because our set of protocols has been extended, we need an extended channel points-to $\overset{\text{sym}}{\rightarrowtail}$, which we define as follows:

$$c \overset{\text{sym}}{\rightarrowtail} q \triangleq \begin{cases} \exists \gamma_1, \gamma_2, \ell.\ \triangleright(c = \ell) * \boxed{\mathsf{end\_inv}\ \gamma_1\ \gamma_2\ \ell} * \triangleright(\mathsf{tok}\,\gamma_1) & \text{if } q = \mathbf{end} \\ c \rightarrowtail q & \text{if } q \in \mathsf{Prot} \end{cases}$$

Here, the following protocol is stored inside our invariant:

$$\mathsf{end\_inv}\ \gamma_1\ \gamma_2\ \ell \triangleq \underbrace{(\ell \mapsto \mathbf{None})}_{\text{before close}} \vee \underbrace{(\ell \mapsto \mathbf{Some}() * (\mathsf{tok}\,\gamma_1 \vee \mathsf{tok}\,\gamma_2))}_{\text{one side has closed}} \vee \underbrace{(\mathsf{tok}\,\gamma_1 * \mathsf{tok}\,\gamma_2)}_{\text{fully closed}}$$

Like the single-shot send-receive protocol, this protocol uses two tokens $\mathsf{tok}\,\gamma_1$ and $\mathsf{tok}\,\gamma_2$, which belong to the two $c \overset{\text{sym}}{\rightarrowtail}$ **end** assertions. Initially, the invariant states that the location ℓ points to **None**. When one side has successfully closed, the invariant states that ℓ points to **Some**(), and the invariant has collected the token of the side that has called close first (because this is nondeterministic, the invariant uses a disjunction $\mathsf{tok}\,\gamma_1 \vee \mathsf{tok}\,\gamma_2$). When both sides have closed, the invariant has both tokens, and no memory points-to (because the memory location has been deallocated). Similarly to the definition of $c \overset{\text{base}}{\rightarrowtail} p$, we add laters in front of $c = \ell$ and $\mathsf{tok}\,\gamma_1$ to make the definition contractive, thereby enabling infinite protocols. With these definitions, we can prove the Hoare specification for the symmetric **sym_close** in a similar way we verified **send1** and **recv1**.

5.2 Send-Close

From an operational point of view, the previous two methods for channel closing are a tiny bit disappointing, because for the last step, a memory location is allocated but not used to communicate any useful message. In this section we develop a channel closing mechanism where the close operation is integrated with the last message send.

This may sound strange at first sight, but upon investigating how channel closing typically works in examples, it hopefully starts to make more sense. Consider an example where party A is communicating a stream of messages to another party B, and A may at every point decide to end the stream. This can be accomplished by sending an additional Boolean along with each message, which determines whether this is the last message or not. When it is the last message, the sender *does not* allocate a continuation channel, and sends () in place of the continuation channel. When the receiver receives a message, they have to inspect the Boolean to determine whether they got a continuation channel or not. This saves one memory allocation and synchronization compared to the previous methods.

While this saving is minor, we argue in favor of it for aesthetic reasons. If one wants to implement the single-shot API on top of the previous session channel API (*i.e.,* the other way around compared to what we have done so far), then a single shot communication would involve one real communication and then one extra allocation and communication to close the channel. In this section we present a channel closing mechanism with which one can implement single-shot channels on top of session channels with no additional synchronizations or allocations. Therefore, with this channel closing mechanism, session channels become a *purely logical* layer over single-shot channels. The implementation of this closing mechanism is very simple, namely the following **send_close** operation:

$$\mathbf{send\_close}\ c\ v \triangleq \mathbf{send1}\ c\ (v, ())$$

There is no corresponding **wait** operation for the other side: as **send_close** simply does not allocate a continuation channel, the other side can use **recv**, which already deallocates the memory

location. For the specification and verification of **send_close**, we use the same $\mathsf{Prot}_{\mathsf{end}}$ protocols:

$$q \in \mathsf{Prot}_{\mathsf{end}} ::= \mathbf{end} \mid p \qquad \text{where } p \in \mathsf{Prot}$$

and extend duality with $\overline{\mathbf{end}} \triangleq \mathbf{end}$ and subprotocols with $\mathbf{end} \sqsubseteq \mathbf{end}$. As before, we define a new channel points-to, this time for the send-close version:

$$c \rightarrowtail^{\mathsf{scl}} q \triangleq \begin{cases} c = () & \text{if } q = \mathbf{end} \\ c \rightarrowtail q & \text{if } q \in \mathsf{Prot} \end{cases}$$

For the **end** protocol, the channel points-to asserts that there is no channel, i.e., the channel is a unit value instead of a pointer to a memory location (this could also be implemented as a null pointer).

These are the specifications for the channel operations with **send_close**:

Channel creation: $\{\mathsf{True}\}\ \mathbf{new}\ ()\ \{c.\ c \rightarrowtail^{\mathsf{scl}} p * c \rightarrowtail^{\mathsf{scl}} \overline{p}\}$

Send message: $\{c \rightarrowtail^{\mathsf{scl}} (!\, x\ \langle v \rangle \{P\}.\ p) * P\ t\}\ \mathbf{send}\ c\ (v\ t)\ \{c'.\ c' \rightarrowtail^{\mathsf{scl}} p\ t\}$ where $p\ t \neq \mathbf{end}$

Send-close: $\{c \rightarrowtail^{\mathsf{scl}} (!\, x\ \langle v \rangle \{P\}.\ p) * P\ t\}\ \mathbf{send\_close}\ c\ (v\ t)\ \{\mathsf{True}\}$ where $p\ t = \mathbf{end}$

Receive message: $\{c \rightarrowtail^{\mathsf{scl}} (?x\ \langle v \rangle \{P\}.\ p)\}\ \mathbf{recv}\ c\ \{w.\ \exists y, c'.\ w = (v\ y, c') * c' \rightarrowtail^{\mathsf{scl}} p\ y * P\ y\}$

The **send** operation now requires that the tail $p\ t$ is *not* **end**, whereas the **send_close** operation requires that $p\ t$ *is* **end**. The receive spec does not concern itself with **end**. Instead, the received message $v\ y$ will contain information about whether the protocol ended or not (such as a Boolean, as described previously). Using logical reasoning about the message, we can then conclude whether the tail protocol $p\ t$ is **end** or not. If it is, then we obtain $c' = ()$, and we do not need to do anything. If it is not **end**, we obtain $c' \rightarrowtail^{\mathsf{scl}} p\ y$ and can continue the protocol.

Unlike **close** with symmetric channel closing from §5.1, the **send_close** operation has been defined in terms of **send1**. The proofs of the specifications therefore also follow straightforwardly from the specifications of **send1** and **recv1**, unlike the proofs for symmetric channel closing.

6 MECHANIZATION

The implementations of channels (§2), the proof that they satisfy their separation logic specifications (§3), the different methods for closing channels (§5), and the verification of all the examples have been fully mechanized using the Coq proof assistant [Coq Team 2021], making use of the Iris separation logic framework.

The mechanization follows the layered design as presented in Fig. 1. The layered design allows our proofs to be simpler compared to previous work on Actris [Hinrichsen et al. 2020]. Only the proofs for single-shot operations **send1**, **recv1** (and the symmetric **sym_close**) involve concurrent separation logic concepts such as ghost state and invariants. All the other proofs are done on top of these specifications, treating the single-shot operations as a black box.

Our protocol definitions are simple compared to Actris we do not need to solve an intricate recursive domain equation [Hinrichsen et al. 2022, §9.7]. At no point do we have to reason about more than one cell in the buffer structure; the multi-shot session protocols simply emerge automatically using composition. Despite this simplification to the Actris model, the different extensions such as subprotocols, guarded recursion, and the different forms of channel closing work seamlessly together. For instance, we can show that an infinitely recursive protocol is a subprotocol of another infinitely recursive protocol, by using guarded recursion and Löb induction.

In total, our Coq mechanization consists of less than 1000 lines of Coq code (including the verification of all examples, counting newlines and comments).

7 RELATED WORK

The origins of our line of work trace back to session types. More directly, our work is inspired by encodings of session types in terms of single-shot synchronization in particular [Kobayashi 2002; Dardha et al. 2017; Jacobs 2022]. Our work is also directly related to dependent protocols and program logics for session protocols. Most notable is the work on Actris [Hinrichsen et al. 2020, 2022], which introduced the notion of *dependent separation protocols*, which we use to specify our session channels. We go over each of these points in more detail below.

Single-shot channels. The encoding of session channels in terms of sequenced single-shot channels originated in the π-calculus. This encoding sends a continuation channel in each message, so that the communication can continue. Kobayashi [2002] showed that session types can be encoded into π-types, and Dardha et al. [2012, 2017] later extended Kobayashi [2002]'s approach. Jacobs [2022] presented a bidirectional version in a λ-calculus.

Similar single-shot primitives have also been used in the implementation of message passing libraries, such as in the work of Scalas and Yoshida [2016]; Padovani [2017]; Kokke and Dardha [2021]; Niehren et al. [2005]. Our implementation of session channels in terms of single-shot channels uses a similar strategy.

Unlike this earlier work, which is either untyped or type-based, we use session protocols in separation logic to verify (partial) functional correctness. Our single-shot channels are not primitive and not built-in to the language, but implemented in terms of low-level memory operations. We take inspiration from the preceding work and subsequently build session channels on top of single-shot channels, and we build session protocols on top of single-shot protocols.

Dependent protocols and session logics. Bocchi et al. [2010] and Toninho et al. [2011] both developed version of (multi-party) session types which incorporate logical binders into the protocols, alongside a first-order decidable assertion language for specifying properties about them. Later, Toninho and Yoshida [2018] and Thiemann and Vasconcelos [2020] expanded on this work by allowing similar binders determine the structure of the remaining protocol, similar to what we do in § 2.4. Compared to our work, the assertion languages are limited in the sense that they cannot describe the delegation of resources (*e.g.,* sending a reference to another thread). Later work [Craciun et al. 2015; Costea et al. 2018] addressed the issue of specifying resource delegation, through the development of a *session logic*, based in separation logic. Their logic allows ascribing channel endpoints with protocols, which in turn can specify resources the be shared, such as other channel endpoints. Compared to our work, they do not support binders, which for one means that they cannot specify protocols referring the dynamically allocated references, like we do in § 2.2. Actris protocols support both binders, delegation, and protocols referring to dynamically allocated references and ghost resources [Hinrichsen et al. 2020], as our protocols do.

Actris. Actris introduced a shared-memory implementation of higher-order session channels, and the notion of dependent separation protocols for the verification of message passing concurrency using program logics, mechanized on top of Iris. Our work focuses primarily on developing a framework in the style of Actris, but with a focus on *layered design*, *elegance*, and *simplicity*. This results in the following key differences between Actris and our work:

- Actris channels implement bi-directional communication using a pair of buffers that are protected by a lock. Our single-shot channels are implemented directly using load and store memory operations, and our session channels and imperative channels are implemented in terms of single-shot channels.
- As a result of this, Actris's dependent separation protocols are defined by solving an intricate recursive domain equation. By contrast, our definition of Prot $\triangleq$ (Send | Recv) $\times$ (Val $\rightarrow$

iProp) is itself non-recursive, yet Actris-style dependent separation protocols can be *defined* as inhabitants of Prot, and automatically support recursive protocols.

- Our notion of subprotocols for single-shot channels is very simple and non-recursive, but automatically lifts to (recursive) session protocols, because session protocols are defined as single-shot protocols. Actris's notion of subprotocols is recursive and more complicated than ours, but also stronger: Actris's implementation of channels with a pair of buffers admits swapping sends over receives (akin to asynchronous subtyping [Mostrous et al. 2009; Mostrous and Yoshida 2015]). Such a transformation is not sound for our single-buffer implementation of channels.
- We achieve a simpler approach by making use of nested invariants, but Actris's solution gave rise the "Actris Ghost Theory" [Hinrichsen et al. 2022, §9.4] for reasoning about session protocols in a way that is disconnected from specific implementations.
- Actris contains a number of convenience features, such as multi-binders and associated tactics, to ease verification of message passing programs in Coq. While such features can be integrated in our Coq development, we preferred to keep the protocols (and verification thereof) simpler, to focus on the layering of channel variants. Even so, our single-binders can simulate multi-binders using tuples, as has been demonstrated throughout the paper.
- While Actris relies on a garbage collector for channel deallocation, we present several manually memory managed solutions for channel closing.

In short, Actris has more features (asynchronous subtyping, ghost theory) and a more convenient implementation in Coq (multi-binders, tactics), but our design achieves the key feature of Actris (dependent separation protocols) in a conceptually simpler and layered manner: once we have defined and verified single-shot channels (which are quite simple and require only the simplest form of ghost resources to verify), we treat them as a black box and develop Actris-style protocols with relative ease and without any further use of ghost state or invariants.

An application of Actris is the verification of the soundness of a session type system via the method of semantic typing [Hinrichsen et al. 2021]. Since the our separation logic specifications for session channels are the same as Actris's, a similar result could be achieved with our development.

Imperative session channels. Related to §2.4, there has also been work on *type systems* for imperative channels, which free the user from having to thread channel variables through their program Saffrich and Thiemann [2022b,a]. The advantage of a type system compared to a program logic is that type checking is automatic, but an advantage of a program logic is its ability to verify functional correctness, and its ability to verify safety of more intricate usage patterns.

REFERENCES

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

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.

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *TOPLAS* 23, 5 (2001), 657–683.

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*. 109–122. https://doi.org/10.1145/1190216.1190235

Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *CONCUR*. 162–176.

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR*. 16–34.

David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: A DSL for Certified Multiparty Computation: From Mechanised Metatheory to Certified Multiparty Processes. In *PLDI*. 237–251. https://doi.org/10.1145/3453483.3454041

David Castro-Perez, Francisco Ferreira, and Nobuko Yoshida. 2020. EMTST: Engineering the Meta-theory of Session Types. In *TACAS (2) (LNCS, Vol. 12079)*. 278–285. https://doi.org/10.1007/978-3-030-45237-7_17

Luca Ciccone and Luca Padovani. 2020. A Dependently Typed Linear π-Calculus in Agda. In *PPDP*. 8:1–8:14. https://doi.org/10.1145/3414080.3414109

The Coq Team. 2021. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.4501022

Andreea Costea, Wei-Ngan Chin, Shengchao Qin, and Florin Craciun. 2018. Automated Modular Verification for Relaxed Communication Protocols. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 284–305. https://doi.org/10.1007/978-3-030-02768-1_16

Florin Craciun, Tibor Kiss, and Andreea Costea. 2015. Towards a Session Logic for Communication Protocols. In *20th International Conference on Engineering of Complex Computer Systems, ICECCS 2015, Gold Coast, Australia, December 9-12, 2015*. IEEE Computer Society, 140–149. https://doi.org/10.1109/ICECCS.2015.33

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session types revisited. In *PPDP'12*. https://doi.org/10.1145/2370776.2370794

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Inf. Comput.* 256 (2017), 253–286. https://doi.org/10.1016/j.ic.2017.06.002

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical step-indexed logical relations. *LMCS* 7, 2 (2011).

Simon J. Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2-3 (2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. 2020. Duality of Session Types: The Final Cut. In *PLACES (EPTCS, Vol. 314)*. 23–33. https://doi.org/10.4204/EPTCS.314.3

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

Matthew A. Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. 2016. An Extensible Approach to Session Polymorphism. *MSCS* 26, 3 (2016), 465–509. https://doi.org/10.1017/S0960129514000231

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-Type Based Reasoning in Separation Logic. *PACMPL* 4, POPL, Article 6 (Dec. 2020), 30 pages. https://doi.org/10.1145/3371074

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *Log. Methods Comput. Sci.* 18, 2 (2022). https://doi.org/10.46298/lmcs-18(2:16)2022

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

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 715)*, Eike Best (Ed.). Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1381)*, Chris Hankin (Ed.). Springer, 122–138. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 273–284. https://doi.org/10.1145/1328438.1328472

Jules Jacobs. 2022. A Self-Dual Distillation of Session Types. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*. 23:1–23:22. https://doi.org/10.4230/LIPIcs.ECOOP.2022.23

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–33. https://doi.org/10.1145/3498662

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 256–269. https://doi.org/10.1145/2951913.2951943

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. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

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 *Proceedings of the 42nd Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP*, Vol. 74. 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Naoki Kobayashi. 2002. Type Systems for Concurrent Programs *(Lecture Notes in Computer Science, Vol. 2757)*. 439–453. https://doi.org/10.1007/978-3-540-40007-3_26

Wen Kokke and Ornela Dardha. 2021. Deadlock-Free Session Types in Linear Haskell *(Haskell 2021)*. https://doi.org/10.1145/3471874.3472979

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

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. 205–217. https://doi.org/10.1145/3009837.3009855

Étienne Lozes and Jules Villard. 2012. Shared Contract-Obedient Endpoints. In *ICE*. 17–31.

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

Dimitris Mostrous and Nobuko Yoshida. 2015. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.* 241 (2015), 227–263. https://doi.org/10.1016/j.ic.2015.02.002

Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 316–332. https://doi.org/10.1007/978-3-642-00590-9_23

Hiroshi Nakano. 2000. A modality for recursion. In *LICS*. 255–266. https://doi.org/10.1109/LICS.2000.855774

Joachim Niehren, Jan Schwinghammer, and Gert Smolka. 2005. A Concurrent Lambda Calculus with Futures *(Lecture Notes in Computer Science, Vol. 3717)*. 248–263. https://doi.org/10.1007/11559306_14

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR*. 49–67.

Susan S. Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *CACM* 19, 5 (1976), 279–285. https://doi.org/10.1145/360051.360224

Luca Padovani. 2017. A simple library implementation of binary sessions. *J. Funct. Program.* 27 (2017), e4. https://doi.org/10.1017/S0956796816000289

Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *FoSSaCS (LNCS, Vol. 9034)*. 3–22. https://doi.org/10.1007/978-3-662-46678-0_1

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-Typed Definitional Interpreters for Linear, Session-Typed Languages. In *CPP*. 284–298. https://doi.org/10.1145/3372885.3373818

Hannes Saffrich and Peter Thiemann. 2022a. Polymorphic Typestate for Session Types. *CoRR* abs/2210.17335 (2022). https://doi.org/10.48550/arXiv.2210.17335

Hannes Saffrich and Peter Thiemann. 2022b. Relating Functional and Imperative Session Types. *Log. Methods Comput. Sci.* 18, 3 (2022). https://doi.org/10.46298/lmcs-18(3:33)2022

Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*. 21:1–21:28. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *POPL* (2019), 30:1–30:29. https://doi.org/10.1145/3290343

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *ESOP (LNCS, Vol. 8410)*. 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

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

Peter Thiemann. 2019. Intrinsically-Typed Mechanized Semantics for Session Types. In *PPDP*. 19:1–19:15. https://doi.org/10.1145/3354166.3354184

Peter Thiemann and Vasco T. Vasconcelos. 2020. Label-dependent session types. *Proc. ACM Program. Lang.* 4, POPL (2020), 67:1–67:29. https://doi.org/10.1145/3371135

Bernardo Toninho. 2015. *A Logical Foundation for Session-Based Concurrent Computation.* Ph.D. Dissertation. Carnegie Mellon University and New University of Lisbon.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, Peter Schneider-Kamp and Michael Hanus (Eds.). ACM, 161–172. https://doi.org/10.1145/2003476.2003499

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *ESOP (LNCS, Vol. 7792)*. 350–369. https://doi.org/10.1007/978-3-642-37036-6_20

Bernardo Toninho and Nobuko Yoshida. 2018. Depending on Session-Typed Processes. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10803)*, Christel Baier and Ugo Dal Lago (Eds.). Springer, 128–145. https://doi.org/10.1007/978-3-319-89366-2_7

Philip Wadler. 2012. Propositions as Sessions. In *ICFP*. 273–286. https://doi.org/10.1145/2364527.2364568

Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 148:1–148:30. https://doi.org/10.1145/3428216