Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing

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

We introduce a linear concurrent separation logic, called **LinearActris**, designed to guarantee deadlock and leak freedom for message-passing concurrency. LinearActris combines the strengths of session types and concurrent separation logic, allowing for the verification of challenging higher-order programs with mutable state through dependent protocols. The key challenge is to prove the adequacy theorem of LinearActris, which says that the logic indeed gives deadlock and leak freedom "for free" from linearity. We prove this theorem by defining a step-indexed model of separation logic, based on *connectivity graphs*. To demonstrate the expressive power of LinearActris, we prove soundness of a higher-order (GV-style) session type system using the technique of logical relations. All our results and examples have been mechanized in Coq.

CCS Concepts: • **Theory of computation** → **Separation logic**; *Program verification*; Programming logic.

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

ACM Reference Format:
Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. *Proc. ACM Program. Lang.* 8, POPL, Article 47 (January 2024), 33 pages. https://doi.org/10.1145/3632889

1 INTRODUCTION

Session type systems [Honda 1993; Honda et al. 1998] allow type checking programs that involve message-passing concurrency. Session types are protocols, which can be seen as sequences of send (**!**) and receive (**?**) actions. They are associated with channels, and express in what order messages of what type should be transferred. For example, the session type **!**$\mathbb{Z}$.**?**$\mathbb{B}$.**end** is given to a channel over which an integer should be sent, after which a boolean is received. More complex session types can be formed with operators for choice ($\oplus$,&), recursion (μ), *etc.*

Aside from ensuring type safety, linear session type systems [Caires and Pfenning 2010; Wadler 2012] can ensure deadlock freedom. That means that well-typed programs cannot end up in a state where all threads are waiting to receive a message from another. Deadlock freedom has been extended to large variety of session type systems [Carbone and Debois 2010; Fowler et al. 2021; Toninho et al. 2013; Toninho 2015; Caires et al. 2013; Pérez et al. 2014; Lindley and Morris 2015, 2016, 2017; Fowler et al. 2019; Das et al. 2018]. The elegance of session type systems is that they give deadlock freedom essentially "for free"—it is obtained from "just" linear type checking. Moreover, session types are compositional—once functions have been type checked, they can be composed by merely establishing that the types agree. A final strength of session types is that deadlock freedom

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.

ACM 2475-1421/2024/1-ART47
https://doi.org/10.1145/3632889

is maintained in a higher-order setting where closures and channels are transferred as first-class data over channels. *The goal of this paper is to extend these advantages to separation logic.*

Linear session types are unique and different from other methods for deadlock freedom—such as lock orders [Dijkstra 1971; Leino et al. 2010; Hamin and Jacobs 2018; Balzer et al. 2019; D'Osualdo et al. 2021], priorities [Kobayashi 1997; Padovani 2014; Dardha and Gay 2018], and global multiparty session types [Honda et al. 2008, 2016]—because they do not require any additional proof obligations involving orders, priority annotations, or global types. Still, other methods neither supersede nor subsume session types in the range of programs they can prove to be deadlock free (§9.1).

The ideas of session types are not limited to type checking, but have previously also been applied to functional verification. Bocchi et al. [2010]; Francalanza et al. [2011]; Lozes and Villard [2012]; Craciun et al. [2015]; Oortwijn et al. [2016]; Hinrichsen et al. [2020, 2022] have developed program logics that incorporate concepts from session types to verify increasingly sophisticated programs with message-passing concurrency. The protocols of these program logics make it possible to put logical conditions on the messages, allowing one to specify the contents (*e.g.,* the message is an even number) instead of just the shape (*e.g.,* it is an integer). The state of the art is the Actris logic and its descendants [Hinrichsen et al. 2020, 2022; Jacobs et al. 2023b], which are embedded in the Iris framework for concurrent separation logic in Coq [Jung et al. 2015, 2016; Krebbers et al. 2017a; Jung et al. 2018b]. The key ingredient of Actris is its notion of *dependent separation protocols*, which inspired by dependent session types, can express dependencies between the data of messages and specify the transfer of resources. For example, the protocol $\mathbf{!}(\ell : \mathsf{Loc}, n : \mathbb{N})\langle \ell \rangle\{\ell \mapsto n\};\ \mathbf{?}\langle n \rangle\{\ell \mapsto (n+1)\};\ \mathbf{end}$ says that a location ℓ with value n should be sent, after which the value n should be received, and the value of ℓ has been incremented.

Since Actris is a full-blown program logic, instead of a type system that aims to have decidable type checking, it can express more protocols and therefore verify safety of more programs than session types. In particular, it can express protocols where the shape (*e.g.,* number of messages) of the protocol depends on the contents of earlier messages. Moreover, Hinrichsen et al. [2021] show that Actris can be used to give a semantic model to prove soundness of (affine) session types using the technique of logical relations in Iris [Timany et al. 2022].

A key ingredient of concurrent separation logics such as Iris (on top of which Actris is built)—and also other separation logic frameworks such as VST [Appel 2014], CFML [Charguéraud 2020], and Bedrock [Chlipala 2013]—is their *adequacy* (a.k.a. soundness) theorem that connects the program logic to the operational semantics. For Iris, the adequacy theorem is [Jung et al. 2018b, §6.4]:

Theorem 1.1. *A proof of* $\{\mathsf{True}\}\ e\ \{\mathsf{True}\}$ *implies that* e *is* ***safe****, i.e., if* $([e], \emptyset) \to_{\mathrm{t}}^{*} ([e_1 \ldots e_n], h)$, *then for each* i *either* e_i *is a value or* (e_i, h) *can step.*

Intuitively this theorem says that the logic is doing its job: a verified program e "cannot go wrong", *i.e.,* it cannot perform illegal operations such as loading from a dangling location (use after free) or use an operator with wrong arguments (*e.g.,* $3 + \lambda x.x$). Formally it says that if e can be verified (*i.e.,* a Hoare triple with trivial precondition can be proved), and the initial configuration $([e], \emptyset)$ (consisting of a single thread e and the empty heap) steps to $([e_1 \ldots e_n], h)$ (consisting of threads $e_1 \ldots e_n$ and heap h), then each thread e_i has either finished (is a value) or can make further progress (perform a step). Illegal operations cannot step, so adequacy guarantees they do not occur.

Despite the strong trust that the adequacy theorem gives in the correctness of the program logic—especially when mechanized in a proof assistant such as Coq—the adequacy theorem of most state-of-the-art program logics says nothing about deadlocks. In Iris, blocking operations (*e.g.,* receiving from a channel whose buffer is empty, or acquiring a lock that has already been acquired) are modeled as busy loops, and thus cannot be distinguished from non-terminating programs. Both deadlocking and non-terminating programs can always step, and are thus trivially safe.

Goal of the paper. Our goal is to build a program logic for partial correctness that (1) enjoys an adequacy theorem that guarantees deadlock freedom for message passing concurrency, (2) combines the strengths of session types and concurrent separation logic to obtain deadlock freedom "for free" from linearity, without any additional proof obligations, and (3) is strong enough to verify challenging programs. By partial correctness we mean that we formalize deadlock freedom as a safety property (it is impossible for all threads to be waiting on a blocking operation), rather than a liveness property (threads are guaranteed to make progress or terminate). Formalizing deadlock freedom as a safety property is similar to the standard property of *global progress* in session type systems [Caires and Pfenning 2010] (§ 10 contains a discussion of safety versus liveness).

Before discussing the desiderata of the program logic, let us investigate the operational semantics and adequacy theorem. To distinguish between deadlock and non-termination, we let the receive operation on a channel block the thread until a message is sent, instead of letting it perform a busy loop. With that change at hand, the adequacy theorem becomes similar to the global progress theorem of session type systems:

Theorem 1.2. *A proof of* $\{\mathsf{Emp}\}\ e\ \{\mathsf{Emp}\}$ *implies that* e *enjoys* ***global progress****, i.e., if* $([e], \emptyset) \to_{\mathrm{t}}^{*} ([e_1 \ldots e_n], h)$*, then either* e_i *is a value for each* i *and* $h = \emptyset$*, or* $([e_1 \ldots e_n], h)$ *can step.*

Instead of requiring each thread to step, which would be false if a thread is genuinely waiting for another thread, we require the configuration as a whole to step. This means that there is always at least one thread that can step, *i.e.,* there is no global deadlock. Additionally, compared to the adequacy theorem for safety, we require the final heap to be empty, which means all channels have been deallocated, *i.e.,* there are no memory leaks. (Note that global progress does not subsume safety, we still need a theorem that ensures the absence of illegal non-blocking operations.)

Our desired adequacy theorem does *not* hold for Iris-based logics such as Actris:

- **The need for linearity.** Iris and Actris are *affine*, which means that resources must be used at most once, but can also be dropped (Iris satisfies the proof rule $P * Q \vdash P$, or equivalently $\mathsf{Emp} \dashv\vdash \mathsf{True}$). Hence one can verify a program that creates a channel with endpoints c_1 and c_2, have one thread perform a receive, and let the other thread perform a no-op:

  Thread 1: $c_1.\mathbf{recv}()$
  Thread 2: do nothing

  This program can be verified in Iris/Actris because using affinity, the ownership of c_2 can be dropped in the second thread. However, this program causes a deadlock: due to the absence of a send, the receive will block indefinitely. In session types this form of deadlock is ruled out by making the system *linear*, which means that resources must be used exactly once, and cannot be dropped until the protocol has been completed.
- **The need for acyclicity** Linearity alone is not enough. If a thread could obtain ownership of both endpoints of a single channel, then it would be able to trivially deadlock itself, by performing the receive before the send. Linearity would not be violated, as the thread would still consume both channel ownership assertions according to the rules of the logic, but the thread would be blocked forever. More generally, if two threads own the endpoints of two channels, and perform a receive followed by send, there would be a deadlock:

  Thread 1: $c_1.\mathbf{recv}();\ d_1.\mathbf{send}(2)$
  Thread 2: $d_2.\mathbf{recv}();\ c_2.\mathbf{send}(1)$

  In session types, Wadler [2012] addresses this problem by combining thread and channel creation into a single construct. Together with linearity, this ensures that channel ownership is *acyclic* in a certain sense, and rules out all deadlocks without need for annotations.

In this paper we introduce **LinearActris**—which amends Actris with the aforementioned restrictions from linear session types to satisfy the goals we stated above. The key challenge that we address in the remainder of the introduction is proving the adequacy theorem of LinearActris.

Key challenge: Proving adequacy. Adequacy is commonly proved by giving a semantic interpretation of propositions and Hoare triples. For sequential separation logic [O'Hearn et al. 2001], propositions are modeled as heap predicates, and the semantics of Hoare triples is defined so that safety and leak freedom follow almost by definition. Since we consider a higher-order program logic, for a concurrent language with dynamic thread and channel spawning, and wish to prove global progress, this simple setup no longer suffices. We should address the following challenges:

- **Circular semantics.** Session types and dependent separation protocols of Actris are higher-order, which means they can specify programs that transfer channels and closures over channels. In Actris one can write $d \rightarrowtail \mathbf{!}(c : \mathrm{Loc})\langle c\rangle\{c \rightarrowtail p\};\ \mathbf{end}$ to say that d is a channel, over which a channel c with protocol p is sent. Here, the protocol p can contain protocol ownership assertions $c \rightarrowtail p'$, where p' can in turn contain protocol ownership assertions. This circularity involves a negative recursive occurrence and cannot be solved in set theory. It is similar to the type-world circularity in models of type systems with higher-order references [Ahmed 2004; Birkedal et al. 2011], and that of storable locks [Hobor et al. 2008] and impredicative invariants [Svendsen and Birkedal 2014], where step-indexing [Appel and McAllester 2001] is used to solve the circularity. The original Actris makes (in part) use of Iris's impredicative invariant mechanism to avoid solving this circularity explicitly.
- **Invariants and linearity.** Iris's invariants are strongly tied to the logic being affine. Jung [2020, Thm 2] presents a paradox showing that a naïve linear version of Iris's invariants cannot be used to obtain even leak-freedom. Bizjak et al. [2019] present Iron, a linear version of Iris with an invariant mechanism that can be used to prove leak-freedom. Aside from not considering deadlock freedom, Iron avoids Jung's paradox by restricting the contents of invariants—namely, invariants cannot contain permissions to deallocate resources. Ownership of the **end** protocol needs to provide permission to deallocate the channel, making Iron's invariants insufficient for our purpose.
- **Invariants and acyclicity.** Linearity alone is not enough to avoid deadlocks—one needs to maintain an invariant that the channel ownership topology is acyclic. Formalizing this acyclicity invariant is a key challenge of the syntactic meta theory of session types [Lindley and Morris 2015, 2016; Fowler et al. 2021; Jacobs et al. 2022a; Jacobs 2022]. Since this prior work is aimed at syntactic theory of type systems, we need to investigate how to incorporate acyclicity of the topology into a semantic model of a program logic. Additionally, in type systems there is a 1-to-1 correspondence between physical references and ownership, but not in program logics. One can create protocols such as $\mathbf{!}(c : \mathrm{Loc})\langle c\rangle\{c \rightarrowtail p\};\ \mathbf{?}\langle()\rangle\{c \rightarrowtail p'\};\ \mathbf{end}$ where a channel reference and ownership is sent, and only an acknowledgment $()$ is returned. This means that the sending thread has to keep a reference to the channel, although it cannot use it before it has received the acknowledgment.

We define a step-indexed linear model of separation logic as the solution of a recursive domain equation [America and Rutten 1989; Birkedal et al. 2010]. To avoid reasoning about step-indices, we work in the pure step-indexed logic with a later modality ($\triangleright$) [Appel et al. 2007; Nakano 2000].

Similar to Iris we define Hoare triples in terms of weakest preconditions. A major difference in the definition of the weakest precondition compared to Iris is that we thread through the weakest preconditions of all threads, as well as the ownership and duality invariants of all channels. This way we can ensure that at all times the threads and channels form an acyclic topology with respect to channel ownership. To formalize acyclicity we use adapt notion of *connectivity graphs* by Jacobs

$$
\begin{array}{rll}
e \in \mathsf{Expr} ::= & x \mid e\ e \mid \lambda x.e \mid (e, e) \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathbf{rec}\ \mathsf{f}\ x.\ e \mid n \mid e + e \mid \cdots & \\
\mid & \mathbf{match}\ e\ \mathbf{with}\ \mathsf{inl}\ x \Rightarrow e;\ \mathsf{inr}\ x \Rightarrow e\ \mathbf{end} \mid \mathbf{let}\ (x_1, x_2) = e\ \mathbf{in}\ e & \\
\mid & \mathbf{fork1}\ e \mid e.\mathbf{send}(e) \mid e.\mathbf{recv}() \mid e.\mathbf{close}() \mid e.\mathbf{wait}() & \text{(Channel operations)} \\
\mid & \mathbf{ref}\ e \mid\ !\,e \mid e \leftarrow e \mid \mathbf{free}\ e \mid \mathbf{assert}\ e & \text{(Heap operations \& assert)}
\end{array}
$$

Fig. 1. The syntax of ChanLang.

et al. [2022a]. Connectivity graphs were originally designed for the syntactic meta theory of type systems for deadlock freedom, but we adapt them so that they can be integrated into a step-indexed separation logic. To simplify the construction of the model and the operational semantics of the language, we base ourselves on the work of Dardha et al. [2012]; Jacobs et al. [2023b]: we use one-shot channels as primitive, and build multi-shot channels on top of those.

Contributions. We introduce **LinearActris**—a concurrent separation logic for proving deadlock- and leak freedom of message-passing programs, essentially offering these guarantees "for free" from linearity, without any additional proof obligations. This involves the following contributions:

- We verify a range of examples of that use channels, closures, and mutable references as first-class data, demonstrating the expressive power of LinearActris (§2).
- We provide a formal description of the proof rules of LinearActris. First for multi-shot channels, and then for one-shot channels. Based on Jacobs et al. [2023b], we derive the logic for multi-shot channels from the one for one-shot channels (§ 3 and 4).
- We provide a formal adequacy proof of LinearActris based on a step-indexed model of separation logic rooted in connectivity graphs [Jacobs et al. 2022a], showing that a derivation in LinearActris ensures deadlock and leak freedom of the program in question (§ 5 and 6).
- We use LinearActris to construct a logical relations model that establishes deadlock freedom for a session-typed language (§7). This contribution has two purposes. First, this provides an application that truly relies on our connectivity based approach to deadlock freedom (and is out of scope for logics based on *e.g.,* lock orders). Second, this shows that every program that can be shown to be deadlock free using a GV-style session type system, can also be shown to be deadlock free using LinearActris. In fact, we go beyond existing GV-like systems by supporting the combination of recursive types, subtyping, term- and session type polymorphism, and unique mutable references (§7).
- We mechanized all our results as well as a number of examples from the original Actris papers in Coq (§8). See Jacobs et al. [2023a] for the sources.

We conclude the paper with related work (§9) and discussion/future work (§10).

2 LINEAR ACTRIS BY EXAMPLE

In this section we present LinearActris with example programs that we verify. We deliberately use very small examples. In our Coq mechanization we show that LinearActris can also be used to prove deadlock freedom of more challenging examples from the Actris papers, in particular, a number of increasingly complicated versions of parallel merge sort (§8).

The programming language that we use in LinearActris is called ChanLang. It has concurrency, bidirectional message passing channels, mutable references, and functional programming constructs (such as lambdas, products, sums, and recursion). The syntax is shown in Fig. 1. ChanLang has the following constructs for message-passing concurrency:

fork $(\lambda c.\, e)$	Fork a new thread that runs e with a new channel a location c, and also return the location of the new channel.
c.**send**(v)	Send message v over the channel at location c.
c.**recv**$()$	Receive a message over the channel at location c.
c.**close**$()$	Close the channel at location c.
c.**wait**$()$	Wait for the channel at location c to be closed.

The c.**recv**() and c.**wait**() operations are blocking, and could thus potentially lead to deadlocks. As is common in session-typed languages like GV [Wadler 2012; Gay and Vasconcelos 2010], our fork operation both spawns the child thread, and sets up a channel for communication between the parent thread and child thread. This will turn out to be important for deadlock freedom (§ 5 and 6).

The following example illustrates how we can use these constructs to fork off a thread that receives a message from the main thread, adds one to it, and sends it back:

$$\mathtt{let}\; c_1 = \mathtt{fork}\; (\lambda c_2.\, c_2.\mathtt{send}(c_2.\mathtt{recv}() + 1);\; c_2.\mathtt{close}())\; \mathtt{in}$$
$$c_1.\mathtt{send}(1);\; \mathtt{assert}(c_1.\mathtt{recv}() == 2);\; c_1.\mathtt{wait}()$$

The **assert** e operation asserts that e evaluates to true, and otherwise it gets stuck. Other illegal operations, such as sending over a closed channel, also get stuck forever. To verify the program, we need to reason about the channels c_1 and c_2. We do so by means of channel ownership assertions $c \rightarrowtail p$, which state that we own a reference to the channel c, and we must interact with it according to the protocol p. Our protocols are *dependent separation protocols* in the style of Actris [Hinrichsen et al. 2020]. We can use the following dual pair of protocols for c_1 and c_2 at the **fork**:

$$c_1 \rightarrowtail\; !\langle 1\rangle;\; ?\langle 2\rangle;\; ?\mathbf{end} \qquad\qquad c_2 \rightarrowtail\; ?\langle 1\rangle;\; !\langle 2\rangle;\; !\mathbf{end}$$

In these protocols, each step is either $!\langle v\rangle$ or $?\langle v\rangle$, indicating that the owner of the reference must **send** or **recv** a value v, respectively. The final $!\mathbf{end}$ / $?\mathbf{end}$ indicates that the protocol is finished, and that the **close** / **wait** operation must be performed.

Quantified protocols. The preceding protocol is inflexible, because it specifies the exact values that must be sent and received. To alleviate this inflexibility, we can use a *quantified protocol* instead:

$$c_1 \rightarrowtail\; !(n:\mathbb{N})\langle n\rangle;\; ?\langle n+1\rangle;\; ?\mathbf{end} \qquad\qquad c_2 \rightarrowtail\; ?(n:\mathbb{N})\langle n\rangle;\; !\langle n+1\rangle;\; !\mathbf{end}$$

This protocol states that if we send n, then we will receive $n + 1$. When verifying a quantified protocol step, the sender can instantiate the quantified variable with any logical value. For example, the sender can instantiate n with 1, and send 1 over the channel. The receiver must be verified to work for any n chosen by the sender.

The continuation of the protocol is allowed to be an arbitrary function of the quantified variables. This can be used to verify examples such as the following:

$$\mathtt{let}\; n = c.\mathtt{recv}()\; \mathtt{in}\; \mathtt{if}\; n < 5\; \mathtt{then}\; c.\mathtt{close}()\; \mathtt{else}\; c.\mathtt{send}(n-5); \ldots$$

The protocol for c will have to have a different length, depending on which branch of the **if** is taken. We can verify this program using the following protocol for c:

$$c \rightarrowtail\; ?(n:\mathbb{N})\langle n\rangle;\; \mathtt{if}\; n < 5\; \mathtt{then}\; (!\mathbf{end})\; \mathtt{else}\; (!\langle n-5\rangle;\; \ldots)$$

Mutable references. In addition to channels, our language has mutable references:

ref v	Allocate a new location in the heap and store the value v in it, and return the location.
$!\,\ell$	Read the value from the location ℓ.
$\ell \leftarrow v$	Write the value v to the location ℓ.
free ℓ	Free the location ℓ.

Illegal operations, such as using a location that has been freed, are modeled as getting stuck forever. Consider the following variant of the preceding example:

$$\begin{aligned}&\mathtt{let}\ c_1 = \mathtt{fork}\ (\lambda c_2.\ \mathtt{let}\ l = c_2.\mathtt{recv}()\ \mathtt{in}\ l \leftarrow\ !\,l + 1;\ c_2.\mathtt{close}())\ \mathtt{in}\\ &\mathtt{let}\ l = \mathtt{ref}\ 1\ \mathtt{in}\ c_1.\mathtt{send}(l);\ c_1.\mathtt{wait}();\ \mathtt{assert}(!\,l == 2);\ \mathtt{free}\ l\end{aligned}$$

We send a mutable reference from the main thread to the forked thread, which increments it. The main thread waits for the forked thread to close its channel, and then asserts that the value of the reference is 2. The reference is then freed by the main thread. LinearActris can prove that this program is safe and does not deadlock. Note that safety relies on the blocking behavior of $c_1.\mathtt{wait}()$, which ensures that the forked thread has finished before the main thread asserts that the value is 2 and frees the reference. The protocols to verify this program are as follows:

$$\begin{aligned}c_1 &\rightarrowtail\ \mathbf{!}(l : \mathrm{Loc}, n : \mathbb{N})\langle l\rangle\{l \mapsto n\};\ \mathbf{?end}\{l \mapsto n+1\}\\ c_2 &\rightarrowtail\ \mathbf{?}(l : \mathrm{Loc}, n : \mathbb{N})\langle l\rangle\{l \mapsto n\};\ \mathbf{!end}\{l \mapsto n+1\}\end{aligned}$$

This time, the protocol uses quantifiers for both the location l and the value n that is initially stored in the location. The protocol states that if we send a location l, then this location will be incremented by 1. The curly brackets {_} indicate the separation logic resources that are sent along with the message. In the protocol for c_1 above, the heap ownership assertion $l \mapsto n$ is transmitted with the initial **send** step, and $l \mapsto n+1$ is received in the **wait** step. As the following example shows, a reference need not be send over the channel, but can also be captured by the closure:

$$\begin{aligned}&\mathtt{let}\ l = \mathtt{ref}\ 1\ \mathtt{in}\ \mathtt{let}\ c_1 = \mathtt{fork}\ (\lambda c_2.\ l \leftarrow\ !\,l + 1;\ c_2.\mathtt{close}())\ \mathtt{in}\\ &c_1.\mathtt{wait}();\ \mathtt{assert}(!\,l == 2);\ \mathtt{free}\ l\end{aligned}$$

We transfer $l \mapsto 1$ to the child thread immediately upon the **fork**, and the protocols simplify to:

$$c_1 \rightarrowtail \mathbf{?end}\{l \mapsto 2\} \qquad\qquad c_2 \rightarrowtail \mathbf{!end}\{l \mapsto 2\}$$

Sending channels over channels. In addition to exchanging references, LinearActris can also reason about programs that send channels over channels. Consider the following example:

$$\begin{aligned}&\mathtt{let}\ d_1 = \mathtt{fork}\ (\lambda d_2.\ \mathtt{assert}(d_2.\mathtt{recv}().\mathtt{recv}() == 2);\ d_2.\mathtt{close}())\ \mathtt{in}\\ &\mathtt{let}\ c_1 = \mathtt{fork}\ (\lambda c_2.\ c_2.\mathtt{send}(2);\ c_2.\mathtt{wait}())\ \mathtt{in}\\ &d_1.\mathtt{send}(c_1);\ d_1.\mathtt{wait}();\ c_1.\mathtt{close}()\end{aligned}$$

The program forks off two threads, which gives the main thread two channels c_1 and d_1. The main thread then sends c_1 over d_1, and waits for d_1 to be closed, and then closes c_1. The first thread receives c_1 from d_2, and then receives on c_1 and asserts that the value is 2, and then closes d_2. The second thread sends 2 over c_2, and then waits for c_2 to be closed.

That this program is safe and does not deadlock can be proven by LinearActris, but this is more subtle than one might think: if we were to swap the two $d_1.\mathtt{wait}();\ c_1.\mathtt{close}()$ operations, then the program would not be safe, as c_1 might be closed before the other threads are done with it. We can verify the example using the following protocols:

$$\begin{aligned}&c_1 \rightarrowtail \mathbf{?}\langle 2\rangle;\ \mathbf{!end} \qquad\qquad c_2 \rightarrowtail \mathbf{!}\langle 2\rangle;\ \mathbf{?end}\\ &d_1 \rightarrowtail \mathbf{!}(c : \mathrm{Loc})\langle c\rangle\{c \rightarrowtail \mathbf{?}\langle 2\rangle;\ \mathbf{?end}\};\ \mathbf{?end}\{c \rightarrowtail \mathbf{!end}\}\\ &d_2 \rightarrowtail \mathbf{?}(c : \mathrm{Loc})\langle c\rangle\{c \rightarrowtail \mathbf{?}\langle 2\rangle;\ \mathbf{?end}\};\ \mathbf{!end}\{c \rightarrowtail \mathbf{!end}\}\end{aligned}$$

The protocol for c_1 and c_2 is simple: we send 2 and then end. The protocol for d_1 is more interesting: we send a (quantified) location c, and also send channel ownership for c, with the same protocol as we chose for c_1. The continuation of the protocol is $\mathbf{?end}\{c \rightarrowtail \mathbf{!end}\}$, which transfers ownership of c back to the main thread, but now at a new protocol.

Storing channels in references. Consider the following variation of the previous example, in which we wrap channel c_1 in a reference:

$$\begin{array}{l}\mathtt{let}\; d_1 = \mathtt{fork}\; (\lambda d_2.\, \mathtt{assert}((!\, d_2.\mathtt{recv}()).\mathtt{recv}() == 2);\; d_2.\mathtt{close}())\; \mathtt{in} \\ \mathtt{let}\; l = \mathtt{ref}\left(\mathtt{fork}\; (\lambda c_2.\, c_2.\mathtt{send}(2);\; c_2.\mathtt{wait}())\right)\; \mathtt{in} \\ d_1.\mathtt{send}(l);\; d_1.\mathtt{wait}();\; (!\, l).\mathtt{close}();\; \mathtt{free}\; l\end{array}$$

We can verify this example using the following protocol:

$$d_1 \rightarrowtail \mathbf{!}(l : \mathsf{Loc}, c : \mathsf{Loc})\langle l\rangle\{l \mapsto c * c \rightarrowtail \mathbf{?}\langle 2\rangle;\; \mathbf{!end}\};\; \mathbf{?end}\{l \mapsto c * c \rightarrowtail \mathbf{!end}\}$$

Sending closures. LinearActris can reason about higher-order programs that send closures, which capture references and channels. Consider the following program, which spawns a thread that receives and runs a closure from the main thread, and then sends the result back:

$$\mathtt{let}\; c_1 = \mathtt{fork}\; \;(\lambda c_2.\, \mathtt{let}\, f = c_2.\mathtt{recv}()\; \mathtt{in}\, c_2.\mathtt{send}(f\; ()); c_2.\mathtt{close}())\; \mathtt{in}\; \ldots$$

The protocol for c_1 is as follows:

$$c_1 \rightarrowtail \mathbf{!}(f : \mathsf{Val}, \Phi : \mathsf{Val} \to \mathsf{aProp})\langle f\rangle\{\mathsf{WP}\; f\; ()\; \{\Phi\}\};\; \mathbf{?}(v : \mathsf{Val})\langle v\rangle\{\Phi\; v\};\; \mathbf{?end}$$

The protocol allows us to send a closure f, provided we also send a weakest precondition assertion $\mathsf{WP}\; f\; ()\; \{\Phi\}$, which ensures that the return value v of f satisfies $\Phi\; v$. We can then receive v, and obtain the resources $\Phi\; v$. This protocol allows the closure f to capture linear resources in its environment, such as channels and references.

3 THE PROOF RULES OF LINEAR ACTRIS

In this section we present the rules of LinearActris. Like Iris, the key component of LinearActris is its weakest precondition $\mathsf{WP}\; e\; \{\Phi\}$ connective. This connective is a separation logic assertion, which states that if the program e is executed in the current heap, then its return value will satisfy predicate Φ. The Hoare triple $\{P\}\; e\; \{\Phi\}$ is syntactic sugar for $P \vdash \mathsf{WP}\; e\; \{\Phi\}$. The adequacy theorem of LinearActris (Theorem 5.4) guarantees safety, deadlock freedom, and leak freedom for e provided we have a proof of $\mathsf{Emp} \vdash \mathsf{WP}\; e\; \{\mathsf{Emp}\}$.

3.1 Basic Separation Logic

Fig. 2 displays the grammar of LinearActris propositions, as well as the basic rules for reasoning about weakest preconditions that involve pure expressions and mutable references. The weakest precondition rules in this figure are fairly standard, so we will only give a brief overview. The rules WP-pure-step and WP-val are the basic rules for reasoning about pure expressions. The rules WP-Löb and WP-rec are used to reason about recursive functions. The WP-bind rule is used to reason about an expression nested inside a (call-by-value) evaluation context. The WP-wand rule can be used to weaken the postcondition, as well as to frame away parts of the precondition. The rules WP-alloc, WP-load, WP-store, and WP-free reason about mutable references. In combination with inference rules for the logical connectives (which are not shown in the figure), these rules handle single-threaded programs, such as programs that manipulate mutable linked lists.

Linearity. An important distinction between LinearActris and Iris/Actris, is that LinearActris is *linear*, whereas Iris is *affine*. This means that in LinearActris, the rule $P \vdash \mathsf{Emp}$ does not hold for all P, whereas in Iris it does.[1] This distinction is important, because the rule $P \vdash \mathsf{Emp}$ can be used to leak resources. For instance if $P = \ell \mapsto v$, then $\ell \mapsto v \vdash \mathsf{Emp}$ can be used to leak the location ℓ. Unlike Iris, LinearActris guarantees leak freedom, and thus forces us to `free` locations. Furthermore, as we shall see shortly, the linearity of LinearActris is crucial for deadlock freedom,

[1]Logically equivalent formulations of the affine rule are $P * Q \vdash P$ or $\mathsf{Emp} \dashv\vdash \mathsf{True}$.

Separation logic propositions:

$$
\begin{array}{rll}
P, Q \in \mathsf{aProp} ::= & \mathsf{True} \mid \mathsf{False} \mid P \land Q \mid P \lor Q & \text{(Propositional logic)} \\
\mid & \forall x.\, P \mid \exists x.\, P \mid x = y & \text{(Higher-order logic with equality)} \\
\mid & P \ast Q \mid P \mathrel{-\!\!\ast} Q \mid \mathsf{Emp} & \text{(Separation logic)} \\
\mid & \triangleright P \mid \mathsf{WP}\ e\ \{\Phi\} & \text{(Step indexing and weakest preconditions)} \\
\mid & \ell \mapsto v \mid \ell \rightarrowtail p & \text{(Heap cell and channel ownership)}
\end{array}
$$

Basic weakest precondition rules:

$$
\textsc{WP-pure-step}\ \dfrac{e_1 \leadsto_{\text{pure}} e_2 \qquad \mathsf{WP}\ e_2\ \{\Phi\}}{\mathsf{WP}\ e_1\ \{\Phi\}}\ast
\qquad
\textsc{WP-val}\ \dfrac{\Phi\ v}{\mathsf{WP}\ v\ \{\Phi\}}\ast
\qquad
\textsc{WP-wand}\ \dfrac{\mathsf{WP}\ e\ \{\Phi\} \quad \ast \quad \forall v.\, \Phi\ v \mathrel{-\!\!\ast} \Psi\ v}{\mathsf{WP}\ e\ \{\Psi\}}\ast
$$

$$
\textsc{WP-rec}\ \dfrac{\mathsf{WP}\ e[x := v][f := \mathbf{rec}\ f\,x.\ e]\ \{\Phi\}}{\triangleright \mathsf{WP}\ (\mathbf{rec}\ f\,x.\ e)\ v\ \{\Phi\}}\ast
\qquad
\textsc{WP-Löb}\ \dfrac{\triangleright P \mathrel{-\!\!\ast} P}{P}\square
\qquad
\textsc{WP-bind}\ \dfrac{\mathsf{WP}\ e\ \{v.\ \mathsf{WP}\ K[v]\ \{\Phi\}\}}{\mathsf{WP}\ K[e]\ \{\Phi\}}\ast
$$

Heap manipulation rules:

$$
\textsc{WP-alloc}\ \dfrac{}{\mathsf{WP}\ \mathbf{ref}\,v\ \{\ell.\ \ell \mapsto v\}}\ast
\qquad
\textsc{WP-load}\ \dfrac{\ell \mapsto v}{\mathsf{WP}\ !\,\ell\ \{w.\ w = v \ast \ell \mapsto v\}}\ast
\qquad
\textsc{WP-store}\ \dfrac{\ell \mapsto v}{\mathsf{WP}\ \ell \leftarrow w\ \{\ell \mapsto w\}}\ast
\qquad
\textsc{WP-free}\ \dfrac{\ell \mapsto v}{\mathsf{WP}\ \mathbf{free}\,\ell\ \{\mathsf{Emp}\}}\ast
$$

Fig. 2. The basic rules of our separation logic.

as it prevents us from dropping the obligation to send a message over a channel (recall, not sending a message means that the receiving end of the channel would block forever).

3.2 Channels and Protocols

Like Actris, LinearActris uses dependent separation protocols for reasoning about channels. The grammar of protocols is displayed in Fig. 3, and their meaning is as follows:

- *Send protocol* $\mathbf{!}(\vec{x})\langle v\rangle\{P\};\ p$. The variables $\vec{x}$ are binders that scope over v, P, and p, that is, these are functions of $\vec{x}$. During the verification of a send operation, we can instantiate $\vec{x}$ with mathematical values of our choosing, and then v must be equal to the physical value that is sent, P is a separation logic proposition that we transfer to the receiver, and p is the new protocol for the channel.
- *Receive protocol* $\mathbf{?}(\vec{x})\langle v\rangle\{P\};\ p$. During the verification of a receive operation, we learn that there exists a choice of mathematical values $\vec{x}$ such that the physical value received equals v, P is a separation logic proposition we receive, and p is the new protocol for the channel.
- *Close protocol* $\mathbf{!end}\{P\}$. During the verification of a close operation, P is a separation logic proposition that we transfer to the other side.
- *Wait protocol* $\mathbf{?end}\{P\}$. During the verification of a wait operation, P is a separation logic proposition that we receive.
- *Recursive protocol* $\mu\alpha.\ p$. This is a recursive protocol, where α is a binder that scopes over p. The recursive protocol can be unfolded by replacing α with p. Recursive protocols with parameters are also supported, we give an example of such a protocol in §3.4.

Dependent separation protocols:

$$\begin{aligned} p \in \mathrm{Prot} ::= {} & \mathbf{!}(\vec{x})\langle v\rangle\{P\};\ p && \text{(Send protocol)}\\ \mid {} & \mathbf{?}(\vec{x})\langle v\rangle\{P\};\ p && \text{(Receive protocol)}\\ \mid {} & \mathbf{!end}\{P\} \mid \mathbf{?end}\{P\} && \text{(Close and wait protocol)}\\ \mid {} & \mu\alpha.\ p \mid \alpha && \text{(Recursive protocol)} \end{aligned}$$

Duality and subprotocols:

$$\begin{aligned} \overline{\mathbf{!}(\vec{x})\langle v\rangle\{P\};\ p} &= \mathbf{?}(\vec{x})\langle v\rangle\{P\};\ \overline{p} \qquad & \overline{\mathbf{!end}\{P\}} &= \mathbf{?end}\{P\} \qquad & \text{(Dual on dependent protocols)}\\ \overline{\mathbf{?}(\vec{x})\langle v\rangle\{P\};\ p} &= \mathbf{!}(\vec{x})\langle v\rangle\{P\};\ \overline{p} & \overline{\mathbf{?end}\{P\}} &= \mathbf{!end}\{P\} & \end{aligned}$$

$$\textsc{Sub-recv}\quad \frac{\forall x_1.\ P_1\ x_1 \mathrel{-\!\!*} \exists x_2.\ (v_1\ x_1 = v_2\ x_2) * P_2\ x_2 * \triangleright(p_1\ x_1 \sqsubseteq p_2\ x_2)}{\mathbf{?}(x_1)\langle v_1\rangle\{P_1\};\ p_1 \sqsubseteq \mathbf{?}(x_2)\langle v_2\rangle\{P_2\};\ p_2}\,*
\qquad
\textsc{Sub-wait}\quad \frac{P_1 \mathrel{-\!\!*} P_2}{\mathbf{?end}\{P_1\} \sqsubseteq \mathbf{?end}\{P_2\}}\,*$$

$$\textsc{Sub-send}\quad \frac{\forall x_2.\ P_2\ x_2 \mathrel{-\!\!*} \exists x_1.\ (v_2\ x_2 = v_1\ x_1) * P_1\ x_1 * \triangleright(p_1\ x_1 \sqsubseteq p_2\ x_2)}{\mathbf{!}(x_1)\langle v_1\rangle\{P_1\};\ p_1 \sqsubseteq \mathbf{!}(x_2)\langle v_2\rangle\{P_2\};\ p_2}\,*
\qquad
\textsc{Sub-close}\quad \frac{P_2 \mathrel{-\!\!*} P_1}{\mathbf{!end}\{P_1\} \sqsubseteq \mathbf{!end}\{P_2\}}\,*$$

$$\textsc{Sub-chan}\quad \frac{\triangleright\, p_1 \sqsubseteq p_2 \quad * \quad c \rightarrowtail p_1}{c \rightarrowtail p_2}\,*$$

Channel weakest precondition rules:

$$\textsc{WP-fork}\quad \frac{\forall c.\ (c \rightarrowtail \overline{p}) \mathrel{-\!\!*} \mathsf{WP}\ e\ c\ \{\mathsf{Emp}\}}{\mathsf{WP}\ \mathtt{\mathbf{fork}}\ e\ \{c.\ c \rightarrowtail p\}}\,*
\qquad
\textsc{WP-send}\quad \frac{P[\vec{x} := \vec{t}\,] \quad * \quad c \rightarrowtail \mathbf{!}(\vec{x})\langle v\rangle\{P\};\ p}{\mathsf{WP}\ c.\mathtt{\mathbf{send}}(v[\vec{x} := \vec{t}\,])\ \{c \rightarrowtail p[\vec{x} := \vec{t}\,]\}}\,*$$

$$\textsc{WP-recv}\quad \frac{c \rightarrowtail \mathbf{?}(\vec{x})\langle v\rangle\{P\};\ p}{\mathsf{WP}\ c.\mathtt{\mathbf{recv}}()\ \{w.\ \exists\vec{t}.\ w = v[\vec{x} := \vec{t}\,] * P[\vec{x} := \vec{t}\,] * c \rightarrowtail p[\vec{x} := \vec{t}\,]\}}\,*$$

$$\textsc{WP-close}\quad \frac{P \ * \ c \rightarrowtail \mathbf{!end}\{P\}}{\mathsf{WP}\ c.\mathtt{\mathbf{close}}()\ \{\mathsf{Emp}\}}\,*
\qquad
\textsc{WP-wait}\quad \frac{c \rightarrowtail \mathbf{?end}\{P\}}{\mathsf{WP}\ c.\mathtt{\mathbf{wait}}()\ \{P\}}\,*$$

Fig. 3. The LinearActris dependent separation protocols and channel rules.

The weakest precondition rules for channels in Fig. 3 work as follows:

- WP-fork: This rule is used to verify a fork operation. The rule states that fork returns a channel c, for which we can choose a protocol p. We must then verify that the thread that is spawned on the other side, operates with its side of the channel according to the *dual* protocol $\overline{p}$, which is the same as p except that all send and receive operations are swapped.

- WP-send: This rule is used to verify a send operation. The rule states that if we have channel ownership $c \rightarrowtail \mathbf{!}(\vec{x})\langle v\rangle\{P\};\ p$, then we can choose an instantiation $\vec{x} := \vec{t}$. The value that we send must equal $v[\vec{x} := \vec{t}]$, and we must give up ownership of the resources described by the proposition $P[\vec{x} := \vec{t}]$. In the postcondition, the channel gets the new protocol $p[\vec{x} := \vec{t}]$.
- WP-recv: This rule is used to verify a receive operation. The rule states that if we have channel ownership $c \rightarrowtail \mathbf{?}(\vec{x})\langle v\rangle\{P\};\ p$, then we can receive a message. In the postcondition, we learn that there exists an instantiation $\vec{x} := \vec{t}$ such that the value that we receive equals $v[\vec{x} := \vec{t}]$, and we obtain the ownership of the resources described by the proposition $P[\vec{x} := \vec{t}]$. The channel gets the new protocol $p[\vec{x} := \vec{t}]$.
- WP-close: This rule is used to verify a close operation. The rule states that if we have channel ownership $c \rightarrowtail \mathbf{!end}\{P\}$, then we can close the channel. We must also provide ownership of the proposition P, which is transmitted to the other side.
- WP-wait: This rule is used to verify a wait operation. The rule states that if we have channel ownership $c \rightarrowtail \mathbf{?end}\{P\}$, then we can wait on the channel, and afterwards we obtain P.

Deadlock and leak freedom. The rules in Fig. 3 are designed to ensure deadlock- and leak freedom. The reader may note that there are no apparent proof obligations for these properties, other than linearity: there are no preconditions that require us to follow a certain lock- or priority order. In § 4 to 6 we will see how the rules in Fig. 3 ensure deadlock freedom and leak freedom.

3.3 Subprotocols

An important feature of Actris are *subprotocols*, analogous to subtyping in type systems. LinearActris also supports subprotocols. The *subprotocol relation* is written $p_1 \sqsubseteq p_2$, and satisfies the rules in Fig. 3. The subprotocol relation lets us make the protocol of a channel more specific: we can turn channel ownership $c \rightarrowtail p_1$ into $c \rightarrowtail p_2$, provided that $p_1 \sqsubseteq p_2$. The rules of Fig. 3 are general, and imply various special cases, *e.g.*, the rules allow us to instantiate a quantifier in a send protocol:

$$\mathbf{!}(n : \mathbb{N})\langle n\rangle;\ \mathbf{?}\langle n+1\rangle;\ \mathbf{?end} \quad\sqsubseteq\quad \mathbf{!}\langle 1\rangle;\ \mathbf{?}\langle 2\rangle;\ \mathbf{?end}$$

Dually, we can abstract a quantifier in a receive protocol:

$$\mathbf{?}\langle 1\rangle;\ \mathbf{!}\langle 2\rangle;\ \mathbf{!end} \quad\sqsubseteq\quad \mathbf{?}(n : \mathbb{N})\langle n\rangle;\ \mathbf{!}\langle n+1\rangle;\ \mathbf{!end}$$

We can apply subprotocols deeper inside the protocol using the special case that if $p_1 \sqsubseteq p_2$, then:

$$\mathbf{!}\langle v\rangle\{P\};\ p_1 \quad\sqsubseteq\quad \mathbf{!}\langle v\rangle\{P\};\ p_2 \qquad\text{and}\qquad \mathbf{?}\langle v\rangle\{P\};\ p_1 \quad\sqsubseteq\quad \mathbf{?}\langle v\rangle\{P\};\ p_2$$

We can also use the subprotocol relation to make the propositions that are transferred more specific. If we have a separating implication $P_1 \mathrel{-\!\!*} P_2$, then we can replace the proposition that is transferred:

$$\mathbf{!}\langle v\rangle\{P_2\};\ p \quad\sqsubseteq\quad \mathbf{!}\langle v\rangle\{P_1\};\ p \qquad\text{and}\qquad \mathbf{?}\langle v\rangle\{P_1\};\ p \quad\sqsubseteq\quad \mathbf{?}\langle v\rangle\{P_2\};\ p$$

3.4 Guarded Recursive Protocols and Choice

Another important feature of Actris and LinearActris is the ability to construct infinite protocols. With the constructs we have see so far, we can construct unbounded protocols and verify programs with them. One can use *well-founded recursion* in the meta-logic (*i.e.*, a `Fixpoint` definition in Coq) to define a recursive function that constructs a protocol, and then use that protocol in the verification of a program. For example, we can construct a protocol that sends n messages, and then closes the channel, for any n determined by the first message:

$$\mathbf{!}(n : \mathbb{N})\langle n\rangle;\ \mathbf{!}\langle n-1\rangle;\ \cdots \mathbf{!}\langle 0\rangle;\ \mathbf{!end}$$

However, well-founded recursion does not allow us to construct truly infinite protocols, such as the following protocol that sends increasing natural numbers forever:

$$\mathbf{!}(n : \mathbb{N})\langle n\rangle;\ \mathbf{!}\langle n+1\rangle;\ \mathbf{!}\langle n+2\rangle;\ \cdots$$

Actris and LinearActris allow us to construct such infinite protocols using *guarded recursion*:

$$p\ n \triangleq \mathbf{!}\langle n\rangle;\ p\ (n+1) \qquad \text{or formally: } p \triangleq \mu\alpha.\,\lambda n.\,\mathbf{!}\langle n\rangle;\ \alpha\ (n+1)$$

This definition is guarded, because the recursive call is guarded by a message send. Our notion of guardedness also allows the recursive call to occur inside the resources. For example:

$$p\ n \triangleq \mathbf{!}(c : \mathrm{Loc})\langle c\rangle\{c \rightarrowtail p\ (n+1)\};\ \mathbf{!end}$$

Guarded recursion is most useful in combination with *choice*, which we can encode using a quantified protocol. This lets us express "services" that can perform a certain action (such as sending a natural number) forever, but allow the receiver to close the channel:

$$p \triangleq \mathbf{!}(n : \mathbb{N})\langle n\rangle;\ \mathbf{?}(b : \mathrm{Bool})\langle b\rangle;\ \mathbf{if}\ b\ \mathbf{then}\ (\mathbf{!end})\ \mathbf{else}\ p$$

4 FROM MULTI-SHOT TO ONE-SHOT CHANNELS

Before discussing the adequacy proof of LinearActris (§ 5 and 6), we first reduce multi-shot channels and protocols to single-shot channels and protocols, inspired by the approach of Dardha et al. [2012] for session types and Jacobs et al. [2023b] for separation logic.

The reason we encode multi-shot channels in terms of one-shot channels is twofold. First, it is easier to prove adequacy of the one-shot logic, because it is simpler. The ideas required are not fundamentally different, but there are fewer cases to handle. Second, we believe that the encoding of multi-shot channels in terms of one-shot channels showcases the flexibility of LinearActris: the encoding involves mutable references and transmitting channels over channels and creating new threads in a non-trivial way. If one considers the examples of § 2 in light of the encoding, one realizes that a lot is going on at run-time, and one might therefore expect it to be difficult to verify deadlock and leak freedom. The encoding shows that LinearActris is flexible enough to modularly build the multi-shot abstraction in terms of one-shot channels.

4.1 Primitive One-Shot Logic

The primitive one-shot channels have the following operations:

fork1 $(\lambda c.\, e)$	Fork a new thread that runs e with a new one-shot channel c, and return c.
send1 $c\ v$	Send message v over the channel c.
recv1 c	Receive a message over the channel c, and free c.

The **send1** $c\ v$ and **recv1** c operations may only be used once per one-shot channel. The operation **recv1** c is blocking.

The primitive one-shot channels are governed by simple one-shot protocols, which are defined in Fig. 4. A one-shot protocol is either $\mathbf{!}\Phi$ or $\mathbf{?}\Phi$, where $\Phi \in \mathrm{Val} \rightarrow \mathrm{aProp}$ is a separation logic predicate that specifies which values are allowed to be transmitted. The dual of $\mathbf{!}\Phi$ is $\mathbf{?}\Phi$, and *vice versa*. The primitive one-shot channel weakest precondition rules are given in Fig. 4. The rules are similar to the rules of LinearActris, except that they are simpler because they do not have to deal with the complexity of multi-shot channels and protocols:

- WP-PRIM-SEND: When we **send1** $c\ v$, we must have channel ownership $c \rightarrowtail_1 \mathbf{!}\Phi$, and we must provide resources $\Phi\ v$ to be transmitted. The postcondition is Emp, because the channel ownership is consumed.

One-shot protocols:

$$p \in \mathsf{Prot} ::= \mathbf{!}\Phi \mid \mathbf{?}\Phi \qquad \text{where } \Phi \in \mathsf{Val} \to \mathsf{aProp} \qquad \text{(Protocols)}$$

$$c \rightarrowtail_1 p \qquad \text{(Channel points-to)}$$

$$\overline{\mathbf{!}\Phi} \triangleq \mathbf{?}\Phi \qquad \overline{\mathbf{?}\Phi} \triangleq \mathbf{!}\Phi \qquad \text{(Duality)}$$

One-shot channel weakest precondition rules:

WP-PRIM-FORK
$$\frac{\forall c.\ (c \rightarrowtail_1 \overline{p}) \mathrel{-\!\!*} \mathsf{WP}\ e\ c\ \{\mathsf{Emp}\}}{\mathsf{WP}\ \mathbf{fork1}\ e\ \{c.\ c \rightarrowtail_1 p\}}\,*$$

WP-PRIM-SEND
$$\frac{\Phi\ v \quad * \quad c \rightarrowtail_1 \mathbf{!}\Phi}{\mathsf{WP}\ \mathbf{send1}\ c\ v\ \{\mathsf{Emp}\}}\,*$$

WP-PRIM-RECV
$$\frac{c \rightarrowtail_1 \mathbf{?}\Phi}{\mathsf{WP}\ \mathbf{recv1}\ c\ \{v.\ \Phi\ v\}}\,*$$

Fig. 4. The primitive one-shot channel rules.

- WP-PRIM-RECV: When we **recv1** c, we must have channel ownership $c \rightarrowtail_1 \mathbf{?}\Phi$, and we obtain resources $\Phi\ v$ where v is the value that was received. The channel ownership is consumed.

4.2 Encoding of Multi-Shot Channels

The multi-shot channels from §3 are implemented in terms of one-shot channels. The implementation is given in Fig. 5. A multi-shot channel endpoint is represented as a mutable reference that stores a one-shot channel. When we send a message v on a multi-shot channel, we create a continuation one-shot channel c', and we send the message (c', v) on the one-shot channel that is stored in the mutable reference. The channel c' is then stored in the mutable reference of the sender, to be used for communicating the next message. On the other side, we receive a message (c', v), and we store c' in the receiver's mutable reference, and then return v. The multi-shot channel is closed by doing a final synchronisation on the one-shot channel without creating a continuation channel, and freeing the mutable reference.

We *define* multi-shot protocols in terms of one-shot protocols, as shown in Fig. 5. The definition for $\mathbf{?}(\vec{x})\langle v\rangle\{P\};\ p$ specifies that there exists an instantiation of the binders $\vec{x}$, such that the message (c, v) is sent over the one-shot channel, which means that the value is specified by v in the protocol. We additionally transmit the resources P, as well as new channel ownership $c \rightarrowtail_1 P$ for the continuation channel at the right protocol. The definition of the send protocol is simply dual. The definitions of the close and wait protocols are special cases of the send and receive protocols, as no continuation channel is created.

Finally, multi-shot channel ownership $\ell \rightarrowtail p$ is defined in terms of heap ownership and one-shot channel ownership, as shown in Fig. 5. The definition states that the mutable reference ℓ stores a one-shot channel c, and that the one-shot channel has protocol $q \sqsubseteq p$. This means that the multi-shot channels support subprotocols, even though the one-shot channels do not.

With these definitions, along with the specifications for one-shot protocols, we can then derive the specifications for multi-shot channels, as presented in Fig. 3.

5 WHY LINEAR ACTRIS IS DEADLOCK FREE: CONNECTIVITY GRAPHS

Now that we have given the rules of the one-shot logic, we cover how it guarantees deadlock- and leak freedom by linearity. We first give the general structure of the adequacy proof, and explain how it uses an invariant that is preserved as the program executes (§5.1). We give an intuition for the principles that the invariant needs to enforce, by going through some faulty examples, and

Multi-shot imperative channel implementation:

$$\begin{aligned}
\texttt{fork}\ e &\triangleq \texttt{ref}\ (\texttt{fork1}\ (\lambda c.\, e\ (\texttt{ref}\, c)))\\
\ell.\texttt{send}(v) &\triangleq \texttt{let}\ c = !\,\ell\ \texttt{in}\ \ell \leftarrow \texttt{fork1}\ (\lambda c'.\, \texttt{send1}\ c\ (c', v))\\
\ell.\texttt{recv}() &\triangleq \texttt{let}\ (c', v) = \texttt{recv1}\ (!\,\ell)\ \texttt{in}\ \ell \leftarrow c';\ v\\
\ell.\texttt{close}() &\triangleq \texttt{send1}\ (!\,\ell)\ ();\ \texttt{free}\ \ell\\
\ell.\texttt{wait}() &\triangleq \texttt{recv1}\ (!\,\ell);\ \texttt{free}\ \ell
\end{aligned}$$

Dependent multi-shot protocol definitions:

$$\begin{aligned}
?(\vec{x})\langle v\rangle\{P\};\ p &\triangleq ?(\lambda w.\, \exists \vec{x}, c.\, w = (c, v) * P * c \rightarrowtail_1 p) && \text{(Receive protocol)}\\
!(\vec{x})\langle v\rangle\{P\};\ p &\triangleq !(\lambda w.\, \exists \vec{x}, c.\, w = (c, v) * P * c \rightarrowtail_1 \overline{p}) && \text{(Send protocol)}\\
?\mathbf{end}\{P\} &\triangleq ?(\lambda w.\, w = () * P) && \text{(Wait protocol)}\\
!\mathbf{end}\{P\} &\triangleq !(\lambda w.\, w = () * P) && \text{(Close protocol)}\\
\ell \rightarrowtail p &\triangleq \exists c, q.\, \ell \mapsto c * c \rightarrowtail_1 q * \triangleright(q \sqsubseteq p) && \text{(Channel points-to)}
\end{aligned}$$

Subprotocols:

$$\begin{aligned}
!\Phi \sqsubseteq !\Psi &\triangleq \forall v.\, \Psi\, v \mathrel{-\!\!*} \Phi\, v & \qquad !\Phi \sqsubseteq ?\Psi &\triangleq \mathsf{False}\\
?\Phi \sqsubseteq ?\Psi &\triangleq \forall v.\, \Phi\, v \mathrel{-\!\!*} \Psi\, v & \qquad ?\Phi \sqsubseteq !\Psi &\triangleq \mathsf{False}
\end{aligned}$$

Fig. 5. Multi-shot channels and protocols in terms of one-shot channels and protocols.

by showing how the notion of connectivity graphs [Jacobs et al. 2022a] is used (§5.2). We finally present how we reason about the preservation of the invariant in terms of connectivity graphs (§5.3). In the next section we will give a more formal presentation of the adequacy proof, including the use of step-indexing to stratify circular definitions (§6).

5.1 General Approach

Our general approach is to define an invariant $I(\vec{e}, h)$, which describes the state of the configuration of threads and heap. The invariant shall satisfy three properties that together imply adequacy. This approach is similar to the technique of progress and preservation for proving type safety [Wright and Felleisen 1994; Pierce 2002; Harper 2016], but our invariant is defined semantically (in terms of the operational semantics of the language) instead of syntactically (in terms of inductively defined judgments). The first of these properties is that the invariant can be established by the weakest precondition of the program:

Lemma 5.1 (Initialization). *If* $\mathsf{Emp} \vdash \mathsf{WP}\ e\ \{\mathsf{Emp}\}$ *holds, then* $I([e], \emptyset)$ *holds.*

That is, the invariant holds for the initial configuration with one thread e and empty heap. The second property is that the invariant is preserved by the steps of our operational semantics:

Lemma 5.2 (Preservation). *If* $I(\vec{e}, h)$ *holds, and* $(\vec{e}, h) \rightarrow_t (\vec{e'}, h')$, *then* $I(\vec{e'}, h')$ *holds.*

The third property is that the invariant implies the conclusion of the adequacy theorem:

Lemma 5.3 (Progress). *If* $I(\vec{e}, h)$ *holds, then either* $(\vec{e}, h)$ *can step, or* $\vec{e}$ *are all values and* $h = \emptyset$.

Together, these three properties imply adequacy, because if we start with the initial configuration, then we can repeatedly apply the preservation theorem to get to a configuration where the invariant holds, after which we can apply the progress theorem to establish adequacy:

Theorem 5.4 (Adequacy ⚙). *If* $\mathsf{Emp} \vdash \mathsf{WP}\ e\ \{\mathsf{Emp}\}$ *is holds, and* $([e], \emptyset) \rightarrow_{\mathrm{t}}^{*} (\vec{e}, h)$*, then either:*

- $(\vec{e}, h) \rightarrow_{\mathrm{t}} (\vec{e}', h')$ *for some* $(\vec{e}', h')$*, or,*
- $\vec{e}$ *are all values and* $h = \emptyset$*.*

In addition to this adequacy theorem, our logic also guarantees safety:

Theorem 5.5 (Safety ⚙). *If* $\mathsf{Emp} \vdash \mathsf{WP}\ e\ \{\mathsf{Emp}\}$ *is holds, and* $([e], \emptyset) \rightarrow_{\mathrm{t}}^{*} (\vec{e}, h)$*, then every thread in* $\vec{e}$ *can either reduce, or is a value, or is blocked on a receive or wait operation.*

The safety theorem is a straightforward consequence of our invariant, so we will not discuss it further. The reader can find the proof in the Coq mechanization [Jacobs et al. 2023a]. In the next subsections we aim to give an intuition of what the invariant $I(\vec{e}, h)$ looks like, and why it is preserved by the operational semantics of our language.

5.2 The Invariant Properties

The proof rules of LinearActris rule out deadlocks and leaks. To prove this, we will define the invariant, as introduced in § 5.1. In this section we investigate the properties that we need the invariant to enforce. We do this by considering program examples that *do* deadlock or leak to identify patterns we need to exclude. This allows us to build up to a formulation of our invariant that is sufficient to make the proof go through, analogous to strengthening the induction hypothesis.

Consider the following example:

$\mathbf{let}\ c_1 = \mathbf{fork1}\ (\lambda c_2.\ ())\ \mathbf{in}\ \mathbf{recv1}\ c_1$

The forked-off thread does nothing, and the main thread waits for the forked-off thread by attempting to receive a message. The problem is that the forked-off thread does not fulfill its obligation to send a message. The proof rules of LinearActris exclude this example because the forked off thread gives a linear channel ownership assertion for c_2 that it must consume, and in this example, there is no operation in the forked-off thread that can consume the ownership assertion. To exclude this pattern our invariant must uphold the following property:

Channel fulfillment: Terminated threads must not hold ownership assertions of channels.

Now consider the following type of deadlock, where both sides try to receive:

$\mathbf{let}\ c_1 = \mathbf{fork1}\ (\lambda c_2.\ \mathbf{recv1}\ c_2)\ \mathbf{in}\ \mathbf{recv1}\ c_1$

The rules of LinearActris exclude this example because upon fork, one thread gets a receive assertion with $?\Phi$ and the other gets a send assertion $!\Phi$, or *vice versa*. Our invariant enforces this with the following property:

Channel duality: Each channel in the configuration is in one of two states:
1. There exist two channel ownership assertions $c \rightarrowtail\ !\Phi$ and $c \rightarrowtail\ ?\Phi$ for that channel, and the channel buffer is empty.
2. There exist only the receiver assertion $c \rightarrowtail\ ?\Phi$, and the channel contains a value v that satisfies $\Phi\ v$.

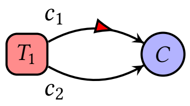
Next, consider the type of deadlock illustrated by the following example:

$\mathbf{let}\ l = \mathbf{ref}\ 1\ \mathbf{in}$
$\mathbf{let}\ c_1 = \mathbf{fork1}\ (\lambda c_2.\ l \leftarrow c_2)\ \mathbf{in}$
$\mathbf{recv1}\ c_1;\ \mathbf{send1}\ (!\,l)\ 2;\ \mathbf{free}\ l$

In this example, the forked-off thread smuggles its own channel back to the main thread by putting it in the reference l. The main thread then attempts to receive, but this will block forever, as the matching send (on $!\,l$) is performed after the receive.

The rules of LinearActris exclude this example because the reference l must be uniquely owned. Since the forked-off thread immediately stores a value in the reference, the forked off thread must get initial ownership over l. The forked-off thread then stores its channel c_2 in the reference, but according to the proof rules of LinearActris, this does not consume the channel ownership assertion for c_2. Hence, the proof rules prevent this example for similar reasons as before.

This example is not yet ruled out by the invariant property above, however, as the invariant has no way of excluding the possibility that the main thread might be holding both channel ownership assertions for c_1 as well as c_2. We depict the situation when the deadlock occurs as follows:



Here, the T_1 node represents the main thread, the C node represents the channel, and the two edges labeled c_1 and c_2 represent ownership of the two endpoints of C. The red triangle on the c_1 edge represents the fact that the thread T_1 is currently blocked on **recv1** c_1. This is a deadlock because T_1 has both endpoints, so **recv1** c_1 will never get unblocked.

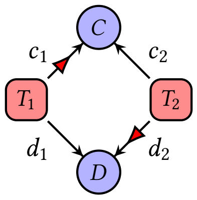
Our invariant rules out this situation with the following property:

Weak channel acyclicity: No thread can hold ownership over both endpoints of a channel.

This property is yet again not enough to guarantee deadlock freedom. In general, it can be the case that there are several threads that are waiting for each other, and that none of them will ever perform the send that the others are waiting for. Consider the following situation:

Thread 1:	**recv1** c_1; **send1** d_1 2	**Ownership:**	$c_1 \rightarrowtail_1 ?\Phi * d_1 \rightarrowtail_1 !\Psi$
Thread 2:	**recv1** d_2; **send1** c_2 1	**Ownership:**	$d_2 \rightarrowtail_1 ?\Psi * c_2 \rightarrowtail_1 !\Phi$

Here, both threads are waiting for each other, but neither of them will ever perform the send that the other is waiting for. This example does not violate the preceding invariant properties as neither thread holds both channel ownership assertions for the same channel. Yet, there is still a deadlock:



This graph shows that thread T_1 is blocked on **recv1** c_1, and thread T_2 is blocked on **recv1** d_2. The problem occurs because the graph is cyclic in an undirected sense. Note that there are no cycles in this directed graph, but deadlocks can already occur if there is a cycle when one is allowed to traverse ownership edges in either direction.

The rules of LinearActris yet again prevent this situation from arising. In general, it may be difficult to see why the proof rules of LinearActris ensure weak channel acyclicity. After all, we can send channel assertions over channels via the transferred resources. One way to understand this is via the following strengthening of the invariant, which generalizes the preceding invariant by considering the *graph* of channel ownership assertions held by the threads:

Strong channel acyclicity (initial version): There exists a *connectivity graph* of channel ownership assertions, where there is an edge from thread T to channel C if T holds a channel ownership assertion $c \rightarrowtail_1 p$ for an endpoint c of C. This graph must be *strongly acyclic.*

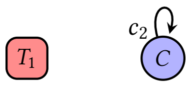
By the term *strongly acyclic*, we mean that there is at most one path from any node to another, even if one is allowed to follow edges backwards.

Leaks. The aforementioned properties are enough to rule out the preceding examples, but there are subtle types of deadlocks that can still occur. The last remaining issue is that we have not yet taken into account the fact that we can store ownership assertions in channels, by transferring them via the send operation. There is thus a danger that we can leak channel ownership assertions circularly into each other, and thus create a cycle of channel ownership assertions. This could cause deadlocks in the same way as the first example in this section: by leaking a send ownership assertion, a send will never happen, and the receiver will block indefinitely.

For this reason, deadlocks are intimately related to *leaks*. It might be tempting to think that linearity alone is enough to rule out leaks, but as we alluded to, this is not the case. Consider what would happen if we had two channel endpoints c_1 and c_2, and do the following:

send1 $c_1\ c_2$

This program would not deadlock, but it would put the channel c_2 in the buffer of c_1. If c_1 and c_2 turned out to be two endpoints *of the same channel*, then this would be a leak, as the channel would never be freed. We can use protocol $\Phi\ v \triangleq \mathsf{True}$ with $c_1 \rightarrowtail_1 !\Phi$ and $c_2 \rightarrowtail_1 ?\Phi$. This protocol allows us to transfer any resource, including the ownership assertion for c_2. Thus, channel ownership for the channel would be stored inside itself, and we would have a leak:



We strengthen our invariant to ensure there cannot be any cyclic ownership between channels:

Strong channel acyclicity (final version): There exists a *connectivity graph* of channel ownership assertions, where there is:
- An edge from a thread T to a channel C if T holds a channel ownership assertion $c \rightarrowtail_1 p$ for an endpoint c of C.
- An edge from a channel C to a channel C' if C contains a message with associated channel ownership assertion $c' \rightarrowtail_1 p$ for an endpoint c' of C'.

This graph must be *strongly acyclic.*

Note that channel ownership should not be confused with having a reference to a channel. A thread can have a reference to a channel without having channel ownership for that channel, and a thread can have channel ownership for a channel without having a reference to that channel.

We can now understand deadlocks and leaks in terms of the connectivity graph:

- **Deadlock.** In order for a thread to be able to perform a receive or wait operation, it must have channel ownership for the channel that it is receiving from. Therefore, if we have a deadlock in which threads are blocked on each other in a circular manner, then there must be a cycle of threads and channels in the connectivity graph.

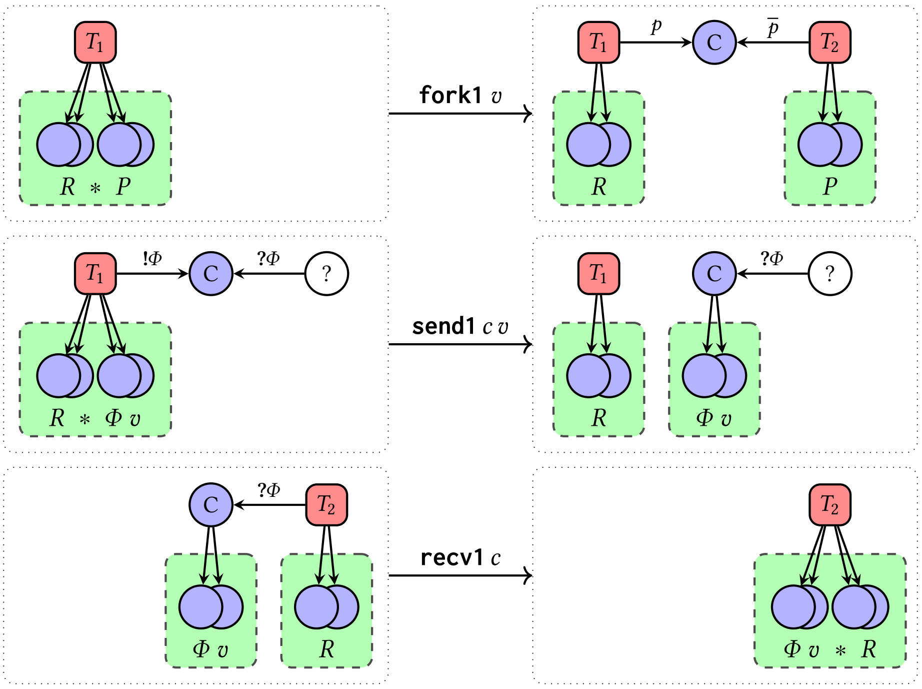


Fig. 6. The one-shot channel operations and the corresponding connectivity graph transformations.

- **Leak.** If, after the program has terminated, there are still channels in the heap, then the channel ownership for them must be stored inside each other in a circular manner, and then there must be a cycle of channels in the connectivity graph.

5.3 Preserving the Invariant

We now discuss how we preserve the invariant by virtue of our program logic rules. The property of **channel fulfillment** is preserved by the fact that we work in a linear logic. The property of **channel duality** is preserved by the fact that we force channels to be dual when allocated.

The property of **strong channel acyclicity** is more intricate, as the connectivity graph must be updated as the program executes. In Fig. 6, we show how the connectivity graph transforms due to each of the one-shot channel operations:

- **Fork.** When thread T_1 does a fork operation, it adds a new thread T_2 to the connectivity graph, and connects it to the original thread via a channel C. The two edges to the channel are labeled with dual protocols p and $\overline{p}$. The original thread T_1 originally owned separation logic resources $R * P$, which may contain ownership of other channels (and mutable references, which we ignore here). This is represented as edges from T_1 to the owned channels. We let P be the ownership of the channel that is transferred to the new thread T_2, while R is the part that thread T_1 keeps for itself. Due to this split of ownership, the fork operation corresponds to a modification of the graph, as shown in the figure. Crucially, if the original graph is strongly acyclic, the resulting graph is still strongly acyclic. Note that this relies on the separation between $R * P$. If we had a channel ownership assertion that occurred both in R and in P, then the resulting graph would not be strongly acyclic.

- **Send.** When thread T_1 performs a send operation on a channel C with protocol $\mathbf{!}\Phi$, it must provide resources $\Phi\ v$, where v is the value it wants to send. The resources $\Phi\ v$ get transferred to the channel, and the thread loses its connection to the channel, because it is one-shot. Therefore, the send operation corresponds to a modification of the graph, as shown in the figure. The reader can see strong acyclicity is preserved.
- **Receive.** When thread T_2 performs a receive operation on a channel C with protocol $\mathbf{?}\Phi$, it receives a value v and resources $\Phi\ v$ from the channel. The channel gets deallocated and removed from the graph, because the channel is one-shot. If the thread initially owned resources R, then afterwards it owns resources $\Phi\ v \ast R$. Note that these resources are separated—this relies crucially on the acyclicity of the graph before the receive operation: if thread T_2 already had channel ownership for some channel C', and additionally got a second channel ownership assertion for C' via $\Phi\ v$, then the original graph would not have been strongly acyclic.

In short, the proof rules of LinearActris ensure that strong acyclicity of the connectivity graph is preserved, and thus its adequacy theorem can ensure that the program is deadlock and leak free. In the next section, we will give an overview of how this is proved formally.

6 FORMAL ADEQUACY PROOF

In this section we give a formal overview of our adequacy proof. We first give a model of the propositions aProp of LinearActris by solving a recursive domain equation in a step-indexed universe of sets [America and Rutten 1989; Birkedal et al. 2010] (§6.1). We then define the invariant that captures the properties presented in §5, which we use in the adequacy proof (§6.2). We then define the LinearActris weakest preconditions, such that it enforces the invariant (§6.3). Finally, we sketch how the weakest precondition rules and the adequacy theorem are proved (§6.4).

6.1 The Step-Indexed Model of Propositions

To map the intuition of the previous section to a formal model of separation logic, we will first give the semantics of the type of propositions. This means we need to define a type aProp with the usual separation logic operators and the connectives $c \rightarrowtail_1 p \in \mathsf{aProp}$ and $\ell \mapsto v \in \mathsf{aProp}$. These connectives assert ownership of outgoing edges to a channel or a heap location in the connectivity graph. To define aProp, we solve the following recursive domain equation:

$$\mathsf{aProp} \simeq (\overbrace{\mathsf{Node} \xrightarrow{\text{fin}} \blacktriangleright(\underbrace{\{\mathbf{!},\mathbf{?}\} \times (\mathsf{Val} \to \mathsf{aProp})}_{\text{protocols } \mathbf{!}\Phi \text{ and } \mathbf{?}\Phi} + \underbrace{\mathsf{Val}}_{\text{ref values}})}^{\text{outgoing edges}}) \to \mathsf{siProp}$$

Before discussing the technicalities (siProp, ▶), let us provide the intuition behind this definition. Recall from §5 that the nodes of the connectivity graph are locations (which can either be channels or references, collectively called *cells*) or threads. Formally, we let $v \in \mathsf{Node} ::= \mathsf{Cell}(\ell) \mid \mathsf{Thread}(tid)$ with $\ell \in \mathsf{Loc}$ and $tid \in \mathbb{N}$. There is a directed edge from node v_1 to v_2 in the connectivity graph if v_1 owns v_2. Edges are labeled with either a protocol ($\mathtt{inl}\ p$) in case of ownership of a channel $\mathsf{Cell}(c)$, or the value of the reference ($\mathtt{inr}\ v$) in case of ownership of a reference $\mathsf{Cell}(\ell)$. A LinearActris proposition should always be seen in the context of a particular thread of channel, where the outgoing edges describe which nodes the thread or channel owns. Threads $\mathsf{Thread}(n)$ are never owned, but are included for conformity with the graph.

The type aProp is not well-defined as an inductive or coinductive definition in the category of sets, because the recursive occurrence of aProp is in negative position. That is why we use the results by America and Rutten [1989]; Birkedal et al. [2010] to solve the recursive domain equation using step-indexing [Appel and McAllester 2001]. The use of step-indexing is evident by the use of (pure) step-indexed propositions (siProp) as our meta logic, and the use of the ► constructor which guards the recursion. This construction is similar to how the model of Iris is constructed, with the crucial difference that Iris considers monotone predicates to obtain an affine logic.

With this definition at hand, we can define the connectives of our separation logic:

$$\begin{aligned}
c \rightarrowtail_1 p &\triangleq \lambda\Sigma.\ \Sigma = \{\mathsf{Cell}(c) \mapsto \mathsf{inl}\, p\} \\
\ell \mapsto v &\triangleq \lambda\Sigma.\ \Sigma = \{\mathsf{Cell}(\ell) \mapsto \mathsf{inr}\, v\} \\
P * Q &\triangleq \lambda\Sigma.\ \exists\Sigma_1, \Sigma_2.\ \Sigma = \Sigma_1 \cup \Sigma_2 \land \mathsf{dom}(\Sigma_1) \cap \mathsf{dom}(\Sigma_2) = \emptyset \land P\,\Sigma_1 \land Q\,\Sigma_2 \\
P \mathrel{-\!\!*} Q &\triangleq \lambda\Sigma.\ \forall\Sigma'.\ \mathsf{dom}(\Sigma) \cap \mathsf{dom}(\Sigma') = \emptyset \Rightarrow P\,\Sigma' \Rightarrow Q(\Sigma \cup \Sigma') \\
P \land Q &\triangleq \lambda\Sigma.\ P\,\Sigma \land Q\,\Sigma
\end{aligned}$$

We have glossed over several technical details here, such as the embedding of A into $\blacktriangleright A$, and that the right hand sides of these definitions live in the step-indexed logic siProp. We refer the interested reader to our Coq mechanization for the full details [Jacobs et al. 2023a].

6.2 The Invariant

The invariant is defined in terms of a connectivity graph, which is a labeled directed graph that is strongly acyclic. Recall that the nodes are the logical objects Node in the configuration. The incoming edges of channels are labeled by the protocols $!\Phi$ and $?\Phi$ appearing in the channel ownership assertions $c \rightarrowtail_1 p$. The incoming edge of a mutable reference is labeled by the value of the reference appearing in the reference ownership assertion $\ell \mapsto v$.

The invariant $I(\sigma)$ on a configuration σ is therefore defined as follows:

$$\mathsf{I}(\sigma) \triangleq \exists G : \mathsf{CGraph}.\ \forall v.\ \mathsf{local\_inv}(\sigma[v], \mathsf{in\_labels}_G(v), \mathsf{out\_edges}_G(v))$$

Here, $\mathsf{in\_labels}_G(v)$ is the multiset of labels on incoming edges of v, and $\mathsf{out\_edges}_G(v)$ is a finite map of outgoing edges of v (as in Jacobs et al. [2022a]). Furthermore, $\sigma[v]$ looks up the physical state associated to the node v in configuration σ. Given $\sigma = (\vec{e}, h)$, the value of $\sigma[v]$ is:

- $\mathsf{Expr}(e)$ if $v = \mathsf{Thread}(\mathit{tid})$ and e is the expression with thread ID tid in $\vec{e}$,
- $\mathsf{Ref}(v)$ if $v = \mathsf{Cell}(\ell)$ and v is the value of the mutable reference at location ℓ in heap h,
- $\mathsf{Chan}(\bot)$ if $v = \mathsf{Cell}(c)$ and no value has been sent to the channel at location c in heap h,
- $\mathsf{Chan}(v)$ if $v = \mathsf{Cell}(c)$ and value v has been sent over the channel at location c in heap h,
- $\bot$ if v is not a valid thread in $\vec{e}$ or cell in h.

The definition of I states that there is a connectivity graph G (that is strongly acyclic), and that for every value of v, the *local invariant* local_inv holds. The local invariant constrains the relation between the physical state of the node v, and the incoming and outgoing edges of v in the graph, and thus relates the graph to the configuration. It is defined as:

$$\begin{aligned}
\mathsf{local\_inv}(\mathsf{Expr}(e), \alpha, \Sigma) &\triangleq \alpha = \emptyset \land \mathsf{WP}_0\ e\ \{\mathsf{Emp}\}\ \Sigma \\
\mathsf{local\_inv}(\mathsf{Ref}(v), \alpha, \Sigma) &\triangleq \alpha = \{v\} \land \Sigma = \emptyset \\
\mathsf{local\_inv}(\mathsf{Chan}(\bot), \alpha, \Sigma) &\triangleq \exists\Phi.\ \alpha = \{!\Phi, ?\Phi\} \land \Sigma = \emptyset \\
\mathsf{local\_inv}(\mathsf{Chan}(v), \alpha, \Sigma) &\triangleq \exists\Phi.\ \alpha = \{!\Phi\} \land \Phi\ v\ \Sigma \\
\mathsf{local\_inv}(\bot, \alpha, \Sigma) &\triangleq \alpha = \emptyset \land \Sigma = \emptyset
\end{aligned}$$

The local invariant for a thread $\mathsf{Expr}(e)$ states that the incoming edges α are empty, and that we have a weakest precondition for e, which owns the outgoing edges Σ (see the end of §6.3 for the difference between the *primitive* $\mathsf{WP}_0\ e\ \{\Phi\}$ and *frame-preserving* $\mathsf{WP}\ e\ \{\Phi\}$). The local invariant for a reference $\mathsf{Ref}(v)$ states that the incoming edges α are the singleton set with value v, and that the outgoing edges Σ are empty. The local invariant for an empty channel $\mathsf{Chan}(\bot)$ states that the incoming edges α are the set containing the dual protocols $!\Phi$ and $?\Phi$, and that the outgoing edges Σ are empty. The local invariant for a channel $\mathsf{Chan}(v)$ with value v states that the incoming edges α are the singleton set containing the protocol $!\Phi$, and that the outgoing edges Σ are owned by the protocol predicate $\Phi\ v$. The local invariant for a logical object that does not exist in the physical configuration, states that the incoming edges α and outgoing edges Σ are empty.

6.3 Weakest Preconditions

We have now defined the invariant, but we still need to define the weakest preconditions, which is the main challenge of proving adequacy. To do so, we first define a *partial invariant* $\mathsf{I}^\circ(\sigma, tid, \Sigma)$, which states that the invariant I holds for all threads and channels in the configuration σ, except for the thread *tid* that our weakest precondition is considering:

$$\mathsf{I}^\circ(\sigma, tid, \Sigma) \triangleq \exists G.\ \forall v.\ \begin{cases} \mathsf{in\_labels}_G(v) = \emptyset \land \mathsf{out\_edges}_G(v) = \Sigma & \text{if } v = \mathsf{Thread}(tid) \\ \mathsf{local\_inv}(\sigma[v], \mathsf{in\_labels}_G(v), \mathsf{out\_edges}_G(v)) & \text{if } v \neq \mathsf{Thread}(tid) \end{cases}$$

The partial invariant $\mathsf{I}^\circ(\sigma, tid, \Sigma)$ states that there is a strongly acyclic connectivity graph G, and that for every node v, the local invariant $\mathsf{local\_inv}$ holds, except for thread *tid*, for which we require that the incoming edges are empty (as threads are never owned), and the outgoing edges are Σ.

Using the partial invariant, we define the *primitive* weakest precondition $\mathsf{WP}_0\ e\ \{\Phi\}$ by cases depending on whether the expression e is a value or not:

$$\begin{aligned} \mathsf{WP}_0\ v\ \{\Phi\}\ \Sigma &\triangleq\ \diamond\,(\Phi\ v\ \Sigma) \\ \mathsf{WP}_0\ e\ \{\Phi\}\ \Sigma &\triangleq \forall tid, \vec{e}, h.\ \triangleright\,\mathsf{I}^\circ((\vec{e}, h), tid, \Sigma) \to \\ &\qquad \diamond\,(\mathsf{reducible\_or\_blocked}(e, h, \Sigma) \land \mathsf{preserved}(e, \vec{e}, h, tid, \Phi)) \\ \mathsf{preserved}(e, \vec{e}, h, tid, \Phi) &\triangleq \forall e', h', \vec{e}_{\mathsf{new}}.\ (e, h) \to_{\mathsf{p}} (e', h', \vec{e}_{\mathsf{new}}) \to \\ &\qquad \triangleright\,\exists \Sigma'.\ \mathsf{I}^\circ((\vec{e} +\!\!+ \vec{e}_{\mathsf{new}}, h'), tid, \Sigma') \land \mathsf{WP}_0\ e'\ \{\Phi\}\ \Sigma' \end{aligned}$$

This definition states that if the expression is a value v, then the weakest precondition holds if the predicate Φ holds for the value v (for technical step-indexing reasons, there is a $\diamond$ modality in front of the predicate, to allow us to remove $\triangleright$ from pure assumptions [Jung et al. 2018b, §5.7]). If the expression e is not a value, then we operate under the assumption that the partial invariant $\mathsf{I}^\circ((\vec{e}, h), tid, \Sigma)$ holds (under the later modality). We must then show that the expression is either reducible or blocked, expressed by the predicate $\mathsf{reducible\_or\_blocked}(e, h, \Sigma)$. This means that e can either step in the context of the heap h, or that e is blocked on a receive operation on a channel for which Σ contains the $?\Phi$ protocol. Secondly, we must show that the invariant and weakest precondition are preserved: if e steps to e', then we must find outgoing edges Σ' such that the partial invariant holds for the new configuration $(\vec{e} +\!\!+ \vec{e}_{\mathsf{new}}, h')$, and the weakest precondition holds for e' under Σ'. Here, $\vec{e}_{\mathsf{new}}$ is the list of new threads that are spawned by the step, and Σ' are the new outgoing edges that are owned by the current thread *tid*.

Recursion. The reader may have noticed that the definition of the weakest precondition $\mathsf{WP}_0\ e\ \{\Phi\}$ and the partial invariant $\mathsf{I}^\circ(\sigma, tid, \Sigma)$ are mutually recursive, and recursive occurrences appear in negative position. We use Iris's guarded fixed-point operator [Jung et al. 2018b, §5.6], which requires all recursive occurrences to appear under a later modality ($\triangleright$).

Framing. The frame rule of separation logic does not hold for $\mathsf{WP}_0\ e\ \{\Phi\}$. Thus, to obtain the LinearActris weakest precondition, we define it as the *frame-preserving* closure [Charguéraud 2020] of the primitive weakest precondition, which satisfies the frame rule:

$$\mathsf{WP}\ e\ \{\Phi\} \triangleq \forall R.\ \overset{?}{\triangleright} R \twoheadrightarrow \mathsf{WP}_0\ e\ \{v.\ R * \Phi\ v\}$$

In this definition, there is a later modality ($\triangleright$) in front of R, but only if e is not a value. This makes sure that we get the *step-framing* rule of Iris: $\triangleright R * \mathsf{WP}\ e\ \{\Phi\} \vdash \mathsf{WP}\ e\ \{v.\ R * \Phi\ v\}$ if $e \notin \mathsf{Val}$.

6.4 Weakest Precondition Rules and Adequacy

With the definition of the weakest precondition connective at hand, we prove the weakest precondition rules of LinearActris. These proofs are relatively complex, as we need to reason about the connectivity graph, and how it is transformed when we perform a step, as shown in Fig. 6.

The adequacy proof (Theorem 5.4) follows the structure sketched in §5, by proving the initialization, preservation, and progress theorems. For the progress theorem, we use the fact that the connectivity graph is acyclic, which means that we can always find a thread that can step. Formally, we apply the principle of *waiting induction* [Jacobs et al. 2022a]. We refer the interested reader to the Coq mechanization for the full details [Jacobs et al. 2023a].

7 SEMANTIC TYPING

This section shows that every program that can be shown to be deadlock free using an advanced session type system, can also be shown to be deadlock free using LinearActris. We prove this result using the "logical approach" to semantic typing [Appel et al. 2007; Dreyer et al. 2011; Jung et al. 2018a; Timany et al. 2022]. In short, we translate a typing derivation of $\vdash e : A$ from a syntactic session type system into a proof of $\mathsf{wp}\ e\ \{[\![A]\!]\}$ in LinearActris, where $[\![A]\!] \in \mathsf{Val} \to \mathsf{aProp}$ is the *semantic interpretation* of the syntactic type A. Due to our adequacy theorem for weakest preconditions (Theorem 5.4), we obtain the corollary that all well-typed programs in the syntactic session type system are deadlock and leak free when executed. This theorem is quite easy to prove, as our program logic does all the heavy lifting.

Our session type system is inspired by the GV family [Wadler 2012; Gay and Vasconcelos 2010], but uses strong updates to track changes to the session types of channels. Moreover, our type system is more expressive than earlier deadlock-free type systems that have appeared in the literature: it supports the combination of session-typed channels with recursive types, subtyping, term- and session type polymorphism, and unique mutable references.

We give a brief overview of the key ingredient of semantic typing (§7.1) before applying it to session types (§7.2).

7.1 Semantic Typing in a Nutshell

A type system involves a set of types $A \in \mathsf{Type}$ and a typing judgment $\vdash e : A$ (we omit typing contexts for brevity). Conventionally both notions are defined syntactically, *i.e.,* the set of types is defined as an inductive type where each constructor corresponds to a type former, and the typing judgment is defined as an inductive relation where each constructor corresponds to a typing rule. The key property of a type system for deadlock freedom is:

Corollary 7.1 (Syntactic type soundness). *If the typing judgment $\vdash e : A$ is derivable for a base type A (integer, boolean), then e is deadlock and leak free.*

To prove this property using semantic typing one carries out the following steps:

(1) Define the *semantic interpretation* $[\![A]\!] \in \mathsf{Val} \to \mathsf{aProp}$ of each syntactic type A. One should think of $[\![A]\!]$ as being the set of values of type A. However, $[\![A]\!]$ is not an ordinary set, but a predicate in separation logic, making it possible to give a meaningful interpretation to types that describe stateful objects, in our case, session types that describe channels.

Term types:

$$\begin{aligned} \mathsf{any} &\triangleq \lambda w.\, \mathsf{Emp} \\ \mathbf{Z} &\triangleq \lambda w.\, w \in \mathbb{Z} \\ A \multimap B &\triangleq \lambda w.\, \forall v.\, \triangleright(A\, v) \mathrel{-\!\!*} \mathsf{wp}\ (w\ v)\ \{B\} \\ \mathsf{chan}\ S &\triangleq \lambda w.\, w \rightarrowtail S \end{aligned}$$

Session types:

$$\begin{aligned} !A.\, S &\triangleq\ !(v : \mathsf{Val})\langle v\rangle\{A\, v\};\ S \\ ?A.\, S &\triangleq\ ?(v : \mathsf{Val})\langle v\rangle\{A\, v\};\ S \end{aligned}$$

Subtyping:

$$\begin{aligned} A <: B &\triangleq \forall v.\, A\, v \mathrel{-\!\!*} B\, v \\ S <: T &\triangleq S \sqsubseteq T \end{aligned}$$

Judgments:

$$\begin{aligned} \Gamma \vDash \sigma &\triangleq \mathop{\huge *}_{(x,A)\in\Gamma} .\ A(\sigma(x)) \\ \Gamma \vDash e : A \dashv \Gamma' &\triangleq \forall \sigma.\, (\Gamma \vDash \sigma) \mathrel{-\!\!*} \\ &\qquad \mathsf{wp}\ e[\sigma]\ \{v.\, A\, v * (\Gamma' \vDash \sigma)\} \end{aligned}$$

Semantic typing rules for terms:

$$\frac{\Gamma_1, x : A \vDash e : B \dashv [\,]}{\Gamma_1 \cdot \Gamma_2 \vDash \lambda x.\, e : A \multimap B \dashv \Gamma_2}$$

$$\frac{\Gamma_1 \vDash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2, x : A \vDash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vDash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : B \dashv \Gamma_3 \setminus x}$$

Semantic typing rules for channels:

$$\frac{\Gamma_1 \vDash e : \mathsf{chan}\ \overline{S} \multimap \mathsf{any} \dashv [\,]}{\Gamma_1 \cdot \Gamma_2 \vDash \mathbf{fork}\ e : \mathsf{chan}\ S \dashv \Gamma_2}$$

$$\frac{\Gamma \vDash e : A \dashv \Gamma', x : \mathsf{chan}\ (!A.\, S)}{\Gamma \vDash \mathbf{send}\ x\ e : \mathbf{1} \dashv \Gamma', x : \mathsf{chan}\ S}$$

$$\frac{}{\Gamma, x : \mathsf{chan}\ (?A.\, S) \vDash \mathbf{recv}\ x : A \dashv \Gamma, x : \mathsf{chan}\ S}$$

Fig. 7. Typing judgements and type formers of the semantic type system.

(2) Define the *semantic typing judgment* $\vDash e : A$ in terms of the interpretation of types. This judgment roughly says that e is deadlock and leak free, and when e terminates with value v, it satisfies $[\![A]\!]\ v$. With a program logic at hand, the semantic typing judgment can simply be defined using the weakest precondition, *i.e.*, $\vDash e : A \triangleq \mathsf{wp}\ e\ \{[\![A]\!]\}$.
(3) Prove the *fundamental theorem*, which says that every expression that is syntactically typed is also semantically typed. That is, $\vdash e : A$ implies $\vDash e : A$. The fundamental theorem is proved by induction on the typing derivation $\vdash e : A$. This means that for each syntactic typing rule (with $\vdash$) we have to prove a semantic version (with $\vDash$). The semantic typing rules are proved using the corresponding weakest precondition rules.

The fundamental theorem almost immediately gives syntactic type soundness. Since the semantic typing judgment is defined in terms of weakest preconditions, it gives us deadlock and leak freedom by adequacy (§7.1). Hence, when composing the fundamental theorem and adequacy, we obtain that every syntactically typed expression is deadlock and leak free, *i.e.*, Corollary 7.1.

To streamline this development, we follow the *foundational approach* to semantic typing, inspired by Appel and McAllester [2001]; Ahmed [2004]; Ahmed et al. [2010]; Jung et al. [2018a]. We omit the syntactic definition of the type system, and immediately define types and the typing judgment in terms of their semantics. Type formers are simply combinators on semantic types, and typing rules are simply lemmas about the semantic typing judgment.

7.2 Type System

An overview of the key definitions appears in Fig. 7. We omit details about unique mutable references, polymorphism, and copy (a.k.a. unrestricted) types for brevity's sake, and refer the interested reader to our Coq mechanization and the affine type system which we adopted [Hinrichsen et al. 2021], as the details revolving these aspects are mostly unchanged.

Type formers. The type system consists of two kinds of types, term types and session types. We have the usual linear term type constructs such as any, $\mathbf{Z}$, and $A \multimap B$, in addition to the channel type $\mathsf{chan}\ S$, which is parametric on a session type S. We support the usual session types such as $\mathbf{!}A.\,S$ and $\mathbf{?}A.\,S$, as well as the ones for closing and branching (omitted for brevity's sake).

In a semantic type system, term types are defined as propositions over values ($\mathsf{Type} \triangleq \mathsf{Val} \to \mathsf{aProp}$). For example, the type $\mathsf{chan}\ S$ is defined in terms of the channel ownership $c \rightarrowtail S$. Session types S are defined using our dependent protocols p. We use the protocol binders to capture that channels exchange values v for which the term type predicate A holds.

Typing judgment. As we work with a language with strong updates, we use a typing judgment $\Gamma \vDash e : A \dashv \Gamma'$ with a pre- *and* post-context $\Gamma, \Gamma' \in \mathsf{List}(\mathsf{String} \times \mathsf{Type})$, similar to RustBelt [Jung et al. 2018a]. Using the post-context can track how types of variables change throughout evaluation.

We use *closing substitutions* to define our typing contexts, as is standard in logical relation models. Closing substitutions $\sigma \in \mathsf{String} \xrightarrow{\text{fin}} \mathsf{Val}$ are finite partial functions that map the free variables of an expression to corresponding values. Closing substitutions come with a judgment $\Gamma \vDash \sigma$, which expresses that the closing substitution σ is well-typed in the context Γ. The judgment says that for every typed variable $(x, A) \in \Gamma$ there is a corresponding value in the closing substitution $\sigma(x)$, for which we own the resources $A(\sigma(x))$.

The typing judgment $\Gamma \vDash e : A \dashv \Gamma'$ is defined using our weakest precondition. That is, given a closing substitution σ and resources $\Gamma \vDash \sigma$ for the pre-context Γ, the weakest precondition holds for e (under substitution with σ), with the postcondition stating that the resources $A\,v$ for the resulting value v are owned separately from the resources $\Gamma' \vDash \sigma$ for the post-context Γ'.

Typing rules. In a semantic type system, every typing rule corresponds to a lemma, which states that if the premises hold semantically, then the conclusion holds semantically. These lemmas are proved using the rules of LinearActris, by unfolding the typing judgment and the type formers, which yields goals that are directly provable using the corresponding weakest precondition rules.

Semantic type soundness. As the semantic typing judgment is defined in terms of weakest precondition, we obtain a type soundness theorem as a direct corollary of adequacy (Theorem 5.4).

Theorem 7.2 (Semantic type soundness ⚙). *If* $[\,] \vDash e : \mathsf{any} \dashv [\,]$ *holds, and* $([e], \emptyset) \to_t^* (\vec{e}, h)$, *then either* $(\vec{e}, h)$ *can step, or* $\vec{e}$ *are all values and* $h = \emptyset$.

This theorem says that our type systems ensures there are no deadlocks and leaks. We obtain a similar type soundness theorem for safety (no illegal non-blocking operations, such as use-after-free) using LinearActris's safety theorem (Theorem 5.5).

8 COQ MECHANIZATION AND EVALUATION

All definitions, theorems, and examples in this paper have been mechanized in Coq using the Iris framework.[2] The full sources are available in our artifact [Jacobs et al. 2023a].

The components of our mechanization and the corresponding line counts are displayed in Table 1. The definition of ChanLang includes the syntax and small-step operational semantics. LinearActris is built on top of an abstract separation logic modeled as step-indexed predicates over a partial commutative monoid (PCM). The abstract separation logic is adapted from Iris, with the crucial difference that Iris considers monotone predicates to obtain an affine logic, while our logic is linear. The mechanization of connectivity graphs is adapted from Jacobs et al. [2022a], but many lemmas had to be ported to the step-indexed setting (*i.e.,* from Coq's Prop to siProp). To construct the propositions aProp of LinearActris we instantiate our abstract separation logic with the PCM of finite maps with disjoint union, and subsequently use Iris's solver for recursive domain equations.

[2]To ease working in a pure-step indexed logic (siProp), we are using a modified version of Iris that attempts to address https://gitlab.mpi-sws.org/iris/iris/-/issues/420. We will upstream our modifications of Iris in future work.

Component	**Section**	**LOC**
Definition of ChanLang with one-shot channels	§2, §4	757
Abstract linear separation logic	§6	693
Connectivity graphs	§5	2350
LinearActris propositions aProp	§6	704
Weakest preconditions and adequacy for one-shot channels	§3, §6	1474
Encoding of the multi-shot logic	§4	1190
Semantic typing	§7	2020
Tactics	§8	568
Miscellaneous	-	379
Examples:		
• Basic examples from this paper	§2	179
• Basic examples from Hinrichsen et al. [2022]	§8	654
• Merge sort from Hinrichsen et al. [2022]	§8	585
• Semantic typing examples from Hinrichsen et al. [2021]	§8	324
Total		11877

Table 1. Overview of the LinearActris Coq mechanization

With these components at hand, we define the weakest precondition connective and prove adequacy for the one-shot logic, and finally define the multi-shot logic and our semantic typing system.

Tactics. Inspired by the original Actris papers, we provide custom tactics for symbolic execution and reasoning about subprotocols. These tactics are built on top of the Iris Proof Mode [Krebbers et al. 2017b, 2018], and are used for the verification of all of our examples.

Basic examples. We mechanize Hoare triples for examples presented in §2, as well as the examples presented in §1 of the Actris 2.0 paper [Hinrichsen et al. 2022]. The proofs of these Hoare triples are the same as those in the original Actris mechanization, but due to our strong adequacy theorem we obtain that these examples satisfy deadlock and leak freedom "for free".

Merge sort examples. We verify the Hoare triples for all merge sort examples from §5 of the Actris 2.0 paper [Hinrichsen et al. 2022]. This includes choice, recursion, higher-order quantification (for generic sorting functions), and delegation. The most advanced version of merge sort recursively creates new child processes for handling the divide-and-conquer part of the merge sort algorithm, and sends the list element-by-element from the parent process to the child processes. We use the following dependent separation protocols, which use the encodings of the choice protocols ($\oplus, \&$) presented in Hinrichsen et al. [2022, §5.3]:

$$\mathsf{p}_{sort} \triangleq \mathsf{p}^{head}_{sort}\ \epsilon \qquad \mathsf{p}^{head}_{sort}\ \vec{x} \triangleq (\,\mathbf{!}\,(x : T,\ v : \mathsf{Val})\langle v\rangle\{I\ x\ v\};\ \mathsf{p}^{head}_{sort}\ (\vec{x} \cdot [x]))\ \oplus\ \mathsf{p}^{tail}_{sort}\ \vec{x}\ \epsilon$$

$$\mathsf{p}^{tail}_{sort}\ \vec{x}\ \vec{y} \triangleq (\,\mathbf{?}(y : T, v : \mathsf{Val})\langle v\rangle\{(\forall i < |\vec{y}|.\ \vec{y}_i \le y) * I\ y\ v\};\ \mathsf{p}^{tail}_{sort}\ \vec{x}\ (\vec{y} \cdot [y]))\ \&_{\{\vec{x} \equiv_{\mathsf{p}} \vec{y}\}}\ \mathbf{!end}$$

Given a total order $(T, \le)$ and an interpretation predicate $I : T \to \mathsf{Val} \to \mathsf{aProp}$, the protocol p_{sort} expresses that the input list is sent, and the sorted list is sent back. The auxiliary protocol $\mathsf{p}^{head}_{sort}\ \vec{x}$ is used for sending the input list, where the parameter $\vec{x}$ keeps track of the elements that have been sent so far. At every iteration, there is a choice ($\oplus$) between (1) sending more elements, and (2) indicating that the whole input list has been sent. The auxiliary protocol $\mathsf{p}^{tail}_{sort}\ \vec{x}\ \vec{y}$ is used for receiving back the sorted list, where the parameter $\vec{y}$ keeps track of the elements that have been received so far. At every iteration, there is a branch (&) between (1) receiving more elements, and (2) protocol termination. The conditions in the protocol ensure that the elements of $\vec{y}$ are returned in sorted fashion, and that when terminated, the resulting list $\vec{y}$ is a permutation of the input list $\vec{x}$.

Semantic typing examples. Inspired by Hinrichsen et al. [2021] we show that the type system from § 7 can be used to type check a computation service that uses choice, recursion, and polymorphism. We use the following session type:

$$\mathsf{S}_{\texttt{compute}} \triangleq (?_{(T:\bigstar)}\,(() \multimap T).\,!T.\,\mathsf{S}_{\texttt{compute}})\ \&\ ?\mathbf{end}$$

The session type says that a client can repeatedly send computations $() \multimap T$ to a service, which returns the result T of forcing the computation. Due to the support for polymorphism, it is possible to send a computation $() \multimap T$ with a different type T in each iteration of the protocol.

9 RELATED WORK

We first discuss other approaches to prove deadlock freedom (§ 9.1), then discuss mechanizations of session types (§ 9.2), and channel implementations (§ 9.3). Finally, we discuss related work on models of separation logic (§ 9.4).

9.1 Proof Methods for Deadlock Freedom

Linear session types. The GV type system [Wadler 2012; Gay and Vasconcelos 2010] and follow-up work [Lindley and Morris 2015, 2016, 2017; Fowler et al. 2019, 2021] ensure deadlock freedom for a functional language with session types by linearity. Earlier work proved deadlock freedom for a linear π-calculus using a graphical approach [Carbone and Debois 2010]. Toninho et al. [2013]; Toninho [2015]'s deadlock-free SILL embeds session-typed processes into a functional language via a monad. Like GV, the seminal paper by Caires and Pfenning [2010] and Toninho [2015]'s PhD thesis spurred a series of derivatives [Caires et al. 2013; Pérez et al. 2014; Das et al. 2018], in which deadlock freedom is guaranteed by linearity. The contribution of our work is to obtain deadlock freedom from linearity in separation logic instead of a type system.

Multiparty session types. Multiparty session types [Honda et al. 2008, 2016] generalize session types from bidirectional channels to n-to-n channels. To ensure deadlock freedom, multiparty session type systems use a consistency check that generalizes the duality condition of binary session types. The consistency check can be performed via projections of a global type, or via an explicit check on a collection of local types [Scalas and Yoshida 2019]. Purely multiparty approaches generally assume a static topology, and thus do not support dynamic creation of threads and channels. This makes them orthogonal in the programs they can establish to be deadlock free compared to linear binary session types (hybrid approaches exist, see below).

Lock orders. Dijkstra originally proposed lock orders as a mechanism to ensure deadlock freedom for his Dining Philosophers problem [Dijkstra 1971]. Lock orders have been incorporated into a number of verification tools and separation logics that support proving deadlock freedom, for example [Leino et al. 2010; Le et al. 2013; Zhang et al. 2016; Hamin and Jacobs 2018]. Lock-order based approaches are orthogonal in expressive strength compared to session types. For instance, it is far from clear how to build a logical relation for a language with session types in terms of a separation logic with lock orders. In the session-typed source language, deadlock freedom is ensured by linearity, and it does not seem possible to translate this into order-based reasoning in the target program logic. Since session types do not have order obligations, it is not clear how the order conditions on the receive operations are justified.

LiLi [Liang and Feng 2016, 2018] is a program logic for proving liveness (and therein deadlock freedom) of concurrent objects using a method they call *definite actions*, which is conceptually similar to lock orders. The definite actions act as obligations which can be used to impose an order of use of multiple (encoded) blocking operations, such as acquisition and releases of locks. The TaDA Live separation logic for proving liveness of concurrent programs [D'Osualdo et al. 2021] uses a similar concept called layers. Similar to LiLi—and thereby lock orders—these layers form a hierarchy of conditions, which can be used to encode an ordering between locks.

Choreographies. Choreographic languages [Montesi 2021; Cruz-Filipe et al. 2021b,a,b] allow the programmer to write a global program that is automatically split into local programs that communicate via channels for which deadlock freedom is guaranteed by construction. Since choreographies are based on program generation, they are very different from our approach.

Usages and obligations. Yet another mechanism for deadlock freedom are usages and obligations [Kobayashi 1997; Igarashi and Kobayashi 1997; Kobayashi et al. 1999; Igarashi and Kobayashi 2001; Kobayashi 2002; Igarashi and Kobayashi 2004], which ensure that channels are used in a non-deadlocking order. In contrast to lock orders, the priority involved in usages and obligations always increases in the order. These mechanisms have also been extended to session-typed languages [Dardha and Gay 2018]. Similar to lock orders, usages and obligations entail additional proof obligations, and as such, are orthogonal to obtaining deadlock freedom from linearity.

Hybrid approaches. Message passing has been extended with locks and sharing [Benton 1994; Villard et al. 2009; Reed 2009b; Lozes and Villard 2011, 2012; Pfenning and Griffith 2015; Balzer et al. 2018, 2019; Hinrichsen et al. 2020; Qian et al. 2021; Rocha and Caires 2021; Jacobs and Balzer 2023]. Some of these approaches ensure deadlock or leak freedom, *e.g.,* via lock orders, linearity, or other checks. Multiparty session types have been combined with linearity to guarantee progress beyond one session [Carbone et al. 2015, 2016, 2017; Jacobs et al. 2022b]. In this paper we used bidirectional channels (built on top of one-shot channels) as the sole concurrency primitive. In future work, we would like to add locks and multiparty session types, inspired by the preceding work (§10).

9.2 Mechanization of Session Types

Hinrichsen et al. [2021] use Actris to prove soundness of a session type system via the method of semantic typing, inspired by RustBelt [Jung et al. 2018a]. We follow a similar approach, but in addition to proving type safety, we prove deadlock and leak freedom. Thiemann [2019] proves type safety of a linear GV-like session type system using dependent types in Agda, Rouvoet et al. [2020] streamline this approach via separation logic. Goto et al. [2016]; Ciccone and Padovani [2020]; Castro-Perez et al. [2020]; Reed [2009a]; Chaudhuri et al. [2019] mechanize π-calculus with session types. These works generally show safety, but Jacobs et al. [2022a]'s Coq mechanization shows deadlock freedom. We generalize their approach of connectivity graphs to the context of separation logic. Lastly, Castro-Perez et al. [2021]; Jacobs et al. [2022b] mechanize multiparty session types.

9.3 Verification of Message-Passing Implementations

While channels are a primitive of our operational semantics, others have verified message-passing implementations that use atomic primitives, such as compare-and-swap or atomic-exchange. Mansky et al. [2017] verifies a message-passing system written in C using VST [Appel 2014; Cao et al. 2018]. Tassarotti et al. [2017] proves the correctness of a compiler for an affine session-typed language, showing that the target terminates iff the source program terminates (under fair scheduling assumptions). In the future, we would like to implement our channels using atomic primitives. In this setting, it is less clear how to formulate the adequacy theorem. As low-level implementations of channels perform busy loops, we would need to model deadlock freedom as a liveness property such as progress under fair scheduling (§10).

Recent work applies Actris to obtain reliable message-passing specifications for channels built on top of UDP-like primitives [Gondelman et al. 2023]. Similarly to the shared memory setting, the implementation busy loops until a message has been successfully transferred over the unreliable network, which can only be guaranteed under fair scheduling and a fair network.

9.4 Linear Models of Separation Logic

The original presentations of sequential separation logic [O'Hearn et al. 2001] and concurrent separation logic (CSL) [O'Hearn 2004; Brookes 2004] use a linear model. For sequential separation logic, linearity gives leak freedom, and with scoped CSL-style invariants this scales to concurrent programs that use parallel composition. When extending the language with more general invariant mechanisms that support unscoped thread creation [Hobor et al. 2008; Svendsen and Birkedal 2014] the situation becomes more complicated. Jung [2020, Thm 2] shows that linearity alone does not give leak freedom, and other mechanisms are needed. The Iron logic [Bizjak et al. 2019] provides such a mechanism: by disallowing deallocation permissions in invariants, leak freedom can be obtained. Unfortunately, ownership of the **end** protocol needs to include permission to deallocate the channel, making Iron's invariants insufficient for higher-order session types.

While all resources in Iris are affine, and all resources in LinearActris are linear, there have been various efforts to make hybrid models of separation logics that have both linear and affine resources [Tassarotti et al. 2017; Cao et al. 2017; Krebbers et al. 2018; Mansky 2022]. Typically they use some form of partial commutative monoids equipped with an order that specifies which resources can be dropped. The model of LinearActris is an instance of the step-indexed ordered resource algebra model by Krebbers et al. [2018], taking the order to be the reflexive relation, meaning no resources can be dropped. An interesting direction for future work is to add a notion of ghost state to LinearActris, for which these hybrid models could be useful.

10 CONCLUSION AND FUTURE WORK

The key strength of LinearActris is deadlock and leak freedom "for free" from linearity, while being otherwise very close to the original Actris logic [Hinrichsen et al. 2020, 2022]. As such, we are able to verify example programs that use higher-order channels (sending channels over channels as messages), higher-order functions (passing closures as arguments to other closures, and sending closures over channels as messages), and higher-order store (sending references over channels as messages), as illustrated in §2. As an added benefit of LinearActris being close to the Actris logic, we were able to port most of the examples from the original Actris papers to LinearActris (§8).

We now discuss some limitations of LinearActris and directions for future work.

Asynchronous subtyping. Actris 2.0 [Hinrichsen et al. 2022] supports asynchronous subtyping of channels, which allows the subtyping rule $?\langle v\rangle;\ !\langle w\rangle;\ p \sqsubseteq\ !\langle w\rangle;\ ?\langle v\rangle;\ p$. This rule allows the user of a channel to perform send steps ahead of time. The reason why this rule is sound in the original Actris framework, is that channels have two separate buffers, one for sending and one for receiving. In the LinearActris logic, we only have one buffer, and messages must enter this buffer in the order specified in the protocol, and hence we cannot support asynchronous subtyping. We believe we could support asynchronous subtyping if we add a second buffer to channels. However, this would introduce complications that are orthogonal to the main contributions of this paper, as we can no longer use the single-shot buffer encoding of channels by Jacobs et al. [2023b].

Other concurrency constructs. LinearActris is designed for message-passing concurrency, and does not support other concurrency constructs such as locks, semaphores, or monitors. The original

Actris logic supports these constructs, in particular, it employs locks to model sharable channels, inspired by manifest sharing in session types [Balzer et al. 2018, 2019]. In the original Actris, these constructs are implemented using busy loops and verified using Iris's mechanisms for ghost state and invariants. When stating deadlock freedom using global progress, it is significantly more complicated to add other concurrency constructs. To ensure that deadlocks can be distinguished from ordinary loops, one would need to add such constructs as primitive blocking operations, and they need to be explicitly handled as part of the connectivity graph. In future work, we would like to pursue this direction. Our reason for believing this to be possible, is that the connectivity graph-based approach to deadlock freedom has been designed to be flexible in the kind of concurrency constructs, and has already been applied to a type system for locks [Jacobs and Balzer 2023].

Multiparty communication. We would like to extend the LinearActris logic with multiparty communication inspired by multiparty session types [Honda et al. 2008]. In prior work, Jacobs et al. [2022b] used connectivity graphs to prove deadlock freedom of a session type system that combines GV-style dynamic thread and channel spawning with multiparty session types. However, we believe that extending these results to separation logic is non-trivial, even without considering deadlock and leak freedom. In particular, it is not clear how global types could be generalized to Actris-style dependent separation protocols.

Liveness. LinearActris guarantees deadlock freedom, but does not guarantee liveness. Deadlock freedom (stated as global progress—the standard way of formulating this property in the session types literature [Caires and Pfenning 2010; Carbone and Debois 2010; Wadler 2012]) means that the program as a whole cannot get stuck on message receives indefinitely, but does not guarantee that the program will eventually terminate or produce a result. In particular, deadlock freedom does not rule out infinite loops written by the user. To guarantee liveness, one needs to prove that loops in the program eventually terminate, or produce a result that counts as progress, and prove that the program cannot get stuck in other ways, such as by waiting for a message that will never arrive. In future work, we plan to investigate whether the LinearActris logic can be extended to guarantee liveness for higher-order message passing, by taking inspiration from liveness logics such as LiLi [Liang and Feng 2016] and TaDa Live [D'Osualdo et al. 2021], and existing work on termination and liveness in Iris [Tassarotti et al. 2017; Spies et al. 2021].

Iris invariants. We hope that our work can be a step towards bringing deadlock and leak freedom to full-fledged separation logics for fine-grained concurrency with Iris-style impredicative invariants [Svendsen and Birkedal 2014]. Recent progress has been made for leak freedom [Bizjak et al. 2019], and termination, as well as termination-preserving refinement [Spies et al. 2021; Tassarotti et al. 2017]. Nevertheless, key challenges related to Iris-style invariants remain. As channels can be seen as a particular type of invariant, we hope that our connectivity graph approach can be generalized, *e.g.,* to a linear form of invariants that are compatible with leak- and deadlock freedom.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV).

DATA AVAILABILITY STATEMENT

The Coq development for this paper can be found in Jacobs et al. [2023a].

REFERENCES

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

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *TOPLAS* (2010). https://doi.org/10.1145/1709093.1709094

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

Andrew W. Appel. 2014. *Program Logics for Certified Compilers.* https://doi.org/10.1017/CBO9781107256552

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *TOPLAS* (2001). https://doi.org/10.1145/504709.504712

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.* https://doi.org/10.1145/1190216.1190235

Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. 2018. A Universal Session Type for Untyped Asynchronous Communication. In *CONCUR.* https://doi.org/10.4230/LIPIcs.CONCUR.2018.30

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *ESOP.* https://doi.org/10.1007/978-3-030-17184-1_22

Nick Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract). In *CSL.* https://doi.org/10.1007/BFb0022251

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed kripke models over recursive worlds. In *POPL.* https://doi.org/10.1145/1926385.1926401

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. The category-theoretic solution of recursive metric-space equations. *TCS* (2010). https://doi.org/10.1016/j.tcs.2010.07.010

Ales Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing obligations in higher-order concurrent separation logic. POPL (2019). https://doi.org/10.1145/3290378

Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *CONCUR.* https://doi.org/10.1007/978-3-642-15375-4_12

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR.* https://doi.org/10.1007/978-3-540-28644-8_2

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *ESOP.* https://doi.org/10.1007/978-3-642-37036-6_19

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR.* https://doi.org/10.1007/978-3-642-15375-4_16

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *JAR* (2018). https://doi.org/10.1007/s10817-018-9457-5

Qinxiang Cao, Santiago Cuellar, and Andrew W. Appel. 2017. Bringing Order to the Separation Logic Jungle. In *APLAS.* https://doi.org/10.1007/978-3-319-71237-6_10

Marco Carbone and Søren Debois. 2010. A Graphical Approach to Progress for Structured Communication in Web Services. In *ICE.* https://doi.org/10.4204/EPTCS.38.4

Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *CONCUR.* https://doi.org/10.4230/LIPIcs.CONCUR.2016.33

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2015. Multiparty Session Types as Coherence Proofs. In *CONCUR.* https://doi.org/10.4230/LIPIcs.CONCUR.2015.412

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2017. Multiparty session types as coherence proofs. *Acta Informatica* (2017). https://doi.org/10.1007/s00236-016-0285-y

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.* 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.* https://doi.org/10.1007/978-3-030-45237-7_17

Arthur Charguéraud. 2020. Separation logic for sequential programs (functional pearl). ICFP (2020). https://doi.org/10.1145/3408998

Kaustuv Chaudhuri, Leonardo Lima, and Giselle Reis. 2019. Formalized Meta-Theory of Sequent Calculi for Linear Logics. *TCS* (2019). https://doi.org/10.1016/j.tcs.2019.02.023

Adam Chlipala. 2013. The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ICFP.* https://doi.org/10.1145/2500365.2500592

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

Florin Craciun, Tibor Kiss, and Andreea Costea. 2015. Towards a Session Logic for Communication Protocols. In *ICECCS.* https://doi.org/10.1109/ICECCS.2015.33

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021a. Certifying Choreography Compilation. In *ICTAC*. https://doi.org/10.1007/978-3-030-85315-0_8

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021b. Formalising a Turing-Complete Choreographic Language in Coq. In *ITP*. https://doi.org/10.4230/LIPIcs.ITP.2021.15

Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *FOSSACS*. https://doi.org/10.1007/978-3-319-89366-2_5

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

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *LICS*. https://doi.org/10.1145/3209108.3209146

Edsger W. Dijkstra. 1971. Hierarchical Ordering of Sequential Processes. *Acta Informatica* (1971). https://doi.org/10.1007/BF00289519

Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. *TOPLAS* (2021). https://doi.org/10.1145/3477082

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical step-indexed logical relations. *LMCS* (2011). https://doi.org/10.2168/LMCS-7(2:16)2011

Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. 2021. Separating Sessions Smoothly. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2021.36

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types Without Tiers. POPL (2019). https://doi.org/10.1145/3290341

Adrian Francalanza, Julian Rathke, and Vladimiro Sassone. 2011. Permission-Based Separation Logic for Message-Passing Concurrency. *LMCS* (2011). https://doi.org/10.2168/LMCS-7(3:7)2011

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *JFP* (2010). https://doi.org/10.1017/S0956796809990268

Leon Gondelman, Jonas Kastberg Hinrichsen, Marío Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. ICFP (2023). https://doi.org/10.1145/3607859

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

Jafar Hamin and Bart Jacobs. 2018. Deadlock-Free Monitors. In *ESOP*. https://doi.org/10.1007/978-3-319-89884-1_15

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.).

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-Type Based Reasoning in Separation Logic. POPL (2020). 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. *LMCS* (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*. https://doi.org/10.1145/3437992.3439914

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP*. https://doi.org/10.1007/978-3-540-78739-6_27

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR*. 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 *ESOP*. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *POPL*. https://doi.org/10.1145/1328438.1328472

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* (2016). https://doi.org/10.1145/2827695

Atsushi Igarashi and Naoki Kobayashi. 1997. Type-Based Analysis of Communication for Concurrent Programming Languages. In *SAS*. https://doi.org/10.1007/BFb0032742

Atsushi Igarashi and Naoki Kobayashi. 2001. A Generic Type System for the Pi-calculus. In *POPL*. https://doi.org/10.1145/360204.360215

Atsushi Igarashi and Naoki Kobayashi. 2004. A Generic Type System for the Pi-calculus. *TCS* (2004). https://doi.org/10.1016/S0304-3975(03)00325-6

Jules Jacobs. 2022. A Self-Dual Distillation of Session Types. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2022.23

Jules Jacobs and Stephanie Balzer. 2023. Higher-Order Leak and Deadlock Free Locks. POPL (2023). https://doi.org/10.1145/3571229

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022a. Connectivity graphs: a method for proving deadlock freedom based on separation logic. POPL (2022). https://doi.org/10.1145/3498662

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022b. Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom. ICFP (2022). https://doi.org/10.1145/3547638

Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2023a. Coq Mechanization of "Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing". https://doi.org/10.5281/zenodo.8415020

Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2023b. Dependent Session Protocols in Separation Logic from First Principles (Functional Pearl). ICFP (2023). https://doi.org/10.1145/3607856

Ralf Jung. 2020. *Understanding and Evolving the Rust Programming Language.* Ph. D. Dissertation. Universität des Saarlandes.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. POPL (2018). https://doi.org/10.1145/3158154

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

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* (2018). 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 *POPL*. https://doi.org/10.1145/2676726.2676980

Naoki Kobayashi. 1997. A Partially Deadlock-Free Typed Process Calculus. In *LICS*. https://doi.org/10.1109/LICS.1997.614941

Naoki Kobayashi. 2002. A Type System for Lock-Free Processes. *I&C* (2002). https://doi.org/10.1006/inco.2002.3171

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *TOPLAS* (1999). https://doi.org/10.1145/330249.330251

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. ICFP (2018). 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 *ESOP*. 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*. https://doi.org/10.1145/3009837.3009855

Duy-Khanh Le, Wei-Ngan Chin, and Yong Meng Teo. 2013. An Expressive Framework for Verifying Deadlock Freedom. In *ATVA*. https://doi.org/10.1007/978-3-319-02444-8_21

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2010. Deadlock-Free Channels and Locks. In *ESOP*. https://doi.org/10.1007/978-3-642-11957-6_22

Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *POPL*. https://doi.org/10.1145/2837614.2837635

Hongjin Liang and Xinyu Feng. 2018. Progress of concurrent objects with partial methods. POPL (2018). https://doi.org/10.1145/3158108

Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *ESOP*. https://doi.org/10.1007/978-3-662-46669-8_23

Sam Lindley and J. Garrett Morris. 2016. Talking Bananas: Structural Recursion For Session Types. In *ICFP*. https://doi.org/10.1145/2951913.2951921

Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools.*

Étienne Lozes and Jules Villard. 2011. Reliable Contracts for Unreliable Half-Duplex Communications. In *Web Services and Formal Methods (WS-FM)*. https://doi.org/10.1007/978-3-642-29834-9_2

Étienne Lozes and Jules Villard. 2012. Shared Contract-Obedient Endpoints. In *ICE*. https://doi.org/10.4204/EPTCS.104.3

William Mansky. 2022. Bringing Iris into the Verified Software Toolchain. https://doi.org/10.48550/arXiv.2207.06574

William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A verified messaging system. OOPSLA (2017). https://doi.org/10.1145/3133911

Fabrizio Montesi. 2021. Introduction to Choreographies. (2021). Accepted for publication by Cambridge University Press.

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

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR*. https://doi.org/10.1007/978-3-540-28644-8_4

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL*. https://doi.org/10.1007/3-540-44802-0_1

Wytse Oortwijn, Stefan Blom, and Marieke Huisman. 2016. Future-based Static Analysis of Message Passing Programs. In *PLACES*. https://doi.org/10.4204/EPTCS.211.7

Luca Padovani. 2014. Deadlock and lock freedom in the linear π-calculus. In *LICS*. https://doi.org/10.1145/2603088.2603116

Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear Logical Relations and Observational Equivalences for Session-Based Concurrency. *I&C* (2014). https://doi.org/10.1016/j.ic.2014.08.001

Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *FoSSaCS*. https://doi.org/10.1007/978-3-662-46678-0_1

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.).

Zesen Qian, G. A. Kavvos, and Lars Birkedal. 2021. Client-Server Sessions in Linear Logic. ICFP (2021). https://doi.org/10.1145/3473567

Jason Reed. 2009a. *A Hybrid Logical Framework*. Ph. D. Dissertation. Carnegie Mellon University.

Jason Reed. 2009b. A Judgmental Deconstruction of Modal Logic. (2009). http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf Unpublished manuscript.

Pedro Rocha and Luís Caires. 2021. *Propositions-as-Types and Shared State*. Technical Report. NOVA LINCS. https://doi.org/10.1145/3473584

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

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

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *PLDI*. https://doi.org/10.1145/3453483.3454031

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *ESOP*. 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*. https://doi.org/10.1007/978-3-662-54434-1_34

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

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2022. A Logical Approach to Type Soundness. Manuscript.

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. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *ESOP*. https://doi.org/10.1007/978-3-642-37036-6_20

Jules Villard, Étienne Lozes, and Cristiano Calcagno. 2009. Proving Copyless Message Passing. In *APLAS*. https://doi.org/10.1007/978-3-642-10672-9_15

Philip Wadler. 2012. Propositions as sessions. In *ICFP*. https://doi.org/10.1145/2364527.2364568

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *I&C* (1994). https://doi.org/10.1006/inco.1994.1093

Dan Zhang, Dragan Bosnacki, Mark van den Brand, Cornelis Huizing, Bart Jacobs, Ruurd Kuiper, and Anton Wijs. 2016. Verifying Atomicity Preservation and Deadlock Freedom of a Generic Shared Variable Mechanism Used in Model-To-Code Transformations. In *MODELSWARD*. https://doi.org/10.1007/978-3-319-66302-9_13

Received 2023-07-11; accepted 2023-11-07