ACTRIS 2.0: ASYNCHRONOUS SESSION-TYPE BASED REASONING IN SEPARATION LOGIC

JONAS KASTBERG HINRICHSEN, JESPER BENGTSON, AND ROBBERT KREBBERS

IT University of Copenhagen, Denmark
e-mail address: jkas@itu.dk

IT University of Copenhagen, Denmark
e-mail address: bengtson@itu.dk

Radboud University and Delft University of Technology, The Netherlands
e-mail address: mail@robbertkrebbers.nl

Abstract. Message passing is a useful abstraction for implementing concurrent programs. For real-world systems, however, it is often combined with other programming and concurrency paradigms, such as higher-order functions, mutable state, shared-memory concurrency, and locks. We present **Actris**: a logic for proving functional correctness of programs that use a combination of the aforementioned features. Actris combines the power of modern concurrent separation logics with a first-class protocol mechanism—based on session types—for reasoning about message passing in the presence of other concurrency paradigms. We show that Actris provides a suitable level of abstraction by proving functional correctness of a variety of examples, including a channel-based merge sort, a channel-based load-balancing mapper, and a variant of the map-reduce model, using concise specifications.

While Actris was already presented in a conference paper (POPL'20), this paper expands the prior presentation significantly. Moreover, it extends Actris to **Actris 2.0** with a notion of *subprotocols*—based on session-type subtyping—that permits additional flexibility when composing channel endpoints, and that takes full advantage of the asynchronous semantics of message passing in Actris. Soundness of Actris 2.0 is proven using a model of its protocol mechanism in the Iris framework. We have mechanised the theory of Actris, together with custom tactics, as well as all examples in the paper, in the Coq proof assistant.

1. Introduction

Message-passing programs are ubiquitous in modern computer systems, emphasising the importance of their functional correctness. Programming languages, like Erlang, Elixir, and Go, have built-in primitives that handle spawning of processes and intra-process communication, while other mainstream languages, such as Java, Scala, F#, and C#, have introduced an Actor model [Hewitt et al. 1973] to achieve similar functionality. In both cases the goal remains the same—help design reliable systems, often with close to constant up-time, using lightweight processes that can be spawned by the hundreds of thousands and that communicate via asynchronous message passing.

Key words and phrases: Message passing, actor model, concurrency, session types, Iris.

Preprint submitted to
Logical Methods in Computer Science

While message passing is a useful abstraction, it is not a silver bullet of concurrent programming. In a qualitative study of larger Scala projects Tasharofi et al. [2013] write:

> We studied 15 large, mature, and actively maintained actor programs written in Scala and found that 80% of them mix the actor model with another concurrency model.

In this study, 12 out of 15 projects did not entirely stick to the Actor model, hinting that even for projects that embrace message passing, low-level concurrency primitives like locks (*i.e.,* mutexes) still have their place. Tu et al. [2019] came to a similar conclusion when studying 6 large and popular Go programs. A suitable solution for reasoning about message-passing programs should thus integrate with other programming and concurrency paradigms.

In this paper we introduce **Actris**—a concurrent separation logic for proving functional correctness of programs that combine message passing with other programming and concurrency paradigms. Actris can be used to reason about programs written in a language that mimics the important features found in aforementioned languages such as higher-order functions, higher-order references, fork-based concurrency, locks, and primitives for asynchronous message passing over channels. The channels of our language are first-class and can be sent as arguments to functions, be sent over other channels (often referred to as delegation), and be stored in references.

Program specifications in Actris are written in an impredicative higher-order concurrent separation logic built on top of the Iris framework [Jung et al. 2015; Krebbers et al. 2017a; Jung et al. 2016, 2018b]. In addition to the usual features of Iris, Actris provides a notion of *dependent separation protocols* to reason about message passing over channels, inspired by binary session types [Honda et al. 1998]. We show that dependent separation protocols integrate seamlessly with other concurrency paradigms, allow delegation of resources, support channel sharing over multiple concurrent threads using locks, and more.

1.1. Message passing in concurrent separation logic.

Over the last decade, there has been much work on extensions of concurrent separation logic with reasoning principles for message passing [Francalanza et al. 2011; Lozes and Villard 2012; Craciun et al. 2015; Oortwijn et al. 2016]. These logics typically include some form of mechanism for writing protocol specifications in a high-level manner, to elegantly reason about message passing in some specific context.

In a different line of work, researchers have developed more expressive extensions of concurrent separation logic that support proving strong specifications of programs involving features such as higher-order functions, fine-grained shared-memory concurrency, and locks. Examples of such logics are TaDA [da Rocha Pinto et al. 2014], iCAP [Svendsen and Birkedal 2014], Iris [Jung et al. 2015], FCSL [Nanevski et al. 2014], and VST [Appel 2014]. However, only a few variants and extensions of these logics provide a high-level reasoning mechanism specific to message-passing concurrency.

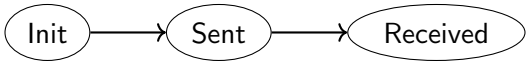
First off, there has been work on the use of Iris-like separation logic to reason about programs that communicate via message passing over a network. The reasoning principles in such logics are geared towards different programming patterns than the ones used in high-level languages like Erlang, Elixir, Go, and Scala. Namely, on networks all data must be serialised, and packets can be lost or delivered out of order. In high-level languages messages cannot get lost, are ensured to be delivered in order, and are allowed to contain many types of data, including functions, references, and even channel endpoints. Two examples of

network logics are Disel by Sergey et al. [2018] and Aneris by Krogh-Jespersen et al. [2020]. Additionally, there has been work on the use of separation logic to prove compiler correctness of high-level message-passing languages. Tassarotti et al. [2017] verified a small compiler of a session-typed language into a language where channel buffers are modelled on the heap.

The primary reasoning principle to model the interaction between processes in the aforementioned logics is the notion of a State Transition System (STS). As a simple example, consider the following program, which is borrowed from Tassarotti et al. [2017]:

$$prog_1 := \texttt{let}\ (c, c') := \texttt{new\_chan}\ ()\ \texttt{in}\ \texttt{fork}\ \{\texttt{send}\ c'\ 42\}\,;\texttt{recv}\ c$$

This program creates two channel endpoints c and c', forks off a new thread, and sends the number 42 over the channel c', which is then received by the initiating thread. Modelling the behaviour of this program in an STS typically requires three states:



The three states model that no message has been sent (Init), that a message has been sent but not received (Sent), and finally that the message has been sent and received (Received). Exactly what this STS represents is made precise by the underlying logic, which determines what constitutes a state and a transition, and how these are related to the channel buffers.

While STSs appear like a flexible and intuitive abstraction to reason about message-passing concurrency, they have their problems:

- Coming up with a good STS that makes the appropriate abstractions is difficult because the STS has to keep track of all possible states that the channel buffers can be in, including all possible interleavings of messages in transit.
- While STSs used for the verification of different modules can be composed at the level of the logic, there is no canonical way of composing them due to their unrestrained structure.
- Finally, STSs are first-order meaning that their states and transitions cannot be indexed by propositions of the underlying logic, which limits what they can express when sending messages containing functions or other channels.

1.2. **Actris 1.0: Dependent separation protocols.** Actris extends separation logic with a notion called *dependent separation protocols*. This notion is inspired by the session type community, pioneered by Honda et al. [1998], where channel endpoints are given types that describe the expected exchanges. Using session types, the channels c and c' in the program $prog_1$ in § 1.1 would have the types $c :$ **?**$\mathbb{Z}$.**end** and $c' :$ **!**$\mathbb{Z}$.**end**, where **!**T and **?**T denotes that a value of type T is sent or received, respectively. Moreover, the types of the channels c and c' are *duals*—when one does a send the other does a receive, and *vice versa*.

While session types provide a compact way of specifying the behaviour of channels, they can only be used to talk about the type of data that is being passed around—not their payloads. In this paper, we build on prior work by Bocchi et al. [2010] and Craciun et al. [2015] to attach logical predicates to session types to say more about the payloads, thus vastly extending the expressivity. Concretely, we port session types into separation logic in the form of a construct $c \rightarrowtail prot$, which denotes ownership of a channel c with dependent separation protocol $prot$. Dependent separation protocols $prot$ are streams of $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ constructors that are either infinite or finite, where finite streams are ultimately terminated by an **end** constructor. Here, v is the value that is being sent or received, P is a separation logic proposition denoting the ownership of the

resources being transferred as part of the message, and the variables $\vec{x}{:}\vec{\tau}$ bind into v, P, and *prot*. The dependent separation protocols for the above example are:

$$c \rightarrowtail ?\langle 42\rangle\{\mathsf{True}\}.\,\mathbf{end} \quad \texttt{and} \quad c' \rightarrowtail !\,\langle 42\rangle\{\mathsf{True}\}.\,\mathbf{end}$$

These protocols state that the endpoint c expects the number 42 to be sent along it, and that the endpoint c' expects to send the number 42. Using this protocol, we can prove that $prog_1$ has the specification $\{\mathsf{True}\}\ prog_1\ \{v.\, v = 42\}$, where v is its resulting value.

Dependent separation protocols $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ are *dependent*, meaning that the tail *prot* can be defined in terms of the previously bound variables $\vec{x}{:}\vec{\tau}$. A sample program showing the use of such dependency is:

$$\begin{aligned} prog_2 := {} & \texttt{let}\ (c, c') := \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ x := \texttt{recv}\ c'\ \texttt{in}\ \texttt{send}\ c'\ (x + 2)\}\,; \\ & \texttt{send}\ c\ 40;\ \texttt{recv}\ c \end{aligned}$$

In this program, the main thread sends the number 40 to the forked-off thread, which then adds two to it, and sends it back. This program has the same specification as $prog_1$, while we change the dependent separation protocol as follows (we omit the dependent separation protocol for the dual endpoint c'):

$$c \rightarrowtail !\,(x : \mathbb{Z})\,\langle x\rangle\{\mathsf{True}\}.\,?\langle x + 2\rangle\{\mathsf{True}\}.\,\mathbf{end}$$

This protocol states that the second exchanged value is exactly the first with two added to it. To do so, it makes use of a dependency on the variable x, which is used to describe the contents of the first message, which the second message then depends on. This variable is bound in the protocol and it is instantiated only when a message is sent. This is different from the logic by Craciun et al. [2015], which does not support dependent protocols. Their logic is limited to protocols analogous to $!\,\langle x\rangle\{\mathsf{True}\}.\,?\langle x + 2\rangle\{\mathsf{True}\}.\,\mathbf{end}$ where x is free, which means the value of x must be known when the protocol is created.

While the prior examples could have been type-checked and verified using the formalisms of Bocchi et al. [2010] and Craciun et al. [2015], the following stateful example cannot:

$$\begin{aligned} prog_3 := {} & \texttt{let}\ (c, c') := \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ l := \texttt{recv}\ c'\ \texttt{in}\ l \leftarrow (!\,l + 2);\ \texttt{send}\ c'\ ()\}\,; \\ & \texttt{let}\ l := \texttt{ref}\ 40\ \texttt{in}\ \texttt{send}\ c\ l;\ \texttt{recv}\ c;\ !\,l \end{aligned}$$

Here, the main thread stores the value 40 on the heap, and sends a reference l over the channel c to the forked-off thread. The main thread then awaits a signal (), notifying that the reference has been updated to 42 by the forked-off thread. This program has the same specification as $prog_1$ and $prog_2$, but the dependent separation protocol is updated:

$$c \rightarrowtail !\,(\ell : \mathsf{Loc})\ (x : \mathbb{Z})\,\langle \ell\rangle\{\ell \mapsto x\}.\,?\langle()\rangle\{\ell \mapsto (x + 2)\}.\,\mathbf{end}$$

This protocol denotes that the endpoints first exchange a reference ℓ, as well as a *points-to* connective $\ell \mapsto x$ that describes the ownership and value of the reference ℓ. To perform the exchange c has to give up ownership of the location, while c' acquires it—which is why it can then safely update the received location to 42 before sending the ownership back along with the notification ().

The type system by Bocchi et al. [2010] cannot verify this program because it does not support mutable state, while Actris can verify the program because it is a separation logic. The logic by Craciun et al. [2015] cannot verify this program because it does not support

dependent protocols, which are crucial here as they make it possible to delay picking the location ℓ used in the protocol until the send operation is performed.

Dependent protocols are also useful to define recursive protocols to reason about programs that use a channel in a loop. Consider the following variant of $prog_1$:

$$\begin{aligned} prog_4 := {} & \texttt{let}\ (c, c') := \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ go\ ()\ := (\texttt{send}\ c'\ (\texttt{recv}\ c' + 2);\ go\ ())\ \texttt{in}\ go\ ()\}; \\ & \texttt{send}\ c\ 18;\ \texttt{let}\ x := \texttt{recv}\ c\ \texttt{in} \\ & \texttt{send}\ c\ 20;\ \texttt{let}\ y := \texttt{recv}\ c\ \texttt{in}\ x + y \end{aligned}$$

The forked-off thread will repeatedly interleave receiving values with sending those values back incremented by two. The program $prog_4$ has the same specification as before, but now we use the following recursive dependent separation protocol:

$$c \rightarrowtail \mu rec.\, !\,(x : \mathbb{Z})\,\langle x\rangle\{\mathsf{True}\}.\, ?\langle x + 2\rangle\{\mathsf{True}\}.\, rec$$

This protocol expresses that it is possible to make repeated exchanges with the forked-off thread to increment a number by two. The fact that the variable x is bound in the protocol is once again crucial—it allows the use of different numbers for each exchange.

Furthermore, Actris inherently includes some features of conventional session types. One such example is the *delegation* of channels as seen in the following program:

$$\begin{aligned} prog_5 := {} & \texttt{let}\ (c_1, c_1') := \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ c := \texttt{recv}\ c_1'\ \texttt{in}\ \texttt{let}\ y := \texttt{recv}\ c_1'\ \texttt{in}\ \texttt{send}\ c\ y;\ \texttt{send}\ c_1'\ ()\}; \\ & \texttt{let}\ (c_2, c_2') := \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ x := \texttt{recv}\ c_2'\ \texttt{in}\ \texttt{send}\ c_2'\ (x + 2)\}; \\ & \texttt{send}\ c_1\ c_2;\ \texttt{send}\ c_1\ 40;\ \texttt{recv}\ c_1;\ \texttt{recv}\ c_2 \end{aligned}$$

This program uses the channel pair c_2, c_2' to exchange the number 40 with the second forked-off thread, which adds 2 to it, and sends it back. Contrary to the programs we have seen before, it uses the additional channel pair c_1, c_1' to delegate the endpoint c_2 to the first forked-off thread, which then sends the number over c_2. While this program is intricate, the following dependent separation protocols describe the communication concisely:

$$\begin{aligned} c_1 \rightarrowtail {} & !\,(c : \mathsf{Val})\,\langle c\rangle\{c \rightarrowtail !\,(x : \mathbb{Z})\,\langle x\rangle\{\mathsf{True}\}.\, ?\langle x + 2\rangle\{\mathsf{True}\}.\, \mathbf{end}\}. \\ & !\,(y : \mathbb{Z})\,\langle y\rangle\{\mathsf{True}\}.\, ?\langle()\rangle\{c \rightarrowtail ?\langle y + 2\rangle\{\mathsf{True}\}.\, \mathbf{end}\}.\, \mathbf{end} \\ c_2 \rightarrowtail {} & !\,(x : \mathbb{Z})\,\langle x\rangle\{\mathsf{True}\}.\, ?\langle x + 2\rangle\{\mathsf{True}\}.\, \mathbf{end} \end{aligned}$$

The first protocol states that the initial value sent must be a channel endpoint c with the protocol used in $prog_1$. This means that the main thread must give up ownership of the channel endpoint c_2, thereby delegating it. The first protocol then expects a value y to be sent, and finally to receive a notification $()$, along with ownership of the channel c_2, which has since taken one step by sending y. Note that c is of the type Val of programming language values due to the programming language being untyped.

Lastly, the dependencies in dependent separation protocols are not limited to first-order data, but can also be used in combination with functions. For example:

$$\begin{aligned} prog_6 := {} & \texttt{let}\ (c, c') := \texttt{new\_chan}\ ()\ \texttt{in} \\ & \texttt{fork}\ \{\texttt{let}\ f := \texttt{recv}\ c'\ \texttt{in}\ \texttt{send}\ c'\ (\lambda().\, f() + 2)\}; \\ & \texttt{let}\ l := \texttt{ref}\ 40\ \texttt{in}\ \texttt{send}\ c\ (\lambda().\, !\,l);\ \texttt{recv}\ c\ () \end{aligned}$$

This program exchanges a value to which 2 is added, but postpones the evaluation by wrapping the computation in a closure. The following protocol is used to verify this program:

$$c \rightarrowtail \mathbf{!}\,(P\,Q : \mathsf{iProp})\,(f : \mathsf{Val})\,\langle f\rangle\{\{P\}\ f\ ()\ \{v.\ v \in \mathbb{Z} * Q(v)\}\}.\\ \quad \mathbf{?}(g : \mathsf{Val})\,\langle g\rangle\{\{P\}\ g\ ()\ \{v.\ \exists w.\ (v = w + 2) * Q(w)\}\}.\,\mathbf{end}$$

The send constructor (**!**) does not just bind the function value f, but also the precondition P and postcondition Q of its Hoare triple. In the second message, a Hoare triple is returned that maintains the original pre- and postconditions, but returns an integer of two higher. To send the function, the main thread would let $P \triangleq \ell \mapsto 40$ and $Q(v) \triangleq (v = 40)$, and prove $\{P\}\ (\lambda().\ !\,\ell)\ ()\ \{Q\}$. This example demonstrates that the state space of dependent separation protocols can be higher-order—it is indexed by the precondition P and postcondition Q of f—which means that they do not have to be agreed upon when creating the protocol, masking the internals of the function from the forked-off thread.

It is worth noting that using dependent recursive protocols it is possible to keep track of a history of what actions have been performed, which, as is shown in § 7, is especially useful when combining channels with locks.

1.3. **Actris 2.0: Subprotocols.** While Actris 1.0's notion of dependent separation protocols is expressive enough to specify advanced exchanges, as indicated by the examples in the previous section, they can only reason about interactions that are strictly dual. In particular, the dual nature of Actris 1.0 requires that:

- Sends ($\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}$) are matched up with receives ($\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}$), and *vice versa*,
- The logical variables $\vec{x}{:}\vec{\tau}$ of matched sends and receives are the same, and,
- The propositions P of matched send and receives are the same.

Reasoning about programs with a more relaxed duality principle has been studied in the session type community, namely in the context of *asynchronous session subtyping* [Mostrous et al. 2009; Mostrous and Yoshida 2015]. A subtyping relation $S_1 <: S_2$ captures that the session type S_2 can be used in place of S_1 when type checking a program. Channel endpoints are then allocated with strictly dual session types, after which either side can be weakened based on the subtyping relation. For one, the subtyping relation captures that sends can be swapped *ahead of* receives $\mathbf{?}T.\mathbf{!}U.S <: \mathbf{!}U.\mathbf{?}T.S$. Swapping sends ahead of receives is safe to do, as the messages are simply enqueued into the corresponding channel buffer earlier than necessary. The following program illustrates such a non-dual yet safe interaction:

$$\begin{aligned} \mathit{prog}_7 := {} & \texttt{let}\ (c, c') := \texttt{new\_chan}\ ()\ \texttt{in}\\ & \texttt{fork}\ \{\texttt{send}\ c'\ 20;\ \texttt{send}\ c'\ (\texttt{recv}\ c' + 2)\};\\ & \texttt{send}\ c\ 20;\\ & \texttt{let}\ x := \texttt{recv}\ c\ \texttt{in}\\ & \texttt{let}\ y := \texttt{recv}\ c\ \texttt{in}\ x + y \end{aligned}$$

Here, both threads first send the value 20, which is enqueued into both of the channel buffers, after which they receive the value of the other thread. After this, they follow a dual behaviour, where the forked-off thread sends a value, which the main thread receives.

In this paper, we show that dependent separation protocols are compatible with the idea of asynchronous session subtyping. This gives rise to **Actris 2.0**, which supports so-called *subprotocols*. Subprotocols are formalised by a preorder $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$, which captures (among others) a notion of swapping sends ahead of receives (provided that the send does

not depend on the logical variables of the receive). We can prove that $prog_7$ results in 42 by picking the following dependent separation protocols:

$$c \rightarrowtail \mathbf{!}\,(x : \mathbb{Z})\,\langle x\rangle\{\mathsf{True}\}.\,\mathbf{?}\langle 20\rangle\{\mathsf{True}\}.\,\mathbf{?}\langle x+2\rangle\{\mathsf{True}\}.\,\mathbf{end} \quad \texttt{and}$$
$$c' \rightarrowtail \mathbf{?}(x : \mathbb{Z})\,\langle x\rangle\{\mathsf{True}\}.\,\mathbf{!}\,\langle 20\rangle\{\mathsf{True}\}.\,\mathbf{!}\,\langle x+2\rangle\{\mathsf{True}\}.\,\mathbf{end}$$

While the main thread satisfies the protocol of c immediately, the forked-off thread does not satisfy the protocol of c', as it sends the first value before receiving. However, it is possible to weaken the protocol of c' using Actris 2.0's notion of subprotocols:

$$\mathbf{?}(x : \mathbb{Z})\,\langle x\rangle\{\mathsf{True}\}.\,\mathbf{!}\,\langle 20\rangle\{\mathsf{True}\}.\,\mathbf{!}\,\langle x+2\rangle\{\mathsf{True}\}.\,\mathbf{end}$$
$$\sqsubseteq \mathbf{!}\,\langle 20\rangle\{\mathsf{True}\}.\,\mathbf{?}(x : \mathbb{Z})\,\langle x\rangle\{\mathsf{True}\}.\,\mathbf{!}\,\langle x+2\rangle\{\mathsf{True}\}.\,\mathbf{end}$$

This gives $c' \rightarrowtail \mathbf{!}\,\langle 20\rangle\{\mathsf{True}\}.\,\mathbf{?}(x : \mathbb{Z})\,\langle x\rangle\{\mathsf{True}\}.\,\mathbf{!}\,\langle x+2\rangle\{\mathsf{True}\}.\,\mathbf{end}$. Since the first send (with value 20) is independent of the variable x bound by the receive, the subprotocol relation follows immediately from the swapping property. Note that it is *not* possible to swap the second send (with value $x+2$) ahead of the receive, as it does in fact depend on variable x bound by the receive.

In addition to allowing the verification of a larger class of programs, Actris 2.0's subprotocols also provide a more extensional approach to reasoning about dependent separation protocols. This is beneficial whenever we want to reuse existing specifications that might use a syntactically different protocol, but that nonetheless logically entail each another. For example, the ordering of logical variables can be changed using the subprotocol relation:

$$\mathbf{!}\,(x : \mathbb{Z})(y : \mathbb{Z})\,\langle (x,y)\rangle\{\mathsf{True}\}.\,\mathit{prot} \sqsubseteq \mathbf{!}\,(y : \mathbb{Z})(x : \mathbb{Z})\,\langle (x,y)\rangle\{\mathsf{True}\}.\,\mathit{prot}$$

Since the subprotocol relation is a first-class logical proposition of Actris 2.0, it also allows the manipulation of separation logic resources, such as moving in ownership. For example, we can show the following *conditional* subprotocol relation:

$$\ell_1' \mapsto 20 \;\mathrel{-\!\!*}$$
$$\mathbf{!}\,(\ell_1, \ell_2 : \mathsf{Loc})\,\langle (\ell_1,\ell_2)\rangle\{\ell_1 \mapsto 20 * \ell_2 \mapsto 22\}.\,\mathit{prot} \sqsubseteq \mathbf{!}\,(\ell_2 : \mathsf{Loc})\,\langle (\ell_1',\ell_2)\rangle\{\ell_2 \mapsto 22\}.\,\mathit{prot}$$

Here, we move the ownership of $\ell_1' \mapsto 20$ into the protocol, to resolve the eventual obligation of sending it, while instantiating the logical variable ℓ_1 with ℓ_1'.

In addition to the demonstrated features, in the rest of this paper we show that Actris 2.0's subprotocol relation is capable of moving resources from one message to another. This gives rise to a principle similar to *framing*, known from conventional separation logic, but applied to dependent separation protocols. Lastly, inspired by the work of Brandt and Henglein [1998], the subprotocol relation is defined coinductively, allowing us to use the principle of Löb induction to prove subprotocol relations for recursive protocols.

1.4. **Formal correspondence to session types.** Even though Actris's notion of dependent separation protocols is influenced by binary session types, this paper does not provide a formal correspondence between the two systems. However, since Actris is built on top of Iris, it forms a suitable foundation for building logical relation models of type systems. In related work by Hinrichsen et al. [2021b], Actris has been used to define a logical relations model of binary session types, with support for various forms of polymorphism and recursion, asynchronous subtyping, references, and locks/mutexes. Similar to RustBelt [Jung et al. 2018a, 2021], the work by Hinrichsen et al. [2021b] gives rise to an extensible approach for proving type safety, which can be used to manually prove the typing judgements of racy, but safe, programs that cannot be type checked using only the rules of the type system.

1.5. **Contributions and outline.** This paper introduces **Actris 2.0**: a higher-order impredicative concurrent separation logic built on top of the Iris framework for reasoning about functional correctness of programs with asynchronous message-passing that combine higher-order functions, higher-order references, fork-based concurrency, and locks. Concretely, this paper makes the following contributions:

- We introduce *dependent separation protocols* inspired by affine binary session types to model the transfer of resources (including higher-order functions) between channel endpoints. We show that they can be used to handle choice, recursion, and delegation (§ 2 to 5).
- We introduce *subprotocols* inspired by asynchronous session subtyping. This notion relaxes duality, allowing channels to send messages before receiving others, and gives rise to a more extensional approach to reasoning about dependent separation protocols, providing more flexibility in the design and reuse of protocols. We moreover show how Löb induction is used to reason about recursive subprotocols (§ 6).
- We demonstrate the benefits obtained from building Actris on top of Iris by showing how Iris's support for ghost state and locks can be used to prove functional correctness of programs using manifest sharing, *i.e.,* channel endpoints shared by multiple parties (§ 7).
- We provide a case study on Actris and its mechanisation in Coq by proving functional correctness of a variant of the map-reduce model by Dean and Ghemawat [2004] (§ 8).
- We give a model of dependent separation protocols in the Iris framework to prove safety and postcondition validity of our Hoare triples (§ 9).
- We provide a full mechanisation of Actris [Hinrichsen et al. 2021a] using the interactive theorem prover Coq. On top of our Coq mechanisation, we provide custom tactics, which we use to mechanise all examples in the paper (§ 10).

1.6. **Differences from the conference version.** This paper is an extension of the paper "Actris: Session-type based reasoning in separation logic" presented at the POPL'20 conference [Hinrichsen et al. 2020]. In this paper we present Actris 2.0, which extends Actris 1.0 with the notion of subprotocols. This extension introduces new logical connectives and proof rules, but also involves a significant overhaul of the original model and its Coq mechanisation. We extend the presentation of the programming language semantics, model and mechanisation substantially, with additional details, considerations, and examples, to give a better understanding of how Actris works and how it can be used. Concretely, this paper includes the following extensions compared to the conference version:

- An overview of subprotocols in the introduction (§ 1.3).
- A new background section on the programming language semantics (§ 2) and Iris (§ 3).
- A section with an expanded overview of Actris (§ 4, moved from § 5).
- A new section on Actris 2.0's notion of subprotocols (§ 6).
- An updated and expanded description of the model of Actris in Iris (§ 9).
- An extension of the section on the Coq mechanisation with sample proofs (§ 10).

2. Programming language semantics

The Iris program logic is parametric in the programming language that is used. As a result there are multiple approaches to extend Iris with support for channels:

- Instantiate Iris with a language that has native support for channels. This approach was carried out in the original Iris paper [Jung et al. 2015] and by Tassarotti et al. [2017].

- Instantiate Iris with a language that has low-level concurrency primitives, but no native support for channels, and implement channels as a library in that language. This approach was carried out by Bizjak et al. [2019] for a lock-free implementation of channels.

In this paper we take the second approach. We implement channels in HeapLang—the default programming language that is shipped with Iris's Coq development [Iris Development Team 2021]. HeapLang is an untyped functional language with high-level features such as higher-order functions, higher-order mutable references, fork-based concurrency, and garbage collection. Due to these high-level features, programs written in HeapLang are reminiscent of those written in high-level programming languages with message passing like Go or Erlang.

Since HeapLang is an untyped language, safety of a program is not obtained by establishing a typing judgement, but by proving a Hoare triple in the Iris/Actris logic. Hinrichsen et al. [2021b] show how logical relations in Actris can be used to define and prove sound a session type system for HeapLang extended with message passing.

We proceed by describing HeapLang's syntax (§ 2.1) and operational semantics (§ 2.2). We then present HeapLang's standard library for spin locks (§ 2.3). We use this lock library to implement channels (§ 2.4), and to write programs that combine message passing with lock-based concurrency (§ 7).

2.1. **Syntax.** The syntax of HeapLang is as follows:

$$
\begin{aligned}
v \in \mathsf{Val} ::= {} & () \mid i \mid b \mid \ell \mid \texttt{rec}\ f\ x := e \mid && (i \in \mathbb{Z}, b \in \mathbb{B}, \ell \in \mathsf{Loc}) \\
& (v_1, v_2) \mid \texttt{inj}_1\ v \mid \texttt{inj}_2\ v \mid \ldots \\
e \in \mathsf{Expr} ::= {} & v \mid x \mid e_1\ e_2 \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \\
& \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{inj}_1\ e \mid \texttt{inj}_2\ e \mid \\
& (\texttt{match}\ e_1\ \texttt{with}\ (\texttt{inj}_1\ x) \Rightarrow e_2 \mid (\texttt{inj}_2\ x) \Rightarrow e_3\ \texttt{end}) \mid \\
& \texttt{ref}\ e \mid !\, e \mid e_1 \leftarrow e_2 \mid && \text{(Mutable state)} \\
& \texttt{fork}\ \{e\} \mid \texttt{CAS}\ e_1\ e_2\ e_3 \mid \ldots && \text{(Concurrency)}
\end{aligned}
$$

We elide the standard boolean and arithmetic operators such as equality, addition, subtraction, and multiplication. We define various notions as syntactic sugar (*i.e.,* as definitions in the meta language by use of $\triangleq$):

$$
\begin{aligned}
\lambda x.\, e &\triangleq \texttt{rec}\ \_\ x := e & e_1;\ e_2 &\triangleq \texttt{let}\ \_ := e_1\ \texttt{in}\ e_2 \\
\texttt{let}\ x := e_1\ \texttt{in}\ e_2 &\triangleq (\lambda x.\, e_2)\ e_1 & \texttt{skipN} &\triangleq \texttt{rec}\ go\ x := \texttt{if}\ 0 < x \\
& & & \qquad \texttt{then}\ go\ (x-1)\ \texttt{else}\ ()
\end{aligned}
$$

We use $\_$ as the anonymous binder that is not used in the body of the binding expression. The `skipN` operation, which performs a given number of no-op program steps, is used in the implementation of channels (§ 2.4) for proof-related reasons (explained in § 9.5). We often write definitions as $f\ x_1 \cdots\ x_n := e$ rather than $f \triangleq \texttt{rec}\ f\ x_1 \cdots\ x_n := e$. For example, we write $\texttt{skipN}\ x := \texttt{if}\ 0 < x\ \texttt{then}\ \texttt{skipN}\ (x-1)\ \texttt{else}\ ()$.

HeapLang includes the usual operations for ML-style references. New references can be allocated using $\texttt{ref}\ e$, dereferenced using $!\, e$, and updated using $e_1 \leftarrow e_2$. Concurrency is supported via $\texttt{fork}\ \{e\}$, which spawns a new thread e that is executed in the background. The language also supports atomic operations like compare-and-set (`CAS`), which are used to implement lock-free data structures and synchronisation primitives, such as the locks (§ 2.3). HeapLang is garbage collected and thus does not have a deallocation operation.

Call-by-value evaluation contexts:

$$
\begin{aligned}
K \in \mathsf{Ctx} ::= {} & \bullet \mid e\ K \mid K\ v \mid (e_1, K) \mid (K, v_2) \mid \texttt{fst}\ (K) \mid \texttt{snd}\ (K) \mid \\
& \texttt{if}\ K\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{inj}_1\ (K) \mid \texttt{inj}_2\ (K) \mid \\
& (\texttt{match}\ K\ \texttt{with}\ (\texttt{inj}_1\ x) \Rightarrow e_2 \mid (\texttt{inj}_2\ x) \Rightarrow e_3\ \texttt{end}) \mid \\
& \texttt{ref}\ (K) \mid !\,K \mid e \leftarrow K \mid K \leftarrow v \mid && \text{(Mutable state)} \\
& \texttt{CAS}\ e_1\ e_2\ K \mid \texttt{CAS}\ e_1\ K\ v_2 \mid \texttt{CAS}\ K\ v_1\ v_2 \mid \ldots && \text{(Concurrency)}
\end{aligned}
$$

Head reductions of HeapLang:

$$
\begin{aligned}
((\texttt{rec}\ f\ x := e)(v); \sigma) &\to_{\mathsf{h}} (e[v/x][(\texttt{rec}\ f\ x := e)/f]; \sigma; [\,]) \\
(\texttt{fst}\ (v_1, v_2); \sigma) &\to_{\mathsf{h}} (v_1; \sigma; [\,]) \\
(\texttt{snd}\ (v_1, v_2); \sigma) &\to_{\mathsf{h}} (v_2; \sigma; [\,]) \\
(\texttt{if true then}\ e_1\ \texttt{else}\ e_2; \sigma) &\to_{\mathsf{h}} (e_1; \sigma; [\,]) \\
(\texttt{if false then}\ e_1\ \texttt{else}\ e_2; \sigma) &\to_{\mathsf{h}} (e_2; \sigma; [\,]) \\
\left(\begin{array}{l}\texttt{match}\ (\texttt{inj}_i\ v)\ \texttt{with} \\ \quad (\texttt{inj}_1\ x) \Rightarrow e_1 \\ \mid (\texttt{inj}_2\ x) \Rightarrow e_2 \\ \texttt{end}\end{array} ; \sigma\right) &\to_{\mathsf{h}} (e_i[v/x]; \sigma; [\,]) && \text{if } i \in \{1, 2\} \\
(\texttt{ref}\ v; \sigma) &\to_{\mathsf{h}} (\ell; \sigma[\ell \leftarrow v]; [\,]) && \text{if } \sigma(\ell) = \bot \\
(!\,\ell; \sigma[\ell \leftarrow v]) &\to_{\mathsf{h}} (v; \sigma[\ell \leftarrow v]; [\,]) \\
(\ell \leftarrow w; \sigma[\ell \leftarrow v]) &\to_{\mathsf{h}} ((); \sigma[\ell \leftarrow w]; [\,]) \\
(\texttt{CAS}\ \ell\ v'\ w; \sigma[\ell \leftarrow v]) &\to_{\mathsf{h}} (\texttt{true}; \sigma[\ell \leftarrow w]; [\,]) && \text{if } v = v' \\
(\texttt{CAS}\ \ell\ v'\ w; \sigma[\ell \leftarrow v]) &\to_{\mathsf{h}} (\texttt{false}; \sigma[\ell \leftarrow v]; [\,]) && \text{if } v \neq v' \\
(\texttt{fork}\ \{e\}; \sigma) &\to_{\mathsf{h}} ((); \sigma; [e])
\end{aligned}
$$

Thread-local and threadpool reductions of HeapLang:

$$
\frac{e_1; \sigma_1 \to_{\mathsf{h}} e_2; \sigma_2; \vec{e}}{K[\,e_1\,]; \sigma_1 \to_{\mathsf{tl}} K[\,e_2\,]; \sigma_2; \vec{e}} \qquad \frac{e_1; \sigma_1 \to_{\mathsf{tl}} e_2; \sigma_2; \vec{e}}{T \cdot [e_1] \cdot T'; \sigma_1 \to_{\mathsf{tp}} T \cdot [e_2] \cdot T' \cdot \vec{e}; \sigma_2}
$$

Figure 1: The operational semantics of HeapLang.

2.2. **Operational semantics.** The small-step operational semantics of HeapLang is presented in Figure 1. The type of program states State is defined as:

$$\sigma \in \mathsf{State} \triangleq \mathsf{Loc} \xrightarrow{\mathsf{fin}} \mathsf{Val}$$

That is, program states are finite partial maps from allocated locations to their stored values.

The *head reduction* $(e_1; \sigma_1 \to_{\mathsf{h}} e_2; \sigma_2; \vec{e})$ describes how an expression $e_1 \in \mathsf{Expr}$ in an initial program state $\sigma_1 \in \mathsf{State}$ reduces to a new expression $e_2 \in \mathsf{Expr}$ in a possibly updated program state $\sigma_2 \in \mathsf{State}$. Additionally, it keeps track of a list of newly spawned threads $\vec{e} \in \mathsf{List}\ \mathsf{Expr}$. The reduction rule $(\texttt{fork}\ \{e\}; \sigma) \to_{\mathsf{h}} ((); \sigma; [e])$ describes how a new thread e is spawned by adding it to the list of newly spawned threads $[e]$. Conversely, the list of newly spawned threads is empty for all of the other reduction rules.

$$
\begin{aligned}
\texttt{new\_lock}\ () &:= \texttt{ref}\ \texttt{false}\\
\texttt{try\_acquire}\ lk &:= \texttt{CAS}\ lk\ \texttt{false}\ \texttt{true}\\
\texttt{acquire}\ lk &:= \texttt{if}\ (\texttt{try\_acquire}\ \ lk)\ \texttt{then}\ ()\ \texttt{else}\ \texttt{acquire}\ lk\\
\texttt{release}\ lk &:= lk \leftarrow \texttt{false}
\end{aligned}
$$

Figure 2: Implementation of locks in HeapLang.

The *thread-local reduction* $(e_1; \sigma_1 \rightarrow_{\mathsf{tl}} e_2; \sigma_2; \vec{e})$ lifts the head reduction to whole expressions. It decomposes the initial expression e_1 into $K[\,e_1'\,]$, where K is a *call-by-value evaluation context* [Felleisen and Hieb 1992] and a head expression e_1'. The head expression e_1' is then reduced, using $(e_1'; \sigma_1 \rightarrow_{\mathsf{h}} e_2'; \sigma_2; \vec{e})$, and the final expression e_2 is set to $K[\,e_2'\,]$. Evaluation contexts (shown in Figure 1) provide a deterministic reduction order of subexpressions. HeapLang reduces right-to-left, meaning that in expressions such as $e_1 \leftarrow e_2$ the expression e_2 reduces before e_1. This is determined by the corresponding evaluation contexts $e \leftarrow K$ and $K \leftarrow v$, which state that we only evaluate sub-expressions of the target location, once the term to store is a value. More precisely, we would initially get $(e_1 \leftarrow \bullet)[\,e_2\,]$. If e_2 reduces to a value v_2 the context syntax dictates that the hole then moves to e_1 yielding $(\bullet \leftarrow v_2)[\,e_1\,]$. If e_1 reduces to a value v_1 we finally end up with the expression $v_1 \leftarrow v_2$, as there is no context syntax where both constituents are values, and this expression can be reduced using a standard head reduction.

Finally, the *threadpool reduction* $(\vec{e_1}; \sigma_1 \rightarrow_{\mathsf{tp}} \vec{e_2}; \sigma_2)$ is the top-level reduction relation that describes the interleaving of threads. It describes how a concurrently running list of threads $\vec{e_1}$, in an initial program state σ_1, reduce to a new list of threads $\vec{e_2}$ in an updated program state σ_2. At each step a thread e_1 is picked non-deterministically from $\vec{e_1}$ and reduced one step to e_2 via the thread-local reduction $(e_1; \sigma_1 \rightarrow_{\mathsf{tl}} e_2; \sigma_2; \vec{e})$. The final list of threads $\vec{e_2}$ is obtained from $\vec{e_1}$ by replacing the expression e_1 with e_2 and appending the list $\vec{e}$ of newly spawned threads to the end.

We refer the interested reader to Iris Development Team [2021, `docs/heap_lang.md`] for more details on the semantics of HeapLang, and to Jung et al. [2018b, §6.1] for details on the language-parametric aspects of Iris.

2.3. **Implementation of locks.** Using HeapLang it is possible to implement various kinds of locks/mutexes. We consider the simplest kind of lock—a spin lock—whose implementation from the HeapLang standard library is shown in Figure 2.

A spin lock implemented using a reference to a boolean, which is `false` if the lock is unlocked, and `true` if the lock is locked. The `new_lock` () operation creates a new lock lk, which is initially unlocked (*i.e.,* `false`). The operation `acquire` lk will atomically (using compare-and-set) take the lock, or loop if the lock is already taken. The `release` lk operation releases the lock so that it may be acquired by other threads.

2.4. **Implementation of channels.** Following the literature on asynchronous session types, the message-passing semantics of our channels is *binary* (communication is between two parties), *asynchronous* (sending messages does not block), *bidirectional* (messages can be in transit in both directions simultaneously), *reliable* (messages are never dropped), and *order preserving* (messages always arrive in the order that they were sent).

```
new_chan () := let (l, r, lk) := (lnil (), lnil (), new_lock ()) in
               ((l, r, lk), (r, l, lk))
   send c v := let (l, r, lk) := c in
               acquire lk;
                 lsnoc l v;
                 skipN (llength r);
               release lk
try_recv c := let (l, r, lk) := c in
               acquire lk;
                 let ret := (if (lisnil r) then (inj1 ()) else (inj2 (lpop l))) in
               release lk; ret
    recv c := match (try_recv c) with
                inj1 () ⇒ recv c
              | inj2 v  ⇒ v
              end
```

Figure 3: Implementation of bidirectional channels in HeapLang.

The implementation of our channels in HeapLang is displayed in Figure 3. It uses locks (§ 2.3) and a linked list library. This list library provides functions for creating a empty list (`lnil`), testing if a list is empty (`lisnil`), computing the length of a list (`llength`), adding an element to the back (`lsnoc`), and popping an element of front (`lpop`). The last two functions mutate the list, instead of creating a copy. The implementation of the list library is standard, and hence elided.

Intuitively, the channels can be thought of as a pair of buffers $(\vec{v}_1, \vec{v}_2)$ of unbounded size. The `new_chan` () operation creates a new channel whose buffers are empty, and returns a tuple of endpoints (c_1, c_2). Bidirectionality is obtained by having one endpoint receive from the others send buffer and *vice versa*. As such, the `send` c_i v operation enqueues the value v in its own buffer, *i.e.,* $\vec{v}_i$, and the `recv` c_i operation dequeues a value from the other buffer, *i.e.,* from $\vec{v}_2$ if $i = 1$ and from $\vec{v}_1$ if $i = 2$. The message passing is asynchronous, as `send` c v will always reduce, while `recv` c will loop as long as the receiving buffer is empty.

More specifically, the `new_chan` function creates new channels by allocating two empty mutable linked lists l and r using `lnil` (), along with a lock lk using `new_lock` (), and returns the tuples (l, r, lk) and (r, l, lk), where the order of the linked lists l and r determines the side of the endpoints. We refer to the list in the left position as the endpoint's own buffer, and the list in the right position as the other endpoint's buffer.

The `send` function sends a value v over a given channel endpoint (l, r, lk), by enqueueing it in the l buffer. The function operates in an atomic fashion by first acquiring the lock via `acquire` lk, thereby entering the critical section, after which the value is enqueued (*i.e.,* appended to the end) of the endpoint's own buffer using the function `lsnoc` l v. The `skipN` (`llength` r) instruction is a no-op that is inserted to aid the proof. We come back to the reason why this instruction is needed in § 9.5.

The `recv` function receives a value over a channel endpoint (l, r, lk), by dequeueing the first value in the r buffer. It does so by performing a loop that repeatedly calls the helper function `try_recv`. This helper function attempts to receive a value atomically, and

fails if there is no value in the other endpoint's buffer. The function `try_recv` acquires the lock with `acquire` lk, and then checks whether the other endpoint's buffer is empty using `lisnil` r. If it is empty, nothing is returned (*i.e.,* $\texttt{inj}_1\,()$), while otherwise the value is dequeued and returned (*i.e.,* $\texttt{inj}_2\,(\texttt{lpop}\ l)$).

Throughout the paper, we often use a combined operation for starting a thread and creating a channel between the parent and child thread:

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

3. The Iris logic

We give a brief introduction to the features of Iris that play an important role in Actris: its support for basic separation logic (§ 3.1), higher-order impredicative separation logic (§ 3.2), guarded recursion and step-indexing (§ 3.3), and Iris's adequacy theorem (§ 3.4). This section does not present new material, so readers that are already familiar with Iris can skip it. An extensive overview of Iris can be found in [Jung et al. 2018b], and a tutorial-style introduction can be found in [Birkedal and Bizjak 2020].

3.1. Basic separation logic.

Propositions in separation logic describe ownership of resources, and can thus intuitively be thought of as predicates over resources. The propositions of Iris $P, Q \in \mathsf{iProp}$ range over an extensible set of resources, which includes the program state. Iris is a higher-order separation logic, so it has the usual logical connectives such as conjunction ($P \land Q$), implication ($P \Rightarrow Q$), universal ($\forall x : \tau.\ P$) and existential ($\exists x : \tau.\ P$) quantification, as well as the connectives of separation logic:

- The *points-to connective* ($\ell \mapsto v$) asserts exclusive resource ownership of a location $\ell \in \mathsf{Loc}$ in the program state, stating that it holds the value $v \in \mathsf{Val}$.
- The *separating conjunction* ($P * Q$) states that P and Q holds for disjoint sets of resources.
- The *separating implication* ($P \mathrel{-\!\!*} Q$) states that by giving up ownership of the resources described by P, we obtain ownership of the resources described by Q. Separating implication is used similarly to implication since (P entails $Q \mathrel{-\!\!*} R$) iff ($P * Q$ entails R).
- The *Hoare triple* $\{P\}\, e\, \{w.\ Q\}$ states that if the initial program state satisfies the precondition P, then (1) the expression e is safe (*i.e.,* does not go wrong), and, (2) if e reduces to a value v, then the final program state satisfies the postcondition $Q[v/w]$. We often omit the binder w in the postcondition if the result is the unit value $()$.

We say that an Iris proposition P *is valid* iff it holds for all resources, *i.e., P* is valid iff True entails P. Note that $P \mathrel{-\!\!*} Q$ is valid iff P entails Q, so we often use the separating implication ($\mathrel{-\!\!*}$) in place of entailment. For readability, we use inference-style rules to denote separation logic rules $(P_1 * \cdots * P_n) \mathrel{-\!\!*} Q$ as:

$$\frac{P_1 \qquad \ldots \qquad P_n}{Q}$$

Iris is an affine separation logic, which means that propositions are upwards closed in the resources, *i.e., P* $* Q$ entails P (rule Affine). Affinity matches up with the use of a garbage-collected programming language—one can simply dispose of an unused points-to connective $\ell \mapsto v$ using rule Affine when a location ℓ is no longer referenced.

While many propositions of separation logic assert exclusive ownership of resources (*e.g.,* $\ell \mapsto v$), others do not (*e.g.,* $t = u$). Propositions that do not assert exclusive ownership

Grammar:

$$\begin{aligned}
\tau, \sigma &::= x \mid 0 \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \mathbb{Z} \mid \mathsf{Type} \mid \forall x : \tau.\, \sigma \mid \\
&\quad \mathsf{Loc} \mid \mathsf{Val} \mid \mathsf{Expr} \mid \mathsf{iProp} \mid \mathsf{List}\ \tau \mid \ldots \\
t, u, P, Q &::= x \mid \lambda x : \tau.\, t \mid t(u) \mid t(\tau) \mid && \text{(Polymorphic lambda-calculus)} \\
&\quad \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid && \text{(Propositional logic)} \\
&\quad \forall x : \tau.\, P \mid \exists x : \tau.\, P \mid t = u \mid && \text{(Higher-order logic with equality)} \\
&\quad P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v \mid \{P\}\, e\, \{v.\, Q\} \mid && \text{(Separation logic)} \\
&\quad \mu x : \tau.\, t \mid \triangleright P \mid \ldots && \text{(Guarded recursion and step indexing)}
\end{aligned}$$

Basic affine separation logic:

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

$$\textsc{Ht-bind}\ \frac{\{P\}\, e\, \{v.\, Q\} \qquad \forall v.\, \{Q\}\, K[\,v\,]\, \{w.\, R\}}{\{P\}\, K[\,e\,]\, \{w.\, R\}}\ K \text{ a call-by-value evaluation context}$$

Heap manipulation:

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

Guarded recursion and step indexing:

$$\textsc{Ht-rec}\ \frac{\{P\}\, e[v/x][\texttt{rec}\ f\ x := e/f]\, \{w.\, Q\}}{\{\triangleright P\}\, (\texttt{rec}\ f\ x := e)\, v\, \{w.\, Q\}} \qquad \triangleright\textsc{-intro}\ \frac{P}{\triangleright P} \qquad \triangleright\textsc{-mono}\ \frac{P \mathrel{-\!\!*} Q}{\triangleright P \mathrel{-\!\!*} \triangleright Q} \qquad \textsc{Löb}\ \frac{\triangleright P \Rightarrow P}{P}$$

$$\mu\textsc{-unfold}\ (\mu x.\, t) = t[\mu x.\, t/x]$$

Figure 4: The grammar and a selection of rules of Iris.

enjoy some useful laws. Separation conjunction ($P * Q$) is logically equivalent to regular conjunction ($P \wedge Q$) if at least one conjunct does not assert exclusive ownership, and separating implication ($P \mathrel{-\!\!*} Q$) is logically equivalent to regular implication ($P \Rightarrow Q$) if the premise P does not assert exclusive ownership.[1] For example, $(t = u) * Q$ and $(t = u) \wedge Q$ are logically equivalent. Since separating conjunction/implication is omnipresent in Iris, we prefer the use of separating conjunction/implication over regular conjunction/implication if both can be used. This is also the convention used in the Iris Coq development.

Iris's notion of resources is not limited to locations in the program state (*i.e.*, $\ell \mapsto v$), but can be extended with user-defined *ghost* resources. We use ghost resources to define Actris's connective $c \rightarrowtail \mathit{prot}$ for exclusive ownership of the channel endpoint c with protocol prot (§ 9), and to reason about programs with non-trivial sharing (§ 7).

[1] Formally, these equivalences hold for the class of *persistent* propositions, see [Jung et al. 2018b, §2.3].

The rules for Hoare triples are mostly standard, but it is worth pointing out the rule for HT-BIND. This rule enables reductions of an expression e, in some evaluation context K, based on the precedence enforced by the evaluation contexts presented in § 2.2.

3.2. **Higher-order impredicative separation logic.** The Iris logic is:

- *Higher-order:* Using Iris's quantifiers $\forall x : \tau.\, P$ and $\exists x : \tau.\, P$ it is not only possible to quantify over first-order types (like $\mathbb{Z}$ and $\mathsf{List}\ \mathbb{Z}$), but over any type, including functions (like $\mathbb{Z} \to \mathbb{Z}$), higher-order functions (like $(\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}$), polymorphic functions (like $\forall T.\, \mathsf{List}\ T \to \mathbb{N}$), Iris propositions ($\mathsf{iProp}$), and Iris predicates (like $\mathbb{Z} \to \mathsf{iProp}$).
- *Impredicative:* Iris's logical connectives can be nested arbitrarily. Notably, $\forall P : \mathsf{iProp}.\, Q$ is an Iris proposition, and not an Iris proposition in a higher universe. Similarly, Hoare triples $\{P\}\, e\, \{v.\, Q\}$ and other Iris connectives like $\mathsf{is\_lock}\ lk\ R$ for lock ownership (§ 7) are first-class Iris propositions themselves.

As we will see in this paper, Actris expands on Iris's support for higher-order impredicative separation logic by allowing the variables $\vec{x}{:}\vec{\tau}$ in the dependent separation protocols $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot$ and $\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot$ to range over any type (including Actris's type of protocols iProto), and the proposition P to contain any Iris/Actris connective (including the Actris connective $c \rightarrowtail prot$ for channel ownership). This is particularly useful to reason about message-passing programs that transfer functions (§ 5.2) and channels (§ 5.5).

To define (pure) functions and predicates used in program specifications, Iris embeds the polymorphic lambda calculus. In the Coq development of Iris, this lambda calculus is obtained via a shallow embedding, and thus comprises the usual Coq data types and functions.[2] We should stress that Iris's lambda calculus is different from our programming language (HeapLang)—the former is typed and pure, whereas the latter is untyped and impure. Consequently, there are two kinds of lambda abstraction ($\lambda x : \tau.\, t$ for Iris and $\lambda x.\, e$ for HeapLang). It should be clear from context which of the lambda abstractions is used.

Figure 4 includes a subset of the Iris grammar. The typing judgement is mostly standard and can be derived from the use of meta variables—we use the meta variables P and Q for propositions (type iProp), the meta variable v for values (type Val), and the meta variables t and u for general terms of any type. Similar to Coq, $\lambda x : \tau.\, t$ is used for both term and type abstraction, and we write $\tau \to \sigma$ for $\forall x : \tau.\, \sigma$ if x is free in σ.

3.3. **Guarded recursion and step-indexing.** Iris is step-indexed [Appel and McAllester 2001; Ahmed 2004], meaning that propositions are indexed by a natural number—referred to as the *step-index*—which is used to stratify a number of semantically cyclic constructs and reasoning principles. Iris employs the logical account of step-indexing [Appel et al. 2007; Dreyer et al. 2011] where the step-index is implicit, and internalised in the logic through the *later modality* ($\triangleright$) [Nakano 2000]. Actris and Iris use step-indexing as follows:

- The principle of Löb induction (rule LÖB) is used to reason about (among others) recursive functions. When proving P, Löb induction lets us assume that a proposition holds *later*, denoted $\triangleright\, P$. The proposition $\triangleright\, P$ is strictly weaker than P, since P entails $\triangleright\, P$ (rule $\triangleright$-INTRO), while the reverse does not hold. The later modality ($\triangleright$) can be eliminated by taking a program step, which is formalised by the Iris proof rule HT-REC. In Actris we use Löb induction to reason about infinite protocols (§ 6.4).

[2] Coq, Iris, and Actris have a predicative Type hierarchy, while propositions are impredicative. For brevity's sake, we omit details about predicativity of Type, as they are standard.

- The guarded recursion operator ($\mu x : \tau.\, t$) lets us construct recursive predicates without a restriction on the variance of x in t. Instead, the variable x should be *guarded*, which means that it should appear under a *contractive* term construct. The prime example of a contractive construct is the later modality ($\triangleright$). The rule μ-UNFOLD says that $\mu x : \tau.\, t$ is in fact a fixpoint of t. Actris's dependent separation protocols $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot$ and $\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot$ are contractive in the tail argument *prot*, and thereby make it possible to use Iris's guarded recursion operator to define recursive protocols (§ 5.4).
- Iris's support for higher-order ghost state [Jung et al. 2016] is used to provide a model of Actris in Iris (§ 9). Additionally, higher-order ghost state is used by Iris to obtain impredicative invariants [Svendsen and Birkedal 2014], which in turn are used to prove the specification of locks [Hobor et al. 2008] used in § 7.

3.4. **Adequacy of Iris.** The adequacy theorem of Iris connects the derivation of Hoare triples to the operational semantics of the programming language. A closed proof of a Hoare triple gives rise to safety and postcondition validity. By safety we mean that the program cannot go wrong, *e.g.,* by resolving an illegal function application (*e.g.,* `true` + 42), or accessing an invalid location (*i.e.,* $!\,\ell$ with $\ell \notin \mathsf{dom}(\sigma)$). Safety is defined formally as:

$$\begin{aligned}\mathsf{safe}\ e \triangleq \forall \sigma, T, \sigma'.\ ([e]; \sigma \to^*_{\mathsf{tp}} T; \sigma') \\ \text{implies } \forall e' \in T.\ (e' \in \mathsf{Val}) \text{ or} \\ (\exists e'', \sigma'', \vec{e}.\ e'; \sigma' \to_{\mathsf{tl}} e''; \sigma''; \vec{e})\end{aligned}$$

This definition is not concerned with whether a program terminates (total correctness).

Postcondition validity means that if the main thread terminates with a value v, then the postcondition holds for that value. This is defined formally as:

$$\begin{aligned}\mathsf{post\_valid}\ (e, \varphi) \triangleq \forall \sigma, v, T, \sigma'.\ (e; \sigma \to^*_{\mathsf{tp}} [v] \cdot T; \sigma') \\ \text{implies } (\varphi\ v)\end{aligned}$$

Theorem 1 Adequacy of Iris**.** *Let* $\varphi \in \mathsf{Val} \to \mathsf{Prop}$ *be a meta-level (i.e., Coq) predicate over values and suppose* $\{\mathsf{True}\}\, e\, \{v.\, \varphi\ v\}$ *is derivable in Iris, then* $\mathsf{safe}\ e$ *and* $\mathsf{post\_valid}\ (e, \varphi)$.

4. The Actris logic

This section describes the core features of Actris 1.0: its dependent separation protocols mechanism (§ 4.1), proof rules (§ 4.2), and its adequacy result (§ 4.3). Actris inherits all features of Iris, which is achieved by defining Actris as an embedded logic in Iris. This means that all of Actris's primitive constructs are defined in Iris, and all of Actris's primitive proof rules are in fact lemmas in Iris. We show how Actris is embedded in Iris in § 9.

4.1. **Dependent separation protocols.** The key feature of Actris is its session-type like dependent separation protocols mechanism. Dependent separation protocols *prot* are streams of $\mathbf{!}\,\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\, prot$ and $\mathbf{?}\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\, prot$ constructors that are either infinite or finite. The finite streams are ultimately terminated by an **end** constructor. The value v denotes the message that is being sent (**!**) or received (**?**), the Iris proposition P denotes the ownership that is transferred along the message, and *prot* denotes the protocol that describes the subsequent messages. The logical variables $\vec{x}{:}\vec{\tau}$ can be used to bind variables in v, P, and

Grammar:

$$\begin{aligned}\tau, \sigma ::= {} & \ldots \mid \mathsf{iProto} \mid \ldots \\ t, u, P, Q, prot ::= {} & \ldots \mid \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot \mid \mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot \mid \mathbf{end} \mid \\ & \overline{prot} \mid prot_1 \cdot prot_2 \mid c \rightarrowtail prot \mid \ldots\end{aligned}$$

Dependent separation protocols:

$$\begin{aligned}\overline{\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot} &= \mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, \overline{prot} & \qquad \overline{\mathbf{end}} &= \mathbf{end} \\ \overline{\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot} &= \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, \overline{prot} & \qquad \overline{\overline{prot}} &= prot \\ (\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot_1) \cdot prot_2 &= \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, (prot_1 \cdot prot_2) & \qquad prot \cdot \mathbf{end} &= prot \\ (\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot_1) \cdot prot_2 &= \mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, (prot_1 \cdot prot_2) & \qquad \mathbf{end} \cdot prot &= prot \\ prot_1 \cdot (prot_2 \cdot prot_3) &= (prot_1 \cdot prot_2) \cdot prot_3 & \qquad \overline{prot_1 \cdot prot_2} &= \overline{prot_1} \cdot \overline{prot_2}\end{aligned}$$

Message passing:

HT-NEW
$$\{\mathsf{True}\}\ \texttt{new\_chan}\ ()\ \{w.\, \exists c_1, c_2.\, w = (c_1, c_2) * c_1 \rightarrowtail prot * c_2 \rightarrowtail \overline{prot}\}$$

HT-SEND
$$\{c \rightarrowtail \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot * P[\vec{t}/\vec{x}]\}\ \texttt{send}\ c\ (v[\vec{t}/\vec{x}])\ \{c \rightarrowtail prot[\vec{t}/\vec{x}]\}$$

HT-RECV
$$\{c \rightarrowtail \mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot\}\ \texttt{recv}\ c\ \{w.\, \exists \vec{y}.\, w = v[\vec{y}/\vec{x}] * c \rightarrowtail prot[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\}$$

Figure 5: The primitive constructs and proof rules of Actris 1.0.

prot. For example, the following dependent separation protocols expresses that a pair of a boolean and an integer reference whose value is at least 10 is sent:[3]

$$\mathbf{!}\,(b : \mathbb{B})\,(\ell : \mathsf{Loc})\,(i : \mathbb{N})\,\langle (b, \ell)\rangle\{\ell \mapsto i * 10 < i\}.\, prot$$

We often omit the proposition $\{P\}$, which simply means it is True.

Apart from the constructors for dependent separation protocols, Actris provides two primitive operations, $\overline{prot}$ and $prot_1 \cdot prot_2$. The $\overline{prot}$ operator denotes the *dual* of a protocol. Similar to conventional session types, it transforms the protocol by changing all sends (**!**) into receives (**?**), and *vice versa*. Taking the dual twice thus results in the original protocol. The operator $prot_1 \cdot prot_2$ *appends* the protocols $prot_1$ and $prot_2$, which is achieved by substituting any **end** in $prot_1$ with $prot_2$.

Channel endpoints are ascribed with dependent separation protocols using the channel endpoint ownership connective $c \rightarrowtail prot$, which captures unique ownership of the channel endpoint c and states that the endpoint follows the protocol *prot*.

4.2. **Actris's proof rules for message passing.** Actris provides proof rules for the three message passing operations `new_chan`, `send`, and `recv` (see § 2.4 for the definition of these operations). The rule HT-NEW allows ascribing any protocol to newly created channels using

[3]Note that $\ell \mapsto i * 10 < i$ is logically equivalent to $\ell \mapsto i \wedge 10 < i$ as $10 < i$ does not describe ownership. As discussed in § 3.1, we prefer the version with separation conjunction.

`new_chan` (), obtaining ownership of $c_1 \rightarrowtail \mathit{prot}$ and $c_2 \rightarrowtail \overline{\mathit{prot}}$ for the respective endpoints. The duality of the protocol guarantees that any receive (**?**) is matched with a send (**!**) by the dual endpoint, which is crucial for establishing safety.

The rule HT-SEND for `send` c w requires the head of the dependent separation protocol of c to be a send (**!**) constructor, and the value w that is sent to match up with the ascribed value. To send a message w, we need to give up ownership of $c \rightarrowtail \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, pick an appropriate instantiation $\vec{t}$ for the variables $\vec{x}{:}\vec{\tau}$ so that $w = v[\vec{t}/\vec{x}]$, give up ownership of the associated resources $P[\vec{t}/\vec{x}]$, and finally regain ownership of the protocol tail $c \rightarrowtail \mathit{prot}[\vec{t}/\vec{x}]$.

The rule HT-RECV for `recv` c is essentially dual to the rule HT-SEND. We need to give up ownership of $c \rightarrowtail \mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, and in return acquire the resources $P[\vec{y}/\vec{x}]$, the return value w where $w = v[\vec{y}/\vec{x}]$, and finally the ownership of the protocol tail $c \rightarrowtail \mathit{prot}[\vec{y}/\vec{x}]$, where $\vec{y}$ is some instantiation of the protocol variables.

Finally, we derive the following specification for the `start` construct from Actris's rule HT-NEW and Iris's rule HT-FORK:

$$\textsc{Ht-start}\quad \frac{\forall c_2.\,\{c_2 \rightarrowtail \overline{\mathit{prot}}\}\ f\ c_2\ \{\mathsf{True}\}}{\{\mathsf{True}\}\ \texttt{start}\ f\ \{c_1.\,c_1 \rightarrowtail \mathit{prot}\}}$$

4.3. **Adequacy of Actris.** By virtue of being an extension of Iris, Actris inherits Iris's adequacy theorem (§ 3.4), which says that a closed proof of a Hoare triple gives rise to safety (programs cannot go wrong) and postcondition validity. In Actris this means that the implementation of the message passing operations (§ 2.4) cannot go wrong, and that transferred messages cannot cause the program to go wrong down the line.

Many conventional session-type systems additionally ensure deadlock freedom—which means that program execution cannot result in a state where all threads are waiting on a message to be sent. Deadlock freedom is ensured through a linear type system and combining thread and channel creation into a `start` primitive. Actris is affine (instead of linear), has a `fork` and `new_chan` primitive (instead of a `start` primitive), and supports locks for channel sharing. Actris thus provides more flexibility in terms of what programs can be written and verified (there exist programs that are deadlock free, but cannot be type-checked using conventional session types, while they can be verified using Actris). On the flip side, using Actris one can prove Hoare triples for programs that deadlock, for example:

$$\{\mathsf{True}\}\ \texttt{let}\ (c, c') ::= \texttt{new\_chan}\ ()\ \texttt{in}\ \texttt{recv}\ c\ \{\mathsf{True}\}$$

Indeed, in our operational semantics programs such as the above are safe. The semantics of both lock acquisition (`acquire`) and message reception (`recv`) is that the thread loops until it succeeds. Loops are considered safe in Iris (and thus also Actris), as the threads in question will continue to take steps, although they will never terminate.

5. A TOUR OF ACTRIS

This section demonstrates the core features of Actris. We introduce and iteratively extend a simple channel-based merge sort algorithm to demonstrate the main features of Actris (§ 5.1–§ 5.6). Note that as the point of the sorting algorithms is to showcase the features of Actris, they are intentionally kept simple and no effort has been made to make them efficient (*e.g.,* to avoid spawning threads for small jobs).

```
sort_service cmp c :=
  let l := recv c in
  if |l| ≤ 1 then send c () else
  let l' := lsplit l in
  let c1 := start (sort_service cmp) in
  let c2 := start (sort_service cmp) in
  send c1 l; send c2 l';
  recv c1; recv c2;
  lmerge cmp l l'; send c ()
```

```
sort_client cmp l :=
  let c :=
    start (sort_service cmp) in
  send c l;
  recv c
```

Figure 6: A channel-based merge sort algorithm (the code for `lmerge` and `lsplit` is standard and thus elided).

5.1. **Basic protocols.** We first prove functional correctness of a simple channel-based merge sort algorithm, whose code is shown in Figure 6. The function `sort_client` $cmp\ l$ takes a comparison function cmp and a linked list l that will be sorted. The function mutates the linked list l, so it returns a unit value $()$ when done. The bulk of the work is done by the `sort_service` $cmp\ c$ function, which takes a channel endpoint c over which it receives a linked list, and over which it sends back $()$ to inform the sender that the list has been sorted. The function `sort_service` is implemented as follows. If the received list is an empty or singleton list, which both are trivially sorted, the function immediately sends back $()$. Otherwise, the list is split into two partitions using `lsplit` l, which mutates the list l to contain the first partition, while returning l' containing the second partition. These partitions are recursively sorted using two newly started instances of `sort_service`. The results of the processes are then requested and merged using `lmerge` $cmp\ l\ l'$, which mutates the list l to contain the merged list. Finally, the unit value $()$ is sent back along the original channel endpoint c.

In order to verify the correctness of the sorting algorithm we first need a specification for the comparison function cmp, which must satisfy the following specification:

$$\begin{aligned}&\mathsf{cmp\_spec}\ (I : T \to \mathsf{Val} \to \mathsf{iProp})\ (R : T \to T \to \mathbb{B})\ (cmp : \mathsf{Val}) \triangleq \\ &\quad (\forall x_1\ x_2.\ R\ x_1\ x_2 \lor R\ x_2\ x_1) * \\ &\quad (\forall x_1\ x_2\ v_1\ v_2.\ \{I\ x_1\ v_1 * I\ x_2\ v_2\}\ cmp\ v_1\ v_2\ \{r.\, r = R\ x_1\ x_2 * I\ x_1\ v_1 * I\ x_2\ v_2\})\end{aligned}$$

This definition is polymorphic in type T. Here, R is a total relation in type T, and I is an interpretation predicate that relates language values to elements of type T. While the relation R dictates the ordering, the interpretation predicate I allows for flexibility about what is ordered. Setting I to *e.g.,* $\lambda x\ v.\, v \mapsto x$ orders references by what they point to in memory, rather than the memory address itself. To specify how lists are laid out in memory we use the following notation:

$$\ell \overset{\text{list}}{\mapsto}_I \vec{x} \triangleq \begin{cases} \ell \mapsto \texttt{inl}\ () & \text{if } \vec{x} = \epsilon \\ \exists v_1\ \ell_2.\, \ell \mapsto \texttt{inr}\ (v_1, \ell_2) * I\ x_1\ v_1 * \ell_2 \overset{\text{list}}{\mapsto}_I \vec{x}_2 & \text{if } \vec{x} = [x_1] \cdot \vec{x}_2 \end{cases}$$

The channel endpoint c adheres to the following dependent separation protocol:

$$\begin{aligned}&\mathsf{sort\_prot}\ (I : T \to \mathsf{Val} \to \mathsf{iProp})\ (R : T \to T \to \mathbb{B}) \triangleq \\ &\quad \mathbf{!}\,(\vec{x} : \mathsf{List}\ T)\ (\ell : \mathsf{Loc})\ \langle \ell \rangle \{\ell \overset{\text{list}}{\mapsto}_I \vec{x}\}.\ \mathbf{?}\vec{y}\ \langle () \rangle \{\ell \overset{\text{list}}{\mapsto}_I \vec{y} * \mathsf{sorted\_of}_R\ \vec{y}\ \vec{x}\}.\ \mathbf{end}\end{aligned}$$

```
sort_service_func c :=
  let cmp := recv c in
  sort_service cmp c

sort_client_func cmp l :=
  let c := start sort_service_func in
  send c cmp; send c l; recv c
```

Figure 7: A version of the sort service that receives the comparison function over the channel.

The protocol describes the interaction of first sending a linked list, and then receiving a unit value () once the list is sorted. The predicate $\mathsf{sorted\_of}_R\ \vec{y}\ \vec{x}$ is true iff $\vec{y}$ is a sorted version of $\vec{x}$ with respect to the relation R. We prove the following specifications of the service and the client:

$$\begin{array}{ll}\{\mathsf{cmp\_spec}\ I\ R\ cmp * c \rightarrowtail \overline{\mathsf{sort\_prot}\ I\ R} \cdot prot\} & \{\mathsf{cmp\_spec}\ I\ R\ cmp * \ell \overset{\text{list}}{\mapsto}_I \vec{x}\} \\ \quad \texttt{sort\_service}\ cmp\ c & \quad \texttt{sort\_client}\ cmp\ \ell \\ \{c \rightarrowtail prot\} & \{\exists \vec{y}.\ \mathsf{sorted\_of}_R\ \vec{y}\ \vec{x} * \ell \overset{\text{list}}{\mapsto}_I \vec{y}\}\end{array}$$

There are two important things to note about these specifications. First, the protocol $\mathsf{sort\_prot}\ I\ R$ is written from the point of view of the client. As such, the precondition for `sort_service` requires that c follows the dual $\overline{\mathsf{sort\_prot}\ I\ R}$. Second, the pre- and postcondition of `sort_service` are generalised to have an arbitrary protocol $prot$ appended at the end. It is important to write specifications this way, so they can be embedded in other protocols. We will see examples of such an embedding in § 5.4 and § 5.5.

The proof of these specifications is almost entirely performed by symbolic execution using the rules HT-NEW, HT-SEND, HT-RECV, and the standard separation logic rules.

Now that we have proven Hoare triples for `sort_service` and `sort_client`, we can use them to prove Hoare triples of other programs that use these functions. Recall that if we use them to prove a Hoare triple of a closed program, we obtain safety and postcondition validity by virtue of Actris's adequacy theorem (§ 3.4).

5.2. **Transferring functions.** The channel-based `sort_service` from the previous section (Figure 6) is parametric on a comparison function. To demonstrate Actris's support for reasoning about functions transferred over channels, we verify the correctness of the function $\texttt{sort\_service}_{func}\ c$ in Figure 7. This function takes a channel endpoint c, over which it receives the comparison function cmp (instead of via a function argument), followed by the list to sort. Similar to the service in § 5.1, it mutates the list, and sends back () when done. To verify this program, we extend the protocol $\mathsf{sort\_prot}$ from § 5.1 as follows:

$$\mathsf{sort\_prot}_{func} \triangleq \mathbf{!}(T : \mathsf{Type})\ (I : T \to \mathsf{Val} \to \mathsf{iProp})\ (R : T \to T \to \mathbb{B})\ (cmp : \mathsf{Val})$$
$$\langle cmp \rangle \{\mathsf{cmp\_spec}\ I\ R\ cmp\}.\ \mathsf{sort\_prot}\ I\ R$$

The new protocol specifies that we first send a comparison function cmp. It includes binders for the polymorphic type T, the interpretation predicate I, and the relation R. The specifications are much the same as before, with the proofs being similar besides the addition of a symbolic execution step to resolve the sending and receiving of the comparison function:

$$\begin{array}{ll}\{c \rightarrowtail \overline{\mathsf{sort\_prot}_{func}} \cdot prot\} & \{\mathsf{cmp\_spec}\ I\ R\ cmp * \ell \overset{\text{list}}{\mapsto}_I \vec{x}\} \\ \quad \texttt{sort\_service}_{func}\ c & \quad \texttt{sort\_client}_{func}\ cmp\ \ell \\ \{c \rightarrowtail prot\} & \{\exists \vec{y}.\ \ell \overset{\text{list}}{\mapsto}_I \vec{y} * \mathsf{sorted\_of}_R\ \vec{y}\ \vec{x}\}\end{array}$$

```
sort_service_rec cmp c :=
  branch c with
   left  ⇒ sort_service cmp c;
           sort_service_rec cmp c
  | right ⇒ ()
  end

sort_client_rec cmp l :=
  let c := start (sort_service_rec cmp) in
  liter (λl'. select c left; send c l'; recv c) l;
  select c right
```

Figure 8: A recursive version of the sort service that can perform multiple jobs in sequence (the code for the function `liter`, which applies a function to each element of the list, is standard and has been elided).

5.3. **Choice.** Branching communication is commonly modelled using the *choice* session types & for branching and ⊕ for selection. We show that corresponding dependent separation protocols can readily be encoded in Actris. At the level of the programming language, the instructions for choice are encoded by sending and receiving a boolean value that is matched using an if-then-else construct:

$$\texttt{select}\ e\ e' \triangleq \texttt{send}\ e\ e'$$

$$\texttt{branch}\ e\ \texttt{with}\ \texttt{left} \Rightarrow e_1 \mid \texttt{right} \Rightarrow e_2\ \texttt{end} \triangleq \texttt{if}\ \texttt{recv}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

The instructions are syntactic sugar, *i.e.,* defined in the meta language (using $\triangleq$), which effectively means that the arguments are evaluated lazily. We define syntactic sugar $\texttt{left} \triangleq \texttt{true}$ and $\texttt{right} \triangleq \texttt{false}$ to be used together with `select` for readability's sake.

Due to the higher-order nature of Actris, the usual protocol specifications for choice from session types can be encoded as regular logical branching within the protocols:

$$prot_1\ {}_{\{Q_1\}}\!\oplus_{\{Q_2\}}\ prot_2 \triangleq \mathbf{!}\,(b : \mathbb{B})\ \langle b\rangle\{\texttt{if}\ b\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2\}.\ \texttt{if}\ b\ \texttt{then}\ prot_1\ \texttt{else}\ prot_2$$

$$prot_1\ {}_{\{Q_1\}}\!\&_{\{Q_2\}}\ prot_2 \triangleq \mathbf{?}(b : \mathbb{B})\ \langle b\rangle\{\texttt{if}\ b\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2\}.\ \texttt{if}\ b\ \texttt{then}\ prot_1\ \texttt{else}\ prot_2$$

We often omit the conditions Q_1 and Q_2, which simply means that they are True. The following rules can be directly derived from the rules Ht-send and Ht-recv:

Ht-select

$$\left\{\begin{array}{l} c \rightarrowtail prot_1\ {}_{\{Q_1\}}\!\oplus_{\{Q_2\}}\ prot_2\ * \\ \texttt{if}\ b\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2 \end{array}\right\}\ \texttt{select}\ c\ b\ \{c \rightarrowtail \texttt{if}\ b\ \texttt{then}\ prot_1\ \texttt{else}\ prot_2\}$$

Ht-branch

$$\frac{\{P * Q_1 * c \rightarrowtail prot_1\}\ e_1\ \{v.\ R\} \qquad \{P * Q_2 * c \rightarrowtail prot_2\}\ e_2\ \{v.\ R\}}{\left\{P * c \rightarrowtail prot_1\ {}_{\{Q_1\}}\!\&_{\{Q_2\}}\ prot_2\right\}\ \texttt{branch}\ c\ \texttt{with}\ \texttt{left} \Rightarrow e_1 \mid \texttt{right} \Rightarrow e_2\ \texttt{end}\ \{v.\ R\}}$$

Apart from branching on boolean values, dependent separation protocols can be used to encode choice on any enumeration type (*e.g.,* lists, natural numbers, days of the week, *etc.*). These encodings follow the same scheme.

5.4. **Recursive protocols.** We now use choice and recursion to verify the correctness of a sorting service that supports performing multiple sorting jobs in sequence. The code of the sorting service $\texttt{sort\_service}_{rec}$ and a possible client $\texttt{sort\_client}_{rec}$ are displayed in Figure 8. The service $\texttt{sort\_service}_{rec}\ cmp\ c$ takes a comparison function *cmp* and a channel endpoint c, and returns (). It contains a loop in which choice is used to either

terminate the service, or to sort an individual list using the channel-based merge sort algorithm `sort_service` from § 5.1. The client $\texttt{sort\_client}_{rec}\ cmp\ l$ takes a comparison function *cmp* and a nested linked list of linked lists l, and returns (). It starts a single instance of the service at channel endpoint c, and then sequentially sends requests to sort each inner linked list l' in l. Finally, the client selects the terminating branch to end the communication with the service. A protocol for interacting with the sorting service can be defined as follows:

$$\mathsf{sort\_prot}_{rec}\ (I : T \to \mathsf{Val} \to \mathsf{iProp})\ (R : T \to T \to \mathbb{B}) \triangleq \\ \mu(rec : \mathsf{iProto}).\ (\mathsf{sort\_prot}\ I\ R \cdot rec) \oplus \mathbf{end}$$

The protocol uses the choice operator $\oplus$ to specify that the client may either request the service to perform a sorting job, or terminate communication with the service. After the job has been finished the protocol proceeds recursively.

We use Iris's operator $\mu x : \tau.\ t$ for guarded recursion (§ 3.3) to define recursive protocols. It is important to recall that—as is usual in logics with guarded recursion—the variable x should appear under a *contractive* term construct in the body t of $\mu x : \tau.\ t$. In our protocol, the recursive variable *rec* appears under the argument of $\oplus$, which is defined in terms of $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ prot$, which, similarly to $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ prot$, is contractive in the tail protocol *prot*. We can then prove the following specifications of the service and the client:

$$\left\{\begin{array}{l}\mathsf{cmp\_spec}\ I\ R\ cmp\ * \\ c \rightarrowtail \overline{\mathsf{sort\_prot}_{rec}\ I\ R} \cdot prot\end{array}\right\} \\ \quad \texttt{sort\_service}_{rec}\ cmp\ c \\ \{c \rightarrowtail prot\}$$

$$\left\{\mathsf{cmp\_spec}\ I\ R\ cmp * \ell \overset{\text{list}}{\mapsto}_J \vec{\vec{x}}\right\} \\ \quad \texttt{sort\_client}_{rec}\ cmp\ \ell \\ \left\{\exists \vec{\vec{y}}.\ |\vec{\vec{y}}| = |\vec{\vec{x}}| * \ell \overset{\text{list}}{\mapsto}_J \vec{\vec{y}} * (\forall i < |\vec{\vec{x}}|.\ \mathsf{sorted\_of}_R\ \vec{\vec{y}}_i\ \vec{\vec{x}}_i)\right\}$$

We let $J \triangleq \lambda \ell'\ \vec{y}.\ \ell' \overset{\text{list}}{\mapsto}_I \vec{y}$ to express that ℓ points to a list of lists $\vec{\vec{x}}$. The proof of the service follows naturally by symbolic execution using the induction hypothesis (obtained from Löb), the rules Ht-branch and Ht-select, and the specification of `sort_service`. Note that we rely on the specification of `sort_service` having an arbitrary protocol as its suffix.

It is worth pointing out that protocols in Actris provide a lot of flexibility. Using just minor changes, we can extend the protocol to support transferring a comparison function over the channel, like the extension made in $\texttt{sort\_client}_{func}$, or in a way such that a different comparison function can be used for each sorting job.

5.5. **Higher-order protocols.** Higher-order communication is a common feature within communication protocols, and particularly the session-types community—it is the concept of transferring a channel endpoint over a channel, often called delegation. Due to the impredicativity of dependent separation protocols in Actris, higher-order reasoning about programs with delegation is readily available. The protocols $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ prot$ and $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ prot$ can simply refer to the channel endpoint ownership $c \rightarrowtail prot'$ in the proposition P.

An example of a program that uses delegation is the $\texttt{sort\_service}_{del}$ variant of the recursive sorting service in Figure 9, which allows multiple sorting jobs to be performed in parallel. The function $\texttt{sort\_service}_{del}\ cmp\ c$ takes a comparison function *cmp*, a channel endpoint c, and returns (). Using the channel endpoint c, a client can request the service to start a new inner sorting service c', which the service delegates over channel endpoint c.

Similar to the client in § 5.4, the client $\texttt{sort\_client}_{del}\ cmp\ l$ takes a comparison function *cmp* and a nested linked list of linked lists l, and returns (). The client starts a connection c to the service, and for each inner list l', it acquires a delegated channel endpoint c', over

```
sort_service_del cmp c :=
  branch c with
   left  ⇒
     let c' :=
       start (sort_service cmp) in
     send c c';
     sort_service_del cmp c
  | right ⇒ ()
  end
```

```
sort_client_del cmp l :=
  let c := start (sort_service_del cmp) in
  let k := lnil () in
  liter (λl'. select c left;
              let c' := recv c in
              send c' l'; lcons c' k) l
  select c right;
  liter recv k
```

Figure 9: A recursive version of the sort service that uses delegation to perform multiple jobs in parallel (the code for the function `lcons`, which pushes an element to the head of a list, has been elided).

which it sends the inner list l' that should be sorted. The client keeps track of all channels to delegated services in a linked list k so that it can wait for all of them to finish (using `liter recv`).

A protocol for the delegation service can be defined as follows, denoting that the client can select whether to acquire a connection to a new delegated service or to terminate:

$$\mathsf{sort\_prot}_{del}\ (I : T \to \mathsf{Val} \to \mathsf{iProp})\ (R : T \to T \to \mathbb{B}) \triangleq \\ \mu(rec : \mathsf{iProto}).\ (\mathbf{?}(c : \mathsf{Val})\ \langle c \rangle \{c \rightarrowtail \mathsf{sort\_prot}\ I\ R\}.\ rec) \oplus \mathbf{end}$$

We can then prove the following specifications of the service and the client:

$$\left\{ \begin{array}{l} \mathsf{cmp\_spec}\ I\ R\ cmp\ * \\ c \rightarrowtail \overline{\mathsf{sort\_prot}_{del}\ I\ R} \cdot prot \end{array} \right\} \\ \quad \mathtt{sort\_service}_{del}\ cmp\ c \\ \{c \rightarrowtail prot\}$$

$$\left\{ \mathsf{cmp\_spec}\ I\ R\ cmp * \ell \overset{\text{list}}{\mapsto}_J \vec{\vec{x}} \right\} \\ \quad \mathtt{sort\_client}_{del}\ cmp\ \ell \\ \left\{ \exists \vec{\vec{y}}.\ |\vec{\vec{y}}| = |\vec{\vec{x}}| * \ell \overset{\text{list}}{\mapsto}_J \vec{\vec{y}} * (\forall i < |\vec{\vec{x}}|.\ \mathsf{sorted\_of}_R\ \vec{\vec{y}}_i\ \vec{\vec{x}}_i) \right\}$$

As before, we let $J \triangleq \lambda \ell'\ \vec{y}.\ \ell' \overset{\text{list}}{\mapsto}_I \vec{y}$ to express that ℓ points to a list of lists $\vec{\vec{x}}$. Once again the proofs are straightforward, as they are simply a combination of recursive reasoning combined with the application of Actris's rules for channels.

5.6. **Dependent protocols.** The protocols we have seen so far have only made limited use of Actris's support for recursion. We now demonstrate Actris's support for dependent protocols, which make it possible to keep track of the history of what messages have been sent and received. We demonstrate this feature by considering a fine-grained version of the channel-based merge-sort service as shown in Figure 10. Like previous versions, the function $\mathtt{sort\_service}_{fg}\ cmp\ c$ takes a comparison function cmp and a channel endpoint c, and returns $()$. However, unlike previous versions, the input list should be transferred element by element over the channel endpoint c to the service, and when done, the service sends back the sorted list element by element. We use choice to indicate whether the whole list has been sent (`right`) or another element remains to be sent (`left`).

The structure of $\mathtt{sort\_service}_{fg}$ is somewhat similar to the coarse-grained merge-sort algorithm that we have seen before. The base cases of the empty or the singleton list are handled initially. This is achieved by waiting for at least two values before starting the recursive sub-services c_1 and c_2. In the base cases the values are sent back immediately,

```
sort_service_fg cmp c :=
  branch c with
    right ⇒ select c right
  | left  ⇒
    let x_1 := recv c in
    branch c with
      right ⇒ select c left; send c x_1;
               select c right
    | left  ⇒
      let x_2 := recv c in
      let c_1 := start (sort_service_fg cmp) in
      let c_2 := start (sort_service_fg cmp) in
      select c_1 left; send c_1 x_1;
      select c_2 left; send c_2 x_2;
      split_fg c c_1 c_2; merge_fg cmp c c_1 c_2
    end
  end

split_fg c c_1 c_2 :=
  branch c with
    right ⇒ select c_1 right;
             select c_2 right
  | left  ⇒
    let x := recv c in
    select c_1 left; send c_1 x;
    split_fg c c_2 c_1
  end

merge_fg cmp c c_1 c_2 :=
  branch c_1 with
    right ⇒ assert false
  | left  ⇒
    let x := recv c_1 in
    merge^aux_fg cmp c x c_1 c_2
  end

merge^aux_fg cmp c x c_1 c_2 :=
  branch c_2 with
    right ⇒ select c left; send c x_1;
             transfer c_1 c
  | left  ⇒
    let y := recv c_2 in
    if cmp x y then
      select c left; send c x;
      merge^aux_fg cmp c y c_2 c_1
    else
      select c left; send c y;
      merge^aux_fg cmp c x c_1 c_2
  end

sort_client_fg cmp l :=
  let c :=
    start (sort_service_fg cmp) in
  send_all c l; recv_all c l
```

Figure 10: A fine-grained version of the sort service that transfers elements one by one (the code for the functions `transfer`, `send_all`, and `recv_all` has been elided).

as they are trivially sorted. The inductive case is handled by starting two sub-services at the channel endpoints c_1 and c_2. First, each of the channel endpoints are sent one of the two initially received elements. The remaining elements are then received by the parent service on c, and forwarded to the sub-services alternatingly on c_1 and c_2, using the function $\texttt{split}_{fg}\ c\ c_1\ c_2$. Once the `right` flag is received, the $\texttt{split}_{fg}$ function terminates, and the algorithm moves to the second phase.

In the second phase, the function $\texttt{merge}_{fg}\ cmp\ c\ c_1\ c_2$ is used to merge the stream of elements returned by the sub-services on c_1 and c_2 and forwards them to the parent service on c. It initially acquires the first element x from the first sub-service on c_1, which it passes to the auxiliary function $\texttt{merge}^{aux}_{fg}$ as the current largest value. The auxiliary function $\texttt{merge}^{aux}_{fg}\ cmp\ c\ x\ c_1\ c_2$ recursively requests a value y from the sub-service from which the current largest value was not acquired from (initially c_2). It then compares x and y using the comparison function cmp, and forwards the smallest element on c. This is repeated until the `right` flag is received from either sub-service, after which the remaining values of the other sub-service are forwarded to the parent service on c using $\texttt{transfer}\ c_1\ c$.

The interface of the client $\texttt{sort\_client}_{fg}\ cmp\ l$ is similar to the one from § 5.1 and 5.2. It takes a comparison function cmp and a linked lists l, sorts the linked list l, and returns () when done. The client sorts the list l by sending its elements to the sort service using the $\texttt{send\_all}\ c\ l$ function (which mutates the list l by removing all of its values and sending them over the channel c), and puts the received values back into the linked list using the $\texttt{recv\_all}\ c\ l$ function (which also mutates the list l). A suitable protocol for proving functional correctness of the fine-grained sorting service is as follows:

$$\mathsf{sort\_prot}_{fg}\ (I : T \to \mathsf{Val} \to \mathsf{iProp})\ (R : T \to T \to \mathbb{B}) \triangleq \mathsf{sort\_prot}^{head}_{fg}\ I\ R\ \epsilon$$

$$\mathsf{sort\_prot}^{head}_{fg}\ (I : T \to \mathsf{Val} \to \mathsf{iProp})\ (R : T \to T \to \mathbb{B}) \triangleq \mu(rec : \mathsf{List}\ T \to \mathsf{iProto}).$$
$$\lambda \vec{x}.\ (\mathbf{!}\ (x : T)\ (v : \mathsf{Val})\ \langle v \rangle \{I\ x\ v\}.\ rec\ (\vec{x} \cdot [x])) \oplus \mathsf{sort\_prot}^{tail}_{fg}\ I\ R\ \vec{x}\ \epsilon$$

$$\mathsf{sort\_prot}^{tail}_{fg}\ (I : T \to \mathsf{Val} \to \mathsf{iProp})\ (R : T \to T \to \mathbb{B}) \triangleq \mu(rec : \mathsf{List}\ T \to \mathsf{List}\ T \to \mathsf{iProto}).$$
$$\lambda \vec{x}\ \vec{y}.\ (\mathbf{?}(y : T)\ (v : \mathsf{Val})\ \langle v \rangle \{(\forall i < |\vec{y}|.\ R\ \vec{y}_i\ y) * I\ y\ v\}.\ rec\ \vec{x}\ (\vec{y} \cdot [y])) \ \&_{\{\vec{x} \equiv_{\mathsf{p}} \vec{y}\}}\ \mathbf{end}$$

The protocol is split into two phases $\mathsf{sort\_prot}^{head}_{fg}$ and $\mathsf{sort\_prot}^{tail}_{fg}$, mimicking the behaviour of the program. The $\mathsf{sort\_prot}^{head}_{fg}$ phase is indexed by the values $\vec{x}$ that have been sent so far. The protocol describes that one can either send another value and proceed recursively, or stop, which moves the protocol to the next phase.

The $\mathsf{sort\_prot}^{tail}_{fg}$ phase is dependent on the list of values $\vec{x}$ received in the first phase, and the list of values $\vec{y}$ returned so far. The condition $(\forall i < |\vec{y}|.\ R\ \vec{y}_i\ y)$ states that the received element is larger than any of the elements that have previously been returned, which maintains the invariant that the stream of received elements is sorted. When the `right` flag is received $\vec{x} \equiv_{\mathsf{p}} \vec{y}$ shows that the received values $\vec{y}$ are a permutation of the ones $\vec{x}$ that were sent, making sure that all of the sent elements have been accounted for.

We can then prove top-level specifications for the service and client that are similar to the coarse-grained version of the channel-based merge sort:

$$\{\mathsf{cmp\_spec}\ I\ R\ cmp * c \rightarrowtail \overline{\mathsf{sort\_prot}_{fg}\ I\ R} \cdot prot\}$$
$$\mathsf{sort\_prot}_{fg}\ c$$
$$\{c \rightarrowtail prot\}$$

$$\{\mathsf{cmp\_spec}\ I\ R\ cmp * \ell \overset{\text{list}}{\mapsto}_I \vec{x}\}$$
$$\texttt{sort\_client}_{fg}\ cmp\ \ell$$
$$\{\exists \vec{y}.\ \ell \overset{\text{list}}{\mapsto}_I \vec{y} * \mathsf{sorted\_of}_R\ \vec{y}\ \vec{x}\}$$

Proving these specifications requires one to pick appropriate specifications for the auxiliary functions to capture the required invariants with regard to sorting. After having picked these specifications, the parts of the proofs that involve communication are mostly straightforward, but require a number of trivial auxiliary results about sorting and permutations.

6. Subprotocols

This section describes **Actris 2.0**, which extends Actris 1.0—as presented in the conference version of this paper [Hinrichsen et al. 2020]—with *subprotocols*, inspired by asynchronous subtyping of session types [Mostrous et al. 2009; Mostrous and Yoshida 2015]. The intention of both of these relations is to capture protocol-preserving changes, that allow for some internal flexibility of how an endpoint fulfills a protocol, while being indistinguishable by the other endpoint. In particular, subprotocols have two key features. First, they exploit the asynchronous semantics of channels by relaxing the notion of duality, thereby making it possible to prove functional correctness of a larger class of programs. Second, they give rise to a more extensional approach to reasoning about dependent separation protocols, as

we can work up to the subprotocol relation rather than equality, thereby providing more flexibility in the design and reuse of protocols.

We first introduce Actris 2.0's subprotocol relation and its proof rules (§ 6.1). These should (similar to the Actris 1.0 logic, presented in § 4) be considered to be primitives of Actris; in § 9.2 we define and prove them in Iris. We then show how subprotocols can be employed to prove a mapper service, which handles requests one at a time, while its client may send multiple requests up front (§ 6.2). Next, we demonstrate how the subprotocol relation allows for the composition of slightly differing protocols, by composing a list reversal service whose protocol is based on a list predicate that does not carry ownership, with a client whose protocol is based on a list predicate that does carry ownership (§ 6.3). Finally, we show that the subprotocol relation is coinductive, and, when combined with Löb induction, can be used to reason about recursive protocols (§ 6.4).

6.1. **The subprotocol relation.** The dependent separation protocols of channel endpoints are picked on channel creation (using the rule Ht-new shown in Figure 5), which then determines how the channel endpoints should interact. To ensure safe communication, Actris adapts the notion of duality from session types, which requires every send (**!**) of one endpoint to be paired with a receive (**?**) for the other endpoint, and *vice versa*. However, working with a channel's protocol and its dual is more restrictive than strictly necessary. Some variations from the original protocol preserve the externally observed interaction, as the other endpoint is agnostic to the variations in question, which will be made clear momentarily. We capture some of these so-called protocol-preserving variations via a new notion—the *subprotocol relation*:

$$prot_1 \sqsubseteq prot_2$$

The subprotocol relation describes that protocol $prot_1$ is *stronger* than $prot_2$, or conversely, that protocol $prot_2$ is *weaker* than $prot_1$. More specifically, this means that $prot_2$ can be used *in place of* $prot_1$ whenever such a protocol is expected during verification. This property is captured by the following monotonicity rule for channel ownership:

$$\frac{c \rightarrowtail prot_1 \qquad prot_1 \sqsubseteq prot_2}{c \rightarrowtail prot_2}$$

The subprotocol relation is inspired by asynchronous subtyping for session types [Mostrous et al. 2009; Mostrous and Yoshida 2015], which allows (1) sending subtypes (contravariance), (2) receiving supertypes (covariance), and (3) swapping sends ahead of receives. These variations preserve the protocol, as (1) the originally expected type that is to be sent can be derived from the subtype, (2) the originally expected type to be received can be derived from the supertype, and (3) sends do not block because channels are buffered in both directions, so messages can be enqueued ahead of time. These variations, including the swapping property, are generalised to dependent separation protocols using the following proof rules:

⊑-send-mono'
$$\frac{\forall \vec{x}{:}\vec{\tau}.\, P_2 \mathrel{-\!\!*} P_1 \qquad \forall \vec{x}{:}\vec{\tau}.\, prot_1 \sqsubseteq prot_2}{\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P_1\}.\, prot_1 \sqsubseteq \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P_2\}.\, prot_2}$$

⊑-recv-mono'
$$\frac{\forall \vec{x}{:}\vec{\tau}.\, P_1 \mathrel{-\!\!*} P_2 \qquad \forall \vec{x}{:}\vec{\tau}.\, prot_1 \sqsubseteq prot_2}{\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P_1\}.\, prot_1 \sqsubseteq \mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P_2\}.\, prot_2}$$

⊑-swap'
$$\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathbf{!}\,\vec{y}{:}\vec{\sigma}\,\langle w\rangle\{Q\}.\, prot \sqsubseteq \mathbf{!}\,\vec{y}{:}\vec{\sigma}\,\langle w\rangle\{Q\}.\,\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot$$

The rules ⊑-SEND-MONO' and ⊑-RECV-MONO' use *separation implication* $P \mathrel{-\!\!*} Q$—which states that ownership of Q can be obtained by giving up ownership of P—to mimic the contra- and covariance of session subtyping. The rule ⊑-SWAP' states that sends can be swapped ahead of receives. To be well-formed, this rule has the implicit side condition that $\vec{x}{:}\vec{\tau}$ does not bind into w and Q, and that $\vec{y}{:}\vec{\sigma}$ does not bind into v and P.

To give an intuition behind the protocol-consistent changes that the above rules capture, consider the following subprotocol derivation:

$$\begin{aligned}
&\mathbf{?}(i : \mathbb{Z})\,\langle i\rangle\{i < 42\}.\,\mathbf{!}\,(j : \mathbb{Z})\,\langle j\rangle\{j > 42\}.\,\mathit{prot} && \sqsubseteq\text{-SEND-MONO'}\\
\sqsubseteq\ &\mathbf{?}(i : \mathbb{Z})\,\langle i\rangle\{i < 42\}.\,\mathbf{!}\,(j : \mathbb{Z})\,\langle j\rangle\{j > 50\}.\,\mathit{prot} && \sqsubseteq\text{-RECV-MONO'}\\
\sqsubseteq\ &\mathbf{?}(i : \mathbb{Z})\,\langle i\rangle\{i < 40\}.\,\mathbf{!}\,(j : \mathbb{Z})\,\langle j\rangle\{j > 50\}.\,\mathit{prot} && \sqsubseteq\text{-SWAP'}\\
\sqsubseteq\ &\mathbf{!}\,(j : \mathbb{Z})\,\langle j\rangle\{j > 50\}.\,\mathbf{?}(i : \mathbb{Z})\,\langle i\rangle\{i < 40\}.\,\mathit{prot}
\end{aligned}$$

Here, we first strengthen the proposition of the send (by increasing the bound from $j > 42$ to $j > 50$), then weaken the proposition of the receive (by reducing the bound from $i < 42$ to $i < 40$), and finally swap the send ahead of the receive.

While the aforementioned rules cover the intuition behind Actris's subprotocol relation, Actris's actual subprotocol rules provide a number of additional features:

(1) They can be used to manipulate the logical variables $\vec{x}{:}\vec{\tau}$ that appear in protocols.
(2) They can be used to transfer ownership of resources in and out of messages.
(3) They can be used to reason about recursive protocols defined using LÖB induction.

The full set of primitive rules for subprotocols is shown in Figure 11. The first four rules account for logical variable manipulation and resource transfer: Rules ⊑-SEND-OUT and ⊑-RECV-OUT generalise over the logical variables $\vec{x}{:}\vec{\tau}$ and transfer ownership of P out of the weaker sending protocol $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, and stronger receiving protocol $\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, respectively. Rule ⊑-SEND-IN weakens a sending protocol $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ by instantiating the logical variables $\vec{x}{:}\vec{\tau}$ and transferring ownership of $P[\vec{t}/\vec{x}]$ into the protocol. Dually, the rule ⊑-RECV-IN strengthens a receiving protocol $\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ by instantiating the logical variables $\vec{x}{:}\vec{\tau}$ and transferring ownership of $P[\vec{t}/\vec{x}]$ into the protocol.

To demonstrate the intuition behind these rules consider the following proof of the subprotocol relation presented in § 1.3, where we transfer ownership of $\ell_1' \mapsto 20$ into a protocol, while instantiating the logical variable ℓ_1 with ℓ_1':

$$\dfrac{\dfrac{}{\ell_1' \mapsto 20 * \ell_2 \mapsto 22 \mathrel{-\!\!*} \ell_1' \mapsto 20 * \ell_2 \mapsto 22}\ \sqsubseteq\text{-SEND-IN}}{\begin{array}{l}\ell_1' \mapsto 20 * \ell_2 \mapsto 22 \mathrel{-\!\!*}\\ \quad \mathbf{!}\,(\ell_1, \ell_2{:}\mathsf{Loc})\,\langle(\ell_1, \ell_2)\rangle\{\ell_1 \mapsto 20 * \ell_2 \mapsto 22\}.\,\mathit{prot} \sqsubseteq \mathbf{!}\,\langle(\ell_1', \ell_2)\rangle.\,\mathit{prot}\end{array}\ \sqsubseteq\text{-SEND-OUT}}$$

$$\begin{array}{l}\ell_1' \mapsto 20 \mathrel{-\!\!*}\\ \quad \mathbf{!}\,(\ell_1, \ell_2{:}\mathsf{Loc})\,\langle(\ell_1, \ell_2)\rangle\{\ell_1 \mapsto 20 * \ell_2 \mapsto 22\}.\,\mathit{prot} \sqsubseteq\\ \quad \mathbf{!}\,(\ell_2{:}\mathsf{Loc})\,\langle(\ell_1', \ell_2)\rangle\{\ell_2 \mapsto 22\}.\,\mathit{prot}\end{array}$$

We first use rule ⊑-SEND-OUT to generalise over the logical variable ℓ_2 and transfer ownership of $\ell_2 \mapsto 22$ out of the weaker protocol (*i.e.,* the send on the RHS), and then use ⊑-SEND-IN to instantiate the logical variables ℓ_1' and ℓ_2 and transfer ownership of $\ell_1' \mapsto 20$ and $\ell_2 \mapsto 22$ into the stronger protocol (*i.e.,* the send on the LHS).

The rules for monotonicity (⊑-SEND-MONO and ⊑-RECV-MONO) and swapping (⊑-SWAP) in Figure 11 differ in two aspects from the rules for monotonicity (⊑-SEND-MONO' and ⊑-RECV-MONO') and swapping (⊑-SWAP') that we have seen in the beginning of this section. First, the actual rules only apply to protocols whose head does not have logical variables

Grammar:

$$t, u, P, Q, prot ::= \ldots \mid prot_1 \sqsubseteq prot_2 \mid \ldots$$

Logical variable manipulation and resource transfer:

$$\frac{\forall \vec{x}{:}\vec{\tau}.\ P \twoheadrightarrow \left(prot_1 \sqsubseteq \mathbf{!}\,\langle v\rangle.\ prot_2\right)}{prot_1 \sqsubseteq \mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ prot_2}\ prot_1 \neq \mathbf{end}\quad (\sqsubseteq\text{-SEND-OUT})$$

$$\frac{P[\vec{t}/\vec{x}]}{\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ prot \sqsubseteq \mathbf{!}\,\langle v[\vec{t}/\vec{x}]\rangle.\ prot[\vec{t}/\vec{x}]}\quad (\sqsubseteq\text{-SEND-IN})$$

$$\frac{\forall \vec{x}{:}\vec{\tau}.\ P \twoheadrightarrow \left(\mathbf{?}\langle v\rangle.\ prot_1 \sqsubseteq prot_2\right)}{\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ prot_1 \sqsubseteq prot_2}\ prot_2 \neq \mathbf{end}\quad (\sqsubseteq\text{-RECV-OUT})$$

$$\frac{P[\vec{t}/\vec{x}]}{\mathbf{?}\langle v[\vec{t}/\vec{x}]\rangle.\ prot[\vec{t}/\vec{x}] \sqsubseteq \mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ prot}\quad (\sqsubseteq\text{-RECV-IN})$$

Monotonicity and swapping:

$$\frac{\triangleright(prot_1 \sqsubseteq prot_2)}{\mathbf{!}\,\langle v\rangle.\ prot_1 \sqsubseteq \mathbf{!}\,\langle v\rangle.\ prot_2}\quad (\sqsubseteq\text{-SEND-MONO})$$

$$\frac{\triangleright(prot_1 \sqsubseteq prot_2)}{\mathbf{?}\langle v\rangle.\ prot_1 \sqsubseteq \mathbf{?}\langle v\rangle.\ prot_2}\quad (\sqsubseteq\text{-RECV-MONO})$$

$$\mathbf{?}\langle v\rangle.\,\mathbf{!}\,\langle w\rangle.\ prot \sqsubseteq \mathbf{!}\,\langle w\rangle.\,\mathbf{?}\langle v\rangle.\ prot\quad (\sqsubseteq\text{-SWAP})$$

Reflexivity and transitivity:

$$prot \sqsubseteq prot\quad (\sqsubseteq\text{-REFL})$$

$$\frac{prot_1 \sqsubseteq prot_2 \qquad prot_2 \sqsubseteq prot_3}{prot_1 \sqsubseteq prot_3}\quad (\sqsubseteq\text{-TRANS})$$

Dual and append:

$$\frac{prot_2 \sqsubseteq prot_1}{\overline{prot_1} \sqsubseteq \overline{prot_2}}\quad (\sqsubseteq\text{-DUAL})$$

$$\frac{prot_1 \sqsubseteq prot_2 \qquad prot_3 \sqsubseteq prot_4}{prot_1 \cdot prot_3 \sqsubseteq prot_2 \cdot prot_4}\quad (\sqsubseteq\text{-APPEND})$$

Channel ownership:

$$\frac{c \rightarrowtail prot_1 \qquad prot_1 \sqsubseteq prot_2}{c \rightarrowtail prot_2}\quad (\sqsubseteq\text{-CHAN-MONO})$$

Figure 11: The grammar and primitive rules of Actris 2.0 for subprotocols.

$\vec{x}{:}\vec{\tau}$ and resources P, *i.e.,* protocols of the shape $\mathbf{!}\,\langle v\rangle.\ prot$ or $\mathbf{?}\langle v\rangle.\ prot$, instead of those of the shape $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ prot$ or $\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ prot$. While this restriction might seem to make the rules more restrictive, the more general rules for monotonicity ($\sqsubseteq$-SEND-MONO' and $\sqsubseteq$-RECV-MONO') and swapping ($\sqsubseteq$-SWAP') are derivable from these simpler rules. This is done using the rules for logical variable manipulation and resource transfer. Second, the actual rules for monotonicity have a later modality ($\triangleright$) in their premise. The later modality makes these rules stronger (by $\triangleright$-INTRO we have that P entails $\triangleright P$), and thereby internalizes its coinductive nature into the Actris logic so LÖB induction can be used to prove subprotocol relations for recursive protocols (§ 6.4).

```
mapper_service f_v c :=
  branch c with
   left  ⇒ let x := recv c in
           let y := f_v x in
           send c y;
           mapper_service f_v c
  | right ⇒ ()
  end
```

```
mapper_client f_v l :=
  let c := start (mapper_service f_v) in
  let n := |l| in
  send_all c l;
  recvN c l n;
  select c right;
```

Figure 12: A mapper service whose verification relies on swapping (the code for the functions `send_all` and `recvN` has been elided).

The remaining rules in Figure 11 express that the subprotocol relation is reflexive (⊑-REFL) and transitive (⊑-TRANS), as well as that the dual operation is anti-monotone (⊑-DUAL) and the append operation is monotone (⊑-APPEND).

Let us consider the following subprotocol relation to provide some further insight into the expressivity of our rules, (where logical variables are omitted for simplicity):

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

Here we extend the protocol $!\,\langle v\rangle\{P\}.\,?\langle w\rangle\{Q\}.\,prot$ with a *frame* R. The proposition R describes resources that can be sent along with the originally expected resources P, and which are reacquired along with the resources Q that are sent back. We demonstrate the usefulness of this notion of framing at the protocol level in § 6.3.

The above subprotocol relation mimics the frame rule of separation logic (HT-FRAME), which makes it possible to apply specifications while maintaining a *frame* of resources R:

$$\frac{\{P\}\ e\ \{w.\,Q\}}{\{P * R\}\ e\ \{w.\,Q * R\}}$$

The frame-like subprotocol relation is proven as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Q * R \twoheadrightarrow\ ?\langle w\rangle.\,prot \sqsubseteq\ ?\langle w\rangle\{Q * R\}.\,prot}\ \sqsubseteq\text{-RECV-IN}}{R \twoheadrightarrow\ ?\langle w\rangle\{Q\}.\,prot \sqsubseteq\ ?\langle w\rangle\{Q * R\}.\,prot}\ \sqsubseteq\text{-RECV-OUT}}{R \twoheadrightarrow\ !\,\langle v\rangle.\,?\langle w\rangle\{Q\}.\,prot \sqsubseteq\ !\,\langle v\rangle.\,?\langle w\rangle\{Q * R\}.\,prot}\ \sqsubseteq\text{-SEND-MONO},\ \triangleright\text{-INTRO}}{P * R \twoheadrightarrow\ !\,\langle v\rangle\{P\}.\,?\langle w\rangle\{Q\}.\,prot \sqsubseteq\ !\,\langle v\rangle.\,?\langle w\rangle\{Q * R\}.\,prot}\ \sqsubseteq\text{-SEND-IN},\ \sqsubseteq\text{-TRANS}}{!\,\langle v\rangle\{P\}.\,?\langle w\rangle\{Q\}.\,prot \sqsubseteq\ !\,\langle v\rangle\{P * R\}.\,?\langle w\rangle\{Q * R\}.\,prot}\ \sqsubseteq\text{-SEND-OUT}$$

We use rule ⊑-SEND-OUT to transfer P and the frame R out of the weaker protocol (*i.e.,* the send on the RHS), and then use rule ⊑-SEND-IN to transfer P into the stronger protocol (*i.e.,* the send on the LHS), leaving us with a context in which we still own the frame R. We then use rule ⊑-SEND-MONO to proceed with the receiving part of the protocol in a dual fashion—we use rule ⊑-RECV-OUT to transfer out Q of the stronger protocol (*i.e.,* the receive on the LHS), and use rule ⊑-RECV-IN to transfer Q and the frame R into the weaker protocol (*i.e.,* the receive on the RHS).

6.2. **Swapping.** Subprotocols make it possible to verify message-passing programs whose order of sends and receives does not match up w.r.t. duality. As an example of such a program, let us consider the mapper service and client in Figure 12. The service `mapper_service` $f_\mathsf{v}\ c$ is a loop, which iteratively receives an element over channel endpoint c, maps the function f_v over that element, and sends the resulting value back. Conversely, the client `mapper_client` $f_\mathsf{v}\ l$ sends all of the elements of the list l up front, and only requests the mapped results back once all elements have been sent. Since the service interleaves the sends and receives, while the client does not, the dependent separation protocols for the service and client cannot be dual of each other. However, the communication between the service and client is in fact safe as messages are buffered. We now show that using subprotocols we can prove that this is indeed the case. We define the protocol based on the communication where sends and receives are interleaved:

$$\begin{aligned}&\texttt{mapper\_prot}\ (I_T : T \to \mathsf{Val} \to \mathsf{iProp})\ (I_U : U \to \mathsf{Val} \to \mathsf{iProp})\ (f : T \to U) \triangleq \\ &\quad \mu(rec : \mathsf{iProto}).\ (\mathbf{!}\,(x : T)\ (v : \mathsf{Val})\,\langle v\rangle\{I_T\ x\ v\}.\,\mathbf{?}(w : \mathsf{Val})\,\langle w\rangle\{I_U\ (f\ x)\ w\}.\ rec) \oplus \mathbf{end}\end{aligned}$$

The protocol is parameterised by representation predicates I_T and I_U that relate HeapLang values to elements of type T and U in the Iris/Actris logic, and a function $f : T \to U$ in Iris/Actris that specifies the behaviour of the HeapLang function f_v. The connection between f and f_v is formalised as:

$$\begin{aligned}&\mathsf{f\_spec}\ (I_T : T \to \mathsf{Val} \to \mathsf{iProp})\ (I_U : U \to \mathsf{Val} \to \mathsf{iProp})\ (f : T \to U)\ (f_\mathsf{v} : \mathsf{Val}) \triangleq \\ &\quad \forall x\, v.\ \{I_T\ x\ v\}\ f_\mathsf{v}\ v\ \{w.\ I_U\ (f\ x)\ w\}\end{aligned}$$

Since `mapper_prot` describes an interleaved sequence of transactions, `mapper_service` can be readily verified against the protocol $\overline{\texttt{mapper\_prot}}$ using just the symbolic execution rules of Actris 1.0 as presented in § 4.2. However, to verify `mapper_client` against the protocol `mapper_prot`, we need to weaken the protocol using the rules for subprotocols of Actris 2.0. Given a list of n elements, the subprotocol relation (together with an intermediate step) that describes this weakening is:

$$\begin{aligned}&\texttt{mapper\_prot}\ I_T\ I_U\ f \\ \sqsubseteq\ &\mathbf{!}\,\langle\texttt{left}\rangle.\,\mathbf{!}\,(x_1 : T)\ (v_1 : \mathsf{Val})\,\langle v_1\rangle\{I_T\ x_1\ v_1\}. && n \text{ times } \mu\text{-UNFOLD and} \\ &\mathbf{?}(y_1 : U)\,\langle y_1\rangle\{I_U\ (f\ x_1)\ y_1\}.\ \cdots && \text{weaken } \oplus \text{ into } \mathbf{!}\,\langle\texttt{left}\rangle \\ &\mathbf{!}\,\langle\texttt{left}\rangle.\,\mathbf{!}\,(x_n : T)\ (v_n : \mathsf{Val})\,\langle v_n\rangle\{I_T\ x_n\ v_n\}. \\ &\mathbf{?}(y_n : U)\,\langle y_n\rangle\{I_U\ (f\ x_n)\ y_n\}. \\ &\texttt{mapper\_prot}\ I_T\ I_U\ f \\ \sqsubseteq\ &\mathbf{!}\,\langle\texttt{left}\rangle.\,\mathbf{!}\,(x_1 : T)\ (v_1 : \mathsf{Val})\,\langle v_1\rangle\{I_T\ x_1\ v_1\}.\ \cdots && n \text{ times } \sqsubseteq\text{-SWAP'} \\ &\mathbf{!}\,\langle\texttt{left}\rangle.\,\mathbf{!}\,(x_n : T)\ (v_n : \mathsf{Val})\,\langle v_n\rangle\{I_T\ x_n\ v_n\}. \\ &\mathbf{?}(y_1 : U)\,\langle y_1\rangle\{I_U\ (f\ x_1)\ y_1\}.\ \cdots \\ &\mathbf{?}(y_n : U)\,\langle y_n\rangle\{I_U\ (f\ x_n)\ y_n\}. \\ &\texttt{mapper\_prot}\ I_T\ I_U\ f\end{aligned}$$

Both steps are proven by induction on n. In the first step, we unfold the recursive protocol n times using μ-UNFOLD, and use the derived rule $(prot_1 \oplus prot_2) \sqsubseteq \mathbf{!}\,\langle\texttt{left}\rangle.\,prot_1$ to weaken the choices to the left choice `left`. Recall from § 5.3 that $\oplus$ is defined in terms of the send protocol ($\mathbf{!}$). This allows us to prove the derived rule $(prot_1 \oplus prot_2) \sqsubseteq \mathbf{!}\,\langle\texttt{left}\rangle.\,prot_1$ using $\sqsubseteq$-SEND-OUT and $\sqsubseteq$-SEND-IN. The second step involves swapping all sends ahead of the receives using the rule $\sqsubseteq$-SWAP'.

The weakened protocol that we have obtained follows the behaviour of the client, making its verification straightforward using Actris's rules for symbolic execution. Concretely, we

```
list_rev_service c :=
  let l := recv c in
  lreverse l; send c ()

list_rev_client l :=
  let c := start list_rev_service in
  send c l; recv c
```

Figure 13: A list reversing service (the code for the function `lreverse` has been elided).

prove the following specifications for the service and the client:

$$\{\mathsf{f\_spec}\ I_T\ I_U\ f\ f_\mathsf{v}\ *\ c \rightarrowtail \overline{\mathtt{mapper\_prot}\ I_T\ I_U\ f} \cdot \mathit{prot}\}\quad \mathtt{mapper\_service}\ f_\mathsf{v}\ c\quad \{c \rightarrowtail \mathit{prot}\}$$

$$\{\mathsf{f\_spec}\ I_T\ I_U\ f\ f_\mathsf{v}\ *\ \ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x}\}\quad \mathtt{mapper\_client}\ f_\mathsf{v}\ \ell\quad \{\ell \overset{\text{list}}{\mapsto}_{I_U} \mathtt{map}\ f\ \vec{x}\}$$

6.3. **Protocol compositionality.** An essential feature of separation logic is the ability to compose specifications of different libraries, so that each library can be defined and verified once against its own specification, while being used in the context of slightly differing specifications and proofs of other libraries. To achieve a similar property for our dependent separation protocols we would similarly like to be able to compose compatible protocols.

A key ingredient that enables such compositionality in traditional separation logic is the frame rule (HT-FRAME). In § 6.1 we demonstrated how subprotocols allow for similar framing in our protocols. In this section we give a more detailed example of such framing in our protocols by considering the service `list_rev_service` c in Figure 13, which receives a linked list over channel endpoint c, reverses it, and sends it back over c.

To specify this service, we could use a protocol similar to the sorting service in § 5.1, defined in terms of the representation predicate $\ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x}$ for linked lists:

$$\mathsf{list\_rev\_prot}_{I_T} \triangleq \mathbf{!}\,(\ell : \mathsf{Loc})(\vec{x} : \mathsf{List}\ T)\ \langle \ell \rangle \{\ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x}\}.\ \mathbf{?}\langle () \rangle \{\ell \overset{\text{list}}{\mapsto}_{I_T} \mathsf{reverse}\ \vec{x}\}.\ \mathbf{end}$$

Although it is possible to verify the service against the protocol $\overline{\mathsf{list\_rev\_prot}_{I_T}}$, this approach is not quite satisfactory. Unlike the sorting service, the reversal service does not access the list elements, but only changes the structure of the list. Hence, there is no need to keep track of the ownership of the elements through the predicate I_T. A self-contained and simpler protocol for this service would instead be the following:

$$\mathsf{list\_rev\_prot} \triangleq \mathbf{!}\,(\ell : \mathsf{Loc})(\vec{v} : \mathsf{List\ Val})\ \langle \ell \rangle \{\ell \overset{\text{list}}{\mapsto} \vec{v}\}.\ \mathbf{?}\langle () \rangle \{\ell \overset{\text{list}}{\mapsto} \mathsf{reverse}\ \vec{v}\}.\ \mathbf{end}$$

Here, $\ell \overset{\text{list}}{\mapsto} \vec{v}$ is a version of the list representation predicate that does not keep track of the resources of the elements, but only describes the structure of the list. It is defined as:

$$\ell \overset{\text{list}}{\mapsto} \vec{v} \triangleq \begin{cases} \ell \mapsto \mathtt{inl}\ () & \text{if } \vec{v} = \epsilon \\ \exists \ell_2.\ \ell \mapsto \mathtt{inr}\ (v_1, \ell_2) * \ell_2 \overset{\text{list}}{\mapsto} \vec{v}_2 & \text{if } \vec{v} = [v_1] \cdot \vec{v}_2 \end{cases}$$

However, once we have verified the service against the simple protocol, the proof of a client might prefer to interact with the list reversal service through the general protocol $\mathsf{list\_rev\_prot}_{I_T}$. Doing so can be achieved by proving the subprotocol relation $\mathsf{list\_rev\_prot} \sqsubseteq \mathsf{list\_rev\_prot}_{I_T}$. To prove this subprotocol relation, we first establish the following relation between the two versions of the list representation predicate:

$$\ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x} \ \dashv\vdash\ (\exists \vec{v}.\ \ell \overset{\text{list}}{\mapsto} \vec{v} * \mathop{\Large *}_{(x,v)\in(\vec{x},\vec{v})} I_T\ x\ v) \qquad \text{(LIST-REL)}$$

Here, $\ast_{(x,v)\in(\vec{x},\vec{v})}$ is the pairwise iterated separation conjunction over two lists of equal length, and $\ast\!\ast$ is a bi-directional separation implication. The above result thus states that $\ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x}$ can be split into two parts, ownership of the links of the list $\ell \overset{\text{list}}{\mapsto} \vec{v}$, and a range of interpretation predicates I_T for each element of the list, and *vice versa*. With this result at hand, the proof of the desired subprotocol relation is carried out as follows:

$$
\begin{aligned}
& \mathsf{list\_rev\_prot} \\
= \; & \mathbf{!}\,(\ell : \mathsf{Loc})(\vec{v} : \mathsf{List\ Val})\,\langle \ell \rangle \big\{ \ell \overset{\text{list}}{\mapsto} \vec{v} \big\}.\, \mathbf{?}\langle () \rangle \big\{ \ell \overset{\text{list}}{\mapsto} \mathsf{reverse}\ \vec{v} \big\}.\, \mathbf{end} \\
\sqsubseteq \; & \mathbf{!}\,(\ell : \mathsf{Loc})(\vec{v} : \mathsf{List\ Val})(\vec{x} : \mathsf{List}\ T)\,\langle \ell \rangle \Big\{ \ell \overset{\text{list}}{\mapsto} \vec{v} * \ast_{(x,v)\in(\vec{x},\vec{v})}\, .I_T\ x\ v \Big\}. \\
& \mathbf{?}\langle () \rangle \Big\{ \ell \overset{\text{list}}{\mapsto} (\mathsf{reverse}\ \ \vec{v}) * \ast_{(x,v)\in(\vec{x},\vec{v})}\, .I_T\ x\ v \Big\}.\, \mathbf{end} \\
\sqsubseteq \; & \mathbf{!}\,(\ell : \mathsf{Loc})(\vec{x} : \mathsf{List}\ T)\,\langle \ell \rangle \big\{ \ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x} \big\}.\, \mathbf{?}\langle () \rangle \big\{ \ell \overset{\text{list}}{\mapsto}_{I_T} \mathsf{reverse}\ \vec{x} \big\}.\, \mathbf{end} \\
= \; & \mathsf{list\_rev\_prot}_{I_T}
\end{aligned}
$$

We first frame the range of interpretation predicates owned by the list $\ast_{(x,v)\in(\vec{x},\vec{v})}\, .I_T\ x\ v$, using an approach similar to the frame example in § 6.1, and then use LIST-REL to combine it with $\ell \overset{\text{list}}{\mapsto} \vec{v}$ and $\ell \overset{\text{list}}{\mapsto} \mathsf{reverse}\ \vec{v}$ for the sending and receiving step, to turn them into $\ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x}$ and $\ell \overset{\text{list}}{\mapsto}_{I_T} \mathsf{reverse}\ \vec{x}$, respectively. Note that the logical variable $\vec{v}$ is changed into $\vec{x}$, using the subprotocol rules for logical variable manipulation. With this subprotocol relation at hand, it is possible to prove the following specifications for the service and client:

$$
\begin{array}{ll}
\big\{ c \rightarrowtail \overline{\mathsf{list\_rev\_prot}} \cdot \mathit{prot} \big\} & \big\{ \ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x} \big\} \\
\quad \texttt{list\_rev\_service}\ c & \quad \texttt{list\_rev\_client}\ \ell \\
\{ c \rightarrowtail \mathit{prot} \} & \big\{ \ell \overset{\text{list}}{\mapsto}_{I_T} \mathsf{reverse}\ \vec{x} \big\}
\end{array}
$$

6.4. **Subprotocols and recursion.** We conclude this section by showing how subprotocol relations involving recursive protocols can be proven using LÖB induction. Recall from § 5 that the principle of LÖB induction is as follows:

$$\frac{\triangleright P \Rightarrow P}{P}$$

By letting P be $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$, we can prove $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$ using the induction hypothesis $\triangleright(\mathit{prot}_1 \sqsubseteq \mathit{prot}_2)$. The later modality ($\triangleright$) ensures that the induction hypothesis is not used immediately, but a monotonicity rule for send ($\sqsubseteq$-SEND-MONO) or receive ($\sqsubseteq$-RECV-MONO) is applied first. This is done typically after unfolding the recursion operator using μ-UNFOLD. The monotonicity rules $\sqsubseteq$-SEND-MONO or $\sqsubseteq$-RECV-MONO contain a later modality ($\triangleright$) in their premise, which makes it possible to strip off the later of the induction hypotheses (by rule $\triangleright$-MONO for monotonicity of $\triangleright$).

Our approach for proving subprotocol relations using LÖB induction is similar to the approach of Brandt and Henglein [1998] for proving subtyping relations for recursive types using coinduction. Brandt and Henglein [1998] however have a syntactic restriction on proofs to ensure that the induction hypothesis is not used immediately (*i.e.,* is used in a *contractive* fashion), while we use the later modality ($\triangleright$) of Iris to achieve that.

To demonstrate how our approach works, we prove $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$, where:

$$\mathit{prot}_1 \triangleq \mu(\mathit{rec} : \mathsf{iProto}).\, (\mathsf{list\_rev\_prot} \cdot \mathit{rec}) \oplus \mathbf{end}$$

$$\mathit{prot}_2 \triangleq \mu(\mathit{rec} : \mathsf{iProto}).\, (\mathsf{list\_rev\_prot}_{I_T} \cdot \mathit{rec}) \oplus \mathbf{end}$$

Here, $\mathsf{list\_rev\_prot}$ and $\mathsf{list\_rev\_prot}_{I_T}$ are the protocols from § 6.3, for which we have already proven $\mathsf{list\_rev\_prot} \sqsubseteq \mathsf{list\_rev\_prot}_{I_T}$. The proof of $prot_1 \sqsubseteq prot_2$ is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} prot_1 \sqsubseteq prot_2}\ \sqsubseteq\text{-APPEND}}{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} \mathsf{list\_rev\_prot} \cdot prot_1 \sqsubseteq \mathsf{list\_rev\_prot}_{I_T} \cdot prot_2}\ \triangleright\text{-MONO}}{\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} \triangleright(\mathsf{list\_rev\_prot} \cdot prot_1 \sqsubseteq \mathsf{list\_rev\_prot}_{I_T} \cdot prot_2)}\ \oplus\text{-MONO}}{\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} (\mathsf{list\_rev\_prot} \cdot prot_1) \oplus \mathbf{end} \sqsubseteq (\mathsf{list\_rev\_prot}_{I_T} \cdot prot_2) \oplus \mathbf{end}}\ \mu\text{-UNFOLD}}{\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} prot_1 \sqsubseteq prot_2}\ \text{LÖB}}{prot_1 \sqsubseteq prot_2}$$

The proof starts with the LÖB rule, followed by unfolding the recursive types with μ-UNFOLD. We then proceed with the following derived rule for monotonicity of selection ($\oplus$):

$$\dfrac{\dfrac{}{\triangleright(prot_1 \sqsubseteq prot_2 \land prot_3 \sqsubseteq prot_4)}\ \oplus\text{-MONO}}{(prot_1 \oplus prot_3) \sqsubseteq (prot_2 \oplus prot_4)}$$

Due to the regular conjunction in the premise, the same resources can be used to prove both branches of $\oplus$. This is sound because only one branch of $\oplus$ will be chosen. The rule $\oplus$-MONO follows from $\sqsubseteq$-SEND-MONO as selection ($\oplus$) is defined in terms of send (**!**).

We continue the main proof with monotonicity of the later modality ($\triangleright$-MONO), which lets us strip off the later of the induction hypothesis $\triangleright(prot_1 \sqsubseteq prot_2)$. We then use $\sqsubseteq$-APPEND, along with $\mathsf{list\_rev\_prot} \sqsubseteq \mathsf{list\_rev\_prot}_{I_T}$, which we have proven in § 6.3. The remaining proof obligation $prot_1 \sqsubseteq prot_2$ follows from the induction hypothesis.

While the protocols in the prior examples are similar in structure, our approach scales to protocols for which that is not the case. For example, consider $prot_1 \sqsubseteq prot_2$, where:

$$prot_1 \triangleq \mu(rec : \mathsf{iProto}).\, \mathbf{!}\,(x : \mathbb{Z})\,\langle x\rangle.\, \mathbf{?}\langle x+2\rangle.\, rec$$

$$prot_2 \triangleq \mu(rec : \mathsf{iProto}).\, \mathbf{!}\,(x : \mathbb{Z})\,\langle x\rangle.\, \mathbf{!}\,(y : \mathbb{Z})\,\langle y\rangle.\, \mathbf{?}\langle x+2\rangle.\, \mathbf{?}\langle y+2\rangle.\, rec$$

Intuitively, these protocols are related, as we can unfold the body of $prot_1$ twice, the body of $prot_2$ once, and swap the second receive over the first send. The proof is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} prot_1 \sqsubseteq prot_2}\ \sqsubseteq\text{-RECV-MONO, } \triangleright\text{-INTRO}}{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} \mathbf{?}\langle x+2\rangle.\, \mathbf{?}\langle y+2\rangle.\, prot_1 \sqsubseteq \mathbf{?}\langle x+2\rangle.\, \mathbf{?}\langle y+2\rangle.\, prot_2}\ \sqsubseteq\text{-SEND-MONO', } \triangleright\text{-INTRO}}{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} \mathbf{!}\, y\,\langle y\rangle.\, \mathbf{?}\langle x+2\rangle.\, \mathbf{?}\langle y+2\rangle.\, prot_1 \sqsubseteq \mathbf{!}\, y\,\langle y\rangle.\, \mathbf{?}\langle x+2\rangle.\, \mathbf{?}\langle y+2\rangle.\, prot_2}\ \sqsubseteq\text{-SWAP', } \sqsubseteq\text{-TRANS}}{prot_1 \sqsubseteq prot_2 \mathrel{-\!\!*} \mathbf{?}\langle x+2\rangle.\, \mathbf{!}\, y\,\langle y\rangle.\, \mathbf{?}\langle y+2\rangle.\, prot_1 \sqsubseteq \mathbf{!}\, y\,\langle y\rangle.\, \mathbf{?}\langle x+2\rangle.\, \mathbf{?}\langle y+2\rangle.\, prot_2}\ \triangleright\text{-MONO}}{\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} \triangleright(\mathbf{?}\langle x+2\rangle.\, \mathbf{!}\, y\,\langle y\rangle.\, \mathbf{?}\langle y+2\rangle.\, prot_1 \sqsubseteq \mathbf{!}\, y\,\langle y\rangle.\, \mathbf{?}\langle x+2\rangle.\, \mathbf{?}\langle y+2\rangle.\, prot_2)}\ \sqsubseteq\text{-SEND-MONO'}}{\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} \mathbf{!}\, x\,\langle x\rangle.\, \mathbf{?}\langle x+2\rangle.\, \mathbf{!}\, y\,\langle y\rangle.\, \mathbf{?}\langle y+2\rangle.\, prot_1 \sqsubseteq \mathbf{!}\, x\,\langle x\rangle.\, \mathbf{!}\, y\,\langle y\rangle.\, \mathbf{?}\langle x+2\rangle.\, \mathbf{?}\langle y+2\rangle.\, prot_2}\ \mu\text{-UNFOLD}}{\triangleright(prot_1 \sqsubseteq prot_2) \mathrel{-\!\!*} prot_1 \sqsubseteq prot_2}\ \text{LÖB}}{prot_1 \sqsubseteq prot_2}$$

Grammar:

$$t, u, P, Q, \mathit{prot} ::= \ldots \mid \mathsf{is\_lock}\ \mathit{lk}\ P \mid \ldots$$

Locks:

$$\{R\}\ \texttt{new\_lock}\ ()\ \{\mathit{lk}.\ \mathsf{is\_lock}\ \mathit{lk}\ R\} \qquad \text{(Ht-new-lock)}$$

$$\{\mathsf{is\_lock}\ \mathit{lk}\ R\}\ \texttt{acquire}\ \mathit{lk}\ \{R\} \qquad \text{(Ht-acquire)}$$

$$\{\mathsf{is\_lock}\ \mathit{lk}\ R * R\}\ \texttt{release}\ \mathit{lk}\ \{\mathsf{True}\} \qquad \text{(Ht-release)}$$

$$\mathsf{is\_lock}\ \mathit{lk}\ R \mathrel{-\!\!*} \mathsf{is\_lock}\ \mathit{lk}\ R * \mathsf{is\_lock}\ \mathit{lk}\ R \qquad \text{(Lock-dup)}$$

Figure 14: The grammar and rules of locks in Iris.

After we use ⊑-send-mono' for the first time, we strip off the later of the induction hypothesis $\triangleright(\mathit{prot}_1 \sqsubseteq \mathit{prot}_2)$, using ▷-mono. Subsequently, when we use ⊑-send-mono' and ⊑-recv-mono, there are no more laters to strip. We therefore instead introduce the laters using ▷-intro before applying the appropriate subprotocol monotonicity rule.

7. Manifest sharing via locks

Since dependent separation protocols and the connective $c \rightarrowtail \mathit{prot}$ for ownership of protocols are first-class objects of the Actris logic, they can be used like any other logical connective. This means that protocols can be combined with any other mechanism that Actris inherits from Iris. In particular, they can be combined with Iris's generic invariant and ghost state mechanism, and can be used in combination with Iris's abstractions for reasoning about other concurrency connectives like locks, barriers, lock-free data structures, *etc.*

In this section we demonstrate how dependent separation protocols can be combined with lock-based concurrency. This combination allows us to prove functional correctness of programs that make use of the notion of *manifest sharing* [Balzer and Pfenning 2017; Balzer et al. 2019], where channel endpoints are shared between multiple parties. Instead of having to extend Actris, we make use of the locks and ghost state that Actris inherits from Iris. We present the basic idea with a simple introductory example of sharing a channel endpoint between two parties (§ 7.1). We then consider a more challenging example of a channel-based load-balancing mapper (§ 7.2).

7.1. Locks and ghost state.

As presented in § 2, HeapLang includes a lock library, with the operations `new_lock` (), `acquire` lk, and `release` lk. The operations satisfy the separation logic specifications shown in Figure 14.

The specifications for locks make use of the representation predicate $\mathsf{is\_lock}\ \mathit{lk}\ R$, which expresses that a lock lk guards the resources described by the proposition R. When creating a new lock one has to give up ownership of R, and in turn, obtains the representation predicate $\mathsf{is\_lock}\ \mathit{lk}\ R$ (Ht-new-lock). The representation predicate can then be freely duplicated so it can be shared between multiple threads (Lock-dup). When entering a critical section using `acquire` lk, a thread gets exclusive ownership of R (Ht-acquire), which has to be given up when releasing the lock using `release` lk (Ht-release). The resources R that are protected by the lock are therefore invariant in-between any of the critical sections. The lock can only ever be acquired by one thread at a time, as `acquire` lk will loop until the

$$\begin{aligned}\texttt{prog\_lock} := \texttt{let}\ c := \texttt{start}\ (\lambda c.\ &\texttt{let}\ lk := \texttt{new\_lock}\ ()\ \texttt{in}\\ &\texttt{fork}\ \{\texttt{acquire}\ lk;\ \texttt{send}\ c\ 21;\ \texttt{release}\ lk\};\\ &\texttt{acquire}\ lk;\ \texttt{send}\ c\ 21;\ \texttt{release}\ lk)\ \texttt{in}\\ &\texttt{recv}\ c + \texttt{recv}\ c\end{aligned}$$

Figure 15: A sample program that combines locks and channels to achieve manifest sharing.

$$\begin{aligned}\mathsf{True} &\Rrightarrow \exists\gamma.\ \mathsf{auth}_\gamma\, 0 && \text{(Auth-init)}\\ \mathsf{auth}_\gamma\, n &\Rrightarrow \mathsf{contrib}_\gamma * \mathsf{auth}_\gamma\,(1+n) && \text{(Auth-alloc)}\\ \mathsf{auth}_\gamma\,(1+n) * \mathsf{contrib}_\gamma &\Rrightarrow \mathsf{auth}_\gamma\, n && \text{(Auth-dealloc)}\\ \mathsf{auth}_\gamma\, n * \mathsf{contrib}_\gamma &\twoheadrightarrow n > 0 && \text{(Auth-contrib-pos)}\end{aligned}$$

Figure 16: The authoritative contribution ghost theory.

lock is released. The Ht-acquire rule reflects this, as the exclusive resources R are only obtained once the function terminates, *i.e.,* when the lock is available.

To show how locks can be used, consider the program `prog_lock` in Figure 15. This program uses a lock to share a channel endpoint between two threads, which each send the integer 21 to the main thread. The following dependent protocol specifies the expected interaction from the point of view of the main thread:

$$\texttt{lock\_prot} \triangleq \mu(rec : \mathbb{N} \to \mathsf{iProto}).\ \lambda n.\ \texttt{if}\ (n = 0)\ \texttt{then}\ \mathbf{end}\ \texttt{else}\ \texttt{?}\langle 21\rangle.\ rec\ (n-1)$$

Here, n denotes the number of messages that should be exchanged. In the example program, n is initially 2. Since $c \rightarrowtail \overline{\texttt{lock\_prot}\ n}$ is an exclusive resource, we need a lock to share it between the threads that send 21. For this we will use the following lock invariant:

$$\mathsf{is\_lock}\ lk\ (\exists n.\ \mathsf{auth}_\gamma\, n * c \rightarrowtail \overline{\texttt{lock\_prot}\ n})$$

The natural number n is existentially quantified since it changes whenever a message is exchanged. To keep track of the number of exchanges that each thread is allowed to make we then need to tie the number n to some local resource. We achieve this by using Iris's *ghost theory* mechanism for creating user-defined ghost state [Jung et al. 2015, 2018b]. In particular, we define two logical connectives $\mathsf{auth}_\gamma\, n$ and $\mathsf{contrib}_\gamma$ using Iris.[4]

The $\mathsf{auth}_\gamma\, n$ fragment can be thought of as an authority that keeps track of the number of ongoing contributions n, while each $\mathsf{contrib}_\gamma$ is a token that witnesses that a contribution is still in progress. This intuition is made precise by the rules in Figure 16. The rule Auth-init expresses that an authority $\mathsf{auth}_\gamma\, 0$ can always be created, capturing that 0 contributions are initially in progress. A fresh ghost identifier γ is given, which is conceptually similar to how we obtain fresh locations for newly allocated references on the physical heap. Using the rules Auth-alloc and Auth-dealloc, we can allocate and deallocate $\mathsf{contrib}_\gamma$ tokens as long as the number n of ongoing contributions in $\mathsf{auth}_\gamma\, n$ is updated accordingly. The rule Auth-contrib-pos expresses that ownership of a token $\mathsf{contrib}_\gamma$ implies that the count n of $\mathsf{auth}_\gamma\, n$ must be positive.

[4]Defining a ghost theory in Iris involves picking an appropriate *resource algebra* with which one can define a set of abstract predicates (here $\mathsf{auth}_\gamma\, n$ and $\mathsf{contrib}_\gamma$). The details of resource algebras are beyond the scope of this paper and can be found in Jung et al. [2018b].

Most of the rules in Figure 16 involve Iris's *view shift* connective $\Rrightarrow$ for performing ghost updates. This is made precise by the structural rules Vs-csq and Vs-frame, which establish the connection between $\Rrightarrow$ and Iris's Hoare triples:

$$\textsc{Vs-csq}\ \frac{P \Rrightarrow P' \qquad \{P'\}\, e\, \{v.\, Q'\} \qquad \forall v.\, Q' \Rrightarrow Q}{\{P\}\, e\, \{v.\, Q\}} \qquad \textsc{Vs-frame}\ \frac{P \Rrightarrow Q}{P * R \Rrightarrow Q * R}$$

With the ghost theory in place, we can now prove suitable specifications for the program. The specification of the top-level program is shown on the right, while the left Hoare triple shows the auxiliary specification of both threads that send the integer 21:

$$\begin{array}{l} \{\mathsf{contrib}_\gamma * \mathsf{is\_lock}\ lk\ (\exists n.\, \mathsf{auth}_\gamma\, n * c \rightarrowtail \overline{\texttt{lock\_prot}\ n})\} \\ \quad \texttt{acquire}\ lk;\ \texttt{send}\ c\ 21;\ \texttt{release}\ lk \\ \{\mathsf{True}\} \end{array} \qquad \begin{array}{l} \{\mathsf{True}\} \\ \quad \texttt{prog\_lock} \\ \{v.\, v = 42\} \end{array}$$

We use rule Ht-new to assign protocol `lock_prot` 2 to the channel. To establish the initial lock invariant, we use the rules Auth-init and Auth-alloc to create the authority $\mathsf{auth}_\gamma\, 2$ and two $\mathsf{contrib}_\gamma$ tokens. The $\mathsf{contrib}_\gamma$ tokens play a crucial role in the proofs of the sending threads to establish that the existentially quantified variable n is positive (using Auth-contrib-pos). Knowing $n > 0$, these threads can establish that the protocol $\overline{\texttt{lock\_prot}\ n}$ has not terminated yet (*i.e.,* is not **end**). This is needed to use the rule Ht-send to prove the correctness of sending 21, and thereby advancing the protocol from $\overline{\texttt{lock\_prot}\ n}$ to $\overline{\texttt{lock\_prot}\ (n-1)}$. Subsequently, the sending threads can deallocate the token $\mathsf{contrib}_\gamma$ (using Auth-dealloc) to decrement the n of $\mathsf{auth}_\gamma\, n$ accordingly to restore the lock invariant.

7.2. **A channel-based load-balancing mapper.** This section demonstrates a more interesting use of manifest sharing. We show how Actris can be used to verify functional correctness of a channel-based load-balancing mapper that maps the HeapLang function f_v over a list. Our channel-based mapper consists of one client that distributes the work, and a number of workers that perform the function f_v on individual elements of the list. To enable communication between the client and the workers, we make use of a single channel. One endpoint is used by the client to distribute the work between the workers, while the other endpoint is shared between all workers to request and return work from the client. The implementation of the workers `par_mapper_worker` f_v lk c, which can be found in Figure 17, consists of a loop over three phases:

(1) The worker notifies the client that it wants to perform work (using `select` c `left`), after which it is then notified (using `branch`) whether there is more work or all elements have been mapped. If there is more work, the worker receives an element x that needs to be mapped. Otherwise, the worker will terminate.
(2) The worker maps the function f_v on x.
(3) The worker notifies the client that it wants to send back a result (using `select` c `right`), and subsequently sends back the result y of mapping f_v on x.

The first and last phases are in a critical section guarded by a lock lk since they involve interaction over a shared channel endpoint. As the sharing behaviour is encapsulated by the worker, we omit the code of the client for brevity's sake.[5]

[5]The entire code is present in the accompanied Coq development [Hinrichsen et al. 2021a].

```
par_mapper_worker f_v lk c :=
  acquire lk;
  select c left;
  branch c with
   right ⇒ release lk
  | left  ⇒ let x := recv c in release lk;      (* acquire work *)
            let y := f_v x in                   (* map it *)
            acquire lk;
              select c right; send c y;         (* send it back *)
            release lk;
            par_mapper_worker f_v lk c
  end
```

Figure 17: A worker of the channel-based mapper service.

$$\mathsf{True} \Rrightarrow \exists\gamma.\, \mathsf{auth}_\gamma\, 0\, \emptyset \qquad \text{(AuthM-init)}$$

$$\mathsf{auth}_\gamma\, n\, X \Rrightarrow \mathsf{auth}_\gamma\, (1+n)\, X * \mathsf{contrib}_\gamma\, \emptyset \qquad \text{(AuthM-alloc)}$$

$$\mathsf{auth}_\gamma\, n\, X * \mathsf{contrib}_\gamma\, \emptyset \Rrightarrow \mathsf{auth}_\gamma\, (n-1)\, X \qquad \text{(AuthM-dealloc)}$$

$$\mathsf{auth}_\gamma\, n\, X * \mathsf{contrib}_\gamma\, Y \Rrightarrow \mathsf{auth}_\gamma\, n\, (X \uplus Z) * \mathsf{contrib}_\gamma\, (Y \uplus Z) \qquad \text{(AuthM-add)}$$

$$Z \subseteq Y * \mathsf{auth}_\gamma\, n\, X * \mathsf{contrib}_\gamma\, Y \Rrightarrow \mathsf{auth}_\gamma\, n\, (X \setminus Z) * \mathsf{contrib}_\gamma\, (Y \setminus Z) \qquad \text{(AuthM-remove)}$$

$$\mathsf{auth}_\gamma\, n\, X * \mathsf{contrib}_\gamma\, Y \twoheadrightarrow n > 0 * Y \subseteq X \qquad \text{(AuthM-contrib-agree)}$$

$$\mathsf{auth}_\gamma\, 1\, X * \mathsf{contrib}_\gamma\, Y \twoheadrightarrow Y = X \qquad \text{(AuthM-contrib-agree1)}$$

Figure 18: The authoritative contribution ghost theory extended with multisets.

A protocol that describes the interaction from the client's point of view is as follows:

$$\begin{aligned}
&\mathsf{par_mapper_prot}\ (I_T : T \to \mathsf{Val} \to \mathsf{iProp})\ (I_U : U \to \mathsf{Val} \to \mathsf{iProp})\ (f : T \to \mathsf{List}\ U) \triangleq \\
&\quad \mu(rec : \mathbb{N} \to \mathsf{MultiSet}\ T \to \mathsf{iProto}).\ \lambda n\ X. \\
&\qquad \texttt{if}\ n = 0\ \texttt{then}\ \mathbf{end}\ \texttt{else} \\
&\qquad (\mathbf{!}\,(x : T)\ (v : \mathsf{Val})\ \langle v \rangle \{I_T\ x\ v\}.\ rec\ n\ (X \uplus \{x\})) \oplus rec\ (n-1)\ X \\
&\qquad\quad {\scriptstyle \{(n=1) \twoheadrightarrow (X=\emptyset)\}}\&{\scriptstyle \{\mathsf{True}\}} \\
&\qquad \mathbf{?}(x : T)\ (\ell : \mathsf{Loc})\ \langle \ell \rangle \{x \in X * \ell \overset{\text{list}}{\mapsto}_{I_U} (f\ x)\}.\ rec\ n\ (X \setminus \{x\})
\end{aligned}$$

Similarly to `mapper_prot` from § 6.2, the protocol is parameterised by representation predicates I_T and I_U, and a function $f : T \to \mathsf{List}\ U$ in the Iris/Actris logic that will be related to f_v through a f_spec specification. Similar to the protocol `lock_prot` from § 7.1, the protocol par_mapper_prot is indexed by the number of remaining workers n. On top of that, it carries a multiset X describing the values currently being processed by all the workers. The multiset X is used to make sure that the returned results are in fact the result of mapping the function f. The condition $(n = 1) \twoheadrightarrow (X = \emptyset)$ on the branching operator (&) expresses that the last worker may only request more work if there are no ongoing jobs.

To accommodate sharing of the channel endpoint between all workers using a lock invariant, we extend the authoritative contribution ghost theory from § 7.1. We do this by adding multisets X and Y to the connectives $\mathsf{auth}_\gamma\, n\, X$ and $\mathsf{contrib}_\gamma\, Y$. These multisets

keep track of the values held by the workers. The rules for the ghost theory extended with multisets are shown in Figure 18. The rules AuthM-init, AuthM-alloc and AuthM-dealloc are straightforward generalisations of the ones we have seen before. The new rules AuthM-add and AuthM-remove determine that the multiset Y of $\mathsf{contrib}_\gamma\ Y$ can be updated as long as it is done in accordance with the multiset X of $\mathsf{auth}_\gamma\ n\ X$. Finally, the AuthM-contrib-agree rule expresses that the multiset Y of $\mathsf{contrib}_\gamma\ Y$ must be a subset of the multiset X of $\mathsf{auth}_\gamma\ n\ X$, while the stricter rule AuthM-contrib-agree1 asserts equality between X and Y when only one contribution remains.

We then prove the following specifications of `par_mapper_worker` and a possible top-level client `par_mapper_client` that uses n workers to map f_v over the linked list ℓ:

$$\left\{\begin{array}{l}\mathsf{f\_spec}\ I_T\ I_U\ f\ f_\mathsf{v} * \mathsf{contrib}_\gamma\ \emptyset\ * \\ \mathsf{is\_lock}\ lk\ \left(\begin{array}{l}\exists n\ X.\ \mathsf{auth}_\gamma\ n\ X\ * \\ c \rightarrowtail \overline{\mathsf{par\_mapper\_prot}\ I_T\ I_U\ f\ n\ X}\end{array}\right)\end{array}\right\}$$
$$\texttt{par\_mapper\_worker}\ f_\mathsf{v}\ lk\ c$$
$$\{\mathsf{True}\}$$

$$\left\{\begin{array}{l}\mathsf{f\_spec}\ I_T\ I_U\ f\ f_\mathsf{v}\ * \\ 0 < n * \ell \overset{\mathsf{list}}{\mapsto}_{I_T} \vec{x}\end{array}\right\}$$
$$\texttt{par\_mapper\_client}\ n\ f_\mathsf{v}\ \ell$$
$$\left\{\exists \vec{y}.\ \vec{y} \equiv_\mathsf{p} \texttt{flatMap}\ f\ \vec{x} * \ell \overset{\mathsf{list}}{\mapsto}_{I_U} \vec{y}\right\}$$

The lock invariant and specification of `par_mapper_worker` are similar to those used in the simple example in § 7.1. The specification of `par_mapper_client` $n\ f_\mathsf{v}\ \ell$ simply states that the resulting linked contains a permutation of performing the map at the level of the logic. To specify that, we make use of $\texttt{flatMap} : (T \to \mathsf{List}\ U) \to (\mathsf{List}\ T \to \mathsf{List}\ U)$, whose definition is standard.

The proof of the client involves allocating the channel with the protocol par_mapper_prot, with the initial number of workers n. Subsequently, we use the rules AuthM-init and AuthM-alloc to create the authority $\mathsf{auth}_\gamma\ n\ \emptyset$ and n tokens $\mathsf{contrib}_\gamma\ \emptyset$, which allow us to establish the lock invariant and to distribute the tokens among the mappers. The proof of the mapper proceeds as usual. After acquiring the lock, the mapper obtains ownership of the lock invariant. Since the worker owns the token $\mathsf{contrib}_\gamma\ \emptyset$, it knows that the number of remaining workers n is positive, which allows it to conclude that the protocol has not terminated (*i.e.,* is not **end**). After using the rules for channels, the rules AuthM-add and AuthM-remove are used to update the authority, which is needed to reestablish the lock invariant so the lock can be released.

8. Case study: map-reduce

As a means of demonstrating the use of Actris for verifying more realistic programs, we present a proof of functional correctness of a simple channel-based load-balancing implementation of the map-reduce model by Dean and Ghemawat [2004].

Since Actris is not concerned with distributed systems over networks, we consider a version of map-reduce that delegates the work over forked-off threads on a single machine. This means that we do not consider mechanics like handling the failure, restarting, and rescheduling of nodes that a version that operates on a network has to consider.

In order to implement and verify our map-reduce version we make use of the implementation and verification of the fine-grained channel-based merge sort algorithm (§ 5.6) and the channel-based load-balancing mapper (§ 7.2). As such, our map-reduce implementation is mostly a suitable client that glues together communication with these services. The purpose of this section is to give a high-level description of the implementation. The actual code and proofs can be found in the accompanied Coq development [Hinrichsen et al. 2021a].

8.1. **A functional specification of map-reduce.** The purpose of the map-reduce model is to transform an input set of type $\mathsf{List}\ T$ into an output set of type $\mathsf{List}\ V$ using two functions f (often called "map") and g (often called "reduce"):

$$f : T \to \mathsf{List}\ (K * U) \qquad g : (K * \mathsf{List}\ U) \to \mathsf{List}\ V$$

An implementation of map-reduce performs the transformation in three steps:

(1) First, the function f is applied to each element of the input set. This results in lists of key/value pairs which are then flattened using a `flatMap` operation (an operation that takes a list of lists and appends all nested lists):

$$\texttt{flatMap}\ f \quad : \quad \mathsf{List}\ T \to \mathsf{List}\ (K * U)$$

(2) Second, the resulting lists of key/value pairs are grouped together by their key (this step is often called "shuffling"):

$$\texttt{group} \quad : \quad \mathsf{List}\ (K * U) \to \mathsf{List}\ (K * \mathsf{List}\ U)$$

(3) Finally, the grouped key/value pairs are passed on to the g function, after which the results are flattened to aggregate the results. This is done using a `flatMap` operation:

$$\texttt{flatMap}\ g \quad : \quad \mathsf{List}\ (K * \mathsf{List}\ U) \to \mathsf{List}\ V$$

The complete functionality of map-reduce is equivalent to applying the following `map_reduce` function on the entire data set:

$$\texttt{map\_reduce} \quad : \quad \mathsf{List}\ T \to \mathsf{List}\ V \quad \triangleq \quad (\texttt{flatMap}\ g) \circ \texttt{group} \circ (\texttt{flatMap}\ f)$$

A standard instance of map-reduce is counting word occurrences, where we let $T \triangleq K \triangleq \mathsf{String}$ and $U \triangleq \mathbb{N}$ and $V \triangleq \mathsf{String} * \mathbb{N}$ with:

$$f : \mathsf{String} \to \mathsf{List}\ (\mathsf{String} * \mathbb{N}) \triangleq \lambda x.\ [(x, 1)]$$

$$g : (\mathsf{String} * \mathsf{List}\ \mathbb{N}) \to \mathsf{List}\ (\mathsf{String} * \mathbb{N}) \triangleq \lambda (k, \vec{n}).\ [(k, \Sigma_{i<|\vec{n}|}.\ \vec{n}_i)]$$

8.2. **Implementation of map-reduce.** The general distributed model of map-reduce is achieved by delegating the phases of mapping, shuffling, and reducing, over a number of worker nodes (*e.g.,* nodes of a cluster or individual CPUs). To perform the computation in a delegated way, there is some work involved in coordinating the jobs over these worker nodes, which is usually done as follows:

(1) Split the input data into chunks and delegate these chunks to worker nodes, that each apply the "map" function f to their given data in parallel. We call these nodes the "mappers".
(2) Collect the complete set of mapped results and "shuffle" them, *i.e.,* group them by key. The grouping is commonly implemented using a parallel sorting algorithm.
(3) Split the shuffled data into chunks and delegate these chunks to worker nodes that each apply the "reduce" function g to their given data in parallel. We call these nodes the "reducers".
(4) Collect and aggregate the complete set of result of the reducers.

Our variant of the map-reduce model is defined as a function $\texttt{map\_reduce}_{\mathsf{V}}\ n\ m\ f_{\mathsf{V}}\ g_{\mathsf{V}}\ \ell$ in HeapLang, which coordinates the work for performing map-reduce on a linked list ℓ between n mappers applying the HeapLang "map" function f_{V}, and m reducers applying the HeapLang "reduce" function g_{V}. To make the implementation more interesting, we prevent

storing intermediate values locally by forwarding/returning them immediately as they are available/requested. The global structure is as follows:

(1) Start n instances of the load-balancing `par_mapper_worker` from § 7, parameterised with the f_{v} function, acting as the mappers. Additionally start an instance of $\texttt{sort\_service}_{fg}$ from § 5.6, parameterised by a concrete comparison function on the keys, corresponding to $\lambda(k_1, \_)\ (k_2, \_).\, k_1 < k_2$. Note that the type of keys are restricted to be integers for brevity's sake.
(2) Perform a loop that handles communication with the mappers. If a mapper requests work, pop a value from the input list. If a mapper returns work, forward it to the sorting service. This process is repeated until all inputs have been mapped and forwarded.
(3) Start m instances of the `par_mapper_worker`, parameterised by g_{v}, acting as the reducers.
(4) Perform a loop that handles communication with the mappers. If a mapper requests work, group elements returned by the sort service. If a mapper returns work, aggregate the returned value in a the linked list. Grouped elements are created by requesting and aggregating elements from the sorter until the key changes.

The aggregated linked list then contains the fully mapped input set upon completion.

8.3. **Functional correctness of map-reduce.** The specification of the map-reduce program that we prove is as follows:

$$\begin{array}{l}\left\{0 < n \ *\ 0 < m \ *\ \mathsf{f\_spec}\ I_T\ I_{\mathbb{Z}*U}\ f\ f_{\mathsf{v}} \ *\ \mathsf{f\_spec}\ I_{\mathbb{Z}*\mathsf{List}\ U}\ I_V\ g\ g_{\mathsf{v}} \ *\ \ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x}\right\} \\ \quad \texttt{map\_reduce}_{\mathsf{v}}\ n\ m\ f_{\mathsf{v}}\ g_{\mathsf{v}}\ \ell \\ \left\{\exists \vec{z}.\, \vec{z} \equiv_p\ \texttt{map\_reduce}\ f\ g\ \vec{x} * \ell \overset{\text{list}}{\mapsto}_{I_V} \vec{z}\right\}\end{array}$$

The $\mathsf{f\_spec}$ predicates (as introduced in § 6.2) establish a connection between the functions f and g in Iris/Actris and the functions f_{v} and g_{v} in HeapLang. These make use of the various interpretation predicates I_T, $I_{\mathbb{Z}*U}$, $I_{\mathbb{Z}*\mathsf{List}\ U}$, and I_V for the types in question. Lastly, the $\ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x}$ predicate determines that the input is a linked list of the initial type T. The postcondition asserts that the result $\vec{z}$ is a permutation of the original linked list $\vec{x}$ applied to the functional specification `map_reduce` of map-reduce from § 8.1.

9. The model of Actris

We construct a model of Actris as a shallow embedding in the Iris framework [Krebbers et al. 2017a; Jung et al. 2015, 2016, 2018b]. This means that the type iProto of dependent separation protocols, the subprotocol relation $prot_1 \sqsubseteq prot_2$, and the connective $c \rightarrowtail prot$ for the channel ownership, are definitions in Iris, and the Actris proof rules are lemmas about these definitions in Iris.

In this section we describe the relevant aspects of the model of Actris. We model the type iProto of dependent separation protocols as the solution of a recursive domain equation, and describe how the operations for dual and composition are defined (§ 9.1). We then define the subprotocol relation $prot_1 \sqsubseteq prot_2$ and prove its proof rules as lemmas (§ 9.2). To connect protocols to the endpoint channel buffers in the semantics we define the *protocol consistency relation*, which ensures that a pair of protocols is consistent with the messages in their associated buffers (§ 9.3). On top of the protocol consistency relation, we define the *Actris ghost theory* for dependent separation protocols (§ 9.4), which forms the key ingredient for defining the connective $c \rightarrowtail prot$ for channel ownership (§ 9.5) that links protocols to the

semantics of channels (§ 2.4). We then show how adequacy follows from the embedding in Iris (§ 9.6). Finally, we show how to solve the recursive domain equation for the type iProto of dependent separation protocols (§ 9.7).

9.1. **The model of dependent separation protocols.** To construct a model of dependent separation protocols, we first need to determine what they mean semantically. The challenging part involves the constructors $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$, whose (higher-order and impredicative) logical variables $\vec{x}{:}\vec{\tau}$ bind into the communicated value v, the transferred resources P, and the tail protocol $prot$. We model these constructors as predicates over the communicated value and the tail protocol. To describe the transferred resources P, we model these protocols as Iris predicates (functions to iProp). This gives rise to the following recursive domain equation:

$$\begin{aligned}\mathsf{action} &::= \mathbf{send} \mid \mathbf{recv}\\ \mathsf{iProto} &\cong 1 + (\mathsf{action} \times (\mathsf{Val} \to \blacktriangleright\mathsf{iProto} \to \mathsf{iProp}))\end{aligned}$$

The left part of the sum type (the unit type 1) indicates that the protocol has terminated, while the right part describes a message that is exchanged, expressed as an Iris predicate. Since the recursive occurrence of iProto in the predicate appears in negative position, we guard it using Iris's *type-level later* ($\blacktriangleright$) operator (whose only constructor is $\mathsf{next} : T \to \blacktriangleright\, T$). The exact way the solution is constructed is detailed in § 9.7. For now, we assume a solution exists, and define the dependent separation protocols constructors as:

$$\begin{aligned}\mathbf{end} &\triangleq \mathtt{inj}_1\,()\\ !\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot &\triangleq \mathtt{inj}_2\,(\mathbf{send}, \lambda w\, prot'.\, \exists\vec{x}{:}\vec{\tau}.\,(v = w) * P * (prot' = \mathsf{next}\ prot))\\ ?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot &\triangleq \mathtt{inj}_2\,(\mathbf{recv}, \lambda w\, prot'.\, \exists\vec{x}{:}\vec{\tau}.\,(v = w) * P * (prot' = \mathsf{next}\ prot))\end{aligned}$$

The definitions of $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ make use of the (higher-order and impredicative) existential quantifiers of Iris to constrain the actual message w and tail $prot'$ so that they agree with the message v and tail $prot$ prescribed by the protocol.

Recursive protocols. Iris's guarded recursion operator $\mu x.\,t$ requires the recursion variable x to appear under a *contractive* term construct in t. Hence, to use Iris's recursion operator to construct recursive protocols, it is essential that the protocols $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ are contractive in the tail $prot$. To show why this is the case, let us first define what it means for a function $f : T \to U$ to be contractive:

$$\forall x, y.\ \triangleright(x = y) \Rightarrow f\ x = f\ y$$

Examples of contractive functions are the later modality $\triangleright : \mathsf{iProp} \to \mathsf{iProp}$ and the constructor $\mathsf{next} : T \to \blacktriangleright\, T$. The protocols $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ are defined so that $prot$ appears below a next, and hence we can prove that they are contractive in $prot$.

Operations. With these definitions at hand, the dual $\overline{(\_)}$ and append $(\_ \cdot \_)$ operations are defined using Iris's guarded recursion operator ($\mu x.\,t$):

$$\overline{(\_)} \triangleq \mu rec.\,\lambda prot.\,\begin{cases}\mathtt{inj}_1\,() & \text{if } prot = \mathtt{inj}_1\,()\\ \mathtt{inj}_2\,(\overline{a}, \lambda w\ prot'.\,\exists prot''. & \text{if } prot = \mathtt{inj}_2\,(a, \Phi)\\ \qquad \Phi\ w\ (\mathsf{next}\ prot'') * & \\ \qquad prot' = \mathsf{next}\ (rec\ prot'')) & \end{cases}$$

$$(\_ \cdot \mathit{prot}_2) \triangleq \mu \mathit{rec}.\, \lambda \mathit{prot}_1.\, \begin{cases} \mathit{prot}_2 & \text{if } \mathit{prot}_1 = \mathtt{inj}_1\,() \\ \mathtt{inj}_2\,(a, \lambda w\ \mathit{prot}'.\, \exists \mathit{prot}''. & \text{if } \mathit{prot}_1 = \mathtt{inj}_2\,(a, \Phi) \\ \qquad \Phi\ w\ (\mathsf{next}\ \mathit{prot}'') * & \\ \qquad \mathit{prot}' = \mathsf{next}\ (\mathit{rec}\ \mathit{prot}'')) & \end{cases}$$

In the above definitions, we let $\overline{\mathbf{send}} \triangleq \mathbf{recv}$ and $\overline{\mathbf{recv}} = \mathbf{send}$.

The base cases of both definitions are as expected. In the recursive cases, we construct a new predicate, given the original predicate Φ. In these new predicates, we quantify over an original tail protocol prot'' such that $\Phi\ w\ (\mathsf{next}\ \mathit{prot}'')$ holds, and unify the new tail protocol prot' with the result of the recursive call $\mathit{rec}\ \mathit{prot}''$.

The equational rules for dual $\overline{(\_)}$ and append $(\_ \cdot \_)$ from Figure 5 are proven as lemmas in Iris using Löb induction. This is possible as the recursive call $\mathit{rec}\ \mathit{prot}''$ appears below a next constructor—since the next constructor is contractive, we can strip-off the later from the induction hypothesis when proving the equality for the tail.

Difference from the conference version. In the conference version of this paper [Hinrichsen et al. 2020], we described two versions of the recursive domain equation for dependent separation protocols: an "ideal" version (as used in this paper), where iProto appears in negative position, and an "alternative" version, where iProto appears in positive position. At that time, we were unable to construct a solution of the "ideal" version, so we used the "alternative" version. In § 9.7 we show how we are now able to solve the "ideal" version.

In the conference version of this paper, the proposition P appeared under a later modality in the definitions of the protocols $!\,\vec{x}\!:\!\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ and $?\vec{x}\!:\!\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, making these protocols contractive in P. This choice was motivated by the ability to construct recursive protocols like $\mu \mathit{rec}.\, !\,(c : \mathsf{Val})\,\langle c\rangle\{c \rightarrowtail \mathit{prot}\}.\,\mathit{prot}'$, where the payload refers to the recursion variable rec. In the current version (without the later modality) we can still construct such protocols, because $c \rightarrowtail \mathit{prot}$ is contractive in prot. We removed the later modality because it is incompatible with the rules $\sqsubseteq$-SEND-OUT and $\sqsubseteq$-RECV-OUT for subprotocols.

9.2. **The model of the subprotocol relation.** We now model the subprotocol relation $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$ from §6. For legibility, we present it in the style of an inference system through its constructors, whereas it is formally defined using Iris's guarded recursion operator ($\mu x.\, t$):

$$\mathtt{inj}_1\,() \sqsubseteq \mathtt{inj}_1\,()$$

$$\frac{\forall v, \mathit{prot}_2.\, \Phi_2\ v\ (\mathsf{next}\ \mathit{prot}_2) \mathrel{-\!\!*} \exists \mathit{prot}_1.\, \Phi_1\ v\ (\mathsf{next}\ \mathit{prot}_1) * \triangleright(\mathit{prot}_1 \sqsubseteq \mathit{prot}_2)}{\mathtt{inj}_2\,(\mathbf{send}, \Phi_1) \sqsubseteq \mathtt{inj}_2\,(\mathbf{send}, \Phi_2)} \qquad \frac{\forall v, \mathit{prot}_1.\, \Phi_1\ v\ (\mathsf{next}\ \mathit{prot}_1) \mathrel{-\!\!*} \exists \mathit{prot}_2.\, \Phi_2\ v\ (\mathsf{next}\ \mathit{prot}_2) * \triangleright(\mathit{prot}_1 \sqsubseteq \mathit{prot}_2)}{\mathtt{inj}_2\,(\mathbf{recv}, \Phi_1) \sqsubseteq \mathtt{inj}_2\,(\mathbf{recv}, \Phi_2)}$$

$$\frac{\forall v_1, v_2, \mathit{prot}_1, \mathit{prot}_2.\, (\Phi_1\ v_1\ (\mathsf{next}\ \mathit{prot}_1) * \Phi_2\ v_2\ (\mathsf{next}\ \mathit{prot}_2)) \mathrel{-\!\!*} \exists \mathit{prot}.\, \triangleright(\mathit{prot}_1 \sqsubseteq\, !\,\langle v_2\rangle.\,\mathit{prot}) * \triangleright(?\langle v_1\rangle.\,\mathit{prot} \sqsubseteq \mathit{prot}_2)}{\mathtt{inj}_2\,(\mathbf{recv}, \Phi_1) \sqsubseteq \mathtt{inj}_2\,(\mathbf{send}, \Phi_2)}$$

To be a well-formed guarded recursion definition, every recursive occurrence of $\sqsubseteq$ is guarded by the later modality ($\triangleright$). Aside from later being required for well-formedness, these laters make it possible to reason about the subprotocol relation using Löb induction; both to prove the subprotocol rules from Figure 11 as lemmas, and for Actris users to reason about

recursive protocols as shown in § 6.4. The relation is defined in a syntax directed fashion (*i.e.,* there are no overlapping rules), and therefore all constructors need to be defined so that they are closed under monotonicity and transitivity.

The first constructor states that terminating protocols ($\mathbf{end} \triangleq \mathtt{inj}_1\,()$) are related. The other constructors concern the protocols $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$ and $\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,prot$, which are modelled as $\mathtt{inj}_2\,(\mathbf{send}, \Phi)$ and $\mathtt{inj}_2\,(\mathbf{recv}, \Phi)$, where $\Phi : \mathsf{Val} \to \blacktriangleright\mathsf{iProto} \to \mathsf{iProp}$ is a predicate over the communicated value and tail protocol. While the actual constructors are somewhat intimidating because they are defined in terms of these predicates in the model, they essentially correspond to the following high-level versions:

$$\frac{\forall \vec{y}{:}\vec{\sigma}.\, P_2 \mathrel{-\!\!*} \exists \vec{x}{:}\vec{\tau}.\, (v_1 = v_2) * P_1 * \triangleright(prot_1 \sqsubseteq prot_2)}{\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v_1\rangle\{P_1\}.\,prot_1 \sqsubseteq \mathbf{!}\,\vec{y}{:}\vec{\sigma}\,\langle v_2\rangle\{P_2\}.\,prot_2}$$

$$\frac{\forall \vec{x}{:}\vec{\tau}.\, P_1 \mathrel{-\!\!*} \exists \vec{y}{:}\vec{\sigma}.\, (v_1 = v_2) * P_2 * \triangleright(prot_1 \sqsubseteq prot_2)}{\mathbf{?}\,\vec{x}{:}\vec{\tau}\,\langle v_1\rangle\{P_1\}.\,prot_1 \sqsubseteq \mathbf{?}\,\vec{y}{:}\vec{\sigma}\,\langle v_2\rangle\{P_2\}.\,prot_2}$$

$$\frac{\forall \vec{x}{:}\vec{\tau}, \vec{y}{:}\vec{\sigma}.\, (P_1 * P_2) \mathrel{-\!\!*} \exists prot.\, \triangleright(prot_1 \sqsubseteq \mathbf{!}\,\langle v_2\rangle.\,prot) * \triangleright(\mathbf{?}\langle v_1\rangle.\,prot \sqsubseteq prot_2)}{\mathbf{?}\,\vec{x}{:}\vec{\tau}\,\langle v_1\rangle\{P_1\}.\,prot_1 \sqsubseteq \mathbf{!}\,\vec{y}{:}\vec{\sigma}\,\langle v_2\rangle\{P_2\}.\,prot_2}$$

To obtain syntax directed rules, the first rule combines $\sqsubseteq$-SEND-OUT, $\sqsubseteq$-SEND-IN, and $\sqsubseteq$-SEND-MONO, and dually, the second rule combines $\sqsubseteq$-RECV-OUT, $\sqsubseteq$-RECV-IN, and $\sqsubseteq$-RECV-MONO. The third rule combines $\sqsubseteq$-RECV-OUT, $\sqsubseteq$-SEND-OUT and $\sqsubseteq$-SWAP and bakes in transitivity, instead of asserting that $prot_1$ and $prot_2$ are equal to $\mathbf{!}\,\langle v_2\rangle.\,prot$ and $\mathbf{?}\langle v_1\rangle.\,prot$, respectively.

The rules from the beginning of this section are defined by generalising the high-level rules to arbitrary predicates. For example, rule $\mathtt{inj}_2\,(\mathbf{send}, \Phi_1) \sqsubseteq \mathtt{inj}_2\,(\mathbf{send}, \Phi_2)$ requires that for any value v and tail protocol $prot_2$ that are allowed by the predicate Φ_2, there is a stronger tail protocol $prot_1$ (*i.e.,* where $prot_1 \sqsubseteq prot_2$), so that the same value v and stronger tail protocol $prot_1$ are allowed by the predicate Φ_1.

The rules in Figure 11 on page 28 are proven as lemmas. Those for logical variable and resource manipulation ($\sqsubseteq$-SEND-OUT, $\sqsubseteq$-SEND-IN, $\sqsubseteq$-RECV-OUT and $\sqsubseteq$-RECV-IN), monotonicity ($\sqsubseteq$-SEND-MONO and $\sqsubseteq$-RECV-MONO), and swapping ($\sqsubseteq$-SWAP) follow almost immediately from the definition, whereas those for reflexivity ($\sqsubseteq$-REFL), transitivity ($\sqsubseteq$-TRANS), and the dual and append operator ($\sqsubseteq$-DUAL and $\sqsubseteq$-APPEND) are proven using LÖB induction.

9.3. **Protocol consistency.** To connect dependent separation protocols to the semantics of channels in § 9.5, we define the *protocol consistency relation* $\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2$, which expresses that protocols $prot_1$ and $prot_2$ are *consistent* w.r.t. channel buffers containing values $\vec{v}_1$ and $\vec{v}_2$. The consistency relation is defined as:

$$\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2 \triangleq \exists prot.$$
$$(\mathbf{?}\langle \vec{v}_{2.1}\rangle.\ldots.\mathbf{?}\langle \vec{v}_{2.|\vec{v}_2|}\rangle.\,prot \sqsubseteq prot_1) * (\mathbf{?}\langle \vec{v}_{1.1}\rangle.\ldots.\mathbf{?}\langle \vec{v}_{1.|\vec{v}_1|}\rangle.\,\overline{prot} \sqsubseteq prot_2)$$

Intuitively, $\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2$ ensures that for all messages $\vec{v}_1 = \vec{v}_{1.1} \ldots \vec{v}_{1.|\vec{v}_1|}$ in transit from the endpoint described by $prot_1$ to the endpoint described by $prot_2$, the protocol $prot_2$ is expecting to receive these message in order (and *vice versa* for $\vec{v}_2$), after which the remaining protocols $prot$ and $\overline{prot}$ are dual. To account for weakening we close the consistency relation under subprotocols (by using $\sqsubseteq$ instead of equality). Closure under the subprotocol relation additionally implicitly captures ownership of the quantifiers and

$$\mathsf{True} \Rrightarrow \exists\gamma.\, (\gamma \mapsto_\bullet prot) * (\gamma \mapsto_\circ prot) \qquad \text{(HO-GHOST-ALLOC)}$$

$$(\gamma \mapsto_\bullet prot) * (\gamma \mapsto_\circ prot') \Rrightarrow (\gamma \mapsto_\bullet prot'') * (\gamma \mapsto_\circ prot'') \qquad \text{(HO-GHOST-UPDATE)}$$

$$(\gamma \mapsto_\bullet prot) * (\gamma \mapsto_\circ prot') \mathrel{-\!\!*} \triangleright(prot = prot') \qquad \text{(HO-GHOST-AGREE)}$$

Figure 19: Ghost theory for higher-order ghost variables in Iris.

resources associated with the messages $\vec{v}_1$ and $\vec{v}_2$. That is, since the subprotocol relations relate the protocol arguments $prot_1$ and $prot_2$ with protocols that specify no quantifiers or resources. More precisely, by the definition of the subprotocol relation (shown in § 9.2), a relation such as $\mathbf{?}\langle v\rangle.\, prot_1 \sqsubseteq \mathbf{?}(\vec{x}{:}\vec{\tau})\,\langle v\rangle\{P\}.\, prot_2$ is equivalent to a separation implication of the form $\mathsf{True} \mathrel{-\!\!*} \exists\vec{x}{:}\vec{\tau}.\, P * \triangleright\, prot_1 \sqsubseteq prot_2$, where the obligation True is trivial, meaning that it implicitly asserts ownership of P.

Finally, closure under the subprotocol relation gives that $\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2$ and $prot_1 \sqsubseteq prot_1'$ implies $\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1'\ prot_2$, and ensures that the consistency relation enjoys the following rules corresponding to creating a channel, sending a message, and receiving a message:

$$\mathsf{prot\_consistent}\ \epsilon\ \epsilon\ prot\ \overline{prot}$$

$$(\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ (\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot_1)\ prot_2 * P[\vec{t}/\vec{x}]) \mathrel{-\!\!*} \triangleright^{|\vec{v}_2|}(\mathsf{prot\_consistent}\ (\vec{v}_1 \cdot [v[\vec{t}/\vec{x}]])\ \vec{v}_2\ prot_1\ prot_2)$$

$$\mathsf{prot\_consistent}\ \vec{v}_1\ ([w] \cdot \vec{v}_2)\ (\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot_1)\ prot_2 \mathrel{-\!\!*} \exists\vec{y}.\, (w = v[\vec{y}/\vec{x}]) * P[\vec{y}/\vec{x}] * \triangleright(\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2)$$

The first rule states that dual protocols are consistent w.r.t. a pair of empty buffers. The second rule states that a protocol $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot_1$ can be advanced to $prot_1$ by giving up ownership of $P[\vec{t}/\vec{x}]$ and enqueueing the value $v[\vec{t}/\vec{x}]$ in the buffer $\vec{v}_1$. Dually, the third rule states that given a protocol $\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\, prot_1$ and a buffer that contains value w as its head, we learn that w is equal to $v[\vec{y}/\vec{x}]$, and that we can obtain ownership of $P[\vec{y}/\vec{x}]$ by advancing the protocol to $prot_1$ and dequeuing the value w from the buffer. Since the relation is symmetric, *i.e.,* if $\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2$ then $\mathsf{prot\_consistent}\ \vec{v}_2\ \vec{v}_1\ prot_2\ prot_1$, we obtain similar rules for the protocol $prot_2$ on the right-hand side.

The last two rules are proven by case analysis on the subprotocol relation ($\sqsubseteq$) in the assumption. Since the subprotocol relation ($\sqsubseteq$) is defined using guarded recursion, we obtain a later modality ($\triangleright$) for each case analysis. To prove the first of the rules, we need to perform a number of case analyses equal to the size of the buffer $\vec{v}_2$, whereas for the second rule we need to perform just a single case analysis. These later modalities are eliminated through the `skipN` operation in the `send` operation, see § 9.5 for further discussion.

9.4. **The Actris ghost theory.** To provide a general interface for making Actris's reasoning principles independent of HeapLang, we employ a standard ghost theory approach in Iris to compartmentalise channel ownership. In § 9.5 we define the connective $c \rightarrowtail prot$ for channel endpoint ownership that links the ghost theory to the buffers of our implementation of channels in HeapLang.

The Actris ghost theory is similar in its interface to the ghost theory for contributions that we used in § 7. We define three new logical connectives—an authority $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$,

$$\mathsf{True} \Rrightarrow \exists \chi.\, \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathsf{prot\_own_r}\ \chi\ \overline{\mathit{prot}} \qquad (\textsc{proto-alloc})$$

$$\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_l}\ \chi\ (!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow \triangleright^{|\vec{v}_2|}\left(\mathsf{prot\_ctx}\ \chi\ (\vec{v}_1 \cdot [v[\vec{t}/\vec{x}]])\ \vec{v}_2\right) * \mathsf{prot\_own_l}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}]) \qquad (\textsc{proto-send-l})$$

$$\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_r}\ \chi\ (!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow \triangleright^{|\vec{v}_1|}\left((\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ (\vec{v}_2 \cdot [v[\vec{t}/\vec{x}]]))\right) * \mathsf{prot\_own_r}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}]) \qquad (\textsc{proto-send-r})$$

$$\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ ([w] \cdot \vec{v}_2) * \mathsf{prot\_own_l}\ \chi\ (?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) \Rrightarrow \triangleright\,\exists \vec{y}.\,(w = v[\vec{y}/\vec{x}]) * P[\vec{y}/\vec{x}] * \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_l}\ \chi\ \mathit{prot} \qquad (\textsc{proto-recv-l})$$

$$\mathsf{prot\_ctx}\ \chi\ ([w] \cdot \vec{v}_1)\ \vec{v}_2 * \mathsf{prot\_own_r}\ \chi\ (?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) \Rrightarrow \triangleright\,\exists \vec{y}.\,(w = v[\vec{y}/\vec{x}]) * P[\vec{y}/\vec{x}] * \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_r}\ \chi\ \mathit{prot} \qquad (\textsc{proto-recv-r})$$

$$\mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathit{prot} \sqsubseteq \mathit{prot}' \twoheadrightarrow \mathsf{prot\_own_l}\ \chi\ \mathit{prot}' \qquad (\textsc{proto-}\sqsubseteq\textsc{-l})$$

$$\mathsf{prot\_own_r}\ \chi\ \mathit{prot} * \mathit{prot} \sqsubseteq \mathit{prot}' \twoheadrightarrow \mathsf{prot\_own_r}\ \chi\ \mathit{prot}' \qquad (\textsc{proto-}\sqsubseteq\textsc{-r})$$

Figure 20: The Actris ghost theory.

and tokens $\mathsf{prot\_own_l}\ \chi\ \mathit{prot}_l$ and $\mathsf{prot\_own_r}\ \chi\ \mathit{prot}_r$—and prove rules about how they can be allocated, updated, and used. Similar to prior ghost theories, the identifier χ associates the connectives to each other. The $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$ connective can be thought of as an authority that governs the global state of the buffers $\vec{v}_1$ and $\vec{v}_2$. The tokens $\mathsf{prot\_own_l}\ \chi\ \mathit{prot}_l$ and $\mathsf{prot\_own_r}\ \chi\ \mathit{prot}_r$ provide local views of the buffers state in terms of the protocols prot_l and prot_r. As we will see in § 9.5, the authority can be shared using a lock, while the tokens provide unique ownership of each endpoint.

To define the connectives of the Actris ghost theory we use Iris's existing ghost theory for higher-order ghost variables, revolving around the two connectives $\gamma \mapsto_\bullet \mathit{prot}$ and $\gamma \mapsto_\circ \mathit{prot}'$, which we call the inner and outer fragments, respectively. As before, the γ is the ghost identifier that associates the connectives. The fragments can be thought of as two pieces of a single variable, which can only be updated in the presence of both fragments. As a result, we know that inner and outer fragment with the same ghost identifier γ always point to the same protocol *prot*. This is made precise by the rules as shown in Figure 19. In particular, higher-order ghost variables are allocated in pairs $\gamma \mapsto_\bullet \mathit{prot}$ and $\gamma \mapsto_\circ \mathit{prot}$ for an identical protocol *prot* (HO-GHOST-ALLOC), and they can only be updated together (HO-GHOST-UPDATE). This means that they will always hold the same protocol (HO-GHOST-AGREE). The subtle part of the higher-order ghost variables is that they involve ownership of a protocol of type iProto, which is defined in terms of Iris propositions iProp. Due to the dependency on iProp (which is covered in detail in § 9.1 and 9.7) the rule HO-GHOST-AGREE only gives the equality between the protocols under a later modality ($\triangleright$).

With Iris's higher-order ghost variables at hand, we can define the Actris ghost theory connectives as:

$$\mathsf{prot\_ctx}\ (\gamma_1,\gamma_2)\ \vec{v}_1\ \vec{v}_2 \triangleq \exists \mathit{prot}_1, \mathit{prot}_2.\, \gamma_1 \mapsto_\bullet \mathit{prot}_1 * \gamma_2 \mapsto_\bullet \mathit{prot}_2 * \triangleright\,\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ \mathit{prot}_1\ \mathit{prot}_2$$

$$\mathsf{prot\_own_l}\ (\gamma_1,\gamma_2)\ \mathit{prot}_l \triangleq \exists \mathit{prot}_l'.\, \gamma_1 \mapsto_\circ \mathit{prot}_l' * \triangleright(\mathit{prot}_l' \sqsubseteq \mathit{prot}_l)$$

$$\mathsf{prot\_own_r}\ (\gamma_1,\gamma_2)\ \mathit{prot}_r \triangleq \exists \mathit{prot}_r'.\, \gamma_2 \mapsto_\circ \mathit{prot}_r' * \triangleright(\mathit{prot}_r' \sqsubseteq \mathit{prot}_r)$$

Since we use two higher-order ghost variables, our identifiers $\chi ::= (\gamma_1, \gamma_2)$ are pairs of Iris ghost identifiers. The authority $\mathsf{prot\_ctx}\ (\gamma_1, \gamma_2)\ \vec{v}_1\ \vec{v}_2$ asserts ownership of the inner fragments of the higher-order ghost variables $\gamma_1 \mapsto_\bullet prot_1$ and $\gamma_2 \mapsto_\bullet prot_2$ for some protocols $prot_1$ and $prot_2$. It then asserts that the buffers $\vec{v}_1$ and $\vec{v}_2$ are consistent with respect to those protocols $prot_1$ and $prot_2$ (via $\mathsf{prot\_consistent}\ \vec{v}_1\ \vec{v}_2\ prot_1\ prot_2$). The tokens $\mathsf{prot\_own_l}\ (\gamma_1, \gamma_2)\ prot_l$ and $\mathsf{prot\_own_r}\ (\gamma_1, \gamma_2)\ prot_r$ respectively assert ownership of the outer higher-order ghost variable fragments $\gamma_1 \mapsto_\circ prot'_l$ and $\gamma_2 \mapsto_\circ prot'_r$. Here $prot'_l$ and $prot'_r$ are protocols that are weaker than the protocol arguments $prot_l$ and $prot_r$ (via $prot'_l \sqsubseteq prot_l$ and $prot'_r \sqsubseteq prot_r$). The explicit weakening under the subprotocol relation may seem redundant, as weakening is already accounted for in $\mathsf{prot\_consistent}$. However, it allows us to weaken the protocols of the tokens without the presence of the authority as shown momentarily. The later modality ($\triangleright$) makes sure that $\mathsf{prot\_own_l}\ (\gamma_1, \gamma_2)\ prot$ and $\mathsf{prot\_own_r}\ (\gamma_1, \gamma_2)\ prot$ are contractive in $prot$.

With the definitions of the ghost theory connectives at hand, we prove the rules of the ghost theory presented in Figure 20. The rule PROTO-ALLOC corresponds to allocation of a buffer pair, the rules PROTO-SEND-L and PROTO-SEND-R correspond to sending a message, and the rules PROTO-RECV-L and PROTO-RECV-R correspond to receiving a message. Finally, the rules PROTO-$\sqsubseteq$-L and PROTO-$\sqsubseteq$-R captures that we can weaken the protocols of the tokens without the presence of the authority. The rules of Figure 20 are proven through a combination of the rules for higher-order ghost state from Figure 19, and the rules for the protocol consistency relation $\mathsf{prot\_consistent}$ from § 9.3.

9.5. **The model of channel ownership.** To link the physical contents of the bidirectional channel c to the Actris ghost theory we define the channel ownership connective as follows:

$$c \rightarrowtail prot \triangleq \exists \chi, l, r, lk.\ \begin{pmatrix}(c = (l, r, lk) * \mathsf{prot\_own_l}\ \chi\ prot) \lor \\ (c = (r, l, lk) * \mathsf{prot\_own_r}\ \chi\ prot)\end{pmatrix} * \\ \mathsf{is\_lock}\ lk\ (\exists \vec{v}_1\ \vec{v}_2.\ l \overset{\text{list}}{\mapsto} \vec{v}_1 * r \overset{\text{list}}{\mapsto} \vec{v}_2 * \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2)$$

The predicate states that the referenced channel endpoint c is either the left (l, r, lk) or the right (r, l, lk) side of a channel, and that we have exclusive ownership of the ghost token $\mathsf{prot\_own_l}\ \chi\ prot$ or $\mathsf{prot\_own_r}\ \chi\ prot$ for the corresponding side. Iris's lock representation predicate $\mathsf{is\_lock}$ (previously presented in § 7) is used to make sharing of the buffers possible. The lock invariant is governed by lock lk, and carries the ownership $l \overset{\text{list}}{\mapsto} \vec{v}_1$ and $r \overset{\text{list}}{\mapsto} \vec{v}_2$ of the mutable linked lists containing the channel buffers, as well as $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$, which asserts protocol consistency of the buffers with respect to the protocols.

With the definition of the channel endpoint ownership along with the ghost theory and lock rules we then prove the channel rules HT-NEW, HT-SEND and HT-RECV from Figure 5. The proofs are carried out through symbolic execution to the point where the critical section is entered, after which the rules of the Actris ghost theory (Figure 20) are used to allocate or update the ghost state appropriately so that it matches the physical channel buffers.

The need for skip instructions. The rules PROTO-SEND-L and PROTO-SEND-R from Figure 20 contain a number of later modalities ($\triangleright$) proportional to the other endpoint's buffer. As explained in § 9.3 these later modalities are the consequence of having to perform a number of case analyses on the subprotocol relation, which is defined using guarded recursion, and thus contains a later modality for each recursive unfolding.

To eliminate these later modalities, we instrument the code of the `send` function with the `skipN` (`llength` r) instruction, which performs a number of skips equal to the size of

the other endpoint's buffer r. The `skipN` instruction has the following specification:

$$\{\triangleright^n P\}\ \texttt{skipN}\ n\ \{P\}$$

Instrumentation with skip instructions is used often in work on step-indexing, see *e.g.,* [Svendsen et al. 2016; Giarrusso et al. 2020]. Instrumentation is needed because current step-indexed logics like Iris unify physical/program steps and logical steps, *i.e.,* for each physical/program step at most one later can be eliminated from the hypotheses. In recent work by Svendsen et al. [2016]; Matsushita and Jourdan [2020]; Spies et al. [2021] more liberal versions of step-indexing have been proposed, but none of these versions of step-indexing have been integrated into the main Coq development of Iris and HeapLang.

9.6. **Adequacy of Actris.** Having constructed the model of Actris in Iris, we obtain the following main result, as first presented in § 3.4:

Theorem 2 Adequacy of Actris**.** *Let* $\varphi \in \mathsf{Val} \to \mathsf{Prop}$ *be a meta-level (i.e., Coq) predicate over values and suppose* $\{\mathsf{True}\}\ e\ \{v.\ \varphi\ v\}$ *is derivable in Iris, then* $\mathsf{safe}\ e$ *and* $\mathsf{post\_valid}\ (e, \varphi)$.

Since Actris is an internal logic embedded in Iris, the proof is an immediate consequence of Iris's adequacy theorem (Theorem 1).

9.7. **Solving the recursive domain equation for protocols.** Recall the recursive domain equation for dependent separation protocols from § 9.1:

$$\mathsf{iProto} \cong 1 + (\mathsf{action} \times (\mathsf{Val} \to \blacktriangleright\mathsf{iProto} \to \mathsf{iProp}))$$

This recursive domain equation shows that iProto depends on the type iProp of Iris propositions. To use types that depend on iProp as part of higher-order ghost state in Iris, such types need to be bi-functorial in iProp. Hence, this means that to construct iProto, in a way that it can be used in combination with the higher-order ghost variables in Figure 19, we need to solve the following recursive domain equation:

$$\mathsf{iProto}(X^-, X^+) \cong 1 + (\mathsf{action} \times (\mathsf{Val} \to \blacktriangleright\mathsf{iProto}(X^+, X^-) \to X^+))$$

Since the recursive occurrence of iProto appears in negative position, the polarity needs to be inverted for iProto to be bi-functorial.

The version of Iris's recursive domain equation solver based on [America and Rutten 1989; Birkedal et al. 2010] as mechanised in Iris's Coq development is not readily able to construct a solution of $\mathsf{iProto}(X^-, X^+)$. Concretely, the solver can only construct solutions of non-parameterised recursive domain equations. While a general construction for solving such recursive domain equations exists [Birkedal et al. 2012, § 7], that construction has not been mechanised in Coq. We circumvent this shortcoming by solving the following recursive domain equation instead, in which we unfold the recursion once by hand:

$$\mathsf{iProto}_2(X^-, X^+) \cong \\ 1 + \Big(\mathsf{action} \times \big(\mathsf{Val} \to \blacktriangleright(1 + (\mathsf{action} \times (\mathsf{Val} \to \blacktriangleright\mathsf{iProto}_2(X^-, X^+) \to X^-))) \to X^+\big)\Big)$$

Here, the polarity in the recursive occurrence is fixed, allowing us to solve $\mathsf{iProto}_2(X^-, X^+)$ using Iris's existing recursive domain equation solver. This is sufficient because a solution of $\mathsf{iProto}_2(X^-, X^+)$ is isomorphic to a solution of $\mathsf{iProto}(X^-, X^+)$.

	Notation on paper	Notation in Coq
Send	$!\, x_1 \ldots x_n \langle v \rangle \{P\}.\, prot$	`<! x_1 .. x_n> MSG v {{ P }}; prot`
Receive	$?x_1 \ldots x_n \langle v \rangle \{P\}.\, prot$	`<? x_1 .. x_n> MSG v {{ P }}; prot`
End	**end**	`END`
Dual	$\overline{prot}$	`iProto_dual prot`
Literals	(), 5, `true`	`#()`, `#5`, `#true`
Logical variables	x, y, z, _	`"x"`, `"y"`, `"z"`, `<>`
Types	1, $\mathbb{N}$, $\mathbb{Z}$	`()`, `nat`, `Z`

Figure 21: Overview of notations in the Actris Coq mechanisation.

10. Coq mechanisation

The definition of the Actris logic, its model, and the proofs of all examples in this paper have been fully mechanised using the Coq proof assistant [Coq Development Team 2020]. In this section we will elaborate on the mechanisation effort (§10.1), and go through the full proof of a message-passing program (§10.2) and a subprotocol relation (§10.3) showcasing the tactics for Actris. We display proofs and proof states taken directly from the Coq mechanisation, which differ in notation from the paper as shown in Figure 21.

10.1. **Mechanisation effort.** The mechanisation of Actris is built on top of the mechanisation of Iris [Krebbers et al. 2017a; Jung et al. 2016, 2018b]. To carry out proofs in separation logic, we use the MoSeL Proof Mode (formerly Iris Proof Mode) [Krebbers et al. 2017b, 2018], which provides an embedded proof assistant for separation logic in Coq. Building Actris on top of the Iris and MoSeL framework in Coq has a number of tangible advantages:

- By defining channels on top of HeapLang, we do not have to define a full programming language semantics, and can reuse all of the program libraries and Coq machinery, including the tactics for symbolic execution of non message-passing programs.
- Since Actris is mechanised as an Iris library we get all of the features of Iris for free, such as the ghost state mechanisms for reasoning about concurrency.
- When proving the Actris proof rules, we can make use of the MoSeL Proof Mode to carry out proofs directly using separation logic, thus reasoning at a high level of abstraction.
- We can make use of the extendable nature of the MoSeL Proof Mode to define custom tactics for symbolic execution of message-passing programs.

These advantages made it possible to mechanise Actris, along with the examples of the paper, with a small Coq development of a total size of about 5000 lines of code (comments and whitespace included). The line count of the different components are shown in Figure 22.

10.2. **Tactic support for session type-based reasoning.** To carry out interactive Actris proofs using symbolic execution, we follow the methodology described in the original Iris Proof Mode paper [Krebbers et al. 2017b]. In particular, this means that the logic in Coq is presented in weakest precondition style rather than using Hoare triples. For handling `send` or `recv` we define the following tactics:

`wp_send (t1 .. tn) with "[H1 .. Hn]"` and `wp_recv (y1 .. yn) as "H"`.

These tactics roughly perform the following actions:

- Find a `send` or `recv` in evaluation position of the program under consideration.

Component	**Sections**	**~LOC**
The Actris model	§ 9.1–§ 9.4	1500
Channel implementation and proof rules	§ 2.4 and 9.5	350
Tactics for symbolic execution	§ 10.2	500
Utilities (linked lists, permutations, *etc.*)	n.a.	450
Authoritative contribution ghost theory	§ 7	150
Recursive domain equation theory solver	§ 9.7	100
Examples:		
• Basic examples	§ 1 and 7.1	400
• Coarse-grained channel-based merge sort	§ 5.1–§ 5.5	250
• Fine-grained channel-based merge sort	§ 5.6	300
• Mapper with swapping	§ 6.2	400
• List reversal	§ 6.3	100
• Channel-based load-balancing mapper	§ 7.2	200
• Channel-based map-reduce	§ 8	300
Total		5000

Figure 22: Overview of components of the Actris Coq mechanisation.

- Find a corresponding $c \rightarrowtail \mathit{prot}$ hypothesis in the separation logic context.
- Normalise the protocol prot using the rules for duals, composition, recursion, and swapping so it has a $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ or $?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ construct in its head position.
- In case of `wp_send`, instantiate the variables $\vec{x}{:}\vec{\tau}$ using the terms `(t1 .. tn)`, and create a goal for the proposition P with the hypotheses `[H1 .. Hn]`. Hypotheses prefixed with `$` will automatically be consumed to resolve a subgoal of P if possible. In case the terms `(t1 .. tn)` are omitted, an attempt is made to determine these using unification.
- In case of `wp_recv`, introduce the variables $\vec{x}{:}\vec{\tau}$ into the context by naming them `(y1 .. yn)`, and create a hypothesis `H` for P.

The implementation of these tactics follows the approach by Krebbers et al. [2017b]. The protocol normalisation is implemented via logic programming with type classes.

As an example we will go through a proof of the following program:

$$\begin{aligned}&\texttt{prog\_ref\_swap\_loop} := \lambda().\ \texttt{let}\ c := \texttt{start}\ (\texttt{rec}\ \mathit{go}\ c' := \texttt{let}\ l := \texttt{recv}\ c'\ \texttt{in}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad l \leftarrow !\,l + 2;\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{send}\ c'\ ();\ \mathit{go}\ c')\ \texttt{in}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\texttt{let}\ l_1 := \texttt{ref}\ 18\ \texttt{in}\ \texttt{let}\ l_2 := \texttt{ref}\ 20\ \texttt{in}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\texttt{send}\ c\ l_1;\ \texttt{send}\ c\ l_2;\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\texttt{recv}\ c;\ \texttt{recv}\ c;\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad!\,l_1 + !\,l_2\end{aligned}$$

Here, the forked-off thread acts as a service that recursively receives locations, adds 2 to their stored number, and then sends back a flag indicating that the location has been updated. The main thread, acting like a client, first allocates two new references, to 18 and 20, respectively, which are both sent to the service after which the update flags are received. It finally dereferences the updated locations, and adds their values together, thus returning 42. To verify this program, we use the following recursive protocol:

$$\texttt{prot\_ref\_loop} \triangleq \mu(\mathit{rec} : \mathsf{iProto}).\ !\,(\ell : \mathsf{Loc})(x : \mathbb{Z})\,\langle \ell\rangle\{\ell \mapsto x\}.\ ?\langle()\rangle\{\ell \mapsto x + 2\}.\ \mathit{rec}$$

```
1  Lemma prog_ref_swap_loop_spec : ∀Φ, Φ #42 -∗ WP prog_ref_swap_loop #() {{ Φ }}.
2  Proof.
3    iIntros (Φ) "HΦ". wp_lam.
4    wp_apply (start_chan_spec prot_ref_loop); iIntros (c) "Hc".
5    - iLöb as "IH". wp_lam.
6      wp_recv (l x) as "Hl". wp_load. wp_store. wp_send with "[$Hl]".
7      do 2 wp_pure _. by iApply "IH".
8    - wp_alloc l1 as "Hl1". wp_alloc l2 as "Hl2".
9      wp_send with "[$Hl1]". wp_send with "[$Hl2]".
10     wp_recv as "Hl1". wp_recv as "Hl2".
11     wp_load. wp_load.
12     wp_pures. by iApply "HΦ".
13 Qed.
```

Figure 23: Proof of message-passing program

The (forked-off) service follows the (dual of) the protocol exactly, while the main thread follows a weakened version. The recursion is unfolded twice, after which the second send has been swapped ahead of the first receive, allowing it to first send both values before receiving:

$$\begin{aligned}\texttt{prot\_ref\_loop} \sqsubseteq\ & !(\ell_1 : \mathsf{Loc})(x_1 : \mathbb{Z})\,\langle \ell_1 \rangle \{\ell_1 \mapsto x_1\}.\\ & !(\ell_2 : \mathsf{Loc})(x_2 : \mathbb{Z})\,\langle \ell_2 \rangle \{\ell_2 \mapsto x_2\}.\\ & ?\langle () \rangle \{\ell_1 \mapsto (x_1 + 2)\}.\\ & ?\langle () \rangle \{\ell_2 \mapsto (x_2 + 2)\}.\,\texttt{prot\_ref\_loop}\end{aligned}$$

The full Coq proof of the program is shown in Figure 23. The proven lemma is logically equivalent to the specification $\{\mathsf{True}\}\,\texttt{prog\_ref\_swap\_loop}\,()\,\{v.\, v = 42\}$, but is presented in weakest precondition style as is common in Iris in Coq. The initial proof state is:

```
--------------------------------------∗
∀Φ, Φ #42 -∗ WP prog_ref_swap_loop #() {{ v, Φ v }}
```

We start the proof on line 3 by introducing the postcondition Φ, and the hypothesis `HΦ: Φ #42`, and then continue by evaluating the lambda expression with `wp_lam`. On line 4 we apply the specification `start_chan_spec`, which is the weakest precondition variant of HT-START for `start` by picking the expected protocol `prot_ref_loop`. This leaves us with two subgoals, separated by bullets "-": one for the forked-off thread, and one for the main thread.

Proof of the forked-off thread. In the proof of the recursively-defined forked-off thread we use `iLöb as "IH"` for LÖB induction on line 5. This leaves us with the proof state:

```
"IH" : ▷(c ⤞ iProto_dual prot_ref_loop -∗
          WP (rec: "go" "c'" :=
                let: "l" := recv "c'" in
                "l" <- ! "l" + #2;;
                send "c'" #();; "go" "c'") c {{ _, True }})
--------------------------------------□
"Hc" : c ⤞ iProto_dual prot_ref_loop
--------------------------------------∗
WP (rec: "go" "c'" :=
      let: "l" := recv "c'" in
      "l" <- ! "l" + #2;;
      send "c'" #();; "go" "c'") c {{ _, True }}
```

We now resolve the application of `c` to the recursive function with `wp_lam`. This lets us strip the later from the Löb induction hypothesis, as the program has taken a step. The proof state is then as follows:

```
"IH" : c ↣ iProto_dual prot_ref_loop -∗ WP prog_rec c {{ _, True }}
--------------------------------------□
"Hc" : c ↣ iProto_dual prot_ref_loop
--------------------------------------∗
WP let: "l" := recv c in
   "l" <- ! "l" + #2;;
   send c #();; prog_rec c {{ _, True }}
```

For brevity's sake we abbreviate the recursive code in `"IH"` as `prog_rec c`.

On line 6 we resolve the proof of the body of the recursive function. So far, the proof only used Iris's standard tactics, we now use the Actris tactic for receive `wp_recv (l x) as "Hl"`, to resolve the receive in evaluation position, introducing the received logical variables `l` and `x`, along with the predicate of the protocol `l ↦ #x` naming it `Hl`. To do so, the protocol is normalised, unfolding the recursive definition once, as well as resolving the dualisation of the head, turning it into a receive as expected. This leads to the following proof state:

```
"IH" : c ↣ iProto_dual prot_ref_loop -∗ WP prog_rec c {{ _, True }}
--------------------------------------□
"Hl" : l ↦ #x
"Hc" : c ↣ iProto_dual (<?> MSG #() {{ l ↦ #(x + 2) }}; prot_ref_loop)
--------------------------------------∗
WP let: "l" := #l in
   "l" <- ! "l" + #2;;
   send c #();; prog_rec c {{ _, True }}
```

We then use the HeapLang tactics `wp_load` and `wp_store` to resolve the dereferencing and updating of the location:

```
"IH" : c ↣ iProto_dual prot_ref_loop -∗ WP prog_rec c {{ _, True }}
--------------------------------------□
"Hl" : l ↦ #(x + 2)
"Hc" : c ↣ iProto_dual (<?> MSG #() {{ l ↦ #(x + 2) }}; prot_ref_loop)
--------------------------------------∗
WP send c #();; prog_rec c {{ _, True }}
```

We then use the Actris tactic `wp_send with "[$Hl]"` to resolve the send operation in evaluation position, by giving up the ownership of `"Hl"`. Again, the protocol is automatically normalised by resolving the dualisation of the receive (**?**) to obtain the send (**!**) as expected.

We finally close the proof of the forked-off thread on line 7. We first take two pure evaluation steps revolving the sequencing of operations with `do 2 wp_pure _` to reach the recursive call. This results in the proof state:

```
"IH" : c ↣ iProto_dual prot_ref_loop -∗ WP prog_rec c {{ _, True }}
--------------------------------------□
"Hc" : c ↣ iProto_dual prot_ref_loop
--------------------------------------∗
WP prog_rec c {{ _, True }}
```

We then use `by iApply "IH"` to close the proof by using the Löb induction hypothesis.

Proof of the main thread. The proof of the main thread follows similarly. On line 8 we use `wp_alloc l1 as "Hl1"` and `wp_alloc l2 as "Hl2"`, to resolve the allocations of the new locations, binding the logical variables of the locations to `l1` and `l2`, and adding hypotheses `"Hl1"` and `"Hl2"` for ownership of these locations to the separation logic proof context. The proof state is then:

```
"HΦ" : Φ #42
"Hc" : c ⤞ prot_ref_loop
"Hl1" : l1 ↦ #18
"Hl2" : l2 ↦ #20
--------------------------------------*
WP send c #l1;; send c #l2;; recv c;; recv c;; ! #l1 + ! #l2 {{ v, Φ v }}
```

On line 9, we resolve the first send operation with the Actris tactic `wp_send with "[$Hl1]"`, by giving up ownership of the location `l1`. Here, the protocol is normalised by unfolding the recursive definition, after which the head symbol is a send (**!**) as expected. The resulting proof state is as follows:

```
"HΦ" : Φ #42
"Hl2" : l2 ↦ #20
"Hc" : c ⤞ (<?> MSG #() {{ l1 ↦ #(18 + 2) }}; prot_ref_loop)
--------------------------------------*
WP send c #l2;; recv c;; recv c;; ! #l1 + ! #l2 {{ v, Φ v }}
```

To resolve the second send operation, we need to weaken the protocol using swapping (rule ⊑-SWAP'), which is taken care of automatically by the Actris tactic `wp_send with "[$Hl2]"`. The normalisation detects that the protocol has a receive (**?**) as a head symbol, and therefore attempts swapping. To do so it steps ahead of the receive (**?**), and unfolds the recursive definition, which results in a send (**!**) as the first symbol after the head. It then detects that there are no dependencies between the two, and can thus apply the swapping rule ⊑-SWAP', moving the send (**!**) ahead of the receive (**?**). With the head symbol now being a send (**!**), the symbolic execution continues as normal, resulting in the proof state:

```
"HΦ" : Φ #42
"Hc" : c ⤞ (<?> MSG #() {{ l1 ↦ #(18 + 2) }};
             <?> MSG #() {{ l2 ↦ #(20 + 2) }}; prot_ref_loop)
--------------------------------------*
WP recv c;; recv c;; ! #l1 + ! #l2 {{ v, Φ v }}
```

On line 10 we then proceed as expected with `wp_recv as "Hl1"` and `wp_recv as "Hl2"`, to resolve the receive operations, giving us back the updated point-to resources:

```
"HΦ" : Φ #42
"Hl1" : l1 ↦ #(18 + 2)
"Hl2" : l2 ↦ #(20 + 2)
"Hc" : c ⤞ prot_ref_loop
--------------------------------------*
WP ! #l1 + ! #l2 {{ v, Φ v }}
```

At line 11 we then continue by using `wp_load` twice to dereference the reacquired and updated locations, and then use trivial symbolic execution using `wp_pures` to resolve the remaining computations. On line 12 we finally close the proof by applying the hypothesis `"HΦ"` about the postcondition.

```
Lemma list_rev_subprot :
  ⊢ (<! (l : loc) (vs : list val)> MSG #l {{ llist l vs }};
     <?> MSG #() {{ llist internal_eq l (reverse vs) }}; END) ⊑
    (<! (l : loc) (xs : list T)> MSG #l {{ llistI IT l xs }};
     <?> MSG #() {{ llistI IT l (reverse xs) }}; END).
Proof.
  iIntros (l xs) "Hl".
  iDestruct (Hlr with "Hl") as (vs) "[Hl HIT]".
  iExists l, vs. iFrame "Hl".
  iModIntro. iIntros "Hl".
  iSplitL.
  { rewrite big_sepL2_reverse_2. iApply Hlr.
    iExists (reverse vs). iFrame "Hl HIT". }
  done.
Qed.
```

Figure 24: Proof of subprotocol relation

10.3. **Tactic support for subprotocols.** While the Actris tactics automatically apply the subprotocol rules during symbolic execution, as shown in § 10.2, we sometimes want to prove subprotocol relations as explicit lemmas. We have tactic support for such proofs as well. We extend the existing MoSeL tactics `iIntros`, `iExists`, `iFrame`, `iModIntro`, and `iSplitL`/`iSplitR` to automatically use the subprotocol rules to turn the goal into an equivalent goal where the regular Iris tactics apply.

- `iIntros (x1 .. xn) "H1 .. Hm"` transforms the subprotocol goal to begin with `n` universal quantification and `m` implications, using the rules $\sqsubseteq$-SEND-OUT and $\sqsubseteq$-RECV-OUT, and then introduces the quantifiers (naming them `x1 .. xn`) into the Coq context, and the hypotheses (naming them `H1 .. Hm`) into the separation logic context.
- `iExists (t1 .. tn)` transforms the subprotocol goal to start with `n` existential quantifiers, using the $\sqsubseteq$-SEND-IN, $\sqsubseteq$-RECV-IN and $\sqsubseteq$-TRANS rules, and then instantiates these quantifiers with the terms `t1 .. tn` specified by the pattern.
- `iFrame "H"` transforms the subprotocol goal into a separating conjunction between the payload predicates of the head symbols of either protocol, using the rules $\sqsubseteq$-SEND-IN and $\sqsubseteq$-RECV-IN, and then tries to solve the payload predicate subgoal using `"H"`.
- `iModIntro` transforms the subprotocol goal into a goal starting with a later modality ($\triangleright$), using the rules $\sqsubseteq$-SEND-MONO and $\sqsubseteq$-RECV-MONO, and then introduces that later by stripping off a later from any hypothesis in the separation logic context.
- `iSplitL/iSplitR "H1 .. Hn"` transforms the subprotocol goal into a separating conjunction between the payload predicates of the head symbols of either protocol, using the $\sqsubseteq$-SEND-IN, $\sqsubseteq$-RECV-IN and $\sqsubseteq$-TRANS rules, and then creates two subgoals. For `iSplitL` the left subgoal is given the hypotheses `H1 .. Hn` from the separation logic context, while the right subgoal is given any remaining hypotheses, and *vice versa* for `iSplitR`.

The extensions of these tactics are implemented by defining custom type class instances that hook into the existing MoSeL tactics as described by Krebbers et al. [2017b].

To demonstrate these tactics, we will go through a proof of the subprotocol relation for the list reversing service presented in § 6.3:

$$\begin{aligned} &\mathbf{!}\,(\ell : \mathsf{Loc})(\vec{v} : \mathsf{List\ Val})\,\langle \ell \rangle \{ \ell \overset{\text{list}}{\mapsto} \vec{v} \}.\, \mathbf{?}\langle () \rangle \{ \ell \overset{\text{list}}{\mapsto} \mathsf{reverse}\ \vec{v} \}.\, \mathbf{end} \\ \sqsubseteq\ &\mathbf{!}\,(\ell : \mathsf{Loc})(\vec{x} : \mathsf{List}\ T)\,\langle \ell \rangle \{ \ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x} \}.\, \mathbf{?}\langle () \rangle \{ \ell \overset{\text{list}}{\mapsto}_{I_T} \mathsf{reverse}\ \vec{x} \}.\, \mathbf{end} \end{aligned}$$

Recall that the following conversion between the list representation predicate with payload $\ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x}$ and one without payload $\ell \overset{\text{list}}{\mapsto} \vec{v}$ holds:

$$\texttt{Hlr} \quad : \quad \ell \overset{\text{list}}{\mapsto}_{I_T} \vec{x} \;\ast\!\ast\; (\exists \vec{v}.\, \ell \overset{\text{list}}{\mapsto} \vec{v} \ast \mathop{\Large\ast}_{(x,v)\in(\vec{x},\vec{v})} I_T\ x\ v)$$

The full Coq proof of the subprotocol relation is shown in Figure 24. The initial proof state is identical to the lemma statement. On line 7 we start by introducing the logical variables `l`, `xs` and the payload `llistI IT l xs` of the weaker protocol with the tactic `iIntros (l xs) "Hl"`. This tactic will implicitly apply the rule $\sqsubseteq$-SEND-OUT, so the goal starts with a universal quantification `∀(l : loc)(xs : list T). llistI IT l xs -∗...`, which is then introduced based on the regular Iris introduction pattern. This gives us:

```
"Hl" : llistI IT l vs
--------------------------------------∗
(<! (l : loc) (vs : list val)> MSG #l {{ llist l vs }};
 <?> MSG #() {{ llist l (reverse vs) }}; END) ⊑
(<!> MSG #l; <?> MSG #() {{ llistI IT l (reverse xs) }}; END)
```

To obtain the payload predicate expected by the stronger protocol, we use the lemma `Hlr`, to derive `llist l vs` and `[∗list] x;v ∈ xs;vs, IT x v` from `llistI l xs` with the tactic `iDestruct (Hlr with "Hl") as (vs) "[Hl HIT]"` on line 8. The resulting proof state is:

```
"Hl" : llist l vs
"HIT" : [∗ list] x;v ∈ xs;vs, IT x v
--------------------------------------∗
(<! (l : loc) (vs : list val)> MSG #l {{ llist l vs }};
 <?> MSG #() {{ llist l (reverse vs) }}; END) ⊑
(<!> MSG #l; <?> MSG #() {{ llistI IT l (reverse xs) }}; END)
```

At line 9 we instantiate the logical variables of the stronger protocol with the logical variables `l` and `vs` using `iExists l, vs`. This will implicitly apply the rules $\sqsubseteq$-SEND-IN and $\sqsubseteq$-TRANS, which makes the goal start with `∃(l : loc) (vs : list val)`, so the existentials can be instantiated. To resolve the payload predicate obligation `llist l vs`, we use `iFrame "Hl"`. This uses the rules $\sqsubseteq$-SEND-IN and $\sqsubseteq$-TRANS to turn the goal into `llist l vs ∗ ...`, where the left subgoal is resolved using `"Hl"`. We then have the following remaining proof state:

```
"HIT" : [∗ list] x;v ∈ xs;vs, IT x v
--------------------------------------∗
(<!> MSG #l; <?> MSG #() {{ llist l (reverse vs) }}; END) ⊑
(<!> MSG #l; <?> MSG #() {{ llistI IT l (reverse xs) }}; END)
```

As the head symbols of both protocols are sends (**!**) with no logical variables or payload predicates, we use `iModIntro` on line 10, which first applies $\sqsubseteq$-SEND-MONO to step over the sends, and then introduces the later modality ($\triangleright$). This gives us the proof state:

```
"HIT" : [∗ list] x;v ∈ xs;vs, IT x v
--------------------------------------∗
(<?> MSG #() {{ llist l (reverse vs) }}; END) ⊑
(<?> MSG #() {{ llistI IT l (reverse xs) }}; END)
```

On line 10, similarly to before, we use `iIntros "Hl"`, to introduce the payload predicate, but this time we do it for the stronger protocol, as dictated by $\sqsubseteq$-RECV-OUT:

```
"HIT" : [∗ list] x;v ∈ xs;vs, IT x v
"Hl" : llist l (reverse vs)
--------------------------------------∗
```

```
(<?> MSG #() ; END) ⊑
(<?> MSG #() {{ llistI IT l (reverse xs) }}; END)
```

To resolve the payload predicate of the weaker protocol, we use `iSplitL "Hl HIT"` on line 11, that first use ⊑-RECV-IN and ⊑-TRANS, to turn the goal into `llistI IT l (reverse xs) ∗ ...`, and then use the goal splitting pattern of Iris, to give us two subgoals, where we use the hypotheses `"Hl"` and `"HIT"` in the left subgoal. The first subgoal is then:

```
"HIT" : [∗ list] x;v ∈ xs;vs, IT x v
"Hl" : llist l (reverse vs)
--------------------------------------∗
llistI IT l (reverse xs)
```

On line 12, we first use the lemma `Hlr` in the right-to-left direction, and then rewrite the hypothesis `"HIT"` using a lemma from the Iris library with `rewrite big_sepL2_reverse_2`. We do this to obtain `[∗list] x;v ∈ reverse xs;reverse vs, IT x v`, in order to match the proof goal. This gives the proof obligation:

```
"HIT" : [∗ list] x;v ∈ reverse xs;reverse vs, IT x v
"Hl" : llist l (reverse vs)
--------------------------------------∗
∃ vs : list val, llist l vs ∗ ([∗ list] x;v ∈reverse xs;vs, IT x v)
```

We finally close the proof on line 13 with `iExists (reverse vs)`, followed by `iFrame "Hl HIT"`, as the goal matches the hypotheses exactly, when picking `reverse vs` as the existential quantification. We then move on to the second subgoal:

```
--------------------------------------∗
(<?> MSG #(); END) ⊑ (<?> MSG #(); END)
```

We resolve this subgoal, on line 14, with the tactic `done`, which tries to close the proof, by automatically applying ⊑-REFL.

11. Related work

This section elaborates on the relation to message passing in separation logic (§ 11.1) and process calculi (§ 11.2), session types (§ 11.3), session subtyping (§ 11.4), endpoint sharing (§ 11.5), and verification of map-reduce (§ 11.6).

11.1. **Message passing and separation logic.** Villard et al. [2009]; Lozes and Villard [2012] present a logic for contract-based reasoning about programs in a small imperative language with bi-directional asynchronous channels. Contracts are represented by finite-state automata with labelled send or receive transitions, equipped with separation logic predicates. Similar to session types (and Actris), contracts have a notion of duality, but unlike Actris they do not support dependencies between messages. Their logic supports ownership transfer (including ownership transfer of channels, akin to delegation), session-type like choice, and a form of recursive contracts. Their language has a close operation for channel deallocation instead of being garbage collected. A restriction to structured concurrency (*i.e.,* par instead of fork-based), structured channel deallocation (*i.e.,* must close both endpoints together) and linear (instead of affine) logic ensures memory-leak freedom. A form of channel sharing is supported, which we further discuss in § 11.5.

Craciun et al. [2015] introduced *session logic*, a variant of separation logic that includes predicates for protocol specifications similar to ours. This work includes support for mutable state, ownership transfer (including ownership transfer of channels, akin to delegation), session-type like choice using a special type of disjunction operator on the protocol level, and a sketch of an approach to verify deadlock freedom of programs. Combined, these features allow them to verify interesting and non-trivial message-passing programs. Their logic as a whole is not higher-order, which means that sending functions over channels is not possible. Moreover, their logic does not support protocol-level logical variables that can connect the transferred message with the tail protocol. It is therefore not possible to model dependent protocols like we do in Actris. Their work includes a notion of subtyping as weakening and strengthening of the payload predicates, however they do not consider swapping, and do not allow manipulation of resources as a part of their subtyping relation. There also exists no support for other concurrency primitives such as locks, which by extension means that manifest sharing is not possible. In Actris we get this for free by building on top of Iris, and reusing its ghost state mechanism. Their work has not been mechanised in a proof assistant, but example programs can be checked using the HIP/SLEEK verifier.

The original Iris paper [Jung et al. 2015] includes a small message-passing language with channels that do not preserve message order. It was included to demonstrate that Iris is flexible enough to handle other concurrency models than standard shared-memory concurrency. Since the Hoare triples for send and receive reason about the entire channel buffer, protocol reasoning must be done via STSs or other forms of ghost state.

Hamin and Jacobs [2019] take an orthogonal direction and use separation logic to prove deadlock freedom of programs that communicate via message passing using a custom logic tailored to this purpose. They do not provide abstractions akin to our session-type based protocols. Instead one has to reason using invariants and ghost state explicitly.

Mansky et al. [2017] verify the functional correctness of a message-passing system written in C using the VST framework in Coq [Appel 2014]. While they do not verify message-passing programs like we do, they do verify that the implementation of their message-passing system is resilient to faulty behaviour in the presence of malicious senders and receivers.

Tassarotti et al. [2017] prove correctness and termination preservation of a compiler from a simple language with session types to a functional language with mutable state, where channels are implemented using references on the heap. This work is also done in Iris in Coq. The session types they consider are more like standard session types, which cannot express functional properties of messages, but only their types.

The Disel logic by Sergey et al. [2018] and the Aneris logic by Krogh-Jespersen et al. [2020] can be used to reason about message-passing programs that work on network sockets. Channels can only be used to send strings, are not order preserving, and messages can be dropped but not duplicated. Since only strings are sent over channels complex data (such as functions) must be marshalled and unmarshalled in order to be sent over the network. Both Disel and Aneris therefore address a different problem than we do.

SteelCore [Swamy et al. 2020] is a framework for concurrent separation logic embedded in the F* language. SteelCore has been used to encode unidirectional synchronous channels that can be typed with protocols akin to session types. Their protocols are defined as a dependent sequence of value obligations with associated separation logic predicates, dictating what can be sent over the channel, including the transfer of ownership. Channels are first-class and can also be transferred (akin to delegation), but their protocols do not include higher-order

protocol-level logical variables, or subtyping. They postulated that their approach scales to bidirectional asynchronous communication, but left that for future work.

11.2. **Separation logic and process calculi.** Another approach to verify message-passing programs is to combine separation logic and process calculus. Neither of the approaches below support delegation or concurrency paradigms other than message passing.

Francalanza et al. [2011] use separation logic to verify programs written in a CCS-like language. Channels model memory location, which has the effect that their input-actions behave a lot like our updates of mutable state with variable substitutions updating the state. As a proof of concept they prove the correctness of an in-place quick-sort algorithm.

Oortwijn et al. [2016] use separation logic and the mCRL2 process calculus to model communication protocols. The logic itself operates on a high level of abstraction and deals exclusively with intraprocess communication where a fractional separation logic is used to distribute channel resources to concurrent threads. Protocols are extracted from code, but there is no formal connection between the specification logic and the underlying language.

11.3. **Session types.** Seminal work on linear type systems for the π-calculus by Kobayashi et al. [1996] led to the creation of binary session types by Honda et al. [1998], and consequentially multiparty session types by Honda et al. [2008].

Later work by Dardha et al. [2012] helped merge the linear type systems of Kobayashi with Honda's session types, which facilitated the incorporation of session types in mainstream programming languages like Go [Lange et al. 2018], OCaml [Padovani 2017; Imai et al. 2019], and Java [Hu et al. 2010]. These works focus on adding session-typed support for message passing in existing languages, but do not target functional correctness.

Bocchi et al. [2010] pushed the boundaries of what can be verified with (multiparty) session types while staying within a decidable fragment of first-order logic. They use first-order predicates to describe properties of values being sent and received. Decidability is maintained by imposing restrictions on these predicates, such as ensuring that nothing is sent that will be invalidated down the line. The constraints on the logic do, however, limit what programs can be verified. The work includes standard subtyping on communicated values and on choices, but no notion of swapping sends ahead of receives.

Caires and Pfenning [2010] discovered a correspondence between intuitionistic linear logic and π-calculus with session types, which was extended with quantifiers and dependent types by Toninho et al. [2011]. These quantifiers range over both terms and propositions of an LF-based logic (Cervesato and Pfenning [1996]), and can be used to specify basic properties of the exchanged values. Toninho and Yoshida [2018] extended this work by allowing the structure of the protocol to depend on the quantifiers. This notion of dependency allows for protocols where the length of the (tail) protocol depends on the values that were previously exchanged, similar to what we do in § 5.6. Finally, Das and Pfenning [2020a,b] developed a dependent session-type system with domain-specific logic for verifying arithmetic properties of programs with message passing.

Another approach to dependent session types was carried out by Thiemann and Vasconcelos [2020] who introduced label-dependent session types. They unify universal and existential quantifiers with the send and receive primitives of conventional session types. Hence, similar to Actris, the choice connectives ($\&$ and $\oplus$) can be derived.

Toninho et al. [2014]; Lindley and Morris [2016] developed session-type systems with termination guarantees in the presence of recursive (session) types. This is achieved by imposing a discipline similar to (co)inductive definitions in Coq and Agda. In contrast, Actris poses no usage discipline on recursive dependent separation protocols, and hence guarantees partial correctness.

11.4. **Session subtyping.** Actris's subprotocol relation is inspired by the notion of session subtyping, for which seminal work was carried out by Gay and Hole [2005]. Mostrous et al. [2009] extended session subtyping to multiparty asynchronous session types, and as part of that, introduced the notion of swapping sends ahead of receives for independent channels. Mostrous and Yoshida [2015] later considered swapping over the same channel in the context of binary session types. Our subprotocol relation is most closely related to the work of Mostrous and Yoshida [2015], although they define subtyping as a simulation on infinite trees, using so-called asynchronous contexts, whereas we define it using Iris's support for guarded recursion. It should be noted that the work by Gay and Hole [2005] differs from the work by Mostrous et al. [2009] and Mostrous and Yoshida [2015] in the orientation of the subtyping relation, as discussed by Gay [2016]. Our subprotocol relation uses the orientation of Gay and Hole [2005].

Session subtyping for recursive type systems is universally carried out as a type simulation on infinite trees [Gay and Hole 2005; Mostrous et al. 2009; Mostrous and Yoshida 2015], which complicates subtyping under the recursion operator. Bernardi et al. [2014] and Gay et al. [2020] provide further insights on this problem, although they primarily investigate duality rather than subtyping.

To reason about recursive subtyping, Brandt and Henglein [1998] present a coinductive formulation of subtyping (which they apply to regular type systems, rather than session types). We use a similar coinductive formulation, but instead of ordinary coinduction, we use Iris's support for guarded recursion, which lets us prove subtyping relations of recursive protocols using Löb induction.

11.5. **Endpoint sharing.** One of the key features of conventional session types is that endpoints are owned by a single thread. While endpoints can be delegated (*i.e.,* transferred from one thread to another), they typically cannot be shared (*i.e.,* be accessed by multiple threads concurrently). However, as demonstrated in § 7, sharing channels endpoints is often desirable, and possible in Actris.

As a simple way to relax this limitation of sharing in conventional session types, Vasconcelos [2012] allows session types of the form $(\mu rec.\, !T.\, rec)$ or $(\mu rec.\, ?T.\, rec)$ to be shared. Lozes and Villard [2012] present a similar idea in the context of their contract-based separation logic (see also § 11.1) by equipping the connective for channel endpoint ownership with a fractional permission. If the fraction is smaller than 1, then the endpoint can be shared, but at the cost of only permitting transitions to the same contract state. Using fractional permissions they prove a lock specification à la Gotsman et al. [2007] of an implementation of locks in terms of channels. This approach to locks is dual to ours in Actris, where we implement channels in terms of locks. Unlike Iris (and Actris), their logic does not support ghost state, so it cannot express complex protocols like the ones from § 7.

In the π-calculus community there has been prior work on endpoint sharing, *e.g.,* by Atkey et al. [2016]; Kobayashi [2006]; Padovani [2014]. The latest contribution in this line of

work is by Balzer and Pfenning [2017]; Balzer et al. [2019], who developed a type system based on session types with support for manifest sharing. Manifest sharing is the notion of sharing a channel endpoint between multiple processes using a lock-like structure to ensure mutual exclusion. Their key idea to ensure mutual exclusion using a type system is to use adjoint modalities to connect two classes of types: types that are linear, and thus denote unique channel ownership, and types that are unrestricted, and thus can be shared. The approach to endpoint sharing in Actris is different: dependent separation protocols do not include a built-in notion for endpoint sharing, but can be combined with Iris's general-purpose mechanisms for sharing, like locks.

11.6. **Verification of map-reduce.** To our knowledge the only verification related to the map-reduce model [Dean and Ghemawat 2004] is by Ono et al. [2011], who made two mechanisations in Coq. The first took a functional model of map-reduce and verified a few specific mappers and reducers, extracted these to Haskell, and ran them using Hadoop Streaming. The second did the same by annotating Java mappers and reducers using JML and proving them correct using the Krakatoa tool [Marché et al. 2004], using a combination of SAT-solvers and the Coq proof assistant. While they worked on verifying specific mappers and reducers, our case study focuses on verifying the communication of a map-reduce model that can later be parameterised with concrete mappers and reducers.

12. Conclusion and future work

In this paper, we have given a comprehensive account of the Actris concurrent separation logic for proving functional correctness of programs that combine message-passing with other programming and concurrency paradigms. The core feature of Actris its its mechanism of dependent separation protocols, which is inspired by session types. Considering the rich literature on session types and concurrent separation logic, we expect there to be many promising directions for future work.

Multi-party. The formalism of multi-party session types [Honda et al. 2008] applies to message-passing communication between more than two parties (threads or processes). The key ingredient of multi-party session types is the notion of a *global protocol*, which specifies the permitted communication for multiple parties of a system. From the global protocol one can then generate *local protocols* for the individual parties. It would be interesting to explore a multi-party version of dependent separation protocols. Prior work by Costea et al. [2018] on multi-party session logic and Zhou et al. [2020] on refined multiparty session types could serve as a starting point.

Deadlock freedom. As discussed in § 4.3, deadlocks are valid behaviours according to the notion of safety used in Iris (and thus Actris). Many conventional session type systems do not consider deadlocks to be valid behaviours, but achieve that at the expense of prohibiting valid (deadlock free) programs that can be verified in Actris.

A direction for future work is to develop a variant of Actris that incorporates the usual restrictions of session-type systems like linearity and a `start` primitive for combined channel and thread creation. To prove an adequacy theorem that ensures that this variant of Actris indeed prohibits deadlocks, one needs to change the model of Actris to ensure acyclicity of the

dependency structure among the threads and channels. This could be achieved by building upon recent work by Bizjak et al. [2019] on linearity in Iris and by Jacobs et al. [2021] on a separation-logic based proof method for deadlock freedom of session types. Additionally, one could consider a version of Actris without garbage collection but with a `close` instruction for channel deallocation, and prove that it indeed guarantees memory-leak freedom.

Another direction for future work is to develop a separation logic that combines session-type based deadlock freedom with lock-order based deadlock freedom to prove deadlock freedom of programs that combine message passing with other concurrency mechanisms like locks. The work by Hamin and Jacobs [2019] on reasoning about lock orders in separation logic, and the work by Balzer et al. [2019] on deadlock freedom for manifest sharing might provide valuable insights, but figuring out how to combine these two approaches with Iris and Actris is a challenging open problem.

Acknowledgments

We thank the anonymous reviewers of both this paper and the POPL'20 conference version for their helpful feedback. We are grateful to Andreea Costea, Daniel Gratzer, Daniël Louwrink, Fabrizio Montesi, Marco Carbone, and the participants of the Iris workshop 2019 for discussions. The third author (Robbert Krebbers) was supported by the Dutch Research Council (NWO), project 016.Veni.192.259.

References

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

Pierre America and Jan J. M. M. Rutten. 1989. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. *JCSS* 39, 3 (1989), 343–375.

Andrew W. Appel. 2014. *Program Logics for Certified Compilers*. Cambridge University Press.

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

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. 109–122.

Robert Atkey, Sam Lindley, and J. Garrett Morris. 2016. Conflation Confers Concurrency. In *Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 32–55.

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *PACMPL* 1, ICFP (2017), 37:1–37:29.

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *ESOP*. 611–639.

Giovanni Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. 2014. On Duality Relations for Session Types. In *TGC (LNCS)*, Vol. 8902. 51–66.

Lars Birkedal and Aleš Bizjak. 2020. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. `https://iris-project.org/tutorial-material.html`.

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. *LMCS* 8, 4 (2012).

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. The Category-Theoretic Solution of Recursive Metric-Space Equations. *TCS* 411, 47 (2010), 4102–4122.

Ales Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing Obligations in Higher-Order Concurrent Separation Logic. *PACMPL* 3, POPL (2019), 65:1–65:30.

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

Michael Brandt and Fritz Henglein. 1998. Coinductive Axiomatization of Recursive Type Equality and Subtyping. *Fundamenta Informaticae* 33, 4 (1998), 309–338.

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR*. 222–236.

Iliano Cervesato and Frank Pfenning. 1996. A Linear Logical Framework. In *LICS*. 264–275.

Coq Development Team. 2020. The Coq Proof Assistant Reference Manual, Version 8.12.0. (2020). `https://coq.inria.fr/distrib/current/refman/`

Andreea Costea, Wei-Ngan Chin, Shengchao Qin, and Florin Craciun. 2018. Automated Modular Verification for Relaxed Communication Protocols. In *APLAS*. 284–305.

Florin Craciun, Tibor Kiss, and Andreea Costea. 2015. Towards a Session Logic for Communication Protocols. In *ICECCS*. 140–149.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*. 207–231.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session Types Revisited. In *PPDP*. 139–150.

Ankush Das and Frank Pfenning. 2020a. Session Types with Arithmetic Refinements. In *CONCUR (LIPIcs)*, Vol. 171. 13:1–13:18.

Ankush Das and Frank Pfenning. 2020b. Verified Linear Session-Typed Concurrent Programming. In *PPDP*. ACM, 7:1–7:15.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*. 137–150.

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

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *TCS* 103, 2 (1992), 235–271.

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

Simon J. Gay. 2016. Subtyping Supports Safe Session Substitution. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 95–108.

Simon J. Gay and Malcolm Hole. 2005. Subtyping for Session Types in the pi Calculus. *Acta Informatica* 42, 2-3 (2005), 191–225.

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

Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala Step-by-Step: Soundness for DOT with Step-Indexed Logical Relations in Iris. *PACMPL* 4, ICFP (2020), 114:1–114:29.

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *APLAS (LNCS)*, Zhong Shao (Ed.), Vol. 4807. 19–37.

Jafar Hamin and Bart Jacobs. 2019. Transferring Obligations Through Synchronizations. In *ECOOP*. 19:1–19:58.

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*. 235–245.

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-type Based Reasoning in Separation Logic. *PACMPL* 4, POPL (2020), 6:1–6:30.

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2021a. Coq Mechanisation of Actris. Available online at `https://gitlab.mpi-sws.org/iris/actris/-/tree/lmcs`.

Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021b. Machine-checked semantic session typing. In *CPP*. 178–198.

Aquinas Hobor, Andrew Appel, and Francesco Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP*. LNCS, Vol. 4960. 353–367.

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP*. 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *POPL*. 273–284.

Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. 2010. Type-Safe Eventful Sessions in Java. In *ECOOP*. 21–25.

Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2019. Session-OCaml: A Session-Based Library with Polarities and Lenses. *Science of Computer Programming* 172 (2019), 135–159.

Iris Development Team. 2021. The Mechanisation of Iris. (2021). `https://gitlab.mpi-sws.org/iris/iris/`

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2021. Connectivity Graphs: A Method for Proving Deadlock Freedom Based on Separation Logic. Manuscript under submission.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *PACMPL* 2, POPL (2018), 66:1–66:34.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *CACM* 64, 4 (2021), 144–152.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *ICFP*. 256–269.

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* 28 (2018), e20.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650.

Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR*. 233–247.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1996. Linearity and the pi-Calculus. In *POPL*. 358–371.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *PACMPL* 2, ICFP

(2018), 77:1–77:30.
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*. 696–723.
Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. 205–217.
Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *ESOP*. 336–365.
Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A Static Verification Framework for Message Passing in Go Using Behavioural Types. *ICSE* (2018), 1137–1148.
Sam Lindley and J. Garrett Morris. 2016. Talking Bananas: Structural Recursion For Session Types. In *ICFP*. 434–447.
Étienne Lozes and Jules Villard. 2012. Shared Contract-Obedient Endpoints. In *ICE*. 17–31.
William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A Verified Messaging System. *PACMPL* 1, OOPSLA (2017), 87:1–87:28.
Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. 2004. The KRAKATOA Tool for Certification of JAVA/JAVACARD Programs Annotated in JML. *JLP* 1-2 (2004), 89–106.
Yusuke Matsushita and Jacques-Henri Jourdan. 2020. Flexible number of logical steps per physical step. `https://gitlab.mpi-sws.org/iris/iris/-/merge_requests/595` Iris merge request.
Dimitris Mostrous and Nobuko Yoshida. 2015. Session Typing and Asynchronous Subtyping for the Higher-Order π-Calculus. *Information and Computation* 241 (2015), 227–263.
Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP*. 316–332.
Hiroshi Nakano. 2000. A Modality for Recursion. In *LICS*. 255–266.
Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP*. 290–310.
Kosuke Ono, Yoichi Hirai, Yoshinori Tanabe, Natsuko Noda, and Masami Hagiya. 2011. Using Coq in Specification and Program Extraction of Hadoop MapReduce Applications. In *SEFM*. 350–365.
Wytse Oortwijn, Stefan Blom, and Marieke Huisman. 2016. Future-based Static Analysis of Message Passing Programs. In *PLACES*. 65–72.
Luca Padovani. 2014. Deadlock and Lock Freedom in the Linear π-Calculus. In *CSL*. 72:1–72:10.
Luca Padovani. 2017. A Simple Library Implementation of Binary Sessions. *JFP* 27, 2010 (2017), e4.
Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and Proving With Distributed Protocols. *PACMPL* 2, POPL (2018), 28:1–28:30.
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*. 80–95.
Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP*. 149–168.

Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite Step-Indexing: Decoupling Concrete and Logical Steps. In *ESOP*. 727–751.

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. *PACMPL* 4, ICFP (2020), 121:1–121:30.

Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. 2013. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?. In *ECOOP*. 302–326.

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP*. 909–936.

Peter Thiemann and Vasco T. Vasconcelos. 2020. Label-Dependent Session Types. *PACMPL* 4, POPL (2020), 67:1–67:29.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent session types via intuitionistic linear type theory. In *PPDP*. 161–172.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2014. Corecursion and Non-Divergence in Session-Typed Processes. In *TGC (LNCS)*, Vol. 8902. 159–175.

Bernardo Toninho and Nobuko Yoshida. 2018. Depending on Session-Typed Processes. In *FOSSACS (LNCS)*, Vol. 10803. 128–145.

Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *ASPLOS*. 865–878.

Vasco Thudichum Vasconcelos. 2012. Fundamentals of session types. *Information and Computation* 217 (2012), 52–70.

Jules Villard, Étienne Lozes, and Cristiano Calcagno. 2009. Proving Copyless Message Passing. In *APLAS*. 194–209.

Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically Verified Refinements for Multiparty Protocols. *PACMPL* 4, OOPSLA (2020).